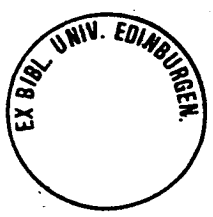
Data Abstraction and the Correctness of Modular Programming

Oliver Schoett

Doctor of Philosophy
University of Edinburgh
1986



Abstract

A theory of data abstraction in modular programming is presented that explains why this technique leads to correct programs.

Data abstraction allows users and implementers to take different views of a specification: While users may depend on a specification as it is, implementers need not provide program entities that satisfy the specification, but merely a "representation" of such entities. This means that the users may be supplied with program entities that do not satisfy the specification, and so an explanation is needed that their code functions correctly nevertheless.

It is shown that data abstraction leads to correct programs if the modules of the programs are "stable", and it is suggested that programming languages for data abstraction should guarantee stability of their modules. The stability criterion corresponds closely to the intuitive idea of "limited access" to encapsulated data types, and to "representation independence" properties of the typed λ-calculus.

The theory is developed in the general framework of an "institution" and uses an abstract notion of "representation". Specifically, the institution of partial many-sorted algebras is considered and the representation relations "behavioural inclusion", "behavioural equivalence" and "standard representation" (the popular concept based on abstraction functions) are studied. These relations are characterized by certain kinds of relations between algebras, called "correspondences", which provide useful practical proof methods for the correctness of data representations.

Behavioural equivalence is found to be superior to standard representation in that "representation bias" of a specification no longer restricts its range of implementations, and in that it allows more constructs to be included in a data abstraction programming language.

Table of Contents

Acknowledgements

My supervisor, Professor Rod Burstall, has given me much-needed encouragement and good advice, and urged me to try to reach the ground of practical examples from the theoretical clouds I was usually drifting in.

It was exciting that just a few steps down the corridor at Edinburgh, Don Sannella and Andrzej Tarlecki were working on program development and data abstraction from a slightly different angle, and I have learned much from their work and our discussions.

Many other people have helped me with their comments, with answers to questions, or by discussing issues with me; in particular, I would like to thank Wilfried Brauer, Helen Fraser, John Gray, Robin Milner, Eugenio Moggi, Tobias Nipkow, Gordon Plotkin, Brian Ritchie, Alistair Sinclair, and Allen Stoughton.

The staff of the Edinburgh University Computer Science Department have been very kind and helpful; in particular, Ms. Eleanor Kerse has helped me to produce the final version of this thesis.

My PhD study at the University of Edinburgh has been funded by the *Studienstiftung des deutschen Volkes* and by the University of Edinburgh. The writing of [Schoett 85] was sponsored by the Science and Engineering Research Council. My new employers, the *Technische Universität München*, have allowed me to continue to work on the thesis. My parents, Ursel and Gerd Schoett, have very generously supported me whenever I needed it, and so enabled me to concentrate on this work.

The enjoyment of one's tools is an essential ingredient of successful work.

Donald E. Knuth [Knuth 81, p. 223]

Chapter 1

Introduction

THIS THESIS presents a theoretical explanation for the correctness of programs obtained from a modular programming discipline using data abstraction. Data abstraction is viewed as a technique that allows users and implementers to take different views of a specification. While users may depend on a specification as it is, implementers are only required to provide program entities that "represent" program entities satisfying the specification, but that need not satisfy the specification themselves. This means that users of a specification may be supplied with program entities that do not satisfy the specification on which they base their correctness arguments. Thus, an explanation is needed that this kind of data abstraction leads to correct programs.

In the following section, the rôle of data abstraction in modular programming is explained in more detail, motivating the view that data abstraction consists in allowing users and implementers to view a specification differently.

Section 1.2 reviews the explanations given in the literature for the correctness of data abstraction. As we shall see, only a few papers have dealt with the problem.

Section 1.3 outlines the approach of this thesis to the problem: A theory of modular program construction is presented in the framework of an "institution"—a notion due to Goguen and Burstall [GB 84] and related to the ideas of "abstract model theory" [Barwise 74]. Data abstraction is based on an abstract "representation" relation between program entities. It is proved that correct programs result from modules that are "stable" and that are "simple implementations" of their specifications. The "simple implementation" property

describes the proof obligation of a programmer, while "stability" is a property to be guaranteed by the programming notation in which the modules are written.

As a concrete example of the abstract theory, partial many-sorted algebras are considered, which model functional programs. Three representation concepts are introduced and compared: "standard representation", the well-known notion based on Hoare's paper [Hoare 72], "behavioural equivalence", a notion which has more recently received attention in the literature, and a new notion, "behavioural inclusion". New proof methods for the behavioural representation concepts are given, and it is shown that these concepts, unlike standard representation, are tolerant of "representation bias" in specifications.

Section 1.4 presents an example of modular program development using data abstraction, which will be used in the remainder of the thesis to illustrate the theoretical concepts. Also, the example shows an important advantage of "abstract model" over "implicit" specifications: Only an abstract model specification of a data type allows one to develop and refine access operations independently of each other.

1.1 Data Abstraction in Modular Programming

Modular programming is a strategy to reduce the difficulty of designing, verifying, or modifying a program by structuring the program into a number of subcomponents, called "modules", with precisely defined interconnections, called "interfaces". The idea is that the correctness of the program as a whole should follow from the correctness of the individual modules with respect to their interfaces, so that the modules can be considered separately. Accordingly, each module has associated interfaces that describe

- the properties the module must guarantee ("export interfaces"), and
- the properties the module may depend on ("import interfaces").

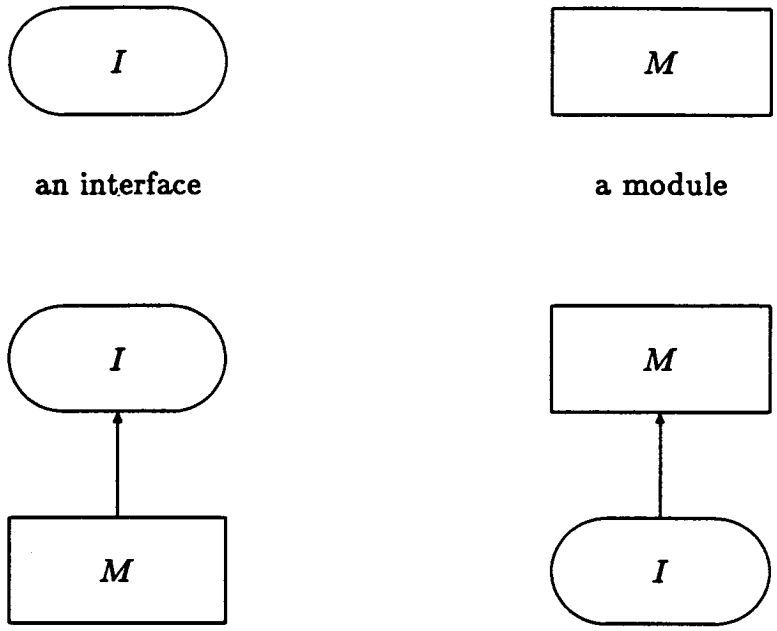


Figure 1-1: The elements of a design graph

An interface thus describes some properties of the program or its components; it may be the the export interface of one and the import interface of a number of other modules.

The relations between the modules and interfaces of a program can be expressed as a graph, which will be called a "design graph". Such a graph consists of the elements shown in Figure 1-1. For example, Figure 1-2 shows the design graph of a program with two modules M and N, where M guarantees the properties described by I (*i. e.*, exports I) and depends on the properties described by L and K (*i. e.*, imports L and K), and the module N exports L and imports J and K. This could be the design graph of a complete program, where I describes the properties required by the program's users, and J and K describe properties guaranteed by the programming environment (*e. g.*, the programming notation and the libraries available).

The graphs considered in this thesis will always be "hierarchical", which means that no loops are possible when travelling along the arrows, or equally,

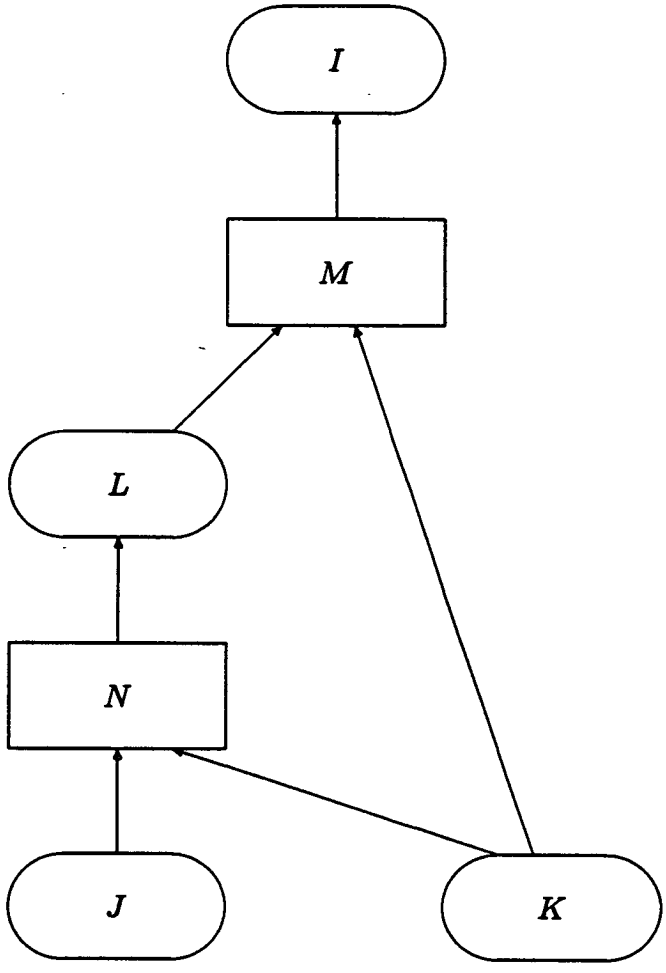


Figure 1-2: An example of a design graph

that the graph can be arranged in levels so that the arrows lead from "lower" to "higher" levels only. Figure 1-2 shows such an arrangement, as all the arrows lead upwards on the page. For the future, this will be adopted as the convention, so that the arrows can be replaced by simple lines.

A hierarchical system structure is generally regarded as highly desirable (*e.g.*, [Dijkstra 68, p. 343 f.], [Parnas 72b, p. 1057 f.], [Dijkstra 72, p. 48–50]), and on page 151–154 below it will be argued that modular programming *requires* a hierarchical system structure, since otherwise the correctness of the composed system could not be inferred from the correctness of the individual modules.

So far, the discussion has largely followed Parnas' discussion of system structure in [Parnas 72a, p. 339 f.]. Note that the issue of modular program structure

is distinct from the issue of design strategy: the structure represented by a design graph could be arrived at in different ways—for example, by "top-down" or by "bottom-up" design. The theory of this thesis deals only with program structure and does not depend on any particular design strategy.

In this thesis, program modules are viewed as defining ("exporting") "program entities", such as data types, data values, data objects, and operations, to be used by other modules or by users of the program. Practically, this means that a module consists of a group of program entity definitions in some programming notation. Many modern programming notations offer constructs to designate such modules—the first of these is SIMULA 67 [DN 66], and the most prominent one is ADA [ANSI 83]. The definitions in a module may be based on program entities that are "imported" from other modules or from the programming environment.

The connections between modules thus consist of program entities that are exported by one and imported by other modules, together with the properties of these program entities that the importing modules may depend on and that the exporting module must guarantee.

Accordingly, interfaces are given as specifications of groups of program entities; such groups in general will be called "structures". A structure that has the properties ascribed to it by an interface is said to "satisfy" the interface or to be a "model" of it; an interface can have any number of models. An interface with no models is called "inconsistent", an interface with several models is called "loose".

The meaning of a modular program is determined by the modules, which must be coded in a programming notation. The interfaces, on the other hand, do not contribute to the meaning of the program, so that they need not be coded in a formal notation. Nevertheless, many authors feel that specifications should be formal, so that they have a well-defined semantics and formal techniques such as machine processing become applicable (*e. g.*, [LZ 75, p. 8 f.]). On the other hand, some argue that most of the benefits of formal specifications can already be achieved by using a "rigorous" method [Jones 80, p. 9–14]. In any case,

the search for appropriate specification notations continues to be of considerable importance in Computer Science.

An important goal in the design of specification notations and in the design of modular programs is to obtain simple interface specifications. "Simplicity" here means not just short, elegant specifications, but also, as Parnas [Parnas 72a] has pointed out, that an interface should contain precisely the right ("designer controlled") amount of information: while too little information in an interface would impede its use, too much information in an interface would tend to make the interface more complex, and would restrict the range of models.

Data abstraction is a technique to simplify interfaces that describe encapsulated data types or data objects. A type or object is said to be "encapsulated", if access to the type or object is restricted to a fixed set of basic operations, called the "access operations" of the type or object, so that any operation involving the type or object must be realized on the basis of these operations.

Of course, every data type or data object in programming comes equipped with a certain set of basic operations, so that it may be said to be encapsulated; the novelty in programming notations that support data abstraction is a feature that allows the programmer to *define* the set of access operations to a data type or object. Usually this is done by declaring the type or object to be "private" to a module: Inside that module, the type or object is defined together with a number of operations, which are defined as usual on the basis of the access operations provided by the type or object definition; outside of the module, however, only those operations of the type or object are available that are explicitly exported from the module. In particular, the basic operations provided by the type or object definition are not available outside the module unless exported explicitly, so that information about the way the type or object is defined is not propagated to the remainder of the program.

The central idea of data abstraction is that the behaviour of a program using an encapsulated data type or data object depends only on those aspects of the encapsulated type or object that constitute its "observable behaviour".

The behaviour of a program consists in the results produced by its possible computations on input values. Specifying or comparing the behaviour of programs is possible only if input and output values belong to fixed, predefined data types or data objects that may be called the "visible" types and objects of the program. The other types and objects of the program affect the program's behaviour only indirectly, namely via the visible results produced by computations involving them.

When data types and data objects are encapsulated by a number of access operations, the computations involving the types or objects can generate values of the types or objects by means of the access operations and use these values as input to further access operations. Some of the access operations may produce result values *outside* the encapsulated types or objects, which can then be further processed by operations outside the encapsulation. Only the result values of these access operations can ultimately affect the behaviour of a program using the encapsulation, because the program's behaviour manifests itself in the visible types and objects, and because these access operations provide the only means to pass from the inside to the outside of the encapsulation.

Thus, the "observable behaviour" of encapsulated types and objects, which alone affects the behaviour of programs using them, consists of the results that the possible combinations of access operations produce in types or objects outside of the encapsulation.

The idea that only the observable behaviour of encapsulated types and objects affects the behaviour of programs using them can be exploited to simplify interfaces describing such types and objects. These interfaces should be "minimal" [LZ 75] or "abstract" [Guttag 77] [Parnas 79] in the sense that they only characterize the observable behaviour of the encapsulated types and objects and avoid specifying unobservable aspects, which could clutter the interface specification, require additional verification effort and even restrict the range of structures satisfying the interface.

Research in data abstraction has to a large extent been concerned with the search for specification notations that allow one to describe abstract interfaces. The ideal here would be a specification notation whose sentences could express only observable properties, so that all specifications in this notation would be completely abstract.

Unfortunately, no such specification notation has been found that would be practically useful.

To appreciate the problems here, consider the classic specification of the stack data type (with access operations *empty*, *push*, *pop*, and *top*) over some element data type *elem* by means of the equations

$$\begin{aligned} &\text{for all } s \in \mathit{stack},\ x \in \mathit{elem}\text{:} \\ &\quad \mathit{top}(\mathit{push}(x, s)) = x, \\ &\quad \mathit{pop}(\mathit{push}(x, s)) = s. \end{aligned}$$

The second equation relates two values of type *stack*. This equality of stacks is not observable to a user of the stack data type, since an equality operation for stacks is not among the access operations given above (this is normal, as the stack data type is perfectly useful without it). In particular, there are representations of the stack data type that do not preserve this equality, such as the familiar representation by an array together with a counter of elements on the stack. In this representation, a term of the form $\mathit{pop}(\mathit{push}(\_, s))$ may result in an array in which a cell has changed compared to s, namely the cell used to hold the element that was pushed on the stack.

We see here that using equations between *stack* values may lead to specifications that are not fully abstract. But simply forbidding equations between stacks would not be a solution either: In the resulting language, only equations between terms of sort *elem* could be written, which would correspond to observable facts. However, the (behaviour of the) stack data type can then no longer be specified in a finite number of axioms, even if full first-order predicate logic is used (I have proved this around 1980, but the proof has not been published and is too complex and remote from the subject of this thesis to be included here).

It is easy to see intuitively why this is impossible: The permitted atomic formulas are equations between terms of sort *elem*. In each such term, only a finite number of *pop* operations can occur, and hence the value of the term is not affected by how the stack deals with values that have been pushed to more than a certain maximal depth beyond the top of the stack. A finite set of sentences of first-order logic contains only a finite number of terms, and so there exists a certain finite depth beyond which the behaviour of the stack does not affect the validity of the formulas any more. In other words, given a finite set of first-order formulas whose atomic formulas are *elem* equations only and that hold for the "true" stack data type, we can define K to be the maximal number of occurrences of *pop* operations in any term of the formulas, and can then construct a model that deals incorrectly with values pushed beyond depth K from the top of the stack, but which nevertheless satisfies all the formulas.

It thus appears that practical interface specifications for encapsulated data types and objects cannot be fully abstract in general, but that they often have to prescribe unobservable facts, such as equality of values of encapsulated types. But such specifications prescribe more than just the observable behaviour of their models, so that some structures may fail to satisfy the specification despite exhibiting correct observable behaviour. An example for this is the array-and-counter representation for stacks mentioned above, which does not satisfy the equation $pop(push(x, s)) = s$.

An indication that this problem has long been recognized in data type theory can be seen in the large number of papers dealing with "implementation" concepts for specifications of encapsulated data types. (Data type theory has mainly dealt with encapsulated types rather than with encapsulated objects; the following discussion therefore concentrates on encapsulated types. Theories of encapsulated types should be easily applicable to abstract objects also, since with each object we can associate the type of its values.) The notion of "implementation" can be seen as a generalization of the usual satisfaction notion for interface specifications, whose purpose is to characterize all the acceptable representations of an encapsulated data type despite the fact that the specification might prescribe unobservable properties.

An "implementation" of a specification can be defined to be a structure representing a type together with its access operations [GHM 78], or a specification of such a structure [EKMP 82] [Ehrich 82] [Ganzinger 83]; very often implementation concepts are based on a relation ("representation") between structures [Milner 71] [Hoare 72] [GTW 78] [SW 82] [Lipeck 83]—an implementation of a specification can then be defined as a structure that "represents" a structure (the "abstract model") satisfying the specification, or as a specification whose models have this property. This approach has the advantage that the representation relation can be considered independently of the specification notation and its semantics.

A natural way of obtaining a representation relation for encapsulated data types is to formalize the notion of the "observable behaviour" of such structures [GGM 76] [Bothe 81] [Reichel 81] [Schoett 81] [GM 82] [SW 83] [Kamin 83]; comparisons of the various notions can be found in [HR 83] and [ST 84a, p. 17]. A representation relation is obtained by requiring the representing structure to have the same observable behaviour as the structure represented, *i. e.*, to be "behaviourally equivalent" to it.

A straightforward way of applying these ideas has been presented by Sannella, Tarlecki, and Wirsing in their "ASL" approach [SW 83] [ST 84a] [ST 85]. ASL is a "kernel specification language" featuring a "behavioural abstraction" operation that may be applied to an arbitrary specification and yields a new specification whose models are defined to be those structures that are behaviourally equivalent to some model of the original specification. Thus, a specification obtained by behavioural abstraction is by definition fully abstract in that only observational aspects of its models are prescribed.

Accordingly, ASL allows one to take a very simple view of data abstraction: to describe an encapsulated data type, one writes a specification P as usual (an arbitrary technique may be used for this, since ASL is "institution-independent", hence can be used with all specification notations), and then applies the behavioural abstraction operator to it, yielding the specification $\mathbf{behaviour}(P)$. The

correct implementations of the encapsulated type are then exactly the models of **behaviour**(P).

As illustrated in [ST 84b], one technique to prove an implementation correct with respect to a specification of the form **behaviour**(P) is to exhibit a model of P and to show that the implementation is behaviourally equivalent to it; this closely corresponds to the usual proofs of data representation correctness.

The view of program development with data abstraction taken in connection with ASL is very simple: A specification is transformed by successive refinement steps until an executable specification, *i. e.*, a program, is obtained. The refinement relation between specifications is simply the inclusion relation between their model classes, so that the program obtained at the end is trivially correct. Data abstraction consists in using the behavioural abstraction operator on specifications, and poses no further methodological problems.

Elegant and simple as the ASL approach to program development may seem, there is a serious problem. While data representation correctness proofs, which consist in proving satisfaction of a specification of the form **behaviour**(P), can be performed as usual, the users of an abstract data type cannot rely on the original specification P, but must base their correctness arguments on the specification **behaviour**(P). The first problem here is that the specification **behaviour**(P) is weaker than P, in that it characterizes the possible representations of the models of P. Secondly, the class of models of **behaviour**(P) is defined semantically, and in general no finite set of axioms can fully describe **behaviour**(P), so that using this specification in correctness arguments may become difficult.

Assume, for example, that P is the classic specification of the stack data type that was discussed above. In the ASL approach, users of the data type would have to use the specification **behaviour**(P), whose models are all those structures that have the same observable behaviour as a model of P, where values of the element data type *elem* are considered observable and those of type *stack* unobservable. As was discussed above, there is no finite set of sentences, even in full first-order logic, that would characterize the models of **behaviour**(P).

The equation $pop(push(x, s)) = s$ of P would not be available to the users, since it describes an unobservable property that is not satisfied in all representations (*e. g.*, it does not hold in the array-and-counter representation discussed above). Hence the elegance and simplicity with which the original specification P describes stacks would largely be lost to the users.

As another example, consider a data type of integers that might be provided as a basic data type by a programming notation. Since nonstandard representations of integers are possible (*e. g.*, a sign-and-magnitude representation with two values representing zero), the users of the integers could not rely on the familiar mathematical laws, but would have to use a different, weaker theory that would chracterize the observable behaviour of the integers.

Thus, the ASL approach causes practical difficulties by not allowing the original specification P of an encapsulated type to be used, but only the weaker and less familiar specification $\mathbf{behaviour}(P)$.

I feel that to achieve the main goal of data abstraction, namely to simplify interfaces describing encapsulated data type and objects, users must be allowed to use the original, "abstract" specifications, despite the fact that these specifications might not be satisfied by the actual representations of the types or objects. Abstract specifications should be written in the form that is most convenient; in particular, data types might be specified abstractly by mathematical models (*e. g.*, integers, tuples, or sequences), and users should be allowed to use the familiar mathematical properties of these models in their correctness arguments.

This thesis is therefore based on the following view of data abstraction:

> Data abstraction in modular programming allows an interface to be viewed in two different ways. While a module importing an interface may depend on all its properties, the module exporting an interface need not guarantee all its properties, but only has to provide a structure "representing" a structure with these properties.

Clearly, data abstraction in this sense implies that modules importing an interface will not always be supplied with a structure having the expected properties.

Thus, an explanation is needed for the correctness of programs designed using data abstraction. The goal of this thesis is to develop a formal theory of modular programming with data abstraction, and to formulate the conditions under which it yields correct programs.

1.2 Approaches to the Correctness Problem

The previous section has discussed the view of program development taken in connection with ASL by Sannella and Tarlecki, where the correctness problem of data abstraction is solved in a simple way, namely by forcing users to rely on a derived specification **behaviour**(P) that characterizes the possible representations of an encapsulated data type specified by P. It was argued that this approach conflicts with the goal of data abstraction, which is to simplify interfaces by allowing "abstract" descriptions of encapsulated types.

A possible remedy of this problem in the approach of Sannella and Tarlecki would be to exhibit proof techniques that would make it as simple to use the specification **behaviour**(P) as it would be to use P itself. To date, however, the proof rules exhibited for the **behaviour** construct allow one to derive only "visible" formulas; that is, formulas whose variables (corresponding to input values) are of visible sort only, and that are obtained by logical connectives and quantification from equations between terms of visible sort [ST 84a, p. 15 and 17]. These proof rules in combination with first-order logic seem to be insufficient for program development; at least if we consider formal proofs (*e. g.*, because an automatical theorem prover is used). The argument showing this is based on the theorem mentioned before (page 8 f.) that the behaviour of the stack data type cannot be specified by a finite number of "visible" first-order formulas. The proof rule given by Sannella and Tarlecki would not allow a finite formal correctness proof of a program that depends on the property that values can be retrieved from a stack after they have been "buried" to an arbitrary depth under other values. A simple example of this problem is that it is not possible to prove formally that $test(n, x) = x$ for all $n \in \mathrm{N}$ and $x \in elem$, where *test* is defined by

the following recursive code on the basis of the stack data type that was given on page 8:

$$test(n\colon \mathbf{N},\ x\colon elem)\colon elem = top(multipop(n,\ multipush(n+1,\ x,\ empty())))$$
$$multipush(n\colon \mathbf{N},\ x\colon elem,\ s\colon stack)\colon stack = \textbf{if } n = 0 \textbf{ then } s \textbf{ else } multipush(n-1,\ x,\ push(x,s))$$
$$multipop(n\colon \mathbf{N},\ s\colon stack)\colon stack = \textbf{if } n = 0 \textbf{ then } s \textbf{ else } multipop(n-1,\ pop(s)),$$

The problem here is that to prove this program correct, we need an infinite number of formulas about the *stack* data type, namely for each $n \in \mathbf{N}$ the formula

$$top(pop^n(push_x^{n+1}(empty()))) = x, \tag{1}$$

where $push_x(s) := push(x,s)$ and $f^0(x) = x$, $f^{n+1}(x) = f(f^n(x))$. In an informal proof, one might argue that the formulas of type (1) are all consequences of the laws of the stack data type, and that since these formulas are "visible" (they are equations between *elem* values), they also hold in the behavioural abstraction of the stack specification. This argument cannot be turned into a formal proof, however, because the inference that all formulas of type (1) hold also in the behavioural abstraction of the stack specification would involve an infinite number of applications of the proof rule given by Sannella and Tarlecki.

Even if the informal proof of (1) just given was considered acceptable, the use of infinite sets of formulas would still make this a more cumbersome form of reasoning than a proof using the stack specification in its original form: By induction on $n \in \mathbf{N}$, one easily proves that

$$multipush(n,\ x,\ push(x,s)) = push(x,\ multipush(n,x,s))$$
$$multipop(n,\ multipush(n,x,s)) = s,$$

and obtains that

$$\begin{aligned} test(n,x) &= top(multipop(n,\ multipush(n+1,\ x,\ empty()))) \\ &= top(multipop(n,\ \textbf{if } n+1 = 0 \textbf{ then } empty() \\ &\qquad \textbf{else } multipush((n+1)-1,\ x,\ push(x,empty())))) \end{aligned}$$

$$= top(multipop(n,\ multipush(n,\ x,\ push(x, empty()))))$$
$$= top(push(x, empty()))$$
$$= x.$$

We see that the proof that $test(n, x) = x$ for all $n \in \mathbf{N}$ and $x \in elem$ is quite simple when the original *stack* specification is used, yet would require infinite sets of formulas of first-order logic when Sannella's and Tarlecki's methodology and their proof rule for the behavioural abstraction operator is used. This would make an informal proof quite cumbersome, and a formal proof (*e. g.*, in an automatic theorem prover) impossible.

Sannella [personal communication, 5 July 1986] suggests that a way to solve this problem with the ASL approach would be to design an institution with sentences powerful enough to express the infinite set (1) of sentences as a single sentence (the notation might be akin to the one actually used in (1) above).

The specification notation of such an institution is likely to be powerful enough to allow most encapsulated data types to be specified in a finite number of axioms that refer only to observable properties. There is still the question, however, of whether this notation will also be useful in practice, or whether the necessary enumerations of terms as in (1) above will be too cumbersome. If the notation turned out to be practically usable, it would be a major advance, because the notation would be an ideal specification notation for encapsulated data types (capable of expressing only observable properties and practically usable), which, as discussed in the previous section, has eluded researchers so far.

Quite often in the literature dealing with abstract data types and their implementation, we find that the problem of the correctness of user programs is not discussed at all or just mentioned in passing, with statements such as "the representation of an encapsulated data type may be changed freely" (*e. g.*, [GTW 78, p. 83] [LB 79, p. 283] [BW 84, p. 268]). This is essentially the informal motivation of data abstraction given in the previous section, which has become part of the "folklore" [Harel 80] of Computer Science:

> **Folklore.** The behaviour of a program in which values of an encapsulated data type are manipulated only by means of the proper access functions depends only on the observable behaviour of the type, and its correctness is not affected by a change of representation of the type.

Such an informal statement, however, is of little use in a formal theory of program development; what such a theory needs is a formal version of the statement that can be rigorously proved, and that can then be used in a proof that modular programming with data abstraction yields correct programs.

But the vagueness of the "folklore" causes problems not only in theory, but also in practice. Suppose, for example, that one wanted to use data abstraction in the design of programs in PASCAL [BS 82] [DS 85]. Since PASCAL does not have an "encapsulation feature" with appropriate visibility rules, coding conventions or a modification of the language are needed to ensure that only the permitted access functions to an encapsulated type are used outside the encapsulation.

But restricting the operations by which an encapsulated type may be accessed would not suffice to guarantee representation independence of user code; the data type constructors **array** and **set** of PASCAL create loopholes that must be closed too.

Consider the two functions

```
function equal1(x, y: T): boolean;
    var a: array[T] of boolean;
    begin a[x] := false;
          a[y] := true;
          equal1 := a[x] end;

function equal2(x, y: T): boolean;
    begin equal2 := x in [y] end;
    {[y] is the set with single member y,
     in is the membership test}.
```

These functions allow one to detect the equality of values of a type T, even if equality was not among the permitted access functions (PASCAL requires that T be an "ordinal type"; this means that T may be *integer*, *boolean*, *char*, an "enumerated type", or a subrange of one of these [BS 82, Section 6.4]). If T represents an abstract type such that several values of T represent the same abstract value, then the two functions may yield different results over the representation type T than over the abstract type, namely when x and y are different values of T that represent the same abstract value.

Thus, PASCAL code using an encapsulated type must not be allowed to use the type in the index position of the **array** type constructor, nor as the argument of the **set** type constructor. Other uses of encapsulated types as arguments for data type constructors, *e. g.*, as the component type of an **array**, **record**, or **file** type definition, do not cause such problems. We see that the notion of what constitutes "proper use" of an encapsulated data type is not as simple as it might have seemed.

A less severe, but still annoying problem exists in ADA [ANSI 83]. In ADA, a data type may be declared either **private** or **limited private** to restrict the visibility of its operations. For a **private** type, the equality operator of the representation remains visible. To allow the usual freedom in the design of representations, the representation equality must not be visible, and the encapsulated type must therefore be declared **limited private**. But this also disallows assignment to variables of the type (called "objects" in ADA). To some extent this can be remedied by declaring a procedure that performs such an assignment; but the use of procedure calls in place of assignment statements is likely to make a program less legible. Also, it remains impossible for user code to initialize variables in connection with their declaration.

The problem results from the fact that the predefined equality and assignment are lumped together in ADA, in that they can be made either both available or both unavailable. From the viewpoint of preserving the correctness of user programs under a change of data representation, it is not necessary to forbid assignment when the representation equality is made invisible, since assigning values of encapsulated types to variables does not cause any problems (although

some care would be necessary if nondeterministic operations were present [Nipkow 86]).

These examples illustrate the dangers of relying on informal ideas only as the basis of data abstraction: programming languages might leave loopholes or introduce unnecessary restrictions. A formalized notion of what constitutes "proper access" to an encapsulated type is not only essential for a theory of data abstraction, but can also help in the design of data abstraction programming languages.

Let us now turn to formal approaches to the correctness problem. While the early papers of Milner [Milner 71] and Hoare [Hoare 72] contain proofs that their methods lead to correct programs, the problem has then been neglected for quite some time.

Milner's method allows one to prove certain relations between programs by constructing a relation called a "simulation" between their state spaces. This method is applicable to programs related by a change in data representation, as illustrated in Milner's Example 2 [Milner 71, p. 485–487]. However, data representations are not proved correct "locally", but only in the context of the whole program. On the one hand, this means that there exists no correctness problem. On the other hand, the method does not correspond to the goal of modular programming to separate the correctness proofs of the data representation and of the code using it, because in a proof of correctness of a data representation, one is forced to consider the program as a whole.

Hoare's method [Hoare 72] does allow one to separate the correctness arguments for data representation and user code. A data representation is proved correct by stating a "representation invariant" and giving an "abstraction function" that maps the representation values satisfying the invariant to the abstract values they represent. Hoare shows that a correct program remains correct when an "abstract variable" (an encapsulated data object) is replaced by its representation [Hoare 72, p. 278 f.] (this section is labelled "Formalities").

Hoare formulates his correctness criterion for "abstract variables", but it can easily be transferred to encapsulated types. This has been done so often

that Hoare's concept may be called the "standard" representation concept for data types (see Section 4.5). However, Hoare's proof that user programs remain correct when abstract variables are replaced by their representations does not immediately generalize to abstract types. The reason is that data types can not only be used to declare variables (to which Hoare's proof would then apply), but also to define new data types. As the PASCAL examples given above illustrate, Hoare's proof idea can only be generalized to data types if one assumes that the available data type constructors are in some sense "well-behaved" [Schoett 81, Section 5.2].

The generalization of Hoare's idea to data types was carried out by myself ([Schoett 81]; a short version in English is [Schoett 83]). The paper uses a fairly different framework from Hoare's in that it deals with "program modules" that allow one to define new data types and operations on the basis of imported data types and operations. Such modules can be specified and implemented independently of each other, and a family of modules can be composed into a module representing their combined semantics. When a module is implemented, the import interface may be taken for granted, while the exported program entities need only "represent" a structure satisfying the export interface [Schoett 81, p. 104 f.]. Hence the theory allows users and implementers to view a specification differently, which means that it deals with data abstraction as viewed in this thesis. Accordingly, in the theory of [Schoett 81], the problem of the correctness of the composition of a system of independently implemented modules appears.

To solve the correctness problem, *i. e.*, to show that the composition of separately implemented modules behaves correctly, the paper imposes the restriction that each of the modules to be composed must have the so-called "homomorphism expansion property" ("HEP") to the effect that when a module is supplied with two different import structures that are related by a "homomorphism" (*i. e.*, a strong partial homomorphism in the terminology of this thesis—Def. 4.4.3), then the two results of the module on these import structures must be related by an extension of that homomorphism [Schoett 81, p. 119].

The paper then argues that program modules written in a concrete programming notation that access their import data types only via the permitted access

functions will satisfy the HEP. To justify this, it is shown informally that familiar data type constructors, such as those for sum and product types, have the HEP, and that functions defined recursively over given base functions have the HEP also.

The weak point of my 1981 paper is the HEP and my attitude that the HEP would always be satisfied in concrete programs.

The first problem with the HEP is that it is sufficient for the composability of module implementations, but stronger than necessary. This thesis will derive "stability" criteria that are both necessary and sufficient, and it will be seen that the HEP is overly restrictive for the behavioural representation concept used in [Schoett 81] (see Theorem 5.4.9 below). The HEP is, however, the appropriate stability criterion for the "standard representation" notion based on Hoare's paper [Hoare 72] (see Theorem 5.4.4 below).

The second problem with my 1981 approach is that the conjecture that the HEP would always be satisfied in concrete programs that use only "well-behaved" type constructors is stated and proved only informally. In particular, the notion of "well-behaved" type constructors should be formalized in order to rule out constructors like set and array of PASCAL. From the discussion of these constructors earlier in this section, it is clear that these constructors must not occur in a programming language supporting data abstraction. Hence the HEP or an improved "stability" criterion should not be regarded as a "natural law", but as a design criterion for data abstraction programming languages. From this point of view, it becomes even more important to derive stability criteria that are necessary and sufficient for the composability of module implementations rather than using *ad hoc* criteria like the HEP, since by using a too restrictive criterion, the design of programming languages could be hampered.

Very similar to the composition of program modules is the composition of "parameterized abstract data types" ("PADTs"), which has been treated extensively in the literature, beginning with [TWW 82] (an earlier version of which appeared in 1978) and [Ehrig *et al.* 80], and influenced by the "theory procedures" of CLEAR [BG 77].

PADTs are based on many-sorted algebras as models of data types and operations of programs, and the semantics of a PADT usually is a functor mapping import algebras to export algebras which are enrichments of the import algebras by new sorts and functions, which model new data types with their access operations. Often, this functor is the free functor defined by a pair of Horn clause specifications (the idea is due to [TWW 82]; the language of infinitary Horn clauses is the most general one that admits the free functor semantics [Tarlecki 83] [MM 84]).

Pairs of specifications describing import and export interfaces of a module or a PADT are also often considered in the literature, usually under the name "parameterized specification". Often, composition operators are defined directly for parameterized specifications (these developments begin with CLEAR [BG 77] [BG 80] [BG 81]).

The composition of PADTs or parameterized specifications involves a "signature morphism" connecting the import interface of one PADT or parameterized specification to the export interface of another one, which allows the entities exported by the latter PADT or parameterized specification to be "renamed" before being imported by the former. Semantically, these signature morphisms cause no new problems.

The correctness problem of data abstraction appears in connection with PADTs or parameterized specifications when we consider the composition of their implementations. After Goguen and Burstall [GB 80], this kind of composition is called "horizontal composition", and so the problem appears in the literature under the name "horizontal composition of implementations" of PADTs or parameterized specifications.

We shall now look at the treatment of the horizontal composition problem in the literature.

One problem we shall encounter is that implementations of PADTs or parameterized specifications can only be composed when the argument of each PADT or parameterized specification, which in general is supplied by another implementation, satisfies the import interface of that PADT or parameterized specification. This is not automatically guaranteed by the horizontal composition

theorems, but usually appears as an explicit condition. But this condition, which we shall call the "fitting condition", implies that each implementation must be designed so that it satisfies the import interfaces of the PADTs or parameterized specifications using it, so that the freedom is lost to design implementations independently of each other.

In [Schoett 81] this problem did not appear, because the modules that are composed do not have any semantic requirement, but are capable of operating on all syntactically correct arguments. Modules in concrete programming notations have this property to a large extent, in that definitions of new program entities have a well-defined semantics for almost every possible semantics of the program entities used in them. Exceptions to this may occur when a "standard" type is used that may not be redefined by the user, or when type constructors impose semantic requirements on their argument types (*e.g.*, the requirement mentioned earlier that in PASCAL the index type of the **array** type constructor and the argument type of the **set** type constructor must be "ordinal types").

We shall therefore also consider what the theorems of the literature say in the special case that the implementation PADTs or parameterized specifications have no semantic requirements, but admit all syntactically correct arguments. In that case, the fitting condition will always be satisfied, so that it is possible to design the implementations of a system independently of each other.

In [SW 82], the fitting condition appears explicitly among the conditions of the horizontal composition theorem, because the theorem requires a "theory morphism" σ': $\underline{R'} \rightarrow \underline{A'}$, where $\underline{R'}$ is the import interface of the implementation of a parameterized specification, and $\underline{A'}$ is the implementation of the argument [SW 82, p. 30]. That σ' is a "theory morphism" means that the requirements expressed in $\underline{R'}$ are consequences of $\underline{A'}$ after renaming according to σ'. This is just the fitting condition.

The horizontal composition theorem of [SW 82] remains interesting, however, if we assume that the implementing parameterized specifications have no semantic requirements. In that case the fitting condition is always satisfied, and it becomes possible to design the implementations of a system independently of each other. In the proof of the theorem, the correctness problem of data ab-

straction is solved by exploiting the fact that the parameterized specifications are written in a certain variant of the specification language CLEAR, in which the "data constraints" have been replaced by "hierarchy constraints" in order to give the language the desired composability properties. This language can thus be said to support data abstraction. It is not clear, however, how the theorems could be transferred to implementations written in other notations such as concrete programming languages, except in the case that there exists a translation of that language into the CLEAR variant investigated by Sannella and Wirsing.

In the theory of Goguen and Meseguer [GM 82], an extremely severe restriction is made. The "fitting morphism" in a horizontal composition must be of the form $F\colon T \to T_{\Lambda 1}$, where T is the import interface of a parameterized specification and of its implementation, and $T_{\Lambda 1}$ is the theory whose signature is that of the abstract argument, but that contains no axioms [GM 82, p. 279]. This means that an implementation importing program entities from another implementation may not rely on any properties of these program entities except their pure syntax. It does not help to assume that the implementing parameterized specifications have no semantic requirements, because the theory assumes that implementations have the same import interfaces as their specifications, so that the specifications of a system could not have any semantic requirements either. This would mean that no nontrivial semantic properties of imported program entities could be used at all, which seems to be an unrealistic restriction to put on program development.

In a paper by Ehrig and Kreowski [EK 83], the horizontal composition problem is split into two parts, called "inner actualization" and "outer parameterization". The "inner actualization" problem is concerned with comparing a parameterized specification and its implementation when they are applied to the same argument, and it is shown in the paper that this results in a correct "induced implementation". The "outer parameterization" problem is concerned with the situation that the results of a parameterized specification and its implementation are passed to another parameterized specification. This problem is closely related to the correctness problem of data abstraction, namely the question of how programs are affected by a change of representation of the encapsulated types

they use. Ehrig and Kreowski, too, point out that a solution to this problem is essential for a theory supporting the horizontal composition of implementations [EK 83, p. 282]. The problem is not solved in the paper, however, so that the theory of horizontal composition remains incomplete.

The composition theorem of [Ganzinger 83] is proved using the assumption that an implementation always satisfies its specification: "In what follows we can always assume that SPEC1 $\subseteq$ SPECP" [Ganzinger 83, p. 346]; SPEC1 is the specification and SPECP is the set of "programs" implementing it (both are sets of equations). Thus, of course, the correctness problem, which is caused by the fact that the program entities supplied by an implementation need not satisfy the export interface on which other modules may depend, does not occur in Ganzinger's treatment. Ganzinger justifies his assumption by arguing that the equations of a specification can be added to the "programs" of an implementation if necessary. But if we try to relate Ganzinger's equational "programs" to programs in a concrete programming notation, we see that the program obtained by adding such equations has very little to do with the old one: If, *e. g.*, we have programs that implement a list data type, adding a commutativity axiom for list concatenation ("$l_1 \circ l_2 = l_2 \circ l_1$") would mean that completely different (and unusual) code for the concatenation operation would have to be written that would cause the commutativity law to be satisfied. With regard to concrete programs, Ganzinger's assumption that SPEC1 $\subseteq$ SPECP cannot be made "without loss of generality", but it is a genuine restriction of the implementation notion.

In Lipeck's thesis [Lipeck 83], a "conservativity" criterion for PADTs is introduced, which serves a similar function to the "homomorphism expansion property" of [Schoett 81], but which is a little more restrictive. Lipeck presents two composition theorems. The first one [Lipeck 83, p. 73] deals only with "simple" PADTs, that is, PADTs whose semantic requirements are trivial (*i. e.*, admit all syntactically correct structures). Since the PADTs to be implemented must also be simple, this means that they may not depend on any semantic properties of their arguments, which is too restrictive for practice (it is similar to the restriction we get in the theory of Goguen and Meseguer [GM 82] if we assume that implementations have no semantic requirements).

Lipeck's second composition theorem [Lipeck 83, p. 78] allows PADTs with nontrivial requirements, but imposes a fitting condition by demanding "top-down-applicability" of the implementation PADTs to their arguments. As Lipeck himself points out, this condition may make it necessary to redesign implementations in order to make them composable with with the other implementations in a system [Lipeck 83, p. 86].

If, however, we assume that the implementation PADTs are "simple", *i. e.*, are "conservative" and have no semantic requirements, we get a composition theorem that is similar to the one given by myself [Schoett 81, p. 119]. Lipeck's "conservativity" criterion for PADTs [Lipeck 83, p. 57] is a little more restrictive than my "homomorphism expansion property" for modules [Schoett 81, p. 119]; on the other hand, the composition mechanisms studied by Lipeck allow renaming of program entities via signature morphisms and are therefore more general than my composition operation for modules.

In summary, the contributions to the correctness problem by the papers dealing with the horizontal composition of PADTs are the following. The papers that do not circumvent the correctness problem deal with it either by exploiting the properties of the particular notation used to define the parameterized specifications that are dealt with (modified CLEAR in [SW 82]), or by postulating a semantic condition for the PADTs that constitute the implementation ("conservativity" in [Lipeck 83]). The papers allow the implementation PADTs or parameterized specifications to have nontrivial semantic requirements; the way these requirements are dealt with is unsatisfactory, however, because their satisfaction is made an explicit "fitting condition" in the horizontal composition theorems. To satisfy this condition, implementations must be developed with a view to the requirements of the PADTs or parameterized specifications that will ultimately depend on them; this means that the implementations in a system cannot be designed independently of each other.

A recent contribution to the correctness problem is a paper by Nipkow [Nipkow 86], which is more general than most other papers in that it deals with

nondeterministic functions. Nipkow presents an implementation concept for encapsulated data type with nondeterministic operations [Nipkow 86, Def. 2.1.4], and then formulates a set of conditions on the semantics of a programming language that is sufficient to ensure that the observable behaviour of a program is not affected by a change of representation of an encapsulated data type [Nipkow 86, Theorems 4.1 and 4.2]. Nipkow shows the usefulness of these conditions by exhibiting a programming notation that satisfies them, and that can therefore soundly be used in connection with his implementation notion for encapsulated nondeterministic data types.

I conjecture that Nipkow's conditions are slightly more restrictive than necessary, because they require that every homomorphism between two models of an encapsulated data type can be extended to a homomorphism on the semantic domains of the language that relates the semantics of all program constructs over the two models. In the proof of Theorem 5.4.9 in this thesis, a data type constructor is presented that can safely be used in a data abstraction language, yet that does not allow to extend a homomorphism between models of its parameter type into a homomorphism between the enriched models produced by the constructor. I conjecture that this type constructor is admissible in Nipkow's framework also, yet that it violates Nipkow's sufficient criteria for a language to be sound.

Another line of research related to the correctness problem of data abstraction is the development of "representation independence" theorems for the typed λ-calculus [Reynolds 83] [Statman 85] and for the second-order typed λ-calculus [Donahue 79] [Haynes 84] [MM 85]. The general form of these theorems is (cf. [MM 85, p. 225]):

> Given a "logical relation" R between two models of the (second-order) typed λ-calculus, then for every closed term, its meanings in the two models are related by R.

These theorems have been called "representation independence" theorems, because they can be used to compare the semantics of programs over different

representations of their basic data types. These representations define λ-calculus models, and if two representations can be connected by a "logical relation", the theorems guarantee that for any program, its meanings over the two representations will be related.

These theorems do not apply directly to the correctness problem of data abstraction, because they deal with the meaning of one program in different models, and not with changes in representation of encapsulated types, which would mean that different, possibly abstract, programs would have to be compared.

Mitchell [Mitchell 86] has recently investigated how the "representation independence" idea can be applied to the correctness problem. Following [MM 85], Mitchell defines "programs" to be closed terms whose type is a member of a fixed set of "program types", and considers two λ-calculus models to be "observationally equivalent", if all programs have the same meaning in them (this is just the idea of "visible" types in data abstraction, cf. Section 4.2 below). In [MM 85], it is shown that observational equivalence of models is characterized by the existence of a logical relation that is the identity on the program types.

Mitchell attempts to transfer this idea to encapsulated data types by calling two definitions of an encapsulated type "observationally equivalent with respect to $\mathcal{R}$", where $\mathcal{R}$ is some given logical relation between models, if whenever the definitions are substituted into some program context, the meanings of the two resulting programs in the two models are related by $\mathcal{R}$. Mitchell presents a sufficient condition for this type of "observational equivalence": Whenever the two type definitions are applied in environments that are related by $\mathcal{R}$, it must be possible to extend $\mathcal{R}$ to the results produced by the two type definitions [Mitchell 86, Theorem 6].

This is almost, but not quite, general enough to solve the correctness problem of data abstraction. For, if we consider encapsulated type definitions that are based on other encapsulated type definitions, the relation $\mathcal{R}$ relating the environments of the "higher level" type definitions will not be fixed, but will be the extended relation relating the results of the lower level type definitions, and will thus depend on the particular representations chosen for the lower level types.

What is needed to solve the correctness problem of data abstraction is that type definitions do not just extend some fixed relation, but an arbitrary one.

It will be shown in Chapter 5 of this thesis that this kind of criterion does indeed provide a sufficient basis to prove that programs containing separately implemented encapsulated data types function correctly (Theorem 5.1.4). The relation between the theory of this thesis and the "representation independence" theories for the λ-calculus will then be further discussed in the Conclusions of this thesis.

1.3 Overview of the Thesis

The thesis presents a theory of modular programming with data abstraction that explains why this discipline leads to correct programs.

The theory is developed on an abstract level, based on the notion of an "institution". An institution comprises "signatures", which represent the syntactic properties of groups of program entities, and for each signature a set of "models", which represent the semantics of the program entities of the signature (note that the "models" of an institution correspond to "structures" in the terminology of the previous two sections). A "specification" or "interface" in an institution is a signature together with a set of models of that signature. The notion is thus independent of any particular specification notation. The "institution" notion used in this thesis differs somewhat from the one introduced by Goguen and Burstall [GB 84] in that "sentences" are omitted and an "inclusion" relation between signatures is added.

To put the theory into a more concrete setting, the institution of "partial many-sorted algebras" will be dealt with in detail. Partial many-sorted algebras model the data types and functions of strongly typed functional programs. A development of such a program is presented in the following section of the Introduction, and in the later chapters, this development will be analysed by means of the theory and provide the examples.

Chapter 2 below presents the basic notion of an "institution" and introduces the institution of partial many-sorted algebras.

In Chapter 3, a general theory of modular programming is developed in the context of an institution. The fundamental concept of this theory is that of a "cell", which consists of an import and an export interface, and which can represent concrete program modules as well as abstract modules or module specifications.

Modular programming is viewed as the construction of a "structured correctness argument", in which each cell is proved correct with respect to its specification, which also is a cell, and where the specification cells form a "decomposition" of a "global" cell that consists of the external import and export interfaces of the system. The purpose of such a "structured correctness argument" is to show that when the cells of the system are "composed" (corresponding to the operation of a compiler or linking loader), the resulting cell is correct with respect to the global cell.

The correctness notion appropriate for modular programming (without data abstraction) is the "refinement" relation between cells. A refinement of a cell M is a cell that produces results satisfying the export interface of M whenever it is supplied with program entities satisfying the import interface of M. With this notion of correctness, all the interfaces of a program will be satisfied by the program entities defined in the program.

Data abstraction enters the picture in Chapter 4. The theory of data abstraction in an institution is based on a "representation relation" between the models, which must satisfy a few simple axioms, but apart from that can be chosen arbitrarily. The idea is that the program entities defined by a program need no longer satisfy the interfaces themselves, but that they only need to "represent" a model satisfying them. This leads to a different correctness notion for cells, which is called "universal implementation". The central theorem of the thesis asserts the "composability of universal implementations"; *i. e.*, that "universal implementation" can be used as the correctness notion in a structured correctness argument: Given a family of cells that are universal implementations of

their specifications, which in turn form a decomposition of a global cell, the composition of the cell family is a universal implementation of the global cell.

In the later sections of Chapter 4, specific representation relations between partial algebras are considered. Section 4.2 argues that to make such representation relations useful for practice, the data types of a program in which it performs its input and output must be distinguished from the other, "internal" types of the program. This leads to a slight modification of the institution of partial many-sorted algebras to the effect that the signatures record a distinction between "visible" and "hidden" data types.

The remaining sections of Chapter 4 introduce three representation relations between partial many-sorted algebras: "behavioural inclusion", "behavioural equivalence", and "standard representation". The first two relations are based on the idea of the "observable behaviour" of an algebra.

Behavioural inclusion reflects the concept of a "partial" or "restricted" representation [KA 84], which may abort when its capacity is exceeded, but that otherwise must deliver correct observable results. Behavioural inclusion is characterized by the existence of a certain kind of relation between the algebras, which is called a "correspondence". To establish a correspondence between two algebras is a practically useful method to prove the correctness of a data representation with respect to behavioural inclusion.

Behavioural equivalence is the equivalence relation between partial algebras induced by behavioural inclusion. This relation holds between two algebras if all computations that start with values of visible types deliver the same observable results. It is the recommended representation relation for practical programming. Behavioural equivalence is characterized by the existence of a "strong correspondence" between the algebras. This yields a practically useful method to prove two algebras behaviourally equivalent.

Standard representation is the criterion due to Hoare [Hoare 72] that has commonly been proposed for the correctness of data representations; it requires the existence of an "abstraction function" from the representation algebra to the algebra it represents. Abstraction functions are strong correspondences that are also partial functions. The standard representation relation is more restric-

tive than behavioural equivalence; this is the reason that specifications with a "representation bias" [Jones 80, p. 259–265] do not allow certain representations to be proved correct using the standard representation criterion. The problem disappears if behavioural equivalence is used as the correctness criterion for data representation.

While the "universal implementation" relation can be used as the correctness notion in a structured correctness argument, and thus provides a theoretical basis for data abstraction in modular programming, it is too strong to be proved in practice for each module of a system. Chapter 5 shows how the "universal implementation" concept can be factored into two components called "simple implementation" and "stability". This decomposition requires not just a "representation relation", but a "representation category" over the models of each signature, where a model is a representation of another one, iff there exists a morphism in the representation category (*i. e.*, a "representation morphism") from the representation to the model it represents.

"Simple implementation" is a relation between cells that reflects the way data representation correctness proofs are performed in practice. In particular, in proofs of the "simple implementation" property of a cell, the imported program entities may be assumed to satisfy the import interface of the cell specification. This reflects the postulate that users of an encapsulated type should be allowed to use the abstract specification of the type rather than only the properties of its representations, which Section 1.1 has explained to be essential for the data abstraction programmig method.

It suffices to prove the "simple implementation" property of a cell with respect to its specification, if the implementation cell is "stable". This criterion admits an elegant characterization; in particular, a cell is stable, if each representation morphism between arguments can be extended to a representation morphism between the results of the cell on these arguments. Stability can serve as a design criterion for programming languages intended to support data abstraction, because if all the modules that can be written in the language are

stable, then programmers need only verify the "simple implementation" property of their modules.

The later sections of Chapter 5 present representation categories for the three representation relations between partial algebras introduced in the previous chapter. The representation morphisms for behavioural inclusion are correspondences, the morphisms for behavioural equivalence are strong correspondences, and the morphisms for standard representation are abstraction functions.

The stability notions for the three representation concepts have the property that a cell is stable, if and only if the respective representation morphisms can be extended from the arguments of the cell to its results. In particular, the stability property for behavioural equivalence turns out to be very similar to the "representation independence" theorems that have been proved for versions of the typed λ-calculus (see the previous section). This suggests that stability is a reasonable requirement to demand of the modules of a "data abstraction" programming language.

It is shown that under certain natural conditions, the stability notions for behavioural inclusion and for behavioural equivalence are equivalent to each other, while the stability notion for standard representation is more restrictive than these. Thus, not only does standard representation provide a more restrictive implementation criterion for the programmer, it also puts more restrictions on the programming languages that can safely be used in connection with it. Since the correspondence proof methods for the behavioural representation concepts are just as simple as the "standard" proof method using abstraction functions, the behavioural representation concepts should replace standard representation as the correctness criterion for data representations in practical programming.

1.4 An Example of Modular Programming with Data Abstraction

The program development in this section illustrates the view of modular programming and data abstraction that was introduced in Section 1.1. Furthermore, the interfaces and modules that occur here will serve as running examples throughout the presentation of the formal theory, so that keeping the present example in mind will help the reader relate the theoretical concepts to practical program development.

The development will proceed in a top-down fashion; that is, interfaces will as far as possible be designed according to the needs of the modules that use them. These needs will usually be determined during the design of these modules, so that the design of a module and of some of its import interfaces will often go hand in hand.

In particular, we will avoid introducing an encapsulated type and its access functions in a single step; since the type will be used by different modules, each of these modules determines an interface specifying those access functions that it needs. These access functions are refined independently of each other, and only after the refinement has reached a level of access functions that are considered "elementary", the complete group of the remaining access functions is considered together in order to design an implementation of the encapsulated type.

This kind of development requires that one can specify and refine the access functions to a type independently of each other. The "mathematical" specification technique that is used here makes this easy, because it allows one to explicitly specify a set of values for an encapsulated type (so that one can write "abstract model" specifications). One can then specify the operations in the traditional manner by referring to their input and output values, although other modes of specification that relate the operations to each other remain possible.

The programming problem to be solved here is the construction of a "dictionary": Given a text as input, that is, a sequence of words, an alphabetically ordered sequence is to be constructed that contains exactly the words occurring in the text, but each word only once.

More abstractly, we assume as given a data type *item* (containing words, for example), and a function

$$leitem\colon item\ item \to bool$$

(where $bool = \{\mathbf{T}, \mathbf{F}\}$ is the standard type of truth values) that is defined everywhere and is such that the relation $\leq \subseteq item \times item$ defined by

$$x \leq y \ :\Longleftrightarrow\ leitem(x, y) = \mathbf{T}$$

is a total ordering[1] on *item* (the symbol "*item*" in mathematical expressions denotes the set of values of type *item*). The relation $< \subseteq item \times item$ is defined in the usual way:

$$\begin{aligned} x < y \ &:\Longleftrightarrow\ x \leq y \ \wedge\ y \not\leq x \\ &\Longleftrightarrow\ leitem(x, y) = \mathbf{T} \ \wedge\ leitem(y, x) = \mathbf{F}. \end{aligned}$$

Figure 1-3 gives the specification of the *item* data type on which our program development is based. The semi-formal notation in which it is written will be used throughout the thesis and deserves some explanation. The whole construct is called an **interface** in agreement with the "interface" notion of Section 1.1, because it characterizes a group of program entities (*bool*, *item*, and *leitem*).

The **signature** part gives the syntactic properties of these program entities: the symbols labelled **sort** denote data types (the name "sort" is used in connection with algebras, see Section 2.2), the symbol *leitem* denotes a function with two arguments of sort *item* and result of sort *bool*.

The **properties** part specifies semantic properties of the program entities. The names of the program entities are used as normal mathematical symbols: a

---

[1]The definitions of all the mathematical concepts and notations used here can be found in Section 2.1.

I_{ITEM} =

interface

signature

bool, *item*: **sort**

leitem: *item item* $\rightarrow$ *bool*

properties

$bool = \{\mathbf{T}, \mathbf{F}\}$

$\mathrm{dom}\, leitem = \prod\langle item, item\rangle$

$\{\,(x,y) \mid leitem(x,y) = \mathbf{T}\,\}$ is a total ordering on *item*

Figure 1-3: The interface I_{ITEM}

type name denotes the set of values of the type, and a function name denotes a partial function from the product of the value sets of the argument types to the value set of the result type. In our example, it is thus a consequence of the **signature** that *leitem*: $\prod\langle item, item\rangle \nrightarrow bool$; in other words, that *leitem* is a partial function from the set of pairs of *item* values to *bool*. Further properties are then given using ordinary mathematical notation. In the example, the type *bool* is defined as the "standard" type of truth values, the function *leitem* yields a result for all possible arguments and describes a total ordering on *item* (namely the ordering $\leq \subseteq item \times item$ that was defined earlier).

The desired "dictionary" operation will have finite sequences of *item* values as argument and result. Such sequences will be written in the form $l = \langle l_1, \ldots, l_n\rangle$ $(n \geq 0)$, and we use the notations

$\mathrm{dom}\langle l_1, \ldots, l_n\rangle = \{1, \ldots, n\}$	(the index set of a sequence)
$\mathrm{ran}\langle l_1, \ldots, l_n\rangle = \{l_1, \ldots, l_n\}$	(the set of "elements" of a sequence)
$\lvert\langle l_1, \ldots, l_n\rangle\rvert = n$	(the length of a sequence)
$\langle l_1, \ldots, l_n\rangle \circ \langle k_1, \ldots, k_m\rangle$ $= \langle l_1, \ldots, l_n, k_1, \ldots, k_m\rangle$	(concatenation of sequences)

The set of all finite sequences of *item* values is written "*item**".

We assume a predefined data type *listitem*, whose values are the finite sequences of *item* values. This type together with its access functions is defined by the interface $I_{LISTITEM}$ shown in Figure 1-4.

We are now ready to specify our design goal, the "dictionary" operation. One requirement is that its result sequence should be strictly ordered according to the ordering $\leq$ described by *leitem*. This property is expressed by the predicate "Ascending" for sequences of *item* values:

$$\text{Ascending}(l) \ :\Longleftrightarrow \text{ whenever } 1 \leq i < j \leq |l|, \text{ then } l_i < l_j$$

(recall that $l_i < l_j$ means that $leitem(l_i, l_j) = \mathbf{T}$ and $leitem(l_j, l_i) = \mathbf{F}$).

The *dictionary* function is specified by the interface I_{DICT} in Figure 1-5. Besides the *dictionary* operation itself, this interface contains a number of other program entities, namely those that are required in the description of this operation. The type *listitem* is included with the specification of its set of values, because it is the argument and result type of *dictionary*. The specification uses the predicate "Ascending", which is defined using the relation $<$ on *item*, which in turn is defined on the basis of *leitem* and *bool*. Hence these program entities are also included in the specification, together with their semantic properties that ensure that the specification of the *dictionary* operation makes sense. Given an argument list l, this operation is required to produce a list containing exactly those *item* values that occur in l ($\text{ran}(dictionary(l)) = \text{ran}\, l$), and that is strictly ordered according to $\leq$ ($\text{Ascending}(dictionary(l))$). In particular, this condition ensures that the resulting list contains each *item* value at most once.

From a practical point of view, it would be reasonable to classify the symbols occurring in an interface such as I_{DICT} into two groups: into those that are to be defined by the interface (here, *dictionary*), and those that only provide the context for this definition. In particular, a programmer should not be misled into producing an implementation of program entities according to an interface in which they occur only as context symbols with perhaps just a subset of their

$I_{LISTITEM}$ =

interface

signature

bool, *item*, *listitem*: **sort**

nil: $\rightarrow$ *listitem*

cons: *item listitem* $\rightarrow$ *listitem*

isnil: *listitem* $\rightarrow$ *bool*

hd: *listitem* $\rightarrow$ *item*

tl: *listitem* $\rightarrow$ *listitem*

properties (x: *item*, $\langle l_1, \ldots, l_n \rangle$: *listitem*)

$bool = \{\mathbf{T}, \mathbf{F}\}$

$listitem = item^*$

$nil() = \langle\rangle$ (the empty list)

$cons(x, \langle l_1, \ldots, l_n \rangle) = \langle x, l_1, \ldots, l_n \rangle$

$$isnil(\langle l_1, \ldots, l_n \rangle) = \begin{cases} \mathbf{T}, & \text{if } n = 0 \\ \mathbf{F}, & \text{if } n > 0 \end{cases}$$

$\operatorname{dom} hd = \operatorname{dom} tl = listitem \setminus \{\langle\rangle\}$

$$\text{for } n \geq 1\text{:} \begin{cases} hd\langle l_1, \ldots, l_n \rangle = l_1 \\ tl\langle l_1, \ldots, l_n \rangle = \langle l_2, \ldots, l_n \rangle \end{cases}$$

Figure 1-4: The interface $I_{LISTITEM}$

properties. For example, it would be nonsense to implement *listitem* according to the interface I_{DICT}.

From the point of view of our theory, however, this distinction is not necessary. Each interface gives properties of some program entities, and it is some module's responsibility to ensure that the interface is satisfied. These responsibilities are recorded in a design graph. This thesis does not attempt to present a complete methodology for the construction of design graphs.

I_{DICT} =

interface

signature

bool, *item*, *listitem*: **sort**

leitem: *item item* $\rightarrow$ *bool*

dictionary: *listitem* $\rightarrow$ *listitem*

properties

$bool = \{\mathbf{T}, \mathbf{F}\}$

dom *leitem* $= \prod\langle item, item\rangle$

$\{(x, y) \mid leitem(x, y) = \mathbf{T}\}$ is a total ordering on *item*

$listitem = item^*$

for all $l \in listitem$:

$\text{ran}(dictionary(l)) = \text{ran}\, l$

$\text{Ascending}(dictionary(l))$

Figure 1-5: The interface I_{DICT}

Some basic rules, however, are that each program entity may be defined only once, and that when a module is to define a group of program entities, all those interfaces have to be exported and hence guaranteed by the module that mention some of these entities and have not already been exported by other modules.

In the example, it would not be appropriate to define *listitem*, because we assumed it to be defined already in the programming environment. Even if that was not the case, *listitem* could not be implemented with only I_{DICT} as the export interface; rather, one would have to take both I_{DICT} and $I_{LISTITEM}$ into account, since they both mention *listitem*.

We now begin to develop a modular program for the *dictionary* operation specified by I_{DICT}, on the basis of the interfaces I_{ITEM} and $I_{LISTITEM}$. That is to say, the program we are designing has to complete the design graph shown in Figure 1-6.

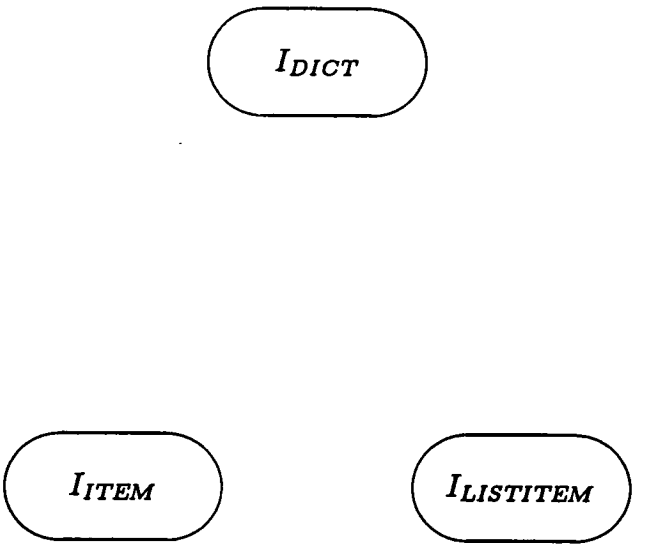


Figure 1-6: The initial design graph of the *dictionary* program development

The development presented here is artificial in the sense that the design decisions are not guided by practical criteria such as efficiency, but are motivated mainly by the desire to keep the whole development reasonably small while illustrating some principal features of a modular program development.

The first design decision is to decompose the *dictionary* function into two functions called *input* and *output*. The *input* function maps the input sequence into an internal data structure, and the *output* function maps this data structure into the desired output sequence. The internal data structure is an element of an encapsulated data type called *store*. In other words, we intend to define the *dictionary* function as the composition of the functions *input* and *output*, where *input* maps a *listitem* value to a *store* value, and *output* maps a *store* value to a *listitem* value. This definition constitutes the first module of the program, which is called M_{DICT} and given in Figure 1-7.

This figure describes a module that might actually have been written in a concrete programming notation; for example, Figure 1-8 shows how M_{DICT} might be coded as a package in ADA [ANSI 83].

M_{DICT} =

module

environment signature

listitem, *store*: **sort**

input: *listitem* → *store*

output: *store* → *listitem*

defined symbols

dictionary: *listitem* → *listitem*

requirements

(none)

result

dictionary = *input* ; *output*

Figure 1-7: The module M_{DICT}

The notation used in Figure 1-7 records such a module, including some facts that in a programming notation might be determined from the context.

The section **environment signature** has the same form as the **signature** section of an interface and lists those program entities that are imported (used, but not defined) by the module with their syntactic properties.

The **defined symbols** section gives the syntax of the additional symbols defined by the module. These symbols are supposed to be directly accessible outside the module, which in ADA is expressed by a **use** clause.

The **requirements** section of our notation gives the semantic properties of the imported program entities that are required *by the code*, which in general are much weaker than those needed to prove the code correct. In the example, the **requirements** section is empty, because the definition *dictionary* = *input* ; *output* does not impose any semantic requirements; it has a well-defined semantics for every interpretation of the environment symbols that matches their syntax. The same holds true for the ADA definition of *dictionary*—this operation is

```
package DICT is
    function dictionary(l: listitem) return listitem;
end;

package body DICT is
    function dictionary(l: listitem) return listitem is
        begin
            return output(input(l));
        end;
end;

use DICT;
```

Figure 1-8: M_{DICT} coded as an ADA package

well-defined in every context in which it is syntactically correct. A nonempty **requirements** section could arise from the use of programming language constructs that impose semantic requirements, and examples of this will occur below (in the modules M_{INPUT}, M_{OUTPUT} and M_{STORE} of Figs. 1-16, 1-18 and 1-19).

Finally, the **result** section of our notation defines the semantics of the **defined symbols** of the module. This section and the **requirements** section use the same mathematical notation as the **properties** section of an interface. The intention is that the definitions of the **result** section model data type and function definitions in a concrete programming notation (Clearly, the definition of *dictionary* in Figure 1-7 matches the ADA definition). To make this correspondence explicit, the function definitions of a module will often be given by recursive code (see Figures 1-16, 1-18 and 1-19 below). This code has the usual meaning of recursive function definitions using a call-by-value semantics.

The module M_{DICT} imports the program entities *store*, *input*, and *output*, which are to be defined by other modules of the program. We still need to record the semantic requirements on these program entities that are necessary for the *dictionary* function defined by M_{DICT} to be correct. Obviously, the

I_{INOUT} =

interface

signature

bool, *item*, *listitem*, *store*: **sort**

leitem: *item item* $\rightarrow$ *bool*

input: *listitem* $\rightarrow$ *store*

output: *store* $\rightarrow$ *listitem*

properties

$bool = \{\mathbf{T}, \mathbf{F}\}$

$\mathrm{dom}\, leitem = \prod\langle item, item\rangle$

$\{(x, y) \mid leitem(x, y) = \mathbf{T}\}$ is a total ordering on *item*

$listitem = (item)^*$

for all $l \in listitem$:

$\mathrm{ran}(output(input(l))) = \mathrm{ran}\, l$

$\mathrm{Ascending}(output(input(l)))$

Figure 1-9: The interface I_{INOUT}

necessary requirement is precisely that the composed function *input* ; *output* has the properties required of the *dictionary* function in I_{DICT}. This requirement is recorded in the interface I_{INOUT} in Figure 1-9. The properties required in this interface are just the ones of I_{DICT} with *dictionary* replaced by the composition of *input* and *output*.

The present stage of the program development can again be shown as a design graph. To the initial interfaces I_{DICT}, I_{ITEM} and $I_{LISTITEM}$, we add the module M_{DICT} with export interface I_{DICT} and import interface I_{INOUT}, obtaining Figure 1-10.

The module M_{DICT} is obviously correct with respect to its import and export interfaces: Whenever it is supplied with program entities satisfying I_{INOUT}, the result of the module, which consists of these entities together with the new function *dictionary*, will satisfy I_{DICT} (the correctness notion indicated here is

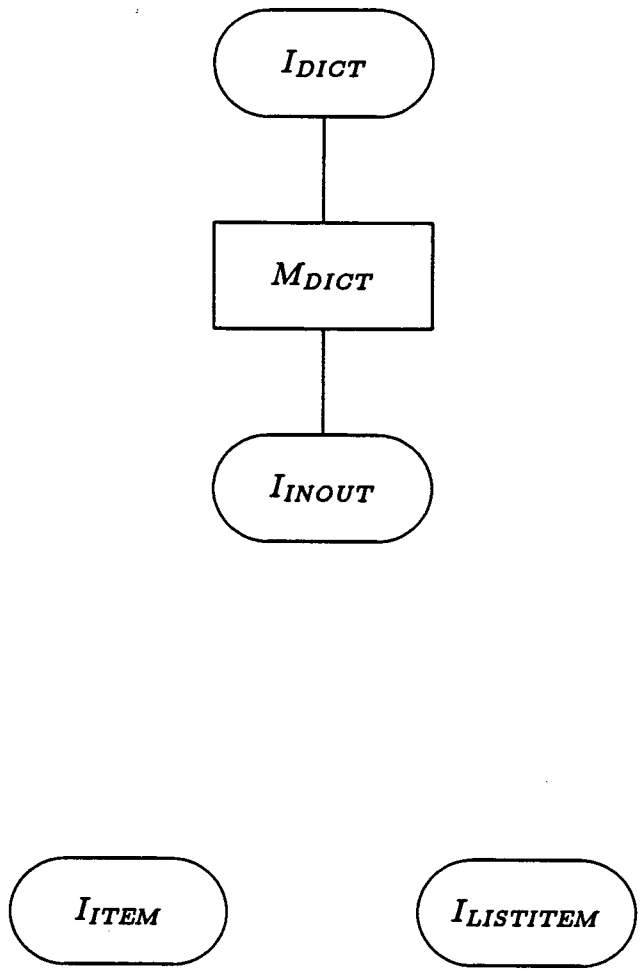


Figure 1-10: The design graph after addition of M_{DICT} and I_{INOUT}

the basic correctness notion for modular programming without data abstraction and is formalized in the "refinement" notion of the theory in Def. 3.1.18). Thus, in the remainder of the program development, we need no longer be concerned with the interface I_{DICT} nor with the module M_{DICT}; it remains to construct program entities so that I_{INOUT} is satisfied.

Our next design step is concerned with the interface I_{INOUT}. We could treat I_{INOUT} as the specification of an encapsulated data type *store* with access operations *input* and *output*, and proceed to implement this type. We shall not do so, however, but decide that *input* and *output* should be expressed on the basis of more elementary access functions to the type *store*.

What should these more elementary access functions be? It seems best not to make an *ad hoc* decision at this point, but to consider the operations *input* and *output* and to select the elementary functions so as to facilitate the design of the code for these functions. This means that we proceed according to the top-down programming strategy, since we intend to define the elementary functions on *store* in the process of designing the code that uses them.

We thus turn to the design of the code for the functions *input* and *output*. Here we face the problem that neither function can be considered on its own, since the target interface only expresses properties of their composition.

Our first goal therefore is to design two separate specifications of *input* and *output*. Together, the two specifications must of course imply the target specification I_{INOUT}. To specify *input* on its own, we have to specify what result values *input* should produce on any argument. These values, however, are of type *store*. Similarly, to specify *output*, we have to specify the desired results for the possible arguments, and the arguments also are values of type *store*. In order to specify *input* and *output* independently of each other, we must therefore be able to talk about values of type *store*, and this requires that we define a set of values for the type. At this point, the only purpose of this value set is to enable us to specify *input* and *output*. Hence the value set for *store* should be chosen so as to simplify these specifications.

A good guide for the decision about the *store* value set is to consider the information that needs to be recorded in the values of the type. In the target interface I_{INOUT} (as well as from our informal description of the problem), it is easy to see that the result $output(input(l))$ for $l \in listitem$ depends only on $\operatorname{ran} l$, that is, the set of *item* values occurring in l. Hence it is sufficient that the *store* values are capable of representing finite sets of *item* values, and the most natural definition of the *store* value set is $store = \mathbf{F}(item)$, *i. e.*, the set of finite sets of *item* values. We can then define $input(l)$ to be $\operatorname{ran} l$, which is the set of *item* values occurring in l, and define the value of *output* on such a set to be the sorted list of values in the set. These specifications are recorded in the interfaces I_{INPUT} and I_{OUTPUT} in Figures 1-11 and 1-12. Note that *input* can

I_{INPUT} =

interface

signature

item, *listitem*, *store*: **sort**

input: *listitem* $\rightarrow$ *store*

properties

listitem = $(item)^*$

store = $\mathbf{F}(item)$

for all $l \in$ *listitem*: $input(l) = \mathrm{ran}\, l$

Figure 1-11: The interface I_{INPUT}

be specified without reference to the ordering on *item*, so that *leitem* and *bool* need not be mentioned in I_{INPUT}.

The interfaces I_{INPUT} and I_{OUTPUT} shall become part of the design graph and make further consideration of I_{INOUT} unnecessary. To this end, it must be proved that I_{INPUT} and I_{OUTPUT} together imply I_{INOUT}. Since correctness proofs are naturally associated with modules in a design graph, it seems best to introduce a module M_{INOUT} that imports I_{INPUT} and I_{OUTPUT} and exports I_{INOUT} (Figure 1-13). Since all the program entities of the export interface I_{INOUT} occur among those of the import interfaces I_{INPUT} and I_{OUTPUT}, the module M_{INOUT} need not define any new program entities. Thus, M_{INOUT} is the empty module (Figure 1-14).

Although this module is empty, its correctness must be proved: We have to show that whenever the module is supplied with program entities that satisfy the import interfaces, then these entities (together with the entities contributed by the module, of which there are none in this case) satisfy the export interface. It is easy to see that this is indeed the case: if we combine the axiom of I_{INPUT} that $input(l) = \mathrm{ran}\, l$ with the axioms of I_{OUTPUT}, the axioms of I_{INOUT} follow immediately.

I_{OUTPUT} =

interface

signature

bool, *item*, *listitem*, *store*: **sort**

leitem: *item item* $\rightarrow$ *bool*

output: *store* $\rightarrow$ *listitem*

properties

bool $= \{\mathbf{T}, \mathbf{F}\}$

dom *leitem* $= \prod\langle item, item\rangle$

$\{(x, y) \mid leitem(x, y) = \mathbf{T}\}$ is a total ordering on *item*

listitem $= (item)^*$

store $= \mathbf{F}(item)$

for all $s \in store$:

$\text{ran}(output(s)) = s$

$\text{Ascending}(output(s))$

Figure 1-12: The interface I_{OUTPUT}

We now turn to the design of the *input* operation specified by I_{INPUT}. This operation has to compute the set of *item* values occurring in its argument list, which is a value of type *listitem*. The operations that are available to inspect *listitem* values are *isnil*, *hd*, and *tl*. This suggests the following program structure for *input*:

$$input(l) = \text{if } isnil(l) \text{ then } \emptyset \text{ else } \{hd\, l\} \cup input(tl\, l)$$

(recall that we defined *store* $= \mathbf{F}(item)$). Thus, we postulate two access functions *empty* and *insert* for the type *store* that are defined by

$$empty() = \emptyset$$

$$insert(x, s) = \{x\} \cup s.$$

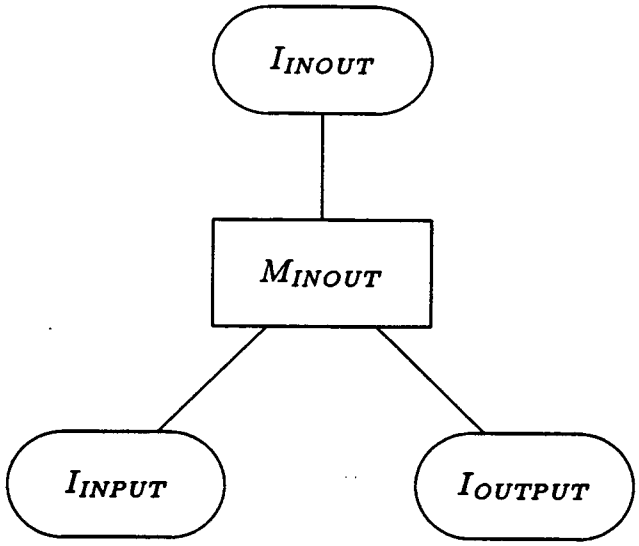


Figure 1-13: The module M_{INOUT} with its import and export interfaces

M_{INOUT} =
module
environment signature
defined symbols
requirements
result

Figure 1-14: The empty module M_{INOUT}

These functions are to be imported by the module defining the *input* operation, and we record their specification in the interface I_{INSERT} of Figure 1-15.

The *input* operation can now be coded as suggested above. This code constitutes the module M_{INPUT} in Figure 1-16. The code given in the **result** section defines *input* recursively and corresponds to code in a programming notation. This code requires that *bool* = {**T**, **F**}, because the **if** construct is used, which requires its first argument to be of the standard type of truth values.

It is worth while to note at this point how the choice of access functions to the type *store* was influenced by the design of the *input* operation. If, for example,

I_{INSERT} =

interface

signature

item, *store*: **sort**

empty: $\rightarrow$ *store*

insert: *item store* $\rightarrow$ *store*

properties

$store = \mathbf{F}(item)$

$empty() = \emptyset$

for all $x \in item$, $s \in store$: $insert(x, s) = \{x\} \cup s$

Figure 1-15: The interface I_{INSERT}

the argument of *input* was not a linear list but a binary tree, the function *insert* would be less useful than an access function

$$union(s, t\text{: } store)\text{: } store = s \cup t,$$

which would allow one to calculate the union of the two *store* values corresponding to the subtrees of a tree.

To conclude the design of the *input* operation, it remains to prove that M_{INPUT} is correct with respect to the import interfaces $I_{LISTITEM}$ and I_{INSERT} and the export interface I_{INPUT}. For this, assume that program entities are given as described by $I_{LISTITEM}$ and I_{INSERT}, and that *input* is defined by the recursive code given in M_{INPUT}. The only nontrivial axiom of I_{INPUT} is then that $input(l) = \text{ran}\, l$ for all $l \in listitem$. It is easy to prove by induction on the length of lists $l \in listitem$ that *input* is total. This allows us to treat the definition of *input* in M_{INPUT} as an equation between values of type *store* (for a detailed treatment of this interpretation of recursive function definitions, see [Cartwright 84]). Using this equation, one proves that $input(l) = \text{ran}\, l$ by induction on the length of l:

M_{INPUT} =

module

environment signature

bool, *item*, *listitem*, *store*: **sort**

isnil: *listitem* → *bool*

hd: *listitem* → *item*

tl: *listitem* → *listitem*

empty: → *store*

insert: *item store* → *store*

defined symbols

input: *listitem* → *store*

requirements

bool = {**T**, **F**}

result

input(*l*) = **if** *isnil*(*l*) **then** *empty*()
else *insert*(*hd l*, *input*(*tl l*))

Figure 1-16: The module M_{INPUT}

- If $|l| = 0$, then $l = \langle\rangle$, hence $isnil(l) = \mathbf{T}$, and $input(l) = empty() = \emptyset = \text{ran}\, l$.

- If $|l| > 0$, then $l \neq \langle\rangle$, hence $isnil(l) = \mathbf{F}$, and

$$\begin{aligned} input(l) &= insert(hd\, l,\ input(tl\, l)) \\ &= insert(l_1,\ input(\langle l_2, \ldots, l_n\rangle)) \\ &= insert(l_1,\ \text{ran}\langle l_2, \ldots, l_n\rangle) \qquad \text{(Inductive Hypothesis)} \\ &= \{l_1\} \cup \{l_2, \ldots, l_n\} \\ &= \{l_1, \ldots, l_n\} \\ &= \text{ran}\, l. \end{aligned}$$

This concludes the correctness proof of M_{INPUT}.

Our next task is to design the *output* operation specified by I_{OUTPUT}. Given a *store* value as input, *i. e.*, a finite set of *item* values, the operation must produce an ordered list containing exactly these values, which is a value of type *listitem*. The operations that may be used to construct *listitem* values are *nil* and *cons*. In the case of a nonempty *store* value s, the output list must be of the form $cons(x,l)$, and since this list must be ordered, the *item* value x must be the minimal element of s, while l must be the ordered list of the other elements of s. This suggests the following program structure:

$$output(s) = \text{ if } s = \emptyset \text{ then } nil() \\ \text{else } cons(\min s,\ output(s \setminus \{\min s\})),$$

where $\min s$ is the $<$-minimal element of s, *i. e.*,

$$\min s \in s \quad \text{and} \quad y \not< \min s \text{ for all } y \in s.$$

We therefore postulate three access operations *isempty*, *min*, and *removemin* to the type *store* that are defined by

$$isempty(s) = \begin{cases} \mathbf{T}, & \text{if } s = \emptyset \\ \mathbf{F}, & \text{if } s \neq \emptyset \end{cases}$$

$$\text{for } s \neq \emptyset: \quad min(s) = \min s$$

$$removemin(s) = s \setminus \{\min s\}.$$

The operation *removemin* is sufficient to code *output* according to the scheme above; we do not need an operation to remove an arbitrary element from a set. It might be possible to implement the specialized operation *removemin* more easily and efficiently than a general "remove" operation. The operation *removemin* is specially designed according to the needs of *output*, which shows the advantages of determining access operations to a type during the design of the modules that use them.

The three new operations on *store* are to be imported by the module defining *output*, and they are specified in the interface I_{MIN} of Figure 1-17.

I_{MIN} =

interface

signature

bool, *item*, *store*: **sort**

leitem: *item item* → *bool*

isempty: *store* → *bool*

min: *store* → *item*

removemin: *store* → *store*

properties

$bool = \{\mathbf{T}, \mathbf{F}\}$

dom *leitem* $= \prod\langle item, item\rangle$

$\{(x,y) \mid leitem(x,y) = \mathbf{T}\}$ is a total ordering on *item*

$store = \mathbf{F}(item)$

for all $s \in store$: $isempty(s) = \begin{cases} \mathbf{T}, & \text{if } s = \emptyset \\ \mathbf{F}, & \text{if } s \neq \emptyset \end{cases}$

for all $s \in store \setminus \{\emptyset\}$:

$min(s) = \min s$

$removemin(s) = s \setminus \{\min s\}$.

Figure 1-17: The interface I_{MIN}

The *output* operation can now be coded as suggested above. This code forms the module M_{OUTPUT} in Figure 1-18.

It remains to prove that M_{OUTPUT} is correct with respect to the import interfaces $I_{LISTITEM}$ and I_{MIN} and the export interface I_{OUTPUT}. Assume that program entities are given as described by $I_{LISTITEM}$ and I_{MIN}, and that *output* is defined by the recursive code of M_{OUTPUT}. The only nontrivial axiom of I_{OUTPUT} is that

for all $s \in store$: $\text{ran}(output(s)) = s$

$\text{Ascending}(output(s))$.

M_{OUTPUT} =

module

environment signature

bool, *item*, *listitem*, *store*: **sort**

leitem: *item item* → *bool*

nil: → *listitem*

cons: *item listitem* → *listitem*

isempty: *store* → *bool*

min: *store* → *item*

removemin: *store* → *store*

defined symbols

output: *store* → *listitem*

requirements

bool = {**T**, **F**}

result

output(*s*) = **if** *isempty*(*s*) **then** *nil*()

else *cons*(*min*(*s*), *output*(*removemin*(*s*)))

Figure 1-18: The module M_{OUTPUT}

A simple induction on the cardinality of sets $s \in store = \mathbf{F}(item)$ shows that *output* is total. Hence the definition of *output* can be regarded as an equation between *listitem* values. The axiom above is now proved by induction on the cardinality of sets $s \in store = \mathbf{F}(item)$.

The inductive hypothesis is that for sets $t \in store$ with $\text{card}(t) < \text{card}(s)$, we have $\text{ran}(output(t)) = t$ and $\text{Ascending}(output(t))$.

If $\text{card}(s) = 0$, then $s = \emptyset$, hence $isempty(s) = \mathbf{T}$ and $output(s) = nil() = \langle\rangle$. Thus $\text{ran}(output(s)) = \text{ran}\langle\rangle = \emptyset = s$, and $\text{Ascending}(output(s)) \iff \text{Ascending}(\langle\rangle)$, which is vacuously true.

If $\mathrm{card}(s) > 0$, then $s \neq \emptyset$, hence $\mathit{isempty}(s) = \mathbf{F}$, and

$$\begin{aligned}\mathit{output}(s) &= \mathit{cons}(\mathit{min}(s),\ \mathit{output}(\mathit{removemin}(s))) \\ &= \mathit{cons}(\min s,\ \mathit{output}(s \setminus \{\min s\})).\end{aligned}$$

Let $s' := s \setminus \{\min s\}$. Since $\min s \in s$, we have $\mathrm{card}(s') < \mathrm{card}(s)$, so by the inductive hypothesis,

$$\begin{aligned}&\mathrm{ran}(\mathit{output}(s')) = s', \quad \text{and} \\ &\mathrm{Ascending}(\mathit{output}(s')).\end{aligned}$$

Thus,

$$\begin{aligned}\mathrm{ran}(\mathit{output}(s)) &= \mathrm{ran}\ \mathit{cons}(\min s, \mathit{output}(s')) \\ &= \{\min s\} \cup \mathrm{ran}\ \mathit{output}(s') \\ &= \{\min s\} \cup s' \\ &= s.\end{aligned}$$

To prove $\mathrm{Ascending}(\mathit{output}(s))$, let $l' = \langle l'_1, \ldots, l'_n \rangle := \mathit{output}(s')$, so that $\mathrm{ran}\, l' = s'$, $\mathrm{Ascending}(l')$, and $l := \mathit{output}(s) = \mathit{cons}(\min s, l') = \langle \min s, l'_1, \ldots, l'_n \rangle$. If $1 \leq i < j \leq |l| = n + 1$, then

- if $i = 1$, then $l_i = l_1 = \min s < l_j$, because $l_j = l'_{j-1} \in \mathrm{ran}\, l' = s' = s \setminus \{\min s\}$
- if $i > 1$, then $l_i = l'_{i-1} < l'_{j-1} = l_j$, because $\mathrm{Ascending}(l')$.

This concludes the correctness proof of M_{OUTPUT}.

After the modules I_{INPUT} and I_{OUTPUT} and the interfaces I_{INSERT} and I_{MIN} have been added to the design graph, the interfaces I_{INSERT} and I_{MIN} are the only ones that still need to be guaranteed (the current design graph can be obtained from Figure 1-20 below by omitting M_{STORE} and the edges leading to it). One could, of course, attempt to express the five access functions to the *store* data type that are specified in these interfaces in terms of yet more elementary access functions, as it was done with *input* and *output*. But we shall not do so and regard the functions *empty*, *insert*, *isempty*, *min* and *removemin* as the elementary access functions of the *store* data type.

Our task is now to design an implementation of the *store* data type. Since the two remaining interfaces I_{INSERT} and I_{MIN} both mention *store*, they must both be export interfaces of the module that defines *store*.

For the design of the *store* implementation, we use the "standard" criterion for the correctness of data representations due to Hoare [Hoare 72]. Given a "representation" data type with access functions that corresponds syntactically to an "abstract" type with its access functions, the correctness of the representation is established as follows:

- A "representation invariant" I is defined, which is a predicate on the values of the representation type.
- An "abstraction function" $\mathcal{A}$ is defined that maps the values of the representation type that satisfy I to values of the abstract type. A value x of the representation type is said to "represent" a value y of the abstract type in case x satisfies I and $\mathcal{A}(x) = y$. Values of types other than the type to be represented (these types are not changed in the representation) are said to "represent" themselves.
- Every access function f is shown to be "compatible" with I and $\mathcal{A}$: Whenever $\langle x_1, \ldots, x_n \rangle$ is an argument tuple for the representation of f and $\langle y_1, \ldots, y_n \rangle$ is an argument tuple for the abstract version of f such that each x_i represents y_i, then the applications of the representation of f to $\langle x_1, \ldots, x_n \rangle$ and of the abstract version of f to $\langle y_1, \ldots, y_n \rangle$ are either both defined or both undefined, and if they are defined, the result value in the representation represents the result value of the abstract application (in particular, the result in the representation must satisfy the representation invariant).

The implementation of the *store* data type that will now be given uses sequences of *item* values to represent *store* values. This choice is motivated by the fact that such sequences are already available as the type *listitem*, so that we can define

$$store := listitem \quad (= item^*).$$

More efficient implementations could be designed based on trees, but this would require the introduction of another data type and thus require another interface of similar complexity to $I_{LISTITEM}$.

The implementation will avoid duplicate elements in the lists and will keep the lists ordered so that the *min* and *removemin* functions become simple. This decision is expressed by the representation invariant

$$I(l) \;:\Longleftrightarrow\; \text{Ascending}(l).$$

The *store* value represented by a list l is just the set of *item* values occurring in l. Hence the abstraction function of the implementation is given by

$$\mathcal{A}(l) := \text{ran}\, l.$$

The main problem in the implementation of the access functions is that the *insert* function must preserve the representation invariant; *i.e.*, the resulting list must be ordered and duplicate-free, provided the argument list is.

The implementation of the *store* data type with its five access functions is given in Figure 1-19. To prove this module correct with respect to the export interfaces I_{INSERT} and I_{MIN} and the import interfaces I_{ITEM} and $I_{LISTITEM}$, one has to show that for every family of import program entities satisfying I_{ITEM} and $I_{LISTITEM}$, the result of M_{STORE} is a representation of a data type satisfying I_{INSERT} and I_{MIN}. The latter, "abstract", data type is almost fully determined by I_{INSERT} and I_{MIN} (assuming fixed import program entities satisfying I_{ITEM} and $I_{LISTITEM}$), except for the behaviour of min and *removemin* when the argument is the empty set, which is not relevant for the code using these functions. Since the implementations of these operations do not yield results on the representation of the empty set (the empty set is represented by the empty list, min is implemented as *hd* and *removemin* is implemented as *tl*), the min and *removemin* operations of the abstract type must also be undefined on the empty set in order for the proof to go through.

The representation invariant I and the abstraction function $\mathcal{A}$ for the correctness proof were given above; to complete the proof, it has to be shown that the five access functions are compatible with I and $\mathcal{A}$. This proof is given in Exam-

M_{STORE} =

module

environment signature

bool, *item*, *listitem*: **sort**

leitem: *item item* → *bool*

nil: → *listitem*

cons: *item listitem* → *listitem*

isnil: *listitem* → *bool*

hd: *listitem* → *item*

tl: *listitem* → *listitem*

defined symbols

store: **sort**

empty: → *store*

insert: *item store* → *store*

isempty: *store* → *bool*

min: *store* → *item*

removemin: *store* → *store*

requirements

$bool = \{\mathbf{T}, \mathbf{F}\}$

result

store = *listitem*

empty = *nil*

insert is defined by the recursive code

$insert(x, l) =$ **if** $isnil(l)$ **then** $cons(x, nil())$
else if $leitem(x, hd\, l)$
then if $leitem(hd\, l, x)$ **then** l
else $cons(x, l)$
else $cons(hd\, l,\ insert(x,\ tl\, l))$

isempty = *isnil*

min = *hd*

removemin = *tl*

Figure 1-19: The module M_{STORE}

ple 4.5.3 of this thesis, based on a formal definition of the "standard" correctness criterion sketched above.

Since M_{STORE} imports only the interfaces I_{ITEM} and $I_{LISTITEM}$, which we assume to be guaranteed by the programming environment, no further unsatisfied interfaces remain, and the design of the *dictionary* program is complete. The complete design graph is shown in Figure 1-20.

Based on a programming environment as described in I_{ITEM} and $I_{LISTITEM}$, the five modules we have designed constitute a complete program that defines a function *dictionary*: *listitem* $\rightarrow$ *listitem*. Naturally, we would like to be able to conclude from the correctness proofs of the individual modules that the *dictionary* function is correct; *i.e.*, causes I_{DICT} to be satisfied.

Here, however, we face the correctness problem of data abstraction. The modules using the *store* data type, *e.g.*, M_{INPUT} and M_{OUTPUT}, have been proved correct under the assumption that they were supplied with program entities satisfying their import interfaces, *e.g.*, I_{INSERT} and I_{MIN}. However, the program entities defined by M_{STORE} do not satisfy these interfaces, so that the correctness proofs of M_{INPUT} and M_{OUTPUT} do not apply in the final system. For example, M_{INPUT} was proved correct under the assumptions (from I_{INSERT}) that $store = \mathbf{F}(item)$, $empty() = \emptyset$, and *insert* a function from $\prod\langle item, \mathbf{F}(item)\rangle$ to $\mathbf{F}(item)$, while in the actual program, we have $store = item^*$, $empty() = \langle\rangle$, and *insert* a function from $\prod\langle item, item^*\rangle$ to $item^*$.

Thus, there is no direct way to infer the correctness of the *dictionary* operation defined by the final program from the correctness of the individual modules, which we proved in the course of this section. Of course, one could consider the final program composed of the five modules and prove the correctness of the *dictionary* operation on the basis of this program. But this would mean that the correctness proofs of the individual modules became redundant, and that the whole program would have to be considered anew. Also, one would then have to consider the program code on the basis of the representation of the *store* data type given in M_{STORE}. This violates the principle of data abstraction discussed in Section 1.1, namely that programs using an encapsulated data type should be

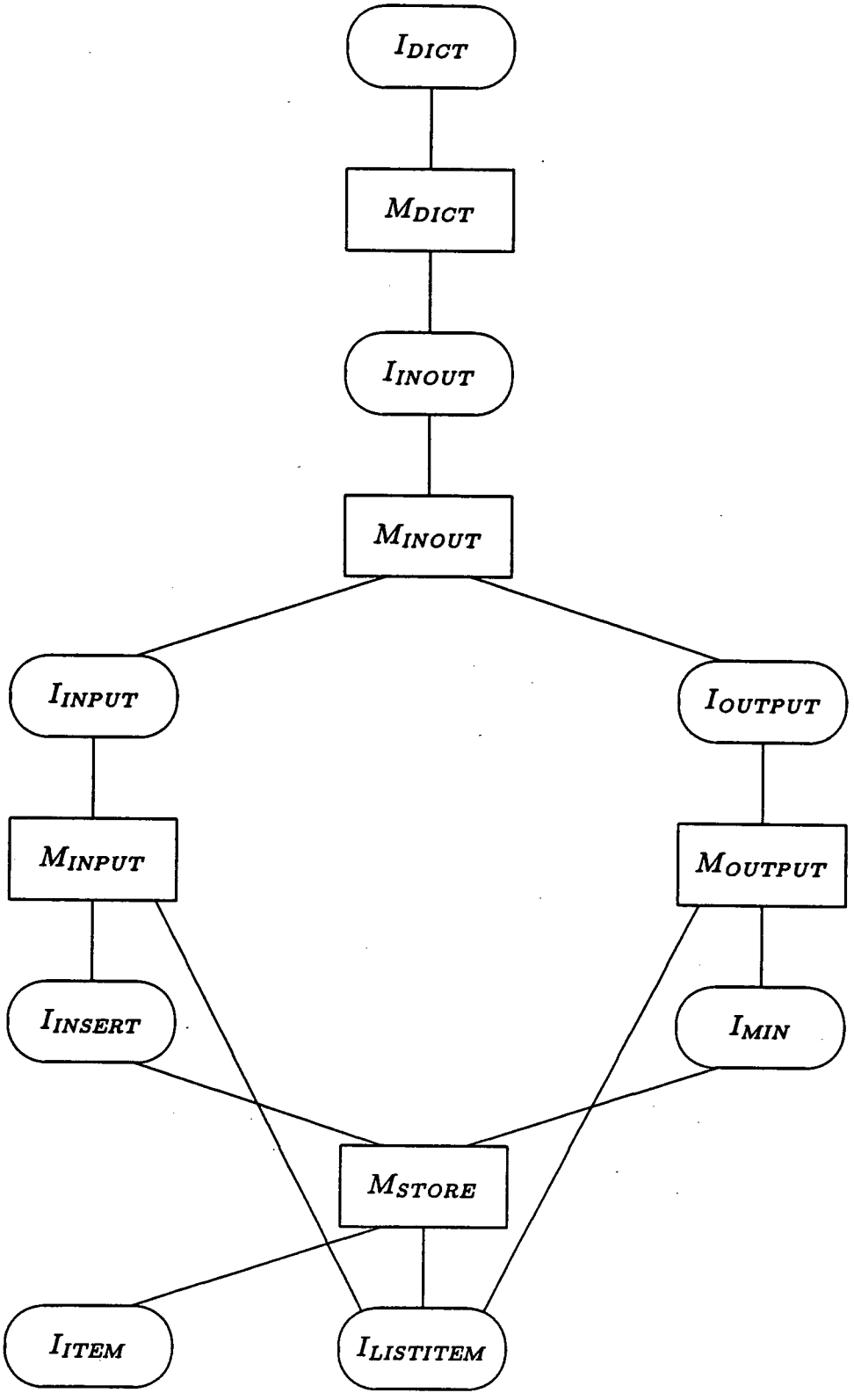


Figure 1-20: The design graph of the *dictionary* program development

proved correct on the basis of the abstract specification of the type rather than the properties of its representations.

The theory of this thesis will present a "stability" criterion for modules that makes it possible to infer the correctness of the *dictionary* operation from the correctness of the individual modules (however, the stability of the modules of this development will not actually be proved). The design graph in Figure 1-20 and the individual module correctness proofs that we have performed so far constitute a "structured correctness argument" as explained in Chapter 3 below (Figure 3-5). The data abstraction technique has helped us by allowing us to use the abstract specification of the *store* data type ($store = \mathbf{F}(item)$) and the access functions specified in I_{INPUT}, I_{OUTPUT}, I_{INSERT} and I_{MIN} in the correctness proofs of M_{INOUT}, M_{INPUT} and M_{OUTPUT}.

Before concluding this section, it seems worthwhile to review an interesting aspect of the design process, namely the way the elementary access functions to the *store* data type were determined. Rather than defining a set of elementary access functions at the time the type *store* was first introduced, *i. e.*, when M_{DICT} was designed, we began with the problem-oriented access functions *input* and *output* and determined the elementary access functions during the design of the code for these functions, *i. e.*, during the design of M_{INPUT} and M_{OUTPUT}. In this way, we arrived at a set of elementary access functions tailored to the needs of our program (cf. pages 47 and 50).

The technique by which we arrived at the set of elementary access functions could be called "access function refinement". When the data type *store* was first introduced in the interface I_{INOUT}, its access functions were *input* and *output*, whose specification was derived immediately from the original problem statement. Later, programs were written for these access functions in terms of simpler access functions that were introduced and specified according to the requirements of these programs. During the process, a new function was introduced only if required by some code under design, and its specification was used to prove the correctness of the code using it before the implementation of the

function was considered. This strict top-down design procedure ensured that all operations were designed according to the needs of the code using them.

Our example thus exhibits a design strategy in which all functions are introduced and specified in connection with the design of the code that uses them, rather than as a bunch of elementary access functions to some newly introduced data type that may or may not be useful for the programs to be designed.

> **Strategy.** (Access function refinement)
> When introducing a new data type, give it access functions that are as closely oriented to your problem as possible. While you consider the functions too complex to be elementary access functions to the type, encode them in terms of simpler access functions that you invent for this purpose. Finally, implement the data type together with the remaining "elementary" access functions as a single module.

The access function refinement strategy influences the choice of a specification technique for encapsulated data types. In order that a function can be refined, *i. e.*, code for it can be written in terms of other functions, it is advantageous that the function be fully specified on its own, rather than implicitly specified via its interaction with other functions. To specify a function on its own, however, one has to consider the relation between its input and output values, and this requires that the ranges of possible input and output values be known. This means that value sets for all the data types accessed by the function must be known, whether encapsulated or not.

In our example, the original specification I_{INOUT} of the functions *input* and *output* only specified them implicitly via their joint effect, and it was not possible at that stage to consider the the functions separately. To make the separate specifications in I_{INPUT} and I_{OUTPUT} possible, a set of values for the type *store* had to be defined (we chose $store = \mathbf{F}(item)$).

Specifications of encapsulated types that define a set of values for the type are called "abstract model" specifications, while specifications that do not do so are said to be "implicit" data type specifications [LZ 75, p. 12].

The strategy of access function refinement seems to work best on the basis of abstract model specifications, because with an explicit value set for each type, the access functions can be specified and refined independently of each other.

Choosing a value set for a newly introduced type is an important design decision, even if the type is to be implemented later so that the chosen value set only occurs in specifications.

The danger here is not so much to choose a value set that is "too small" and thus insufficient to represent the required information, because then the access functions could not be expressed in terms of this value set. In our example, it would not have been possible to specify *input* and *output* separately so that together they would satisfy I_{INOUT}; which means that the problem would have been noticed immediately in the correctness proof of M_{INOUT}.

Rather more serious is the danger of introducing a value set that is overly complex and that preserves more information than necessary. As will be discussed on page 258 f. below, this need not prejudice the choice of an implementation if "behavioural" correctness concepts and the appropriate proof techniques are used. However, an overly complex value set might misguide the refinement of the access functions of a type.

If, for example, in the *dictionary* problem we had chosen I_{INOUT} as follows:

$$\begin{aligned} &\mathit{store} = \mathit{item}^* \\ &\mathit{input} = \mathrm{Id}(\mathit{item}^*) \\ &\mathit{output} \text{ specified the same way as } \mathit{dictionary}, \end{aligned} \qquad (*)$$

we could still express *output* in terms of functions *isempty*, *min* and *removemin* specified analogously to the ones in I_{MIN}, but we might then have chosen to refine *min* and *removemin* further; *e. g.*, using a function *isempty* and functions *first* and *rest* specified like *hd* and *tl* of $I_{LISTITEM}$. These functions, which cannot be introduced in the actual specification $\mathit{store} = \mathbf{F}(\mathit{item})$, preclude the implementation of *store* given in M_{STORE}, because they require that all the information about the input list be represented in the *store* (since it can be reconstructed by *isempty*, *first* and *rest*). Thus, a less storage-efficient implementation of *store* would be necessary.

It is possible to verify that the value set of a type does not represent irrelevant information by applying the following "bias test": an abstract model specification for a data type is "unbiased" and hence free of redundancy, if the access functions allow one to generate all values of the encapsulated type, and if they allow one to distinguish the values from each other. The first condition is generally called the "reachability" or "no junk" requirement; the second condition is what Jones calls the "bias test" in [Jones 80, Ch. 15].

It is important to apply this bias test as soon as the set of values for the type is introduced. In our example, the alternative specification (*) would have shown a bias in the test, because the *output* function could not discriminate lists with the same range (*i. e.*, set of *item* values occurring in it). After the indicated refinement steps that introduced the functions *isempty*, *first* and *rest*, the *store* data type would not have shown a bias in the test, since these functions allow to discriminate its values.

In general, abstract model specifications and the access function refinement strategy should work best for data types for which one can determine an "unbiased" set of values. On the other hand, implicit data type specifications should work best when one can determine the elementary access functions to a type at an early stage. Thus, the choice between an abstract model or an implicit specification of a data type is likely to depend on the situation. It would seem advantageous, therefore, to use a specification notation that does not prejudice this choice by enforcing either abstract model or implicit specifications. The "mathematical" specification notation used in this thesis allows one to specify arbitrary properties of data types and functions, and thus to write both abstract model specifications (*e. g.*, I_{INSERT} combined with I_{MIN}) and implicit specifications (I_{INOUT}) of encapsulated data types with equal ease.

Chapter 2

Signatures, Algebras, and Institutions

THIS CHAPTER introduces the basic notions of the theory. "Algebraic signatures" and "algebras", familiar from the literature on abstract data types, serve as models of the program entities in functional programs, such as the ones presented in Section 1.4. However, several key notions and theorems will be developed on a more abstract level, based on the notion of an "institution".

2.1 Mathematical Concepts and Notations

This section presents the basic mathematical concepts and notations from set theory and category theory that will be used in this thesis. Its purpose is not to give a tutorial introduction, but to serve as a concise reference. That is, the reader is supposed to be familiar with the notions of "set", "category" and "functor", and is advised to just skim or skip this section. When a concept is first used in the remainder of the thesis, a short explanation is usually given; the present section may then be consulted for the precise definition.

The theory of this thesis is based on set theory in its "standard" ZFC axiomatization ("Zermelo-Fraenkel set theory with the Axiom of Choice"). Presentations of these axioms can be found in [Kunen 80, p. xv f], [Levy 79, Section I.5], [Barwise 77, Chapter B.1], and in many other places. The important property of ZFC to keep in mind is that there is just one kind of objects, called "sets" (there are other axiom systems that use "classes" or both "sets" and "classes" as their

basic notions). On page 70 below, an axiom will be added to ZFC that allows it to deal with "large sets" or "classes" as well.

The symbol ":=" is used as a concise way of stating definitions—the left hand side contains one or more "new" symbols (usually letters) that become defined such that the left hand side equals the right hand side. For example, in the phrase "If $x \in M$, then $y := f(x)$ satsfies ...", the letter "y" becomes defined to equal $f(x)$. Similarly, the symbol " $:\Longleftrightarrow$ " is used to define predicates.

Here are some of the notations that will be used.

$\mathbf{N} = \{0, 1, 2, \ldots\}$	set of *natural numbers*
$\mathbf{P}(A) = \{ X \mid X \subseteq A \}$	*power set* of a set A
$\mathbf{F}(A) = \{ X \mid X \subseteq A$ and X finite$\}$	set of finite subsets of A
$(x, y) = \{\{x\}, \{x, y\}\}$	*ordered pair* (the precise definition is not relevant; what matters is that $(x, y) = (x', y')$ iff $x = x'$ and $y = y'$)
$A \times B = \{ (x, y) \mid x \in A$ and $x \in B \}$	*cartesian product* of sets A and B
$A \setminus B = \{ x \in A \mid x \notin B \}$	*set difference*
if $A \cap B = \emptyset$:	
$A + B = A \cup B$	*sum* of two disjoint sets (this notation is used when A and B are disjoint in order to help the reader)
if A is finite:	
$\mathrm{card}(A) \in \mathbf{N}$	the *cardinality* of A, *i. e.*, the number of elements of A.

A *relation* is a set consisting entirely of ordered pairs. If R is a relation, then

$$x \, R \, y \;:\Longleftrightarrow\; (x, y) \in R.$$

The *domain* and *range* of a relation R are given by

$$\begin{aligned} \operatorname{dom} R &:= \{\, x \mid \exists y\colon (x,y) \in R \,\} \\ \operatorname{ran} R &:= \{\, y \mid \exists x\colon (x,y) \in R \,\}. \end{aligned}$$

The *converse* of a relation R is

$$R^{\cup} := \{\, (y,x) \mid (x,y) \in R \,\}.$$

The *composition* of two relations R and S is

$$R\,;S := \{\, (x,z) \mid \exists y\colon (x,y) \in R \text{ and } (y,z) \in S \,\}.$$

The *restriction* of a relation R to a set M is

$$R/M := \{\, (x,y) \in R \mid x \in M \,\}.$$

The *image* of a set M under a relation R is

$$R(\!|M|\!) := \{\, y \mid \exists x \in M\colon (x,y) \in R \,\} = \operatorname{ran}(R/M).$$

A *function* is a relation f that satisfies

$$(x,y) \in f \wedge (x,z) \in f \implies y = z;$$

that is, if for every x there is at most one y such that $(x,y) \in f$. For $x \in \operatorname{dom} f$, the unique y such that $(x,y) \in f$ is called the *value* of f on x, and is written "$f(x)$" or sometimes just "$f\,x$". All of the concepts described above for relations also apply to functions. In particular, the notation "$f\,;g$" denotes the composition of the two functions f and g in diagrammatic order, so that $(f\,;g)(x) = g(f(x))$. A function f is a *total function* or *mapping* from a set A to a set B (written"$f\colon A \to B$"), if

$$\operatorname{dom} f = A \quad \text{and} \quad \operatorname{ran} f \subseteq B,$$

it is a *partial function* from A to B (written "$f\colon A \nrightarrow B$"), if

$$\operatorname{dom} f \subseteq A \quad \text{and} \quad \operatorname{ran} f \subseteq B.$$

A function f is *injective*, if its converse $f^{\cup}$ is a function (that is, if $(x,z) \in f$ and $(y,z) \in f$ imply $x = y$). A (total or) partial function $f\colon A \nrightarrow B$ is *surjective*, if $\operatorname{ran} f = B$; it is *bijective*, if it is total, injective, and surjective (note that the property "total" depends on the source set A, "surjective" depends on the target set B, and "bijective" depends on both the source set A and the target set B).

If A and B are sets such that $A \subseteq B$, then the unique function $f\colon A \to B$ satisfying $f(x) = x$ for all $x \in A$ is called the *inclusion* function from A to B; the inclusion from a set A to itself is called the *identity* function on A and is written "$\mathrm{Id}(A)$".

A *family* is a function. Special terminology and notation are used for families: the domain of a family is called its *index set*, the elements of the domain are the *indices*. A family is called *empty*, *finite*, or *infinite* according to whether its index set is empty, finite, or infinite. The range of a family is called its set of *values* or *elements* (by abuse of language). A family x with index set I is written "$\langle x_i \rangle_{i \in I}$"; the notation "$x_i$" stands for $x(i)$, that is, the value of x for the index $i \in I$. If the range of a family is a subset of a set M, the family is called a *family of elements* of M.

Some mathematical notations have families of sets as arguments: if $M = \langle M_i \rangle_{i \in I}$ is a family of sets, then

$$\prod M = \prod_{i \in I} M_i := \{ \langle x_i \rangle_{i \in I} \mid \forall i \in I\colon x_i \in M_i \}$$ *product* of a family of sets

$$\bigcup M = \bigcup_{i \in I} M_i := \{ x \mid \exists i \in I\colon x \in M_i \}$$ *union* of a family of sets

if the M_i are pairwise disjoint:

$$\sum M = \sum_{i \in I} M_i := \bigcup_{i \in I} M_i$$ *sum* of a family of pairwise disjoint sets

if $I \neq \emptyset$:

$$\bigcap M = \bigcap_{i \in I} M_i := \{ x \mid \forall i \in I\colon x \in M_i \}$$ *intersection* of a nonempty family of sets.

A *sequence* is a family whose index set is an initial segment (*i. e.*, a $\leq$-downward closed subset, see below) of the set $\mathbf{N}_1 := \{1, 2, 3, \ldots\}$. The index set of a

finite sequence x is of the form $\{1,\ldots,n\}$ for $n \in N$; the number n is called the *length* of x, written "$|x|$". A finite sequence x of length n is also called an *n-tuple* and is written "$\langle x_1,\ldots,x_n\rangle$" or sometimes just "$x_1,\ldots,x_n$". In particular, a *pair* is a 2-tuple, a *triple* is a 3-tuple, etc.

Since it is the traditional notation, the application of a function f to an n-tuple $\langle x_1,\ldots,x_n\rangle$, which should be written "$f(\langle x_1,\ldots,x_n\rangle)$" or "$f\langle x_1,\ldots,x_n\rangle$", will often be written "$f(x_1,\ldots,x_n)$". In particular, the application of f to the 0-tuple $\langle\rangle$ will be written "$f()$".

If x and y are finite sequences, their *concatenation*, written "$x \circ y$", is defined by

$$x \circ y := x \cup \{\, (i + |x|,\, y_i) \mid i \in \operatorname{dom} y \,\};$$

this is a finite sequence whose length is the sum of the lenghts of x and y. If M is any set, the finite sequences of elements of M are called *words* over M; the notation "M^*" denotes the set of all words over M, *i. e.*,

$$M^* = \{\, \langle x_1,\ldots,x_n\rangle \mid n \in \mathbf{N} \text{ and } x_i \in M \text{ for all } i \in \{1,\ldots,n\} \,\}.$$

Let $<$ be a relation. A set M is *$<$-downward closed*, if $x \in M$ and $y < x$ imply $y \in M$. A *$<$-minimal* element of a set M is an $x \in M$ such that no $y \in M$ satisfies $y < x$. The relation $<$ is *well-founded*, if every nonempty set has a $<$-minimal element.

A relation $<$ is *transitive*, if $x < y$ and $y < z$ imply $x < z$. The *transitive closure* of a relation $<$ is the intersection of all transitive relations that include $<$, that is,

$$\ll := \{\, (x,y) \mid \text{whenever } R \text{ is a transitive rel. and } < \subseteq R, \text{ then } (x,y) \in R \,\};$$

this is well-defined, because $(\operatorname{dom}< \cup \operatorname{ran}<) \times (\operatorname{dom}< \cup \operatorname{ran}<)$ is a transitive relation including $<$. It is easily seen that $\ll$ is a transitive relation including $<$.

A *preordering* on a set M is a relation $\sqsubseteq \subseteq M \times M$ that is transitive and satisfies $x \sqsubseteq x$ for all $x \in M$. The pair $\langle M, \sqsubseteq\rangle$ is called a *preorder*, if $\sqsubseteq$ is a preordering on M. A preordering $\sqsubseteq$ on a set M that is *symmetric*, *i. e.*, for which $x \sqsubseteq y$ implies $y \sqsubseteq x$, is called an *equivalence relation* on M. A preordering $\sqsubseteq$

on a set M for which $x \sqsubseteq y$ and $y \sqsubseteq x$ imply $x = y$ is called a *partial ordering* on M; if $\sqsubseteq$ is a partial ordering on M, then $\langle M, \sqsubseteq \rangle$ is called a *partial order*. If $\sqsubseteq$ is a partial ordering on M such that for all $x, y \in M$, we have $x \sqsubseteq y$ or $y \sqsubseteq x$, then $\sqsubseteq$ is a *total ordering* on M and $\langle M, \sqsubseteq \rangle$ is a *total order*.

Let $\langle M, \sqsubseteq \rangle$ be a preorder, and let N be a subset of M. A *minimal* (*maximal*) element of N is an $x \in N$ such that for all $y \in N$ such that $y \neq x$ we have $y \not\sqsubseteq x$ ($x \not\sqsubseteq y$) (note that this is slightly different from the definition of "minimal" given above for $<$-type relations. The notation will always make it clear which definition is meant). A *lower bound* (*upper bound*) of N is an $x \in M$ such that $x \sqsubseteq y$ ($y \sqsubseteq x$) for all $y \in N$. A *least* (*greatest*) element of N is an $x \in N$ that is a lower bound (upper bound) of N. These notions also apply to families of elements of M instead of subsets of M—for this, a family is identified with its range.

In a partial order $\langle M, \sqsubseteq \rangle$, the *least upper bound* (*greatest lower bound*) of a set $N \subseteq M$, written "$\bigsqcup N$" ("$\bigsqcap N$"), is the least element among the upper bounds (greatest element among the lower bounds) of N; if such an element exists, it is unique. A partial order in which every pair of elements has a least upper bound and a greatest lower bound is called a *lattice*; one also writes "$x \sqcup y$" for $\bigsqcup\{x, y\}$ and "$x \sqcap y$" for $\bigsqcap\{x, y\}$ and calls $\sqcup$ and $\sqcap$ the *join* and *meet* operation of the lattice.

A *chain* in a partial order $\langle M, \sqsubseteq \rangle$ is a set $N \subseteq M$ (or a family of elements of M, which as before is identified with its range) that is totally ordered by $\sqsubseteq$, *i. e.*, is such that $x \sqsubseteq y \vee y \sqsubseteq x$ for all $x \in N$. We will often employ "Zorn's Lemma", according to which

> a partial order in which every chain has an upper bound
> has a maximal element;

it is equivalent to the axiom of choice (see, *e. g.*, [Levy 79, p. 161], [Barwise 77, p. 355]).

Besides the concepts from set theory just presented, this thesis will use some basic concepts from category theory. They can all be found in the first 35 pages

of Mac Lane's book [Mac Lane 71]. However, notation and terminology will be changed slightly in order to avoid confusion in this thesis: a morphism in a category will have a "source" and a "target" instead of a "domain" and a "codomain", and the composition of two morphisms $f\colon A \to B$ and $g\colon B \to C$ is written "$f\,;g$" (*i. e.*, in diagrammatic order) rather than "$g \circ f$" or "gf".

A *category* consists of a set A of *arrows* or *morphisms*, a set O of *objects*, two mappings $\mathrm{src}, \mathrm{trg}\colon A \to O$ mapping each arrow to its *source* and *target* object, respectively, a map $\mathrm{id}\colon O \to A$ mapping each object to its *identity* arrow, and a *composition* operation $;\colon \prod\langle A, A\rangle \nrightarrow A$ such that

$$\mathrm{dom}\,; = \{\,\langle f, g\rangle \mid \mathrm{trg}\, f = \mathrm{src}\, g\,\},$$

and for all $x \in O$ and $f \in A$:

$$\mathrm{src}(\mathrm{id}(X)) = \mathrm{trg}(\mathrm{id}(X)) = X$$
$$\mathrm{id}(\mathrm{src}\, f)\,;f = f\,;\mathrm{id}(\mathrm{trg}\, f) = f,$$

and for $\langle f, g\rangle$ and $\langle g, h\rangle$ in $\mathrm{dom}\,;$:

$$\mathrm{src}(f\,;g) = \mathrm{src}\, f$$
$$\mathrm{trg}(f\,;g) = \mathrm{trg}\, g$$
$$(f\,;g)\,;h = f\,;(g\,;h).$$

If C is a category, then $|C|$ is its set of objects, and for $x, y \in |C|$, the *hom-set* $C(X,Y)$ is the set of arrows from X to Y in C, *i. e.*,

$$C(X,Y) = \{\,f \mid f \text{ an arrow of } C,\ \mathrm{src}\, f = X \text{ and } \mathrm{trg}\, f = Y\,\}.$$

We also write "$f\colon X \to Y$ in C" to express that $f \in C(X,Y)$.

A special kind of categories are the *categories of sets*, where the set O of objects is a set of sets (in ZFC, this is always true), and the set A of arrows consists of all the total functions between these sets (to be precise, A consists of triples $\langle S, f, T\rangle$, where f is a total function from S to T), with composition the usual composition of functions. Note that a category of sets is completely determined by its set of objects.

Before presenting some examples of categories of sets, a foundational problem has to be solved. We would like to have at our disposal something like the "category of all sets"; unfortunately, this category would have the "set of all sets" as its set of objects, which does not exist in ZFC.

We shall adopt Mac Lane's solution to this problem [Mac Lane 71, p. 21–24], which is to postulate the existence of a "universe", that is, a set closed under all the usual operations of set theory. Such a universe can be said to contain all the sets of interest in "normal" mathematics, and hence it is a useful approximation to the idea of the "set of all sets".

To be precise, a *universe* is a set U such that

$\bigcup U \subseteq U$ (equivalently, $X \in U \implies X \subseteq U$),
for all $X \in U$: $\bigcup X \in U$ and $\mathbf{P}\, X \in U$,
$\mathbf{N} \in U$, and
whenever f is a function with $\operatorname{dom} f \in U$ and $\operatorname{ran} f \subseteq U$, then $\operatorname{ran} f \in U$.

Now ZFC set theory is augmented by the axiom

there exists a universe.

For the remainder of the thesis, $\mathbf{U}$ is a fixed universe. The elements of $\mathbf{U}$ are called *small sets*; similarly, mathematical objects such as functions or categories are called *small*, if they are elements of $\mathbf{U}$. The subsets of $\mathbf{U}$ (which may or may not be small sets), are called *classes*.

This gives us two important categories of sets:

Set is the category of sets whose objects are the small sets (*i. e.*, $|\mathbf{Set}| = \mathbf{U}$),
Cls is the category of sets whose objects are the classes (*i. e.*, $|\mathbf{Cls}| = \mathbf{P}(\mathbf{U})$).

Another useful kind of categories are the *preorder categories*, in which every hom-set has at most one element. These categories correspond to preorders, for if C is a preorder category, then its object set $|C|$ is preordered by the relation $X \sqsubseteq Y \;:\Longleftrightarrow\; C(X,Y) \neq \emptyset$, and if $\langle M, \sqsubseteq \rangle$ is a preorder, a preorder category is obtained by taking M as the object set, $\sqsubseteq$ as the arrow set (recall that $\sqsubseteq$ is a set

of ordered pairs) and by putting $\mathrm{src}(x,y) := x$, $\mathrm{trg}(x,y) := y$, $\mathrm{id}(x) := (x,x)$, and $(x,y)\,;(y,z) := (x,z)$.

If in a a preorder category C we have for all objects X, Y that $C(X,Y) \neq \emptyset$ and $C(Y,X) \neq \emptyset$ imply $X = Y$ (so that the preordering $\sqsubseteq$ of the category is a partial ordering on $|C|$), then C is called a *partial order category.*

One partial order category we shall use is **SetIncl**, whose objects are the small sets, and whose arrows are all the inclusion mappings between small sets.

With each category C is associated the *opposite* category C^{op}, which has the same object and arrow sets, but in which the notions of "source" and "target" are swapped and the order of arguments of composition is reversed:

$$\begin{aligned}&\text{If} \quad f\colon X \to Y \ (\textit{i.e.}, \mathrm{src}\, f = X \text{ and } \mathrm{trg}\, f = Y) \text{ and } f\,;g = h \text{ in } C,\\ &\text{then } f\colon Y \to X \ (\textit{i.e.}, \mathrm{src}\, f = Y \text{ and } \mathrm{trg}\, f = X) \text{ and } g\,;f = h \text{ in } C^{\mathrm{op}}.\end{aligned}$$

To avoid confusion, an arrow f of C^{op} is usually written "f^{op}", so that we obtain the laws

$$f\colon X \to Y \text{ in } C \iff f^{\mathrm{op}} : Y \to X \text{ in } C^{\mathrm{op}}$$

and

$$f\,;g = h \text{ in } C \iff g^{\mathrm{op}}\,;f^{\mathrm{op}} = h^{\mathrm{op}} \text{ in } C^{\mathrm{op}}.$$

A *functor* F from a category C to a category B (notation: "$F\colon C \to B$") consists of an *object function* mapping the C-objects to B-objects and an *arrow function* mapping the C-arrows to B-arrows (both of these mappings are usually written "F" also) such that whenever $X \in |C|$ and f and g are composable arrows of C, we have

$$\begin{aligned}F(\mathrm{id}(X)) &= \mathrm{id}(F(X))\\ F(f\,;g) &= F(f)\,;F(g).\end{aligned}$$

For example, this thesis will use the functor $(\_)^+\colon \mathbf{Set} \to \mathbf{Set}$ which maps a small set S to $\prod\langle S^*, S\rangle$ (*i.e.*, the set of all pairs $\langle s,r\rangle$ with $s \in S^*$ and $r \in S$),

and which maps a function $f\colon S \to T$ to the function $f^+\colon S^+ \to T^+$ defined by $f^+\langle\langle s_1,\ldots,s_n\rangle, r\rangle = \langle\langle f\,s_1,\ldots,f\,s_n\rangle, f\,r\rangle$.

A *category of categories* is a category whose objects are categories, and whose arrows are all the functors between them.

For example, we shall use the category of categories **LCat**, whose objects are all the large categories, that is, categories whose object and arrow sets are classes.

An *inclusion functor* is a functor whose object and arrow functions are inclusion functions. A category S is a *subcategory* of a category C, if there exists an inclusion functor from S to C (in other words, if the object and arrow sets of S are subsets of those of C, and the source, target, identity and composition operations of S are restrictions of those of C).

For example, the category **SetIncl** is a subcategory of **Set**, and **Set** is a subcategory of **Cls**.

2.2 Algebraic Signatures and Algebras

This section presents the notions "algebraic signature", "signature morphism" and "algebra", which are familiar in abstract data type theory; and it illustrates the way signatures and algebras model the data structures in functional programs, such as the ones of Section 1.4.

The signature notion is ubiquitous in the literature on abstract data types. In this thesis, I shall use the name "algebraic signature", because the term "signature" will be used in a more general sense in the next section. The notation I use is adopted from [BR 83, p. 221] and [Reichel 84, Def. 2.2.1].

Recall that if α is a function, then $\mathrm{dom}\,\alpha$ is its domain and $\mathrm{ran}\,\alpha$ is its range. If S is a set, S^* is the set of finite sequences over S, that is, of sequences $s = \langle s_1,\ldots,s_n\rangle$, where $n \in \mathbf{N}$, and $s_i \in S$ for all i in $\{1,\ldots,n\}$; the set S^+ is

defined as $S^+ := \prod\langle S^*, S\rangle$. If S and T are disjoint sets, their union is written "$S + T$".

2.2.1 Definition. An *algebraic signature* is a pair

$$\Sigma = \langle S, \alpha\rangle,$$

where α is a function with $\operatorname{ran}\alpha \subseteq S^+$, and $S \cap \operatorname{dom}\alpha = \emptyset$.

An algebraic signature will often be written in the form

$$\Sigma = \langle S,\ \alpha\colon F \to S^+\rangle,$$

which defines $F := \operatorname{dom}\alpha$. The elements of $S + F$ (note that S and F are disjoint) are the *symbols* of Σ; the elements of S are the *sort symbols* (or *sorts*), those of F the *function symbols* of Σ.

The map $\alpha\colon F \to S^+$ maps each function symbol to its *type* in Σ, and if $\alpha(f) = \langle s_1 \ldots s_n, r\rangle$ for some $f \in F$, $s_1, \ldots, s_n, r \in S$, write:

$$f\colon s_1 \ldots s_n \to r \text{ in } \Sigma,$$

and call the word $s_1 \ldots s_n$ the *source* (or *arity*) and r the *target* of f in Σ. □

Algebraic signatures have appeared throughout the program development in Section 1.4: The "signature" part of every interface given there is just an algebraic signature, given in a notation which is not the one suggested by the definition, but easy to read, unambiguous, and widely used in the literature. This notation will be used throughout this thesis to denote particular signatures. Its translation into the notation suggested by the definition is illustrated in the following example.

2.2.2 Example (Meaning of the conventional notation for signatures).
The signature of the interface I_{DICT} (Figure 1-5), which states the original problem of the program development in Section 1.4, is written in the conventional

notation:

signature

bool, item, listitem: **sort**

leitem: *item item* → *bool*

dictionary: *listitem* → *listitem*

This denotes the signature

$$\Sigma_{DICT} = \langle S, \alpha: F \to S^+ \rangle,$$

where

$$S = \{bool, item, listitem\},$$

$$F = \{leitem, dictionary\},$$

$$\alpha(leitem) = \langle item\ item, bool \rangle,$$

$$\alpha(dictionary) = \langle listitem, listitem \rangle.$$

The meaning of a line of the form "f: $s_1 \ldots s_n \to r$" is thus just as defined in Definition 2.2.1. □

The notion of an "algebra" that will now be defined is known in the literature under the name "partial many-sorted algebra" ([Burmeister 82, p. 350 f.], [BR 83, p. 221][1], [Wirsing *et al.* 83, p. 4], [Reichel 84, Def. 2.2.10], [KA 84, p. 321]).

Recall that "f: $S \nrightarrow T$" means that f is a partial function from S to T, *i.e.*, that f is a function, $\mathrm{dom}\, f \subseteq S$ and $\mathrm{ran}\, f \subseteq T$.

2.2.3 Definition. Let $\Sigma = \langle S, \alpha: F \to S^+ \rangle$ be an algebraic signature. An *algebra* of signature Σ (also called *Σ-algebra*) is a function A with domain $S+F$,

---

[1]The definition of a "partial algebra" in this paper contains a crucial printing error: The requirement for a partial algebra must be "$\mathrm{dom}\,\sigma^A \subseteq A_{s_1} \times \cdots \times A_{s_n}$", not "$\mathrm{dom}\,\sigma^A = A_{s_1} \times \cdots \times A_{s_n}$", as stated in the paper. With the latter condition, a "partial algebra" would be the same as a "total algebra".

such that whenever

$$f\colon s_1 \ldots s_n \to r \text{ in } \Sigma,$$

then

$$A(f)\colon \Big(\prod_{i\in\{1,\ldots,n\}} A(s_i)\Big) \nrightarrow A(r).$$

A Σ-algebra A maps a symbol z of Σ to its *interpretation* in A, written "A_z". The interpretations of the sort and function symbols of Σ are the *carriers* (often called *sorts*, by abuse of language) and *functions* of A, respectively. □

Algebras, too, have occurred throughout the program development in Section 1.4: On page 34 f. I mentioned that in the "**properties**" part of an interface, it is implicitly understood that a sort symbol denotes the set of values of a data type, and that a function symbol denotes a partial function from the (product of the) sets denoted by the input sort symbols to the set denoted by the output sort symbol. This just means that the interpretations of the symbols in the "**properties**" part of an interface are understood to form an algebra; an interface therefore describes properties of an algebra.

In general, an interface may be "satisfied" by any number of algebras. The case that an interface describes just one algebra is quite common—for example, the interfaces $I_{LISTITEM}$ (Figure 1-4), I_{DICT} (Figure 1-5), I_{INPUT} (Figure 1-11), I_{OUTPUT} (Figure 1-12), and I_{INSERT} (Figure 1-15) all describe a single algebra of the signature given in their "signature" part, given a fixed interpretation of the symbols "*item*" and "*leitem*". On the other hand, the interfaces I_{ITEM} (Figure 1-3), I_{INOUT} (Figure 1-9) and I_{MIN} (Figure 1-17) are "loose" in that they are satisfied by more than one algebra.

2.2.4 Example. (Recall that if f is a relation (in particular, a function), then $f^{\cup}$ is the converse of f, and $f(\!|S|\!) = \{\, y \mid \exists x \in S\colon (x,y) \in f \,\}$ is the image of S under f).

If the interpretations A_{item} of *item* and $A_{leitem}\colon \prod\langle A_{item}, A_{item}\rangle \to \{\mathbf{T},\mathbf{F}\}$ of *leitem* are given such that $\{\,(x,y) \mid A_{leitem}(x,y) = \mathbf{T}\,\}$ is a total ordering

on A_{item}, then the following algebra A is the unique algebra of signature Σ_{DICT} (Example 2.2.2) that satisfies the interface I_{DICT} (Figure 1-5):

$A_{bool} = \{\mathbf{T}, \mathbf{F}\}$

A_{item} : ⟨as given⟩

$A_{listitem} = (A_{item})^*$

A_{leitem} : ⟨as given⟩

$A_{dictionary}$: $(A_{item})^* \to (A_{item})^*$
maps $l \in (A_{item})^*$ to the unique $d \in (A_{item})^*$ that contains the elements occurring in l in ascending order, *i. e.*, the list d such that $\operatorname{ran} d = \operatorname{ran} l$ and Ascending(d). □

It can be seen here that the notations used to present an algebra in this thesis are the same as those used in the specifications of Section 1.4. That is to say, the language of rigorous program specifications is the same as the mathematical language used throughout this thesis. This makes the expressive power and flexibility of conventional mathematical notation available for program specification.

The general rôle of algebras in this thesis is to model the data types and operations of concrete programs. The functions of an algebra may be partial, that is, they may be undefined for some argument tuples. In such a case, no result value is delivered that could be further processed by other operations of the algebra. Hence, partiality represents "abortive" situations in which no further processing is done within a program (*i. e.*, in which nontermination or an abnormal program exit occurs).

In the theory of abstract data types, it is customary to use total, rather than partial, algebras to model data structures. In a total algebra, the functions always return result values, and so there is no direct way of indicating abortive situations.

The main technical advantage of total algebras is that their theory is simpler and better developed; while there is just one notion of "homomorphism", "subalgebra", and "congruence" for total algebras, there is an embarrassing wealth of variants of these notions for partial algebras ([Grätzer 79, p. 80], [Burmeister 82, p. 306]), so that a choice has to be made which variant to use for a certain purpose. This affects, for example, the "initial algebra semantics" of specifications (originally proposed in [GTW 78]), which depends on a homomorphism concept between algebras.

In Klaeren's book [Klaeren 83, p. 92–93], we find an argument in favour of total algebras from a "software engineering" viewpoint. Partiality of an operation, Klaeren argues, jeopardizes the "robustness" of a program (*i. e.*, its ability to cope with exceptional situations), because the application of a function to arguments for which it is undefined results in an abortive situation, in which no explicit error handling can take place.

Read contrapositively, Klaeren's argument is that for the design of robust programs, total algebras should be used: Since the functions of a total algebra return values in all circumstances, including exceptions, and since these values must then be processed further within a program, the programmer is forced to provide explicit exception handling, and encouraged to provide sensible error messages.

However, this argument in favour of total algebras ignores the fact that in a partial algebra, the same provisions for error handling are possible as in a total algebra—the designer is free to decide whether to provide a return value in an exceptional situation, or to treat the situation as abortive by making the operation that detects it undefined in this case. Total algebras do not offer this choice, and this can cause a number of problems.

First, the use of total algebras makes programming more complex, as it enforces consideration of exceptions that might otherwise have been neglected. Consider for example programming in a language where a value retrieved from memory might not necessarily be the expected one, but also the exception value for "memory failure" ("corrupt data"). This would force the programmer to

consider the handling of memory failures throughout the program. Only in few applications would the increased security justify the increase in program complexity.

A second problem in trying to make operations produce result values in all possible exception situations is that malfunctions may occur within the code that detects and handles exceptions itself. Here, the assumption that "every malfunction causes the production of a result value" seems to lead to an infinite regress.

A third problem is that many programming languages have basic operations that are partial, because they are capable of causing an abnormal program exit without giving the programmer a way of preventing this by "trapping" the exception (*e.g.*, division might cause a program exit when the denominator is zero). The most adequate model for such an operation is a partial function that does not return a value if an unrecoverable exception occurs. In a total algebra, the operation would have to be modelled by a total function. But the value returned by this function in a situation where its concrete counterpart aborts is totally fictitious. Since this fictitious value nevertheless has to be processed in some way by the other functions in the algebra, we obtain an algebraic model that is more complicated than necessary, that is no longer in correspondence with the concrete operations, and that therefore invites mistakes.

A similar problem occurs in cases where operations fail to terminate. Again, in the total algebra approach, the operation would have to be modelled as returning a fictitious value, "$\perp$", say, in this case. This in turn creates fictitious terms, such as "$f(\perp)$", that do not correspond to the application of a function to a value in the concrete program.

Nontermination is accepted by Klaeren as a reason for considering partial functions [Klaeren 83, p. 143 f.]. He argues, however, that the "basic" operations of a data structure are always total, and that nontermination can occur only in "derived" operations.

He thus creates a distinction between "basic" and "derived" operations of a program, which seems a high conceptual price to pay. In particular, an abstract

data type becomes less "abstract", if its operations are divided into "basic" and "derived" ones. Such a distinction is unnecessary from a user's point of view, rather, it is an "implementation detail" that should not enter into specifications.

As we have seen, the attempt to rule out abortive exception situations is problematic. In particular, abortive exceptions have the advantage of simplicity over exceptions that return values, as no special data values and no error handling are required. This suggest a useful rôle for abortive exceptions in program design, when used with deliberation.

In general, a tradeoff is necessary in program design between simplicity and robustness. The choice which exceptions to regard as abortive, and which to handle within a program, is an essential design decision. The range of exceptions that a program is designed to cope with may be called the "exception scope" of the program.

Some examples may illustrate the "exception scope" concept: In a compiler, failure of the syntax analysis would certainly have to be in the exception scope of the program and thus to be properly recognized and handled. In consequence, other exceptions will also have to be handled properly, for example, a failure in looking up a symbol in the symbol table, as this may occur during the compilation of a syntactically incorrect program.

On the other hand, there are exceptions which one would regard as being outside the scope of the compiler, such as, for example, corruption of a symbol table entry due to a memory failure. One would usually expect the underlying hardware to be designed in such a way that such errors are extremely unlikely to penetrate into the compiler's operation (this could, for example, be achieved by stopping the processor if a hardware malfunction was detected), and thus allow oneself the freedom to omit such errors from consideration in the design of the compiler.

In other programs, one might well make a different choice: Consider for example software controlling vital functions of an aircraft. Here, where memory failures are both more likely and potentially more disastrous, it would be mandatory to provide proper handling of situations where corrupt data are detected.

As a conclusion of the preceding discussion, I would suggest the following guideline for the use of partiality.

> **Strategy.** When designing a program, decide about the "exception scope" of the program, that is, about the range of exceptions to be anticipated and handled within the program. The functions of the algebraic model must yield well-defined "exception values" for exceptions inside the scope. Exceptions outside the scope should be modelled by partiality in order to keep the algebraic model simple.

The advantage of partial algebras is that they allow one to model abortive exception situations in an elegant way. In contrast, total algebras enforce the introduction of fictitious result values in such situations, and results of applying operations to these values have to be defined, although such applications are impossible in a concrete program.

The final concept to be introduced in this section is that of a "signature morphism", which is ubiquitous in abstract data type theory. A signature morphism maps the symbols of one signature onto the symbols of another, such that the type information is preserved. The notion of a signature being a "subsignature" of another can also be introduced on the basis of signature morphisms.

Recall that the operation $(\_)^+$ is defined for functions as well as for sets: if $f\colon S \to T$, then $f^+\colon S^+ \to T^+$ maps $\langle s_1 \dots s_n, r\rangle$ to $\langle (fs_1)\dots(fs_n),(fr)\rangle$ (it is usually obvious from the context which version of $(\_)^+$ is meant). The operation $(\_)^+$ is a functor $(\_)^+\colon \mathbf{Set} \to \mathbf{Set}$, that is, we have $(\mathrm{Id}(S))^+ = \mathrm{Id}(S^+)$ and $(f\,;g)^+ = f^+\,;g^+$ for all sets S and composable functions f and g.

2.2.5 Definition (Signature Morphisms, Subsignatures).
Let $\Sigma = \langle S, \alpha\colon F \to S^+\rangle$ and $\Sigma' = \langle S', \alpha'\colon F' \to (S')^+\rangle$ be algebraic signatures. A *signature morphism* from Σ to Σ' is a map $\sigma\colon S + F \to S' + F'$, such that

$$\sigma(\!|S|\!) \subseteq S', \qquad \sigma(\!|F|\!) \subseteq F',$$

and whenever $f\colon s_1 \dots s_n \to r$ in Σ, then

$$(\sigma f)\colon (\sigma s_1)\dots(\sigma s_n) \to (\sigma r) \text{ in } \Sigma'$$

(More concisely: $\alpha\,;\sigma^+ = \sigma\,;\alpha'$).

If σ is a signature morphism from Σ to Σ', we write "σ: $\Sigma \to \Sigma'$". If σ is an inclusion map of sets (*i. e.*, $\sigma = (S + F \subseteq S' + F')$), we call σ an *inclusion morphism* from Σ to Σ', and say that Σ is *included* in Σ', or a *subsignature* of Σ', and write "$\Sigma \sqsubseteq \Sigma'$". If $\Sigma \sqsubseteq \Sigma'$, we write "$(\Sigma \sqsubseteq \Sigma')$" for the unique inclusion morphism $(\Sigma \sqsubseteq \Sigma')$: $\Sigma \to \Sigma'$. □

The following proposition expresses some well-known properties of signature morphisms. The straightforward proof is omitted.

2.2.6 Proposition. *If σ: $\Sigma \to \Sigma'$ and σ': $\Sigma' \to \Sigma''$ are two signature morphisms, their composition (as functions) is a signature morphism $(\sigma\,;\sigma')$: $\Sigma \to \Sigma''$.*

If $\Sigma = \langle S,\ \alpha\colon F \to S^+\rangle$, then the identity map $\mathrm{Id}(S + F)$ is an inclusion morphism from Σ to itself. This morphism is a two-sided identity for the composition of morphisms.

A set of algebraic signatures as objects with the signature morphisms (inclusion morphisms) between them as arrows forms a category. □

2.2.7 Definition. Let "**ASig**" denote the category of small algebraic signatures with arrows the signature morphisms between them; let "**AIncl**" denote the category of small algebraic signatures with arrows the signature inclusions between them. □

The following properties of the inclusion relation are derived easily from the definition.

Recall that if α is a relation (in particular, a function) and S is a set, then $\alpha/S = \{\,(x,y) \in \alpha \mid x \in S\,\}$ is the restriction of α to S.

2.2.8 Proposition. *An algebraic signature $\Sigma = \langle S,\ \alpha\colon F \to S^+\rangle$ is included in an algebraic signature $\Sigma' = \langle S',\ \alpha'\colon F' \to (S')^+\rangle$, if and only if $S \subseteq S'$, $F \subseteq F'$, and $\alpha \subseteq \alpha'$ (or equivalently, $\alpha'/F = \alpha$).*

Every set of algebraic signatures is partially ordered by inclusion. □

If Σ is a subsignature of Σ', the sort symbols of Σ are sort symbols of Σ', and every function symbol $f\colon s_1 \dots s_n \to r$ in Σ is a function symbol of the same type in Σ'. However, not every subset of the symbols of Σ' defines a subsignature of Σ'—only those that with every function symbol $f\colon s_1 \dots s_n \to r$ of Σ' contain the sort symbols $s_1 \dots s_n$ and r as well (we might call such subsets "closed under α'"). These subsets are in 1-1 correspondence with the subsignatures of Σ'.

An important property of signature morphisms is that they allow the "translation" of algebras in the opposite direction, as stated in the following well-known proposition.

2.2.9 Proposition. *If $\sigma\colon \Sigma \to \Sigma'$ is a signature morphism, and A is a Σ'-algebra, then the functional composition $\sigma\,;A$ is a Σ-algebra. If A and Σ are small, so is $\sigma\,;A$.*

If both $\sigma\colon \Sigma \to \Sigma'$ and $\sigma'\colon \Sigma' \to \Sigma''$ are signature morphisms, then for any Σ''-algebra A: $\sigma\,;(\sigma'\,;A) = (\sigma\,;\sigma')\,;A$.

If $\sigma = \mathrm{Id}(\Sigma)\colon \Sigma \to \Sigma$, then $\sigma\,;A = A$ for all Σ'-algebras A. □

From this proposition it follows that we obtain a functor by associating with each signature morphism $\sigma\colon \Sigma \to \Sigma'$ between small signatures the map $\overline{\sigma}$ from the set of small Σ'-algebras to the set of small Σ-algebras defined by: $A \mapsto \sigma;A$.

2.2.10 Definition. Let $\mathbf{Alg}\colon \mathbf{ASig}^{\mathrm{op}} \to \mathbf{Cls}$ be the functor whose object function maps $\Sigma \in |\mathbf{ASig}|$ to the class of small Σ-algebras, and whose arrow function maps $\sigma\colon \Sigma \to \Sigma'$ in $\mathbf{ASig}$ to the function $\overline{\sigma}\colon \mathbf{Alg}(\Sigma') \to \mathbf{Alg}(\Sigma)$ defined by: $A \mapsto \sigma\,;A$.

If $\Sigma \sqsubseteq \Sigma'$ in $\mathbf{ASig}$ and $A \in \mathbf{Alg}(\Sigma')$, we call $A/\Sigma := \mathbf{Alg}((\Sigma \sqsubseteq \Sigma')^{\mathrm{op}})(A)$ the *reduct* of A to Σ. □

The "translation" of a Σ'-algebra A' along a signature morphism $\sigma\colon \Sigma \to \Sigma'$ can be described by saying that in the translated algebra $A := (\sigma\,;A')$ each symbol z

has the interpretation that $\sigma(z)$ has in A': $A_z = A'_{\sigma(z)}$. That A is a Σ-algebra follows from the "type preservation" properties of σ, *i. e.*, from the fact that σ maps sort symbols to sort symbols, function symbols to function symbols, and "preserves types" (if f: $s_1 \ldots s_n \to r$ in Σ, then (σf): $(\sigma s_1) \ldots (\sigma s_n) \to (\sigma r)$ in Σ').

If $\sigma = (\Sigma \sqsubseteq \Sigma')$ is an inclusion, the symbols of Σ form a subset of those of Σ', and the algebra A assigns to each symbol of Σ the same interpretation as A'—this is why A is called a "reduct" of A'. Note that a "reduct" is not the same as a "subalgebra" in this thesis: A "subalgebra" of an algebra A will later be defined to be an algebra with the same signature as A, in which each sort (and the graph of each function) is a subset of the corresponding sort (or function graph) of A (see Definition 4.3.5).

2.3 Institutions

This section introduces the notion of an "institution", which generalizes the setting of the previous section. Many of the definitions and theorems of this thesis will not be developed on the "concrete level" of algebraic signatures and partial algebras, but on the "abstract level" of institutions.

There are two reasons for this generalization. First, algebraic signatures and algebras are best suited to model a "pure" functional programming language, as used in the program development of Section 1.4. In other programming notations, there may be program entities other than just sorts and functions, for example variables, labels, exceptions, parameterized types (type constructors) etc. In general, a signature would state the syntactical ("type") attributes of the names denoting program entities in an environment. Hence, for a richer programming notation, a signature would have to comprise more classes of symbols than just sort and function symbols, and to provide adequate type information for symbols of those classes. Similarly, the semantic models for program entities would not just be algebras, but structures characterizing the semantics of the

program entities listed in a signature. Such structures in general will be called "models" of a signature.

By developing the theory as far as possible on the basis of signatures and models in general, I hope to make the theory applicable or at least easily transferable to modular programming in richer programming notations. However, this thesis will deal only with one concrete instance of the general setting, namely with algebraic signatures and algebras.

A second reason for developing the theory as far as possible on the abstract level is that this makes the mathematical development more elegant—the basic theorems about composition of specifications and implementations can be proved without consideration of the details of algebraic signatures and algebras, and this induces the same kind of simplification that data abstraction does in programming. An illustration of this simplification is that the proof of Theorem 4.1.7 in the present thesis requires 11 pages (page 189–200), while theorems analogous to this theorem and its Corollary 4.1.12, but significantly weaker, were proved in [Schoett 81] in two separate proofs requiring 5 pages [p. 124–129] and 7 pages [p. 61–68], respectively.

Developing the theory on an abstract level not only makes it simpler and more general, it also emphasizes the properties of signatures and models that are fundamental for modular programming, and separates them from the particular properties of algebraic signatures and algebras. This is in accordance with the "axiomatic method" as characterized by Bourbaki [Bourbaki 66, p. 3]:

> "La méthode axiomatique permet, lorsqu'on a affaire à des êtres mathématiques complexes, d'en dissocier les propriétés et de les regrouper autour d'un petit nombre de notions, c'est-à-dire, ... de les classer suivant les *structures* auxquelles elles appartiennent".[1]

---

[1] Translation, based on p. 9 of the English translation: "The axiomatic method allows us, when we are concerned with complex mathematical objects, to separate their properties

The basic idea behind the institution concept is to abstract from the details of the syntax and the semantic models of a logical or programming notation. This idea has inspired the development of "abstract model theory" in mathematics [Barwise 74], it has been introduced to computer science by Burstall and Goguen ([BG 80, p. 307 f.], [GB 84]), and has since been used by others (*e.g.*, [MM 84], [ST 85]).

In particular, the term "institution" for the abstract setting is borrowed from Goguen and Burstall [GB 84]. Two apologies must be made for this:

First, the notion used here differs in two important ways from the notion used by Goguen and Burstall:

- there is no notion of "sentences"; following an idea of Lipeck [Lipeck 83, p. 15 f.], specifications are treated as sets of models,
- there is a partial "inclusion" ordering on the set of signatures, so that lattice-like operations on signatures can be performed.

Sentences have been omitted from an institution, because they seem unnecessary for the theory; the inclusion ordering is the basis for operations on signatures that do not need explicitly stated signature morphisms. Despite these changes, however, the purpose of the institution notion in this thesis is the same as the purpose of Goguen and Burstall: to work in an axiomatic framework that abstracts from the details of particular semantic models and of particular specification notations. This fact has led me to adopt the name "institution" nevertheless.

Second, the institution notion I present is not a mature mathematical concept. Rather, I have collected in the definition of this notion the properties I needed in the development of the abstract layer of the theory (consisting of the present section, Chapter 3, and Sections 4.1 and 5.1). This has resulted in a somewhat inelegant set of axioms for the "institution" notion, "proof-generated" in the sense of [Lakatos 76, p. 127 f.]. The reader is thus asked not to ponder too

---

and regroup them around a small number of concepts, that is to say, ... to classify them according to the *structures* to which they belong".

deeply the significance of the individual axioms, particularly in Definition 2.3.5, but to regard them as documenting the requirements of the theory developed in this thesis. It is to be expected that in further developments of the theory, the axioms will be replaced by a stronger and perhaps more elegant set of properties.

In the following, an "institution" will be defined as a triple $\langle Sig, Incl, Mod \rangle$ satisfying certain axioms. Here *Sig* is a category of "signatures" and "signature morphisms", *Incl* a subcategory of *Sig*, consisting of "inclusion morphisms" only, and *Mod* a "model functor" from Sig^{op} to a category of sets. An example of an institution is $\langle$**ASig, AIncl, Alg**$\rangle$, which the reader is asked to keep in mind as an illustration of the concept.

Recall that a category I is a partial order category, if each hom-set $I(S,T)$ has at most one element, and if $I(S,T) \neq \emptyset$ and $I(T,S) \neq \emptyset$ imply $S = T$.

2.3.1 Definition. A *partially ordered category* is a pair $\langle C, I \rangle$ of categories, such that I is a subcategory of C, $|I| = |C|$, and I is a partial order category.

In a partially ordered category $\langle C, I \rangle$, call the arrows of I *inclusions*, and if $j\colon S \to T$ is an inclusion, say that S is *included* in T and write "$S \sqsubseteq T$". If $S \sqsubseteq T$, write "$(S \sqsubseteq T)$" for the unique inclusion from S to T, and if $\alpha\colon T \to U$ is an arrow of C, let $\alpha/S := (S \sqsubseteq T)\,;\alpha$ be the *restriction* of α to S. □

The pair $\langle$**ASig, AIncl**$\rangle$ is a partially ordered category, and the inclusion relation agrees with the inclusion relation of Definition 2.2.5. Another example of a partially ordered category is the pair $\langle$**Set, SetIncl**$\rangle$ (**Set** is the category of small sets and functions, **SetIncl** is the subcategory of **Set** that contains only the inclusion functions, cf. Section 2.1). Here, the inclusion relation is set inclusion, and the restriction of a morphism (*i. e.*, a function) to a set agrees with the standard notion of "restriction" (see Section 2.1).

In an institution, the inclusion relation between signatures is required to have some additional properties. One of these, the property of being "compatibly

complete", will now be introduced in the general context of partial orders, in order to derive some of its consequences first.

Recall that if $\langle M, \sqsubseteq \rangle$ is a partial order, an element S of M is said to be an "upper bound" of a set $P \subseteq M$, if and only if $T \sqsubseteq S$ for all $T \in P$. The element S is the "least upper bound" of P, if S is the least element among the upper bounds of P, that is, if $S \sqsubseteq T$ whenever T is an upper bound of P. The notions of "lower bound" and "greatest lower bound" are defined analogously.

These notions apply to *families* of elements of M as well—a bound of $\langle S_i \rangle_{i \in I}$ is the same as a bound of $\{ S_i \mid i \in I \}$. It does not really matter whether one considers bounds of sets or of families, since every set can be represented as a family (indexed by itself), and every family by the set of its elements (its range, to be precise). We will hence use the "bound" concept for sets and families indiscriminately.

2.3.2 Definition. Let $\sqsubseteq$ be a partial ordering on a set M. A set or family of elements of M is $\sqsubseteq$-*compatible* ("compatible", for short), if it has an upper bound according to $\sqsubseteq$.

The ordering $\sqsubseteq$ is *compatibly complete*, if every compatible set (and hence every compatible family) has a least upper bound. Let "$\bigsqcup P$" denote the least upper bound of a compatible set $P \subseteq M$, and let "$\bigsqcup_{i \in I} S_i$" denote the least upper bound (also called the *join*) of a compatible family $\langle S_i \rangle_{i \in I}$ of elements of M (written "$S \sqcup T$" in the binary case). □

Readers familiar with lattices might note that a compatibly complete partial order $\langle M, \sqsubseteq \rangle$ becomes a complete lattice when a new "top" element is added that is greater than all the elements of M. Conversely, each complete lattice is a compatibly complete partial order, and removing the top element from a complete lattice with at least two elements still leaves a compatibly complete partial order. The following proposition and its proof have well-known analogues in complete lattices.

2.3.3 Proposition. *In a compatibly complete partial order every nonempty set (and hence every nonempty family) has a greatest lower bound.*

Proof. Let $\langle M, \sqsubseteq \rangle$ be a compatibly complete partial order, and let $P \subseteq M$ be nonempty. Let Q be the set of all lower bounds of P. Every element of P is an upper bound of Q. Since P is nonempty, Q is compatible, and thus possesses a least upper bound $K := \bigsqcup Q$. We show that K is the greatest lower bound of P. First, K is a lower bound of P, because if $T \in P$, then T is upper bound of Q, and hence $K = \bigsqcup Q \sqsubseteq T$. Second, if S is any lower bound of P, then $S \in Q$, and hence $S \sqsubseteq K$. □

2.3.4 Definition. If $\sqsubseteq$ is a compatibly complete partial order on some set, let "$S \sqcap T$" denote the greatest lower bound (also called the *meet*) of two elements S and T of that set. □

An important property of the join operator is that the result of an arbitrary expression using binary and general joins (and no other operators) depends only on the *set* of elements occurring in the expression and is independent of the arrangement of the operators. From this, one easily obtains that the familiar laws of commutativity, associativity, and idempotence hold whenever the signatures involved in an expression are compatible.

The meet operator possesses analogous properties. It is subject to the restriction, however, that the number of elements combined in a meet expression must not be zero.

In the remainder of this thesis, these properties of join and meet will be used without mentioning them explicitly.

After this brief treatment of properties of compatibly complete partial orders, we are now ready to state the syntactic properties of an "institution".

2.3.5 Definition. An *institution syntax* is a partially ordered category $\langle Sig, Incl \rangle$ that satisfies the following axioms.

(a) (Compatible Completeness)

The partial ordering $\sqsubseteq$ on $|Sig|$ defined by *Incl* is compatibly complete.

Let "$S \sim T$" mean that S and T are compatible signatures, use "$\bigsqcup$", "$\sqcup$" to denote general and binary joins of compatible (families of) signatures, and "$\sqcap$" to denote binary meets.

(b) (Distributivity)

If $\langle S_i \rangle_{i \in I \cup \{0\}}$ is a compatible family of Sig-objects, then

$$S_0 \sqcap (\bigsqcup_{i \in I} S_i) = \bigsqcup_{i \in I} (S_0 \sqcap S_i).$$

(c) (Renaming)

Whenever S, T, and U are Sig-objects such that $S \sim T$ and $S \sim U$, then there exists an object $\hat{U}$, such that

$$\hat{U} \sim S \sqcup T,$$
$$\hat{U} \sqcap (S \sqcup T) = U \sqcap S,$$

together with Sig-isomorphisms

$$j\colon \hat{U} \to U, \qquad k\colon \hat{U} \sqcup S \to U \sqcup S,$$

such that

$$(\hat{U} \sqsubseteq \hat{U} \sqcup S)\,;k = j\,;(U \sqsubseteq U \sqcup S)$$
$$(S \sqsubseteq \hat{U} \sqcup S)\,;k = (S \sqsubseteq U \sqcup S)$$
$$(U \sqcap S \sqsubseteq \hat{U})\,;j = (U \sqcap S \sqsubseteq U)$$

(in other words, the diagram shown in Figure 2-1 commutes).

In an institution syntax $\langle Sig, Incl \rangle$, call the objects of Sig *signatures* and the morphisms of Sig *signature morphisms*, and if $S \sqsubseteq T$, say that S is a *subsignature* of T. We regard $\sqsubseteq$ as the "standard" partial ordering on the set of signatures, and hence we shall speak of "compatible" signatures, and of "least upper bounds", "joins" and "meets" of signatures, always implicitly referring to $\sqsubseteq$. □

The general idea behind this definition is that the signatures of an institution syntax represent the "type environments" of a programming language; each signature corresponds to a set of program symbols ("identifiers") with associated

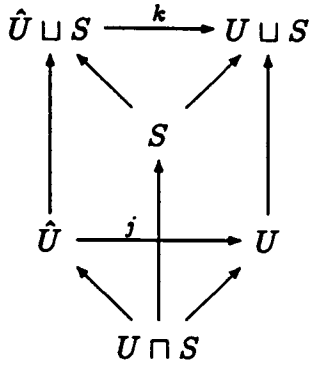


Figure 2-1: Diagram postulated by the renaming axiom

type information. This type information would allow one to determine whether code that uses the symbols of a signature was syntactically well-formed.

The signature morphisms mentioned in the axioms above are either inclusions or isomorphisms or compositions of these. No other kinds of signature morphisms will be used in the general theory based on institutions, although an institution syntax might contain such morphisms. A subsignature of a signature S may be thought of as containing a subset of the symbols of S; a signature isomorphic to S may be thought of as a "renaming" of S, which could be obtained from S by performing a one-to-one substitution of symbols.

Note that axiom (b) of the definition requires distributivity only in the case that S_0 is compatible with the S_i $(i \in I)$, although both sides of the equation would be defined even if S_0 was incompatible with $\bigsqcup_{i \in I} S_i$. In fact, in this thesis the meet operator $\sqcap$ will never be applied to incompatible signatures—compatibility will be regarded as a prerequisite for forming both joins and meets of signatures. The only exception to this is Theorem 2.3.6 (c) below, which characterizes the meet of algebraic signatures regardless of their compatibility.

Note also that the inclusion

$$S_0 \sqcap (\bigsqcup_{i \in I} S_i) \sqsupseteq \bigsqcup_{i \in I} (S_0 \sqcap S_i).$$

is trivially true, because the family $\langle S_0 \sqcap S_i \rangle_{i \in I}$ is bounded by $S_0 \sqcap (\bigsqcup_{i \in I} S_i)$.

While the completeness and distributivity axioms will be used frequently in this thesis, the renaming axiom is rather specialized: it will only be used once, in

the proof of Lemma 4.1.9. This lemma says that the presence of "hidden" internal symbols in a system may safely be ignored when looking at that system from the outside, a fact which is essential for the composability of implementations (Theorem 4.1.7).

To help understand the renaming axiom, here is a rough illustration of its meaning. Think of T and U as two separate name spaces of some program (*e.g.*, T might represent the declarations in force at some point in the program, and U might represent the declarations visible inside a block declared at that point), and of S as the entities that T and U have in common (in the example, S would be the set of global symbols visible inside the block, that is, the set of symbols that do not receive new definitions). What the axiom says then is that there is a "renaming" $\hat{U}$ of U, such that no clashes occur between $\hat{U}$ and T ($\hat{U} \sim S \sqcup T$), the symbols of $U \sqcap S$ have not been changed ($\hat{U} \sqcap S = U \sqcap S$), and $\hat{U}$ and T share symbols only via S ($\hat{U} \sqcap T \sqsubseteq U \sqcap S$). In the example, this would be a renaming of the local symbols of the block that makes them disjoint from the symbols of the global name space T.

Thus, in general, the renaming axiom expresses the possibility of avoiding clashes between local and global declarations by renaming the local declarations while preserving the "connections" between the local and the global declarations.

It is also worthwhile to note that the renaming axiom is symmetric between T and U in its assumptions as well as in its conclusions. Defining $\hat{T} := T$, $j' := \mathrm{Id}(T)\colon \hat{T} \to T$, and $k' := \mathrm{Id}(T \sqcup S)$, we can substitute T for U, $\hat{T}$ for $\hat{U}$, j' for j and k' for k in the conclusion of the axiom, and obtain statements that are also valid.

This symmetry indicates that clashes between local and global definitions can equally well be avoided by renaming the global declarations instead of the local ones; the proof of Lemma 4.1.9 does this, because it is formally simpler.

We now verify that the pair $\langle \mathbf{ASig}, \mathbf{AIncl} \rangle$ is an institution syntax.

2.3.6 Theorem. *The pair* $\langle \mathbf{ASig}, \mathbf{AIncl} \rangle$ *is an institution syntax. In particular,*

(a) *A family* $\langle \Sigma_i \rangle_{i \in I}$ *of small algebraic signatures* $\Sigma_i = \langle S_i, \alpha_i \colon F_i \to S_i^+ \rangle$ *is compatible, if and only if:*

$$\bigcup_{i \in I}(S_i + F_i) \text{ is small,} \qquad (\bigcup_{i \in I} S_i) \cap (\bigcup_{i \in I} F_i) = \emptyset, \qquad \bigcup_{i \in I} \alpha_i \text{ is a function.}$$

In particular, if $I = \{1,2\}$, *the signatures* Σ_1 *and* Σ_2 *are compatible, if and only if*

$$S_1 \cap F_2 = S_2 \cap F_1 = \emptyset, \quad \text{and} \quad \forall x \in F_1 \cap F_2 \colon \alpha_1(x) = \alpha_2(x).$$

(b) *If* $\langle \Sigma_i \rangle_{i \in I}$ *is a compatible family of small algebraic signatures, such that* $\Sigma_i = \langle S_i, \alpha_i \colon F_i \to S_i^+ \rangle$, *then*

$$\bigsqcup_{i \in I} \Sigma_i = \langle \bar{S}, \bar{\alpha} \colon \bar{F} \to \bar{S}^+ \rangle,$$

where

$$\bar{S} = \bigcup_{i \in I} S_i, \qquad \bar{\alpha} = \bigcup_{i \in I} \alpha_i, \qquad \bar{F} = \bigcup_{i \in I} F_i.$$

(c) *If* $\langle \Sigma_i \rangle_{i \in I}$ *is a nonempty family of small algebraic signatures, such that* $\Sigma_i = \langle S_i, \alpha_i \colon F_i \to S_i^+ \rangle$, *then*

$$\prod_{i \in I} \Sigma_i = \langle \bar{S}, \bar{\alpha} \colon \bar{F} \to \bar{S}^+ \rangle,$$

where

$$\bar{S} = \bigcap_{i \in I} S_i, \qquad \bar{\alpha} = \bigcap_{i \in I} \alpha_i, \qquad \bar{F} = \operatorname{dom} \bar{\alpha}.$$

If in addition the family is compatible, then $\bar{F} = \bigcap_{i \in I} F_i$.

The proof uses the following lemma.

2.3.7 Lemma. *If* $k \colon \Sigma \to \Sigma'$ *is an arrow in* **ASig** *(i. e., a signature morphism between small algebraic signatures) which is a bijection between the symbol sets of* Σ *and* Σ', *then the inverse* k^{-1} *of this bijection is the inverse signature morphism to* k; *in particular,* k *and* k^{-1} *are* **ASig***-isomorphisms.*

Proof. Let $\Sigma = \langle S,\ \alpha\colon F \to S^+\rangle$ and $\Sigma' = \langle S',\ \alpha'\colon F' \to (S')^+\rangle$. Then k is, by assumption, a bijection from $S + F$ to $S' + F'$. We show that k^{-1} also is a signature morphism. It is then clear that it is the inverse of the signature morphism k, and that k and k^{-1} are therefore **ASig**-isomorphisms.

We have $k^{-1}(\!|S'|\!) \subseteq S$, because otherwise k would map an element of F to an element of S'; analogously, we have $k^{-1}(\!|F'|\!) \subseteq F$. Finally, since $\alpha\,;k^+ = k\,;\alpha'$, we have

$$\alpha'\,;(k^{-1})^+ = k^{-1}\,;k\,;\alpha'\,;(k^{-1})^+ = k^{-1}\,;\alpha\,;k^+\,;(k^{-1})^+ = k^{-1}\,;\alpha.$$

Hence k^{-1} is a signature morphism from Σ' to Σ. □

Proof of Theorem 2.3.6.

The pair $\langle$**ASig**, **AIncl**$\rangle$ is a partially ordered category. We shall verify the three axioms of the "institution syntax" definition (Def. 2.3.5) and in the process prove the formulae given in the theorem.

We first prove formula (a) of the theorem. Let $\langle\Sigma_i\rangle_{i\in I}$ be a family of small algebraic signatures, where $\Sigma_i = \langle S_i,\ \alpha_i\colon F_i \to S_i^+\rangle$. Suppose first that $\langle\Sigma_i\rangle_{i\in I}$ is compatible. Then we can choose a small algebraic signature $\hat{\Sigma} = \langle\hat{S},\ \hat{\alpha}\colon \hat{F} \to \hat{S}^+\rangle$ that is an upper bound of the family. But then $\bigcup_{i\in I}(S_i + F_i)$ is small, because it is a subset of the small set $\hat{S} + \hat{F}$; we have $\bigcup_{i\in I} S_i \cap \bigcup_{i\in I} F_i \subseteq \hat{S} \cap \hat{F} = \emptyset$; and $\bigcup_{i\in I}\alpha_i$ is a function, because it is a subset of the function $\hat{\alpha}$.

Conversely, suppose that the three conditions given in (a) are true. We verify that $\bar{\Sigma} := \langle\bigcup_{i\in I} S_i, \bigcup_{i\in I}\alpha_i\rangle$ is a small algebraic signature. Clearly, it is then an upper bound of the family $\langle\Sigma_i\rangle_{i\in I}$.

Define $\bar{S} := \bigcup_{i\in I} S_i$, $\bar{\alpha} := \bigcup_{i\in I}\alpha_i$, and $\bar{F} := \operatorname{dom}\bar{\alpha} = \bigcup_{i\in I}\operatorname{dom}\alpha_i = \bigcup_{i\in I} F_i$. The set $\bar{S}$ is small, because is is a subset of $\bigcup_{i\in I}(S_i + F_i)$, which is small by assumption. $\bar{\alpha}$ is a function mapping the elements of $\bar{F}$ to small sets. Hence the elements of $\bar{\alpha}$ are small, and $\bar{\alpha}$ contains exactly one such element per element of the small set $\bar{F}$. It follows that $\bar{\alpha}$ and $\bar{\Sigma}$ are small. We have $\bar{S}\cap\bar{F} = \emptyset$ by assumption. Because $\operatorname{ran}\bar{\alpha} = \bigcup_{i\in I}\operatorname{ran}\alpha_i \subseteq \bigcup_{i\in I} S_i^+ \subseteq \bar{S}^+$, $\bar{\alpha}$ is a function from $\bar{F}$ to $\bar{S}^+$. Hence $\bar{\Sigma}$ is a small algebraic signature.

The conditions given for the special case $I = \{1,2\}$ are easily seen to be equivalent to the general ones in this case.

Axiom (a) (Compatible Completeness):

We prove formula (b) of the theorem, which implies the compatible completeness property. Let $\langle \Sigma_i \rangle_{i \in I}$ be a compatible family of small algebraic signatures. Then we can choose a small algebraic signature $\hat{\Sigma} = \langle \hat{S}, \hat{\alpha} \rangle$ that is an upper bound of the Σ_i ($i \in I$). We have $S_i \subseteq \hat{S}$ and $\alpha_i \subseteq \hat{\alpha}$ for all $i \in I$. Define $\bar{S} := \bigcup_{i \in I} S_i$ and $\bar{\alpha} := \bigcup_{i \in I} \alpha_i$. As $\bar{S} \subseteq \hat{S}$ and $\bar{\alpha} \subseteq \hat{\alpha}$, both $\bar{S}$ and $\bar{\alpha}$ are small. We have $\operatorname{ran} \alpha = \bigcup_{i \in I} \operatorname{ran} \alpha_i \subseteq \bigcup_{i \in I} S_i^+ \subseteq \bar{S}^+$, and $(\operatorname{dom} \bar{\alpha}) \cap \bar{S} \subseteq (\operatorname{dom} \hat{\alpha}) \cap \hat{S} = \emptyset$. Hence $\langle \bar{S}, \bar{\alpha} \rangle$ is a small algebraic signature. Clearly, it is the least upper bound of the family $\langle \Sigma_i \rangle_{i \in I}$.

Axiom (b) (Distributivity):

We first prove formula (c) of the theorem. Let $\Sigma_i = \langle S_i, \alpha_i \colon F_i \to S_i^+ \rangle$ for $i \in I$, and let $\bar{\Sigma}$ be defined as in formula (c). We show that $\bar{\Sigma}$ is a small algebraic signature. It then follows trivially that $\bar{\Sigma}$ is the greatest lower bound of the Σ_i ($i \in I$), because the sort set of any lower bound must be a subset of every S_i, hence of $\bar{S}$, and the type map of any lower bound must be a subset of every α_i, hence of $\bar{\alpha}$.

Clearly, $\bar{\Sigma}$ is small. Further,

$$\begin{aligned} \operatorname{ran} \bar{\alpha} &= \operatorname{ran} \bigcap_{i \in I} \alpha_i \\ &\subseteq \bigcap_{i \in I} \operatorname{ran} \alpha_i \\ &\subseteq \bigcap_{i \in I} S_i^+ \\ &= \left(\bigcap_{i \in I} S_i \right)^+ \\ &= \bar{S}^+, \end{aligned}$$

and $\bar{\alpha}$ is a function, because it is a subset of some α_i, which is a function. Hence $\bar{\alpha} \colon \bar{F} \to \bar{S}^+$. Since $\bar{S} \cap \bar{F} \subseteq S_i \cap F_i = \emptyset$ for some i, $\bar{S}$ and $\bar{F}$ are disjoint, and hence $\bar{\Sigma}$ is a small algebraic signature.

We have $\bar{F} = \operatorname{dom} \bar{\alpha} = \operatorname{dom} \bigcap_{i \in I} \alpha_i \subseteq \bigcap_{i \in I} \operatorname{dom} \alpha_i = \bigcap_{i \in I} F_i$. Suppose now that the family $\langle \Sigma_i \rangle_{i \in I}$ is compatible. We can then pick an upper bound $\hat{\Sigma} = \langle \hat{S}, \hat{\alpha} \colon \hat{F} \to \hat{S}^+ \rangle$ of the family. To show that $\bigcap_{i \in I} F_i \subseteq \bar{F}$, consider

any $f \in \bigcap_{i\in I} F_i$, and let $\tau := \hat{\alpha}(f)$. Then for any i in I: $f \in F_i = \operatorname{dom}\alpha_i$, and $\alpha_i \subseteq \hat{\alpha}$, hence $\alpha_i(f) = \tau$. It follows that $(f,\tau) \in \bigcap_{i\in I}\alpha_i = \bar{\alpha}$, hence $f \in \operatorname{dom}\bar{\alpha} = \bar{F}$. This concludes the proof of formula (c) of the theorem.

The proof of the distributivity axiom (axiom (b) of Def. 2.3.5) is now straightforward. Let $\Sigma_i = \langle S_i,\ \alpha_i\colon F_i \to S_i^+\rangle$ for $i \in I \cup \{0\}$, and assume that $\langle\Sigma_i\rangle_{i\in I}$ is compatible (compatibility of Σ_0 with $\bigsqcup_{i\in I}\Sigma_i$ is not required in this proof). Then

$$\begin{aligned}
\Sigma_0 \sqcap \Big(\bigsqcup_{i\in I}\Sigma_i\Big) &= \Sigma_0 \sqcap \Big\langle \bigcup_{i\in I} S_i,\ \bigcup_{i\in I}\alpha_i\Big\rangle \\
&= \Big\langle S_0 \cap \big(\bigcup_{i\in I} S_i\big),\ \alpha_0 \cap \big(\bigcup_{i\in I}\alpha_i\big)\Big\rangle \\
&= \Big\langle \bigcup_{i\in I}(S_0 \cap S_i),\ \bigcup_{i\in I}(\alpha_0 \cap \alpha_i)\Big\rangle \\
&= \bigsqcup_{i\in I}\langle S_0 \cap S_i,\ \alpha_0 \cap \alpha_i\rangle \\
&= \bigsqcup_{i\in I}(\Sigma_0 \sqcap \Sigma_i).
\end{aligned}$$

Axiom (c) (Renaming):

Suppose that three small algebraic signatures Σ_0, Σ_1 and Σ_2 are given, where $\Sigma_i = \langle S_i,\ \alpha_i\colon F_i \to S_i^+\rangle$ for $i \in \{0,1,2\}$, and suppose that

$$\Sigma_0 \sim \Sigma_1 \quad \text{and} \quad \Sigma_0 \sim \Sigma_2$$

(Σ_0, Σ_1 and Σ_2 correspond to S, T and U in the renaming axiom).

Construct a small algebraic signature $\hat{\Sigma}_2$, together with signature isomorphisms

$$j\colon \hat{\Sigma}_2 \to \Sigma_2, \qquad k\colon \hat{\Sigma}_2 \sqcup \Sigma_0 \to \Sigma_2 \sqcup \Sigma_0$$

as follows: Pick a set R of the same cardinality as $(S_2 + F_2) \setminus (S_0 + F_0)$, such that $R \cap ((S_0 + F_0) \cup (S_1 + F_1)) = \emptyset$.[1] Let $i\colon R \to (S_2 + F_2) \setminus (S_0 + F_0)$ be a

[1]For example, R could be constructed in the following way: Because $D := \operatorname{dom}((S_0 + F_0) \cup (S_1 + F_1))$ is a set, we can pick $x \notin D$. Define $R := \{x\} \times ((S_2 + F_2) \setminus (S_0 + F_0))$.

bijection. Define

$$k := (i + \mathrm{Id}(S_0 + F_0))\colon R + (S_0 + F_0) \to (S_2 + F_2) \cup (S_0 + F_0)$$

(i and $\mathrm{Id}(S_0+F_0)$ are disjoint, because they are functions with disjoint domains). The map k is bijective, with inverse

$$k^{-1} = (i^{-1} + \mathrm{Id}(S_0 + F_0))\colon (S_2 + F_2) \cup (S_0 + F_0) \to R + (S_0 + F_0).$$

Now let $\hat{S}_2 := k^{-1}(\!|S_2|\!)$, and $\hat{F}_2 := k^{-1}(\!|F_2|\!)$, and define $\hat{\alpha}_2\colon \hat{F}_2 \to \hat{S}_2^+$ as the composition of the diagram

$$\hat{F}_2 \xrightarrow{k/\hat{F}_2} F_2 \xrightarrow{\alpha_2} S_2^+ \xrightarrow{(k^{-1})^+/S_2^+} \hat{S}_2^+,$$

i.e., let $\hat{\alpha}_2 := (k/\hat{F}_2)\,;\alpha_2\,;((k^{-1})^+/S_2^+)$.

The sets $\hat{S}_2$ and $\hat{F}_2$ are small and disjoint, because S_2 and F_2 are. Hence $\hat{\Sigma}_2 := \langle \hat{S}_2,\ \hat{\alpha}_2\colon \hat{F}_2 \to \hat{S}_2^+ \rangle$ is a small algebraic signature.

We have

$$\begin{aligned}
\hat{F}_2 \cap (F_0 \cup F_1) &= k^{-1}(\!|F_2|\!) \cap (F_0 \cup F_1) \\
&= \big(k^{-1}(\!|F_2 \setminus (S_0 + F_0)|\!) + k^{-1}(\!|F_2 \cap (S_0 + F_0)|\!)\big) \cap (F_0 \cup F_1) \\
&= \big(i^{-1}(\!|F_2 \setminus (S_0 + F_0)|\!) + (F_2 \cap (S_0 + F_0))\big) \cap (F_0 \cup F_1) \\
&= (F_2 \cap (S_0 + F_0)) \cap (F_0 \cup F_1) \\
&\qquad\qquad \text{(as } i^{-1}(\!|F_2 \setminus (S_0 + F_0)|\!) \subseteq R, \text{ and } R \cap (F_0 \cup F_1) = \emptyset) \\
&= (F_2 \cap F_0) \cap (F_0 \cup F_1) \\
&= F_2 \cap F_0.
\end{aligned}$$

Consider $x \in \hat{F}_2 \cap (F_0 \cup F_1)$. Then $x \in F_2 \cap F_0$, therefore $\alpha_2(x) = \alpha_0(x) \in S_0^+$, and hence

$$\begin{aligned}
\hat{\alpha}_2(x) &= ((k^{-1})^+/S_2^+)(\alpha_2((k/\hat{F}_2)(x))) \\
&= ((k^{-1})^+/S_2^+)(\alpha_2(x)) && \text{(because } x \in F_0) \\
&= \alpha_2(x) && \text{(because } \alpha_2 x \in S_0^+).
\end{aligned}$$

Hence we have shown that

$$\text{if } x \in \hat{F}_2 \cap (F_0 \cup F_1)\ [= F_2 \cap F_0], \text{ then } \hat{\alpha}_2(x) = \alpha_2(x) = \alpha_0(x). \qquad (*)$$

We shall now show that $\hat{\Sigma}_2$ has the properties required of $\hat{U}$ in the renaming axiom.

First, $\hat{\Sigma}_2 \sim \Sigma_0 \sqcup \Sigma_1$ according to formula (a) of the theorem, because

$$\begin{aligned}
\hat{S}_2 \cap (F_0 \cup F_1) &= k^{-1}(\!|S_2|\!) \cap (F_0 \cup F_1) \\
&= \left(k^{-1}(\!|S_2 \setminus (S_0 + F_0)|\!) + k^{-1}(\!|S_2 \cap (S_0 + F_0)|\!)\right) \cap (F_0 \cup F_1) \\
&= \left(i^{-1}(\!|S_2 \setminus (S_0 + F_0)|\!) + (S_2 \cap (S_0 + F_0))\right) \cap (F_0 \cup F_1) \\
&\subseteq (R + S_0) \cap (F_0 \cup F_1) \\
&= R \cap (F_0 \cup F_1) + S_0 \cap (F_0 \cup F_1) \\
&= \emptyset + \emptyset \\
&= \emptyset,
\end{aligned}$$

because, by a symmetrical argument,

$$\hat{F}_2 \cap (S_0 \cup S_1) = \emptyset,$$

and because, according to (*), $\alpha_2(x) = \alpha_0(x)$ for $x \in \hat{F}_2 \cap (F_0 \cup F_1)$.

Second, we show that $\hat{\Sigma}_2 \cap (\Sigma_0 \sqcup \Sigma_1) = \Sigma_2 \cap \Sigma_0$. By formulas (b) and (c), we have

$$\hat{\Sigma}_2 \cap (\Sigma_0 \sqcup \Sigma_1) = \langle \hat{S}_2 \cap (S_0 \cup S_1),\ \hat{\alpha}_2 \cap (\alpha_0 \cup \alpha_1) \rangle.$$

By an argument analogous to the proof above that $\hat{F}_2 \cap (F_0 \cup F_1) = F_2 \cap F_0$, one shows that

$$\hat{S}_2 \cap (S_0 \cup S_1) = S_2 \cap S_0,$$

which is the sort set of $\Sigma_2 \cap \Sigma_0$. Finally,

$$\begin{aligned}
\hat{\alpha}_2 \cap (\alpha_0 \cup \alpha_1) &= (\hat{\alpha}_2 / k^{-1}(\!|F_2|\!)) \cap (\alpha_0 \cup \alpha_1) \\
&= \left(\hat{\alpha}_2 / R + \hat{\alpha}_2 / (F_2 \cap (S_0 + F_0))\right) \cap (\alpha_0 \cup \alpha_1) \\
&= (\hat{\alpha}_2 / (F_2 \cap F_0)) \cap (\alpha_0 \cup \alpha_1) && \text{(as } R \cap (F_0 \cup F_1) = \emptyset\text{)} \\
&= (\alpha_2 / (F_2 \cap F_0)) \cap (\alpha_0 \cup \alpha_1) && \text{(due to (*))} \\
&= (\alpha_2 / (F_2 \cap F_0)) \cap ((\alpha_0 \cup \alpha_1) / (F_2 \cap F_0)) \\
&= (\alpha_2 / (F_2 \cap F_0)) \cap (\alpha_0 / (F_2 \cap F_0)), && \text{(because } \Sigma_0 \sim \Sigma_1\text{)} \\
&= (\alpha_2 \cap \alpha_0) / (F_2 \cap F_0) \\
&= \alpha_2 \cap \alpha_0,
\end{aligned}$$

which is the type map of $\Sigma_2 \sqcap \Sigma_0$. Hence $\hat{\Sigma}_2 \sqcap (\Sigma_0 \sqcup \Sigma_1) = \Sigma_2 \sqcap \Sigma_0$.

Next, we show that k is an **ASig**-isomorphism $k\colon \hat{\Sigma}_2 \sqcup \Sigma_0 \to \Sigma_2 \sqcup \Sigma_0$. Since k is a bijection from $(\hat{S}_2 + \hat{F}_2) \cup (S_0 + F_0)$ to $(S_2 + F_2) \cup (S_0 + F_0)$, according to Lemma 2.3.7 it is sufficient to show that k is a signature morphism.

To see that k is a signature morphism, observe first that $k(\!|\hat{S}_2 \cup S_0|\!) = k(\!|\hat{S}_2|\!) \cup k(\!|S_0|\!) \subseteq S_2 \cup S_0$, which is the sort set of $\Sigma_2 \sqcup \Sigma_0$, and that analogously $k(\!|\hat{F}_2 \cup F_0|\!)$ is a subset of the set of function symbols of $\Sigma_2 \sqcup \Sigma_0$. Finally, for $f\colon s_1 \dots s_n \to r$ in $\hat{\Sigma}_2 \sqcup \Sigma_0$:

- if $f \in F_0$, then $(\hat{\alpha}_2 \cup \alpha_0)(f) = \alpha_0(f) \in S_0^+$, and thus $k^+((\hat{\alpha}_2 \cup \alpha_0)(f)) = \alpha_0(f) = (\alpha_2 \cup \alpha_0)(k(f))$,
- if $f \in \hat{F}_2$, then $k^+((\hat{\alpha}_2 \cup \alpha_0)(f)) = k^+(\hat{\alpha}_2(f)) = k^+((k^{-1})^+(\alpha_2(k(f)))) = \alpha_2(k(f)) = (\alpha_2 \cup \alpha_0)(k(f))$.

Thus, $(\hat{\alpha}_2 \cup \alpha_0)\ ;\ k^+ = k\ ;\ (\alpha_2 \cup \alpha_0)$. Hence k is a signature morphism from $\hat{\Sigma}_2 \sqcup \Sigma_0$ to $\Sigma_2 \sqcup \Sigma_0$.

Consider now the signature morphism

$$k/\hat{\Sigma}_2\colon \hat{\Sigma}_2 \to \Sigma_2 \sqcup \Sigma_0.$$

The associated mapping is $k/(\hat{S}_2 + \hat{F}_2)$, which is a bijection between $\hat{S}_2 + \hat{F}_2$ and $S_2 + F_2$. Hence the map $k/(\hat{S}_2 + \hat{F}_2)$ defines a signature morphism

$$j\colon \hat{\Sigma}_2 \to \Sigma_2,$$

and because the map is bijective, Lemma 2.3.7 yields that j is an **ASig**-isomorphism between $\hat{\Sigma}_2$ and Σ_2.

It remains to verify the three equations relating k and j in the renaming axiom. For this, it is sufficient to show that the respective symbol mappings are the same, and since the symbol mappings of inclusion morphisms are inclusion mappings, it remains to show that k agrees with j on $\hat{S}_2 + \hat{F}_2$, that k is the identity on $S_0 + F_0$, and that j is the identity on $(S_2 + F_2) \cap (S_0 + F_0)$. All this is trivial.

Hence the renaming axiom holds in $\langle \mathbf{ASig}, \mathbf{AIncl} \rangle$, which is therefore an institution syntax. □

A simpler example of an institution syntax is the pair $\langle \mathbf{Set}, \mathbf{SetIncl} \rangle$.

2.3.8 Theorem. *The pair* $\langle \mathbf{Set}, \mathbf{SetIncl} \rangle$ *is an institution syntax.*

Proof. We can employ Theorem 2.3.6, because **Set** is isomorphic to the full subcategory of **ASig** whose objects are the signatures without function symbols, and **SetIncl** is isomorphic to the analogous full subcategory of **AIncl**. It is clear from the formulas (b) and (c) of Theorem 2.3.6, that these subcategories are closed under formation of **ASig**-joins and -meets. From this one easily deduces that the three axioms of the "institution syntax" definition (Def. 2.3.5) hold for the subcategories of **ASig** and **AIncl** that contain only the signatures without function symbols, and hence hold for the isomorphic categories **Set** and **SetIncl**. □

We are now ready to define the notion of an "institution".

Recall that if C is a category, then C^{op} is the opposite category, and if $\sigma\colon S \to T$ is an arrow of C, then $\sigma^{\mathrm{op}}\colon T \to S$ is the arrow of C^{op} that corresponds to σ.

2.3.9 Definition (Institution).
A *preinstitution* is a triple

$$\langle Sig, Incl, Mod \rangle,$$

where $\langle Sig, Incl \rangle$ is an institution syntax, and Mod is a functor from Sig^{op} to a category of sets.

The terminology of an institution syntax (Def. 2.3.5) carries over to a preinstitution, that is, the objects of Sig are called "signatures", the morphisms of Sig are called "signature morphisms", and the partial ordering on signatures defined by $Incl$ is called "inclusion" and written "$\sqsubseteq$". If S is a signature, an element A of $Mod(S)$ is a *model* of signature S (or just an "S-model"). If $\sigma\colon S \to T$ is

a signature morphism, let $\overline{\sigma} := Mod(\sigma^{op})$: $Mod(T) \to Mod(S)$. If $S \sqsubseteq T$ and $A \in Mod(T)$, then $A/S := \overline{(S \sqsubseteq T)}(A) \in Mod(S)$ is the *reduct* of A to S. If $S \sqsubseteq T$, $A \in Mod(S)$ and $B \in Mod(T)$ such that $B/S = A$ (*i. e.*, A is a reduct of B), then A is *included* in B (written "$A \sqsubseteq B$").

A preinstitution is an *institution*, if it has the following "completeness" property: whenever $\langle S_i \rangle_{i \in I}$ is a nonempty compatible family of signatures, and $A_i \in Mod(S_i)$ for $i \in I$, such that for all $i, j \in I$:

$$A_i/(S_i \sqcap S_j) = A_j/(S_i \sqcap S_j),$$

then there exists a unique $\bar{A} \in Mod(\bigsqcup_{i \in I} S_i)$ satisfying $\bar{A}/S_i = A_i$ (*i. e.*, $A_i \sqsubseteq \bar{A}$) for all $i \in I$. This model $\bar{A}$ will be denoted "$\bigsqcup_{i \in I} A_i$" and called the *join* of the family $\langle A_i \rangle_{i \in I}$ (The proper notation would be "$\bigsqcup_{i \in I} \langle S_i, A_i \rangle$", but the signatures S_i ($i \in I$) will always be known from the context). □

Here is the basic relation between reduct (of models) and restriction (of signature morphisms).

2.3.10 Proposition. *Let $\langle Sig, Incl, Mod \rangle$ be a preinstitution. If $S \sqsubseteq T$ and σ: $T \to U$ is a signature morphism, then for every $A \in Mod(U)$:*

$$(\overline{\sigma}\, A)/S = \overline{(\sigma/S)}\, A.$$

Proof. $(\overline{\sigma}\, A)/S = \overline{(S \sqsubseteq T)}(\overline{\sigma}\, A) = \overline{((S \sqsubseteq T)\,;\sigma)}\, A = \overline{(\sigma/S)}\, A.$ □

As explained before, a signature S can be thought of as representing a "type environment" of a programming language, that is, a set of program symbols with associated type information. A model of signature S is supposed to represent an "environment", in which semantic values are associated with the symbols of S, conforming to the type information in S. A model thus represents the semantics of a self-contained set of definitions in the programming language.

A signature morphism σ: $S \to T$ defines a "translation" map $\overline{\sigma} = Mod(\sigma^{op})$ from T-models to S-models. If σ is thought of as a map from the symbols of S to the symbols of T, the translation $\overline{\sigma}\, A$ of a T-model A could be obtained by assigning to a symbol x of S the semantic value assigned to $\sigma(x)$ in A.

The completeness property of an institution allows one to construct a model by combining models that represent the contributions of different sections of a program (*e. g.*, different modules). Variants of the completeness axiom are well known in sheaf theory, where they distinguish "sheaves" from "presheaves" [FS 79, p. 346].[1]

2.3.11 Theorem. *The triple* $\langle$**ASig, AIncl, Alg**$\rangle$ *is an institution. In particular,*

(a) *If* $\Sigma \sqsubseteq \Sigma'$, $\Sigma = \langle S, \alpha\colon F \to S^+\rangle$, *and* $A \in Alg(\Sigma')$, *then* A/Σ *is the restriction of the mapping* A *to* $S + F$.

(b) *If* $\langle \Sigma_i \rangle_{i \in I}$ *is a nonempty compatible family of small algebraic signatures, and* $A_i \in \mathbf{Alg}(\Sigma_i)$ *for* $i \in I$, *such that for all* $i, j \in I$: $A_i/(\Sigma_i \sqcap \Sigma_j) = A_j/(\Sigma_i \sqcap \Sigma_j)$, *then* $\bar{A} := \bigcup_{i \in I} A_i$ *is the unique algebra of signature* $\bigsqcup_{i \in I} \Sigma_i$ *that satisfies* $\bar{A}/\Sigma_i = A_i$ *for all* $i \in I$ *(in particular,* $\bigsqcup_{i \in I} A_i = \bigcup_{i \in I} A_i$ *for* $I \neq \emptyset$*).*

Proof. By Theorem 2.3.6, the pair $\langle$**ASig, AIncl**$\rangle$ is an institution syntax. By definition, **Alg** is a functor from $\mathbf{ASig}^{\mathrm{op}}$ to a category of sets. It is clear that clause (b) of the theorem implies the completeness property of an institution. The two clauses of the theorem will now be proved.

Clause (a): Suppose that $\Sigma = \langle S, \alpha\colon F \to S^+\rangle$ and $\Sigma' = \langle S', \alpha'\colon F' \to (S')^+\rangle$, and that $A \in \mathbf{Alg}(\Sigma')$. Then

$$\begin{aligned} A/\Sigma &= \mathbf{Alg}((\Sigma \sqsubseteq \Sigma')^{\mathrm{op}})(A) \\ &= \overline{(\Sigma \sqsubseteq \Sigma')}(A) \\ &= (\Sigma \sqsubseteq \Sigma')\,;A \\ &= \big((S + F) \subseteq (S' + F')\big)\,;A \\ &= A/(S + F). \end{aligned}$$

---

[1] The close connection between institutions and sheaves was noted and pointed out to me by John Gray

Clause (b): Let $\langle \Sigma_i \rangle_{i \in I}$ be a nonempty compatible family of small algebraic signatures, where $\Sigma_i = \langle S_i, \alpha_i \colon F_i \to S_i^+ \rangle$, and let $A_i \in \mathbf{Alg}(\Sigma_i)$ for $i \in I$, such that for all $i, j \in I$: $A_i/(\Sigma_i \sqcap \Sigma_j) = A_j/(\Sigma_i \sqcap \Sigma_j)$.

By Theorem 2.3.6 (b),

$$\bigsqcup_{i \in I} \Sigma_i = \Big\langle \bigcup_{i \in I} S_i, \bigcup_{i \in I} \alpha_i \colon \bigcup_{i \in I} F_i \to (\bigcup_{i \in I} S_i)^+ \Big\rangle.$$

Now let $\bar{A} := \bigcup_{i \in I} A_i$. This is a relation, and

$$\operatorname{dom} \bar{A} = \bigcup_{i \in I} \operatorname{dom} A_i = \bigcup_{i \in I} (S_i + F_i) = \bigcup_{i \in I} S_i \cup \bigcup_{i \in I} F_i = \bigcup_{i \in I} S_i + \bigcup_{i \in I} F_i.$$

To see that $\bar{A}$ is a function, suppose that (x, y) and (x, z) are elements of $\bar{A}$. We can pick $i, j \in I$, such that $(x, y) \in A_i$ and $(x, z) \in A_j$. Then $x \in \operatorname{dom} A_i \cap \operatorname{dom} A_j = (S_i + F_i) \cap (S_j + F_j) = (S_i \cap S_j) + (F_i \cap F_j)$. By Theorem 2.3.6 (c), this means that x is a symbol of $\Sigma_i \sqcap \Sigma_j$. Hence $y = A_i(x) = (A_i/(\Sigma_i \sqcap \Sigma_j))(x) = (A_j/(\Sigma_i \sqcap \Sigma_j))(x) = A_j(x) = z$. Thus, $\bar{A}$ is a function.

To see that $\bar{A}$ is an algebra of signature $\bigsqcup_{i \in I} \Sigma_i$, consider $f \colon s_1 \dots s_n \to r$ in $\bigsqcup_{i \in I} \Sigma_i$, i.e., $f \in \bigcup_{i \in I} F_i$ and $(\bigcup_{i \in I} \alpha_i)(f) = \langle s_1 \dots s_n, r \rangle$. Pick $i \in I$ such that $f \in F_i$. Then $\alpha_i(f) = \langle s_1 \dots s_n, r \rangle$, $A_i(f) = \bar{A}(f)$, $A_i(s_k) = \bar{A}(s_k)$ for $k \in \{1, \dots, n\}$, and $A_i(r) = \bar{A}(r)$. Because A_i is an algebra,

$$A_i(f) \colon \Big(\prod_{k \in \{1, \dots, n\}} A_i(s_k) \Big) \mathrel{+\!\!\to} A_i(r),$$

and this is equivalent to

$$\bar{A}(f) \colon \Big(\prod_{k \in \{1, \dots, n\}} \bar{A}(s_k) \Big) \mathrel{+\!\!\to} \bar{A}(r).$$

Thus, $\bar{A}$ is an algebra of signature $\bigsqcup_{i \in I} \Sigma_i$. Using clause (a), one immediately obtains that $\bar{A}/\Sigma_i = A_i$ for all $i \in I$.

To see that $\bar{A}$ is the only such algebra, let $B \in \mathbf{Alg}(\bigsqcup_{i \in I} \Sigma_i)$ be such that $B/\Sigma_i = A_i$ for all $i \in I$.

Both $\bar{A}$ and B are mappings with domain $(\bigcup_{i \in I} S_i) + (\bigcup_{i \in I} F_i)$. Consider an element x of this set. We can pick $i \in I$ such that $x \in S_i + F_i$. Hence

$$\bar{A}(x) = \Big(\big((S_i + F_i) \subseteq (\bigcup_{i \in I} S_i + \bigcup_{i \in I} F_i) \big) \,;\, \bar{A} \Big)(x)$$

$$
\begin{aligned}
&= (\bar{A}/\Sigma_i)(x) \\
&= A_i(x) \\
&= (B/\Sigma_i)(x) \\
&= \Big(((S_i + F_i) \subseteq (\bigcup_{i\in I} S_i + \bigcup_{i\in I} F_i))\,;B\Big)(x) \\
&= B(x).
\end{aligned}
$$

This means that $\bar{A} = B$, hence $\bar{A}$ is unique. □

A simpler example of an institution is obtained from the institution syntax $\langle \mathbf{Set}, \mathbf{SetIncl}\rangle$ by defining the models of a signature (*i. e.*, a set) S to be the small functions with domain S, *i. e.*, the maps from S to the universe $\mathbf{U}$.

2.3.12 Definition. Let $\mathbf{SetMod}\colon \mathbf{Set}^{\mathrm{op}} \to \mathbf{Cls}$ be the functor that maps a set S to the class $(S \to \mathbf{U})$ of mappings from S to $\mathbf{U}$, and $f\colon S \to T$ in $\mathbf{Set}$ to the map $\mathbf{SetMod}(f^{\mathrm{op}})\colon (T \to \mathbf{U}) \to (S \to \mathbf{U})$ defined by $k \mapsto f\,;k$. □

2.3.13 Theorem. *The triple* $\langle \mathbf{Set}, \mathbf{SetIncl}, \mathbf{SetMod}\rangle$ *is an institution.*

Proof. The proof of the theorem that $\langle \mathbf{Set}, \mathbf{SetIncl}\rangle$ is an institution syntax (Theorem 2.3.8) was based on the isomorphism between the category of small sets and the category of small algebraic signatures without function symbols. This isomorphism commutes with the two model functors, that is, the set $(S \to \mathbf{U})$ of small functions on a set S is identical to the set of small algebras of signature $\langle S, \emptyset\rangle$, and the model map of a map $f\colon S \to T$ in $\mathbf{Set}$ is the same as the model map of the corresponding signature morphism $f\colon \langle S, \emptyset\rangle \to \langle T, \emptyset\rangle$ in $\mathbf{ASig}$.

From this observation and the fact that $\langle \mathbf{ASig}, \mathbf{AIncl}, \mathbf{Alg}\rangle$ is an institution, it easily follows that $\langle \mathbf{Set}, \mathbf{SetIncl}, \mathbf{SetMod}\rangle$ also is an institution. □

Chapter 3

Modular Systems

THIS CHAPTER presents a theory of modular programming that is based on the concept of a "cell". Both module specifications and program modules are cells in the theory, and cells therefore act in the rôle of specifications as well as in the rôle of the objects specified.

The fundamental relation between cells in modular programming is that of "refinement", which generalizes the "satisfaction" relation between a module and its specification to cells.

In Section 3.2 the "decomposition" of a cell into a system of cells is described; this concept expresses how the module specifications of a modular system must fit together.

Section 3.3 defines the "composition" of a system of cells, which reflects the way program modules would be composed by a compiler.

In Section 3.4 it is shown that the concepts presented so far form a sound discipline for modular programming: decomposing a cell into a cell system, refining the cells of that system, and composing the refined cells yields a refinement of the original cell. This theorem generalizes previous theorems by myself [Schoett 81] and by Back and Mannila [BM 84].

The final section, Section 3.5, analyses the relation between decomposition and composition further. In particular, it is shown that, apart from some syntactic restrictions, decomposition is the most general design criterion for the specifications of a modular system that makes the final, composed cell a refinement of its specification.

The mathematical theory of this chapter is developed exclusively on the abstract level of an institution (except, of course, for the examples). Since all definitions and theorems refer to a single, but arbitrary, institution, the following convention is adopted.

Convention. Throughout this chapter, the triple $\langle Sig, Incl, Mod \rangle$ is assumed to be an institution. The concepts that depend on an institution (such as "signature", "inclusion", or "model") are implicitly assumed to refer to the institution $\langle Sig, Incl, Mod \rangle$. □

3.1 Cells and Refinement

This section introduces the "cell" concept and the "refinement" relation between cells.

3.1.1 Definition. A *cell signature* is a pair of compatible signatures.

In a cell signature $\langle E, D \rangle$, call E the *environment signature*, and call D the *definition signature* ("signature of defined entities"). □

The meaning of a cell signature $\langle E, D \rangle$ is that a cell of that signature will assume the program entities of E to be present, and will contribute entities so that the entities of D will be present as well. That is, the cell will define those entities of D that do not occur already in E.

More formally, this is described in the following definition: A "site" for $\langle E, D \rangle$ is a signature onto which a cell of signature $\langle E, D \rangle$ can be fitted, and the "result signature" of $\langle E, D \rangle$ on a site is that site enriched by the contribution of the cell.

3.1.2 Definition. Let $\langle E, D \rangle$ be a cell signature. A *site* for $\langle E, D \rangle$ is a signature F, compatible with $E \sqcup D$, such that

$$F \sqcap (E \sqcup D) = E.$$

If F is a site for $\langle E, D \rangle$, the signature $F \sqcup D$ is called the *result signature* of $\langle E, D \rangle$ on F. □

The following proposition gives an alternative characterization of the "site" notion.

3.1.3 Proposition. *Let $\langle E, D\rangle$ be a cell signature, and let F be a signature. Then F is a site for $\langle E, D\rangle$, if and only if*

$$F \sqsupseteq E, \qquad F \sim D, \quad \text{and} \quad F \sqcap D \sqsubseteq E.$$

Proof. If F is a site for $\langle E, D\rangle$, then $F \sim (E \sqcup D)$ and $F \sqcap (E \sqcup D) = E$. From this the three clauses above follow trivially.

Conversely, suppose that $\langle E, D\rangle$ is a cell signature, and F a signature such that $F \sqsupseteq E$, $F \sim D$, and $F \sqcap D \sqsubseteq E$. Then $E \sqcup D \sqsubseteq F \sqcup D$, hence $E \sqcup D \sim F$, and

$$F \sqcap (E \sqcup D) = (F \sqcap E) \sqcup (F \sqcap D) = E \sqcup (F \sqcap D) = E. \qquad \square$$

In this proposition it can be seen that a site F for a cell signature $\langle E, D\rangle$ must satisfy three conditions:

- F must contain all the program entities required by a cell of signature $\langle E, D\rangle$ $(F \sqsupseteq E)$,
- F must be syntactically compatible with the entities newly contributed by the cell $(F \sim D)$,
- F must not already contain any of the entities to be contributed by the cell $(F \sqcap D \sqsubseteq E)$.

By virtue of these conditions, a cell of signature $\langle E, D\rangle$ can enrich the signature F without conflicts—think of F as representing a self-contained set of declarations in a programming language, and of $\langle E, D\rangle$ as representing a set of (not necessarily self-contained) declarations that may follow F on the same level of block structure, where the programming language forbids multiple declarations of a symbol on the same level of block structure.

In the program development of Section 1.4, cell signatures (in the institution **⟨ASig, AIncl, Alg⟩**) have occurred in connection with modules. Each module given there has an **environment signature** and a set of **defined symbols**. The cell signature that characterizes the module syntactically has the **environment signature** as environment signature, and its definition signature is obtained by adding the **defined symbols** to the environment signature.

3.1.4 Example. The module M_{DICT} of Figure 1-7 was described syntactically as follows:

environment signature

listitem, store: **sort**

input: *listitem* → *store*

output: *store* → *listitem*

defined symbols

dictionary: *listitem* → *listitem*.

This description defines the cell signature $\langle E_{DICT}, D_{DICT}\rangle$, where E_{DICT} is as given under **environment signature** above, and D_{DICT} is E_{DICT} enriched by

dictionary: *listitem* → *listitem*. □

Another cell signature connected with a module is obtained from the interfaces that specify the module. If we define E to be the join of the signatures of the interfaces on which the module depends, and D to be the join of the signatures of the interfaces to be satisfied by the module, we obtain a cell signature $\langle E, D\rangle$, which may be called the cell signature of the "specification" of the module.

3.1.5 Example. The module M_{DICT} depends on the interface I_{INOUT} (Figure 1-9), and its result is specified by the interface I_{DICT} (Figure 1-5). The cell signature of the "specification" of M_{DICT} therefore is

$$\langle \Sigma_{INOUT}, \Sigma_{DICT}\rangle,$$

(Σ_{DICT} is given in Example 2.2.2, Σ_{INOUT} is the signature of I_{INOUT}, shown in Figure 1-9), or explicitly:

environment signature

bool, *item*, *listitem*, *store*: **sort**

leitem: *item item* $\rightarrow$ *bool*

input: *listitem* $\rightarrow$ *store*

output: *store* $\rightarrow$ *listitem*

definition signature

bool, *item*, *listitem*: **sort**

leitem: *item item* $\rightarrow$ *bool*

dictionary: *listitem* $\rightarrow$ *listitem*.

In this cell signature, the definition signature does not fully include the environment signature as in the previous example. The set of "defined entities" of this cell signature, that is, the set of entities occurring in the **definition signature**, but not in the **environment signature**, contains only the symbol *dictionary*, and is the same as in the previous example. □

We now turn to the semantic aspects of the "cell" concept. A cell will consist of two "interfaces", in the sense of the following definition.

3.1.6 Definition. An *interface* (also called a "specification") is a pair

$$\langle S, P \rangle,$$

where S is a signature and $P \subseteq Mod(S)$.

By abuse of language, we will usually let "P" denote the interface $\langle S, P \rangle$, call P an "interface of signature S" or just "S-interface", and call S the "signature of P", written "Sig(P)". □

All the interfaces that occurred in the *dictionary* program development fit this definition (for the institution $\langle$**ASig**, **AIncl**, **Alg**$\rangle$). We used the informal nota-

tion scheme

interface
signature
⋮
properties
⋮ .

In this scheme, the **signature** part gives the signature of the interface, the **properties** part gives a predicate that characterizes the model set of the interface.

The idea of using interfaces as defined here in place of specifications in some specification language was adopted from Lipeck [Lipeck 83, p. 15 f.], who calls the analogous concept a "class" („*Klasse*"). Since every possible specification language must determine which models satisfy a specification, every specification of models of a certain signature in every language gives rise to an interface (Lipeck's "classes" are slightly more restrictive, as they must be closed under model isomorphisms).

Conversely, an interface may be regarded as a "sentence" of a very general (in fact, *the* most general) specification language. For example, one obtains an "institution" in the sense of Goguen and Burstall [GB 84, p. 229] (in the simplified version, where $Mod(S)$ is a set rather than a category) by defining the set of "sentences" of a signature to be the set of interfaces, "satisfaction" as the element relation ($\in$), and the "translation" of a sentence $\langle S, P\rangle$ along a signature morphism $\sigma\colon S \to T$ by means of the "satisfaction condition" of Goguen and Burstall [GB 84, p. 229]:

$$\sigma\langle S, P\rangle := \langle T, \{\, A \in Mod(T) \mid Mod(\sigma^{\mathrm{OP}})(A) \in P \,\}\rangle.$$

The following definition treats interfaces as if they were sentences. Note, however, that the "projection" operation is not a special case of "translation" as described above, but rather analogous to the derive operation of ASL [ST 85, p. 15].

3.1.7 Definition (Projection and conjunction of interfaces).
Let P be an interface of signature S, and let $T \sqsubseteq S$. The *projection* of P onto T, written "$P/\!\!/T$", is the T-interface defined by the set of T-reducts of the elements of P:

$$P/\!\!/T := Mod((T \sqsubseteq S)^{\mathrm{op}})(\!|P|\!) = \{\, A/T \mid A \in P \,\}.$$

Let $\langle P_i \rangle_{i \in I}$ be a family of interfaces with compatible signature family $\langle S_i \rangle_{i \in I}$. The *conjunction* $\bigwedge_{i \in I} P_i$ of the interfaces P_i ($i \in I$) is the interface of signature $\bigsqcup_{i \in I} S_i$ defined by

$$\bigwedge_{i \in I} P_i := \{\, A \in Mod(\bigsqcup_{i \in I} S_i) \mid \forall i \in I\colon A/S_i \in P_i \,\}.$$

The conjunction of two interfaces P and Q is written "$P \wedge Q$". □

Note that in this definition, language is heavily "abused" in the sense of Definition 3.1.6: projections and conjunctions of interfaces depend on their signatures, although the notations "$P/\!\!/T$" and "$\bigwedge_{i \in I} P_i$" do not explicitly mention them (the proper notations "$\langle S, P \rangle/\!\!/T$" and "$\bigwedge_{i \in I} \langle S_i, P_i \rangle$" would be rather clumsy).

The following propositions state some basic monotonicity properties of the conjunction operation.

3.1.8 Proposition (*"the more components in a conjunction, the less models"*).
Let $\langle P_i \rangle_{i \in I}$ be a family of interfaces with compatible signature family $\langle S_i \rangle_{i \in I}$, let $J \subseteq I$, and let $T \sqsubseteq \bigsqcup_{i \in J} S_i$. Then

$$(\bigwedge_{i \in J} P_i)/\!\!/T \supseteq (\bigwedge_{i \in I} P_i)/\!\!/T.$$

Proof. Consider $A \in (\bigwedge_{i \in I} P_i)/\!\!/T$. By definition, there exists $B \in \bigwedge_{i \in I} P_i$, such that $B/T = A$. But then $B/\bigsqcup_{i \in J} S_i \in \bigwedge_{i \in J} P_i$, because for each $i \in J$: $(B/\bigsqcup_{i \in J} S_i)/J = B/J \in P_j$. Hence

$$A = B/T = (B/\bigsqcup_{i \in J} S_i)/T \in (\bigwedge_{i \in J} P_i)/\!\!/T.$$ □

3.1.9 Proposition *("Conjunction is monotonic in its arguments")*
Let $\langle P_i\rangle_{i\in I}$ *and* $\langle P_i'\rangle_{i\in I}$ *be interface families with signature families* $\langle S_i\rangle_{i\in I}$ *and* $\langle S_i'\rangle_{i\in I}$, *such that* $\langle S_i'\rangle_{i\in I}$ *is compatible and* $S_i \sqsubseteq S_i'$ *for all* $i \in I$, *and let* $T \sqsubseteq \bigsqcup_{i\in I} S_i$. *If* $P_i'/\!\!/S_i \subseteq P_i$ *for all* $i \in I$, *then*

$$(\bigwedge_{i\in I} P_i')/\!\!/T \subseteq (\bigwedge_{i\in I} P_i)/\!\!/T.$$

Proof. Assume that $P_i'/\!\!/S_i \subseteq P_i$ for all $i \in I$. If $A \in (\bigwedge_{i\in I} P_i')/\!\!/T$, then there exists $B \in \bigwedge_{i\in I} P_i'$, such that $B/T = A$. But $B/\bigsqcup_{i\in I} S_i \in \bigwedge_{i\in I} P_i$, because for each $i \in I$:

$$(B/\bigsqcup_{i\in I} S_i)/S_i = B/S_i = (B/S_i')/S_i \in P_i'/\!\!/S_i \subseteq P_i,$$

and hence

$$A = B/T = (B/\bigsqcup_{i\in I} S_i)/T \in (\bigwedge_{i\in I} P_i)/\!\!/T. \qquad \square$$

Note that the "converse" of this proposition is false in general: If

$$\forall i \in I\colon\ P_i \subseteq P_i'/\!\!/S_i,$$

it does not follow that

$$(\bigwedge_{i\in I} P_i)/\!\!/T \subseteq (\bigwedge_{i\in I} P_i')/\!\!/T.$$

Here is a counterexample in the institution $\langle$**Set**, **SetIncl**, **SetMod**$\rangle$: Let

$$I = \{1,2\}, \quad S_1 = S_2 = \emptyset, \quad P_1 = P_2 = \{\emptyset\},$$
$$S_1' = S_2' = \{x\}, \quad P_1' = \{\{(x,1)\}\}, \quad P_2' = \{\{(x,2)\}\},$$

so that P_1 and P_2 both specify the unique $\emptyset$-model $\emptyset$, P_1' prescribes that the interpretation of x (an arbitrary symbol) be 1, and P_2' prescribes that it be 2.

Then

$$(P_1 \wedge P_2)/\emptyset = \{\emptyset\}/\emptyset = \{\emptyset\},$$

but

$$(P_1' \wedge P_2')/\emptyset = \emptyset/\emptyset = \emptyset.$$

The specification $P_1' \wedge P_2'$ is empty, because P_1' and P_2' prescribe different interpretations for x. Yet the projections of P_1' and P_2' to the empty signature

(that is, P_1 and P_2) do not exhibit this conflict any more and have a nonempty intersection.

We are now ready to define "cells".

3.1.10 Definition (Cell).
A *cell* is a pair of interfaces whose signatures are compatible. In a cell $\langle Q, R\rangle$, call Q the *requirement interface* and R the *result interface*. The cell signature $\langle E, D\rangle := \langle \mathrm{Sig}(Q), \mathrm{Sig}(R)\rangle$ is the *signature* of $\langle Q, R\rangle$, also written "$\mathrm{Sig}\langle Q, R\rangle$", and $\langle Q, R\rangle$ is called a "cell of signature $\langle E, D\rangle$" or an "$\langle E, D\rangle$-cell". □

The intended interpretation of a cell $\langle Q, R\rangle$ is that it can be applied in contexts that contain program entities satisfying the requirement interface Q, and that it will then contribute program entities such that the result interface R also is satisfied. This is captured in the following definition.

3.1.11 Definition (Base and result).
Let $\langle Q, R\rangle$ be a cell of signature $\langle E, D\rangle$. A *base* for $\langle Q, R\rangle$ is a model A of signature F, such that

$$F \text{ is a site for } \langle E, D\rangle, \quad \text{and} \quad A/E \in Q.$$

A *result* of $\langle Q, R\rangle$ on the base A is a model B of signature $F \sqcup D$ that satisfies

$$B/F = A \quad \text{and} \quad B/D \in R$$

(more concisely: $B \in \{A\} \wedge R$). □

In general, a cell may have any number of results (including zero) on any base. The following definition classifies cells according to the number of results they have.

3.1.12 Definition. A cell is

consistent, if for every base it has at least one result,

single-valued, if for every base it has at most one result,

a *module*, if for every base it has exactly one result

(*i. e.*, if it is both consistent and single-valued). □

This definition employs quantification over all bases of a cell, that is, over models of possibly many different signatures. A simpler characterization is given by the following proposition, whose criteria involve only models of the environment and result signature of a cell.

3.1.13 Proposition. *Let $\langle Q, R\rangle$ be a cell of signature $\langle E, D\rangle$, and let $\_/E$: $Q \wedge R \to Q$ be the reduct function. Then*

$$\begin{aligned}
\langle Q,R\rangle \text{ is consistent} &\iff Q \subseteq (Mod(E) \wedge R)/\!\!/E \\
&\iff _/E \text{ is surjective,} \\
\langle Q,R\rangle \text{ is single-valued} &\iff _/E \text{ is injective,} \\
\langle Q,R\rangle \text{ is a module} &\iff _/E \text{ is bijective.}
\end{aligned}$$

Proof.

First Line: Suppose $\langle Q, R\rangle$ is consistent. Consider $A \in Q$. Since A is a base for $\langle Q, R\rangle$, there exists a result B of $\langle Q, R\rangle$ on A. Clearly, $B \in Mod(E) \wedge R$. Hence

$$A = B/E \in (Mod(E) \wedge R)/\!\!/E.$$

It follows that $Q \subseteq (Mod(E) \wedge R)/\!\!/E$.

Conversely, suppose that $Q \subseteq (Mod(E) \wedge R)/\!\!/E$. Let A be a base for $\langle Q, R\rangle$, and let F be the signature of A. Since $A/E \in Q$, there exists $B \in (Mod(E) \wedge R)$, such that $B/E = A/E$. The intersection of the signatures of B and of A is E, because F is a site for $\langle E, D\rangle$. Hence by the completeness property of an institution there exists $C \in Mod(F \sqcup D)$, such that $C/F = A$ and $C/(E \sqcup D) = B$. But then C is a result of $\langle Q, R\rangle$ on A, because $C/F = A$ and $C/D = B/D \in R$. As A was an arbitrary base for $\langle Q, R\rangle$, the cell is consistent.

Second Line: Assume again that $\langle Q, R\rangle$ is consistent. Let $A \in Q$. Since A is a base for $\langle Q, R\rangle$, we can pick a result B of $\langle Q, R\rangle$ on A. Of course, $B \in Q \wedge R$. Since $B/E = A$, we have $A \in \mathrm{ran}(\_/E)$. Since A was an arbitrary member of Q, $\_/E$ is surjective.

Conversely, if $\_/E$ is surjective, then

$$(Mod(E) \wedge R)/\!\!/E \supseteq (Q \wedge R)/\!\!/E = Q.$$

By the first line of the proposition, $\langle Q, R\rangle$ is consistent.

Third Line: Assume that $\langle Q, R\rangle$ is single-valued. Let A and B be elements of $Q \wedge R$ such that $A/E = B/E$. Then A and B are results of $\langle Q, R\rangle$ on the base A/E, and the single-valuedness of $\langle Q, R\rangle$ implies that $A = B$. Hence $\_/E\colon Q \wedge R \to Q$ is injective.

Now assume that $\_/E$ is injective. Let A and B be two results of $\langle Q, R\rangle$ on a base C of signature F. Because

$$A/E = (A/F)/E = C/E = (B/F)/E = B/E,$$

and because $A/(E \sqcup D)$ and $B/(E \sqcup D)$ are elements of $Q \wedge R$, the injectivity of $\_/E$ implies that $A/(E \sqcup D) = B/(E \sqcup D)$; in particular, $A/D = B/D$. Since A and B are of signature $F \sqcup D$, and $A/F = C = B/F$ also, the completeness property of an institution (uniqueness of joins) implies that $A = B$. Since A and B were arbitrary, $\langle Q, R\rangle$ is single-valued.

Fourth Line: This follows trivially from the previous two lines. □

We have encountered cells in the institution $\langle$**ASig, AIncl, Alg**$\rangle$ in the *dictionary* program development. Each **module** given there (M_{DICT}: Fig. 1-7, M_{INOUT}: Fig. 1-14, M_{INPUT}: Fig. 1-16, M_{OUTPUT}: Fig. 1-18, M_{STORE}: Fig. 1-19) was presented using the following scheme:

module
  environment signature
  ⋮

defined symbols
⋮
requirement
⋮
result
⋮

In this scheme, the sections **environment signature** and **defined symbols** describe a cell signature $\langle E, D \rangle$, as explained in Example 3.1.4. The **requirement** section states properties of an algebra of the **environment signature**, and hence defines an interface Q of signature E; the **result** section states properties of an algebra of signature D (*i. e.*, the combination of **environment signature** and **defined symbols**), which define an interface R of signature D. The cell defined by such a scheme is $\langle Q, R \rangle$.

The following example illustrates how a concrete program module, expressed in some programming notation, is rendered as a cell in the theory.

3.1.14 Example. In the *dictionary* program development of Section 1.4, the following definition of the *input* operation was given:

$$\begin{aligned} \mathit{input}(l) = \text{if } \mathit{isnil}(l) \text{ then } & \mathit{empty}() \\ \text{else } & \mathit{insert}(\mathit{hd}\, l, \mathit{input}(\mathit{tl}\, l)) \end{aligned}$$

Except for minor syntactic details, this is a recursive definition of the function

$$\mathit{input}\colon \mathit{listitem} \to \mathit{store}$$

in a functional programming notation such as, for example, the functional subsets of ALGOL 68 [van Wijngaarden *et al.* 76], ML [HMM 86] or the "Algorithmic Language" („*Algorithmische Sprache*") of the Munich CIP Project [CIP 85]. In the development of Section 1.4, this code forms the body of a program module (in general, a program module could contain any number of type and operation definitions).

First, we discuss how the cell signature of this module is obtained. The environment signature must contain all the program entities that are used in the

code. This includes all the source and target sorts of the operations used. The operations used are *isnil*, *empty*, *insert*, *hd*, and *tl*. The type of these operations is obtained from the context. The environment signature that results is:

environment signature

bool, *item*, *listitem*, *store*: **sort**

isnil: *listitem* → *bool*

hd: *listitem* → *item*

tl: *listitem* → *listitem*

empty: → *store*

insert: *item store* → *store*.

This is just the environment signature given in Figure 1-16.

The result signature is obtained from the environment signature by adding the program entities defined by the code, here *input*. As it is unnecessary to list the symbols of the environment signature again, this can simply be recorded as follows:

defined symbols

input: *listitem* → *store*.

We now turn to the semantics of the cell. First, what are its requirements? One might think of naming here the assumptions made in the correctness proof of the module, that is, the assumptions recorded in the interfaces $I_{LISTITEM}$ (Figure 1-4) and I_{INSERT} (Figure 1-15). But these interfaces are not part of the code itself, and so the resulting cell would no longer be a direct representation of the program code (the proper rôle of the interfaces $I_{LISTITEM}$ and I_{INSERT} with respect to the code is explained in Example 3.1.15 below).

The requirements of the code itself are the semantic prerequisites that are necessary to ensure that the code is correct and has a valid semantics in the programming notation used. In the example, there are no such requirements concerning the types *item*, *listitem*, and *store*, and the operations *isnil*, *hd*, *tl*, *empty*, and *insert*, because the code is valid whatever the interpretation of these symbols is (assuming these interpretations conform to the type information in the signature).

However, the type *bool* cannot have an arbitrary interpretation, because it must be compatible with the **if** construct used in the code. The **if** construct is regarded here as a fixed part of the language rather than as an arbitrary function, because a function corresponding to the **if** construct would have to have arguments of "higher order" types ([Schoett 81, p. 50 f.], [BW 83, p. 145–149]).

This special status of **if** also gives a special status to the type *bool*: Because the **if** construct is a fixed part of the programming notation, the type *bool* also must have a fixed definition, to enable the **if** construct to interpret values of type *bool* (in practical programming notations this restriction is reflected in the fact that even if it is possible to redefine the type *bool*, the new type of that name cannot become acceptable in the first argument position of **if**).

Assuming that the predefined type *bool* has the value set $\{\mathbf{T}, \mathbf{F}\}$, we can record the restriction on *bool* as follows.

requirement

$$bool = \{\mathbf{T}, \mathbf{F}\}.$$

Finally, consider the result interface of the cell. This is to be a predicate that characterizes the result defined by the code for all interpretations of the environment symbols that satisfy the requirements.

In the presentation of the module M_{INPUT} in Figure 1-16, we simply wrote down the code for the *input* operation to characterize the result:

result

$$input(l) = \textbf{if } isnil(l) \textbf{ then } empty() \textbf{ else } isnil(hd\ l, input(tl\ l)).$$

This code can be read as the following predicate:

> "The interpretation of *input* is the function that the code defines according to the semantics of the programming notation on the basis of the interpretation of the environment symbols."

With the conventional semantics of recursive definitions, this means that the interpretation of *input* must be the least fixpoint of the functional equation

$$input = \lambda l.\ \text{if } isnil(l) \text{ then } empty()$$
$$\text{else } insert(hd\, l, input(tl\, l)),$$

where partial functions are partially ordered by the inclusion relation between their graphs.

We have reviewed the meaning of the four sections **environmentsignature, definedsymbols, requirements,** and **result** of the description of the cell M_{INPUT} given in Figure 1-16. □

It can be seen in this example that cells which represent concrete program modules will usually be consistent and single-valued, that is, "modules" in the sense of Definition 3.1.12:

Given a concrete module, as code in some programming notation, the requirement interface of the cell representing it records the conditions necessary for the code to be a well-formed definition in the programming notation. Hence, the code defines a result for every environment that satisfies the requirement, and the associated cell is therefore consistent.

Also, a definition in a programming notation normally has unique semantics. Hence, for every environment satisfying the requirements of the code, the code defines a unique result, and hence there is a unique extension of the environment that satisfies the result interface of the associated cell. The cell therefore is a "module" as defined above.

In a modular program development, a second cell can be associated with a module. This cell, which might be called the "specification" of the module, is formed from the interfaces of the modular design to which the module is related. The interfaces that specify the module's result form the result interface of the specification cell, the interfaces that specify the properties of the module's environment on which the module may rely form the requirement interface of the specification cell.

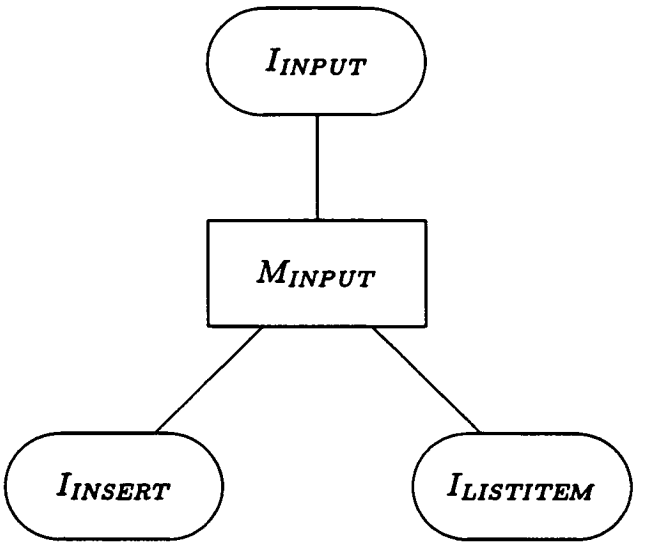


Figure 3-1: The module M_{INPUT} and the interfaces specifying it

3.1.15 Example. In the *dictionary* program development of Section 1.4, the module M_{INPUT} is specified by the interfaces I_{INPUT}, I_{INSERT}, and $I_{LISTITEM}$, as shown in Figure 3-1. The "specification" of M_{INPUT}, obtained by combining the three interfaces, is the following cell $\mathcal{M}_{INPUT}$:

$$\mathcal{M}_{INPUT} = \textbf{cell}$$
$$\textbf{requirement}$$
$$I_{INSERT} \wedge I_{LISTITEM}$$
$$\textbf{result}$$
$$I_{INPUT}$$

The cell $\mathcal{M}_{INPUT}$ represents the interfaces via which M_{INPUT} is related to the other modules of the system. For the correctness of the modular design, only $\mathcal{M}_{INPUT}$ is relevant, not M_{INPUT}; the only information about M_{INPUT} that is distributed across the system is that contained in $\mathcal{M}_{INPUT}$. We might say that $\mathcal{M}_{INPUT}$ "encapsulates" M_{INPUT} and presents an abstract view of it to the rest of the system, as illustrated in Figure 3-2.

On the other hand, this "abstract" view of M_{INPUT} is only relevant in the original modular design, it is irrelevant for the semantics of the final program.

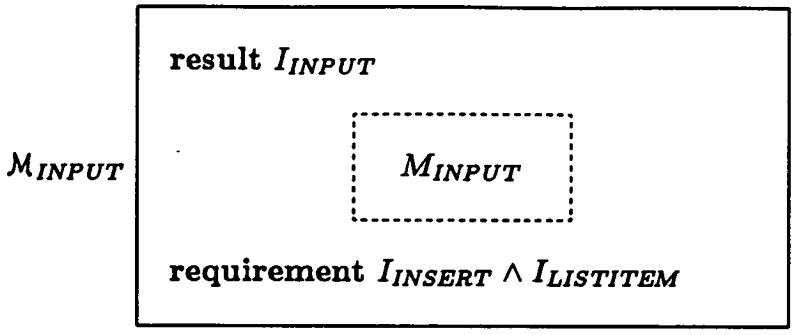


Figure 3-2: $\mathcal{M}_{INPUT}$ viewed as "encapsulating" M_{INPUT}

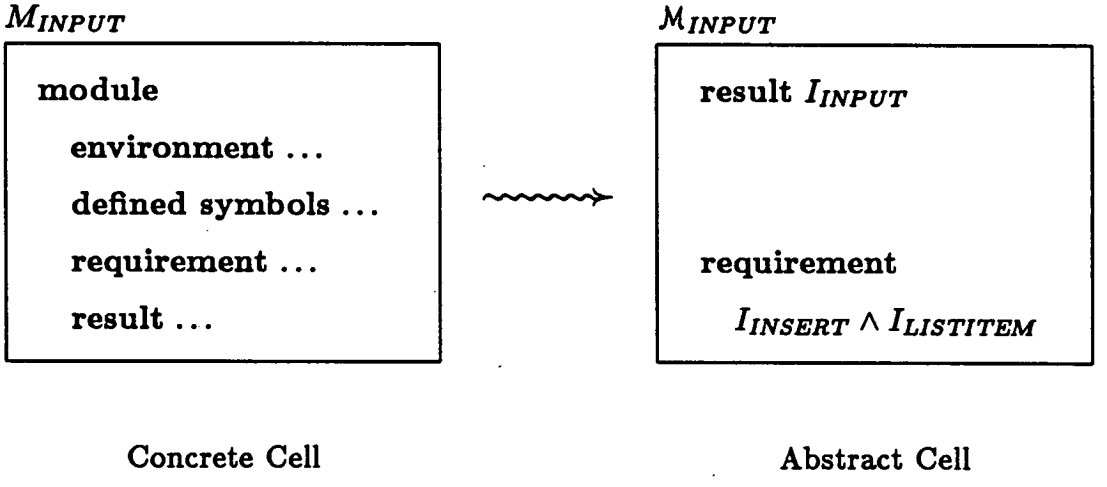


Figure 3-3: Concrete and abstract cell side by side

Only M_{INPUT}, not $\mathcal{M}_{INPUT}$ contributes to this semantics. In general, therefore, we shall show the "concrete" cell M_{INPUT} and the "abstract" cell $\mathcal{M}_{INPUT}$ side by side, as in Figure 3-3.

The arrow "$\rightsquigarrow$" in the figure indicates that a certain relation must hold between the concrete and the abstract cell—the concrete cell must be "correct" with respect to the abstract cell, its specification. The correctness notion for modular programming, called "refinement", will be defined later in this section (Definition 3.1.18). Data abstraction is based on a different notion of correctness, called "implementation". This relation will be studied in chapters 4 and 5 to follow.

If we form the specification cells for all the five modules of the *dictionary* system (Figure 1-20), we obtain the view of the modular design shown in Figure 3-4. This figure shows the five cells M_{DICT}, M_{INOUT}, M_{INPUT}, M_{OUTPUT}, and M_{STORE}, with the dependence relation derived from the design graph of Figure 1-20, surrounded by yet another cell $\bar{M}$. This cell is composed of the interfaces via which the system as a whole is related to the outside; this cell may be called the "global specification" of the system. Formally, the relation between the global specification and the individual module specifications is given by the "decomposition" notion (Definition 3.2.10 below).

A figure analogous to Figure 3-4 could be drawn showing the five concrete modules M_{DICT}, M_{INOUT}, M_{INPUT}, M_{OUTPUT}, and M_{STORE}.

However, while the specifications of Figure 3-4 are related to a global specification by the "decomposition" relation, the rôle of the concrete modules is different: they are to be composed in a constructive way to yield a cell as result that represents the combined semantics of the modules. This "composition" operation, which is analogous to the composition of modules performed by a compiler, will be defined in Definition 3.3.6 below.

The relationship between decomposition and composition, between systems of specifications and their concretizations, is illustrated in Figure 3-5. All the entities displayed in this figure are cells. On the right, a family of specifications is shown that decomposes the global specification; on the left, a family of cells whose composition is the "composed cell". The wavy arrows at the bottom indicate that each "concrete" cell must be correct with respect to its specification.

The fundamental question in this situation is: assuming the relationships hold as shown in the figure, is the composed cell correct with respect to the global specification?

If we let "correctness" mean "refinement", a positive answer to the question can be regarded as asserting the correctness of modular programming as a programming discipline. In different theories, positive answers have previously been given by Schoett [Schoett 81, Thm. 4.2.6] and by Back and Mannila [BM 84], and in the present theory, the affirmative answer will be given by the theorem

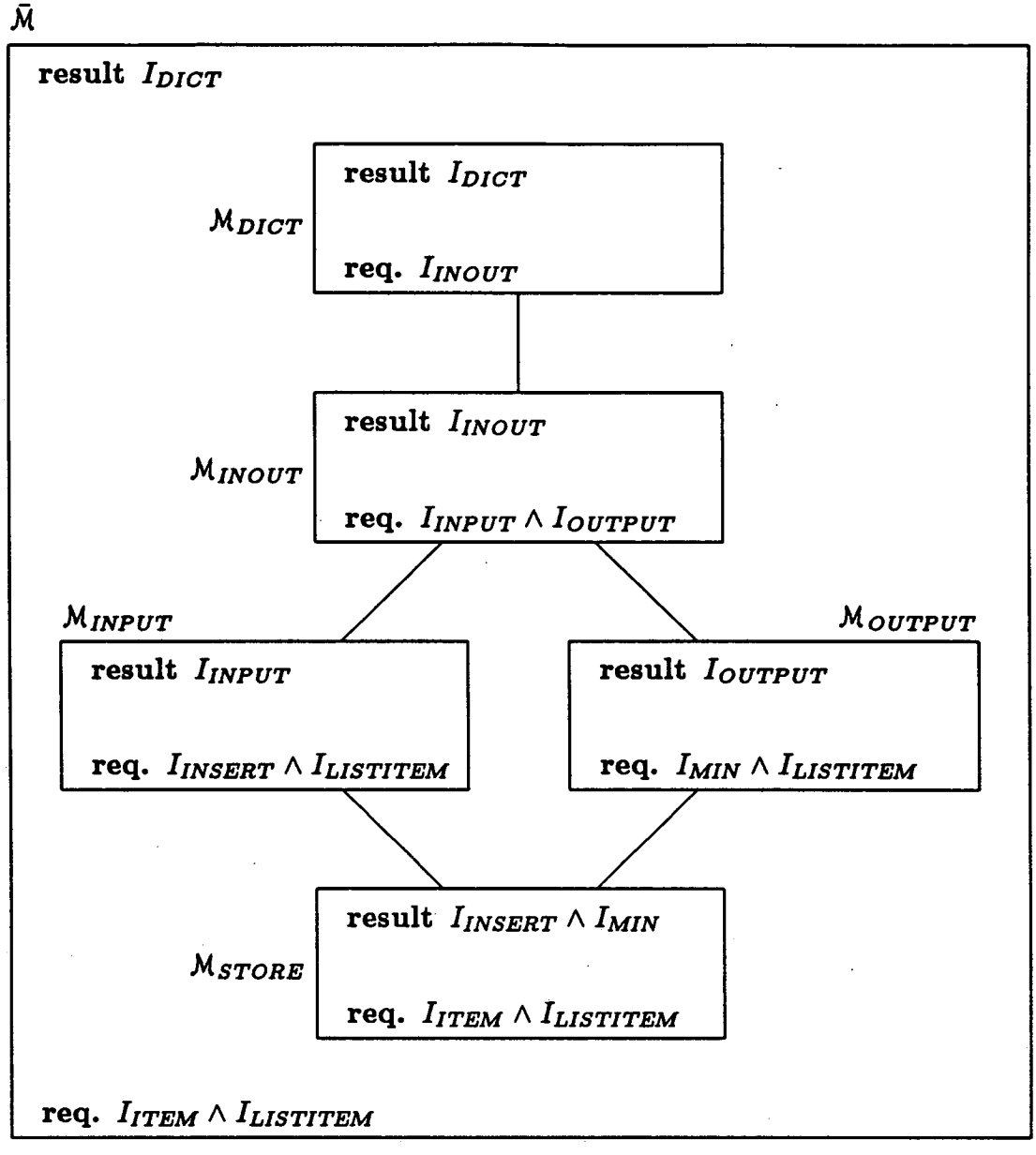


Figure 3-4: The specification cells of the *dictionary* system

asserting the "composability of refinements", which is discussed in Section 3.4 below (and proved afterwards in Section 4.1).

Section 4.1 presents a composability theorem for a different correctness concept: "universal implementation". Chapters 4 and 5 will show that this asserts the correctness of modular programming with data abstraction.

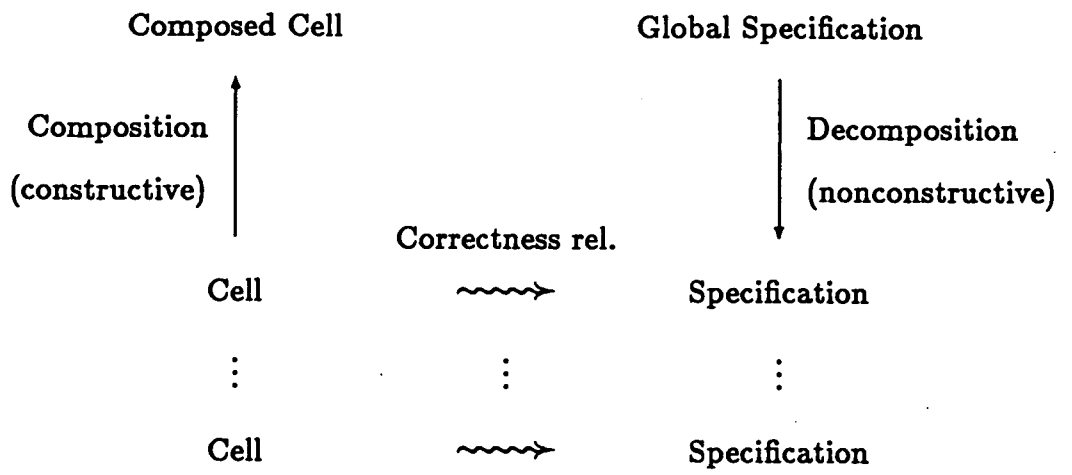


Figure 3-5: The structured correctness argument for a modular system

It is important to realize that while Figure 3-5 shows the *logical* relations between the elements of a modular programming project, it does not necessarily indicate a *temporal* sequence of design steps starting at the top right and proceeding clockwise around the figure (this was the view I advocated in [Schoett 81, p. 138 f.]). For example, the global specification is not necessarily complete at the beginning, because the requirement interface might become enlarged by requirements of the individual module specifications during the design process. Similarly, each individual module specification might contain requirements that were found only during the design of the concrete code for that module. Also, during the design of the code for some modules, the need for other modules might become apparent. All these points are illustrated by the *dictionary* program development of Section 1.4.

For practical purposes it might therefore be better to view modular program design as the stepwise construction of a design graph like the one in Figure 1-20. The theory of this thesis, however, will deal with cells and their relationships as shown in Figure 3-5. The present discussion has illustrated how design graphs can be mapped into cell systems, and thus shown how to apply the cell-based theory to practical program construction. □

In the next two definitions, the "refinement" relation between cells will be defined. This is the correctness relation that must hold in modular programming between a module and its specification, where both the module and the specification are viewed as cells.

The first definition deals with cell signatures only.

3.1.16 Definition. Let $\langle E, D\rangle$ be a cell signature. A *syntactic refinement* of $\langle E, D\rangle$ is a cell signature $\langle E', D'\rangle$ such that whenever F is a site for $\langle E, D\rangle$, then F is a site for $\langle E', D'\rangle$, and $F \sqcup D' = F \sqcup D$ (*i. e.*, the two cell signatures have the same result signature on F). □

Here are some basic properties of the syntactic refinement relation.

3.1.17 Proposition.

(a) *The syntactic refinement relation is a preordering on the set of cell signatures.*

(b) *If* $\langle E', D'\rangle$ *is a syntactic refinement of* $\langle E, D\rangle$, *then* $E' \sqcup D' \sqsubseteq E \sqcup D$.

(c) $\langle E', D'\rangle$ *is a syntactic refinement of* $\langle E, D\rangle$, *if and only if*

$$E \text{ is a site for } \langle E', D'\rangle \quad \text{and} \quad E \sqcup D' = E \sqcup D.$$

Proof. It follows trivially from the definition that the syntactic refinement relation is transitive and reflexive and hence a preordering.

To prove (b), let $\langle E', D'\rangle$ be a syntactic refinement of $\langle E, D\rangle$. Since E is a site for $\langle E, D\rangle$, E also is a site for $\langle E', D'\rangle$, therefore $E' = E \sqcap (E' \sqcup D') \sqsubseteq E$, and hence $E' \sqcup D' \sqsubseteq E \sqcup D' = E \sqcup D$.

To prove (c), suppose that $\langle E', D'\rangle$ is a syntactic refinement of $\langle E, D\rangle$. The signature E is a site for $\langle E, D\rangle$, and hence E is a site for $\langle E', D'\rangle$, and $E \sqcup D' = E \sqcup D$.

Conversely, suppose that E is a site for $\langle E', D'\rangle$, and that $E \sqcup D' = E \sqcup D$. Let F be any site for $\langle E, D\rangle$, *i. e.*, $F \sim E \sqcup D$ and $F \sqcap (E \sqcup D) = E$. Since

$E' \sqcup D' \sqsubseteq E \sqcup D$ by (b), it follows that $F \sim E' \sqcup D'$ and

$$\begin{aligned} F \sqcap (E' \sqcup D') &= F \sqcap (E \sqcup D) \sqcap (E' \sqcup D') \\ &= E \sqcap (E' \sqcup D') \\ &= E', \end{aligned}$$

Thus, F is a site for $\langle E', D' \rangle$. Finally,

$$\begin{aligned} F \sqcup D' &= F \sqcup E \sqcup D' \\ &= F \sqcup E \sqcup D \\ &= F \sqcup D, \end{aligned}$$

and hence $\langle E', D' \rangle$ is a syntactic refinement of $\langle E, D \rangle$. □

The idea behind the definition of "syntactic refinement" is that if $\langle E', D' \rangle$ is a syntactic refinement of $\langle E, D \rangle$, then a cell of signature $\langle E', D' \rangle$ can be substituted for a cell of signature $\langle E, D \rangle$ without syntactic problems:

First, the cell of signature $\langle E', D' \rangle$ fits on any site for $\langle E, D \rangle$. In particular, this implies that $E' \sqsubseteq E$, *i. e.*, the environment signature of the syntactic refinement is contained in the environment signature of the cell signature it refines. E' may be smaller than E—this would just mean that a cell of signature $\langle E', D' \rangle$ does not use as many program entities as a cell of signature $\langle E, D \rangle$.

The second condition of "syntactic refinement", that $F \sqcup D' = F \sqcup D$ for every site F, means that the results of cells of signature $\langle E', D' \rangle$ and of signature $\langle E, D \rangle$ on a base of signature F have the same signature, *i. e.*, that the contributions of cells of the two kinds are syntactically the same.

At first, it might seem reasonable to require only that $F \sqcup D' \sqsupseteq F \sqcup D$, *i. e.*, that cells of the refined signature contribute at least as much as cells of the original signature. However, additional program entities in $F \sqcup D'$ might clash with program entities defined elsewhere in the system—for example, $F \sqcup D'$ might no longer be compatible with signatures that were compatible with $F \sqcup D$.

3.1.18 Definition. Let $\langle Q, R \rangle$ be a cell. A *refinement* of $\langle Q, R \rangle$ is a cell $\langle Q', R' \rangle$ such that $\mathrm{Sig}\langle Q', R' \rangle$ is a syntactic refinement of $\mathrm{Sig}\langle Q, R \rangle$, and whenever A is

a base for $\langle Q, R\rangle$, then

A is a base for $\langle Q', R'\rangle$,
there exists a result of $\langle Q', R'\rangle$ on A, and
every result of $\langle Q', R'\rangle$ on A is a result of $\langle Q, R\rangle$ on A. □

Here are some basic properties of the refinement relation.

3.1.19 Proposition.

(a) *The refinement relation is transitive.*

(b) *If a cell has a refinement, then it is consistent.*

(c) *A cell is a refinement of itself if and only if it is consistent.*

(d) *The refinement relation is a preordering on the set of consistent cells.*

(e) *A cell* $\langle Q', R'\rangle$ *of signature* $\langle E', D'\rangle$ *is a refinement of a cell* $\langle Q, R\rangle$ *of signature* $\langle E, D\rangle$, *if and only if*

$\langle E', D'\rangle$ *is a syntactic refinement of* $\langle E, D\rangle$,
$Q/E' \subseteq (Q' \wedge R')/E'$, *and*
$(Q \wedge R')/D \subseteq R$.

Proof.

Part (a): Consider three cells $\langle Q, R\rangle$, $\langle Q', R'\rangle$, and $\langle Q'', R''\rangle$ with signatures $\langle E, D\rangle$, $\langle E', D'\rangle$, and $\langle E'', D''\rangle$, and suppose that $\langle Q'', R''\rangle$ is a refinement of $\langle Q', R'\rangle$, and that $\langle Q', R'\rangle$ is a refinement of $\langle Q, R\rangle$. By Proposition 3.1.17 (a), $\langle E'', D''\rangle$ is a syntactic refinement of $\langle E, D\rangle$.

Let A be a base for $\langle Q, R\rangle$. Then A is a base for $\langle Q', R'\rangle$, hence A is a base for $\langle Q'', R''\rangle$ and there exists a result of $\langle Q'', R''\rangle$ on A. If B is a result of $\langle Q'', R''\rangle$ on A, then B is a result of $\langle Q', R'\rangle$ on A, and hence a result of $\langle Q, R\rangle$ on A.

Hence $\langle Q'', R''\rangle$ is a refinement of $\langle Q, R\rangle$.

Part (b): Let $\langle Q', R'\rangle$ be a refinement of $\langle Q, R\rangle$, and let A be a base for $\langle Q, R\rangle$. Then A is a base for $\langle Q', R'\rangle$, there exists a result B of $\langle Q', R'\rangle$ on A, and B also is a result of $\langle Q, R\rangle$ on A. Since A was arbitrary, $\langle Q, R\rangle$ is consistent.

Part (c): Putting $\langle Q', R' \rangle$ equal to $\langle Q, R \rangle$ in the definition of "refinement", the definition simplifies to: "whenever A is a base for $\langle Q, R \rangle$, there exists a result of $\langle Q, R \rangle$ on A" (the other conditions are trivially true). This is just "$\langle Q, R \rangle$ is consistent".

Part (d): By (c), the refinement relation is reflexive on the set of consistent cells; by (a), it is transitive.

Part (e): Let $\langle Q, R \rangle$ be a cell of signature $\langle E, D \rangle$, and $\langle Q', R' \rangle$ be a cell of sigan-ture $\langle E', D' \rangle$.

Suppose first that $\langle Q', R' \rangle$ is a refinement of $\langle Q, R \rangle$. By definition, $\langle E', D' \rangle$ is a syntactic refinement of $\langle E, D \rangle$.

To see that $Q/E' \subseteq (Q' \wedge R')/E'$, consider $A \in Q$. A is a base for $\langle Q, R \rangle$, and hence a base for $\langle Q', R' \rangle$, and we can pick a result B of $\langle Q', R' \rangle$ on A. B also is a result of $\langle Q, R \rangle$ on A. Now $B/E' = (B/E)/E' = A/E' \in Q'$, because A is a base for $\langle Q', R' \rangle$, and $B/D' \in R'$, because B is a result of $\langle Q', R' \rangle$ on A. Hence

$$A/E' = (B/E)/E' = B/E' = (B/(E' \sqcup D'))/E' \in (Q' \wedge R')/E'.$$

Since A was an arbitrary element of Q, it follows that $Q/E' \subseteq (Q' \wedge R')/E'$.

To see that $(Q \wedge R')/D \subseteq R$, consider $B \in (Q \wedge R')$. Then B/E is a base for $\langle Q, R \rangle$, hence a base for $\langle Q', R' \rangle$, and B is a result of $\langle Q', R' \rangle$ on B/E. Hence, B is a result of $\langle Q, R \rangle$ on B/E, and thus $B/D \in R$.

Conversely, suppose that the three conditions of the proposition are satisfied, and let A be a base of signature F for $\langle Q, R \rangle$. Then F is a site for $\langle E, D \rangle$ and hence for $\langle E', D' \rangle$. Since

$$A/E' = (A/E)/E' \in Q/E' \subseteq (Q' \wedge R')/E' \subseteq Q',$$

A is a base for $\langle Q', R' \rangle$.

Since $A/E' \in (Q' \wedge R')/E'$, we can pick $B \in (Q' \wedge R')$ such that $B/E' = A/E'$. We can join A and B, because the intersection of their signatures is $F \sqcap (E' \sqcup D') = E'$, and $B/E' = A/E'$. Now $A \sqcup B$ is a result of $\langle Q', R' \rangle$ on A,

because $A \sqcup B \in Mod(F \sqcup (E' \sqcup D')) = Mod(F \sqcup D')$, $(A \sqcup B)/F = A$, and $(A \sqcup B)/D' = B/D' \in R'$.

Finally, suppose that B is a result of $\langle Q', R' \rangle$ on A. Then B also is a result of $\langle Q, R \rangle$ on A, because $B/F = A$ and (as $B/E = (B/F)/E = A/E \in Q$ and $B/D' \in R'$)

$$\begin{aligned} B/D &= (B/(E \sqcup D))/D = (B/(E \sqcup D'))/D \\ &\in (Q \wedge R')/D \subseteq R. \end{aligned}$$

Thus, $\langle Q', R' \rangle$ is a refinement of $\langle Q, R \rangle$. □

The idea behind the "refinement" notion is that a refinement of a cell can be substituted for that cell without problems: The refinement will be applicable to all bases of the original cell, and every result the refinement can produce could also have been produced by the original cell.

A variant of the definition, which was tried at first, would not require that a refinement have a result on every base of the original cell. Then "refinement" would be a preordering on all cells, not just the consistent ones. It turns out, however, that this refinement notion does not have the desired composition properties (see Example 3.5.10).

3.2 Cell Systems and Decomposition

This section deals with the design of modular systems. Modular systems are viewed as families of cells that fit together syntactically, that is, whose signatures form a "signature system". Semantically, the design of modular sytems is formalized in the "decomposition" concept: A cell system is a decomposition of a cell (the "global" cell, which contains the external interfaces of the system), if the external requirement interface guarantees that all cells of the system are supplied with proper bases, and if the combined results of the cells guarantee that the external result interface is satisfied.

First, we deal with the syntactic compatibility conditions for a cell system, that is, we deal with cell signatures only. A family of cell signatures determines a "syntactic dependence relation", according to which a cell M depends on a cell N if M uses program entities defined by N. In the design of modular systems, more general "dependence relations" are admitted, which are extensions of the syntactic dependence relation, and which allow additional dependencies between cells to be specified (with the purpose, *e.g.*, of simplifying proofs).

Recall that a relation $<$ is *well-founded* if and only if every nonempty set has a $<$-minimal element, that is, if and only if for all $M \neq \emptyset$ there exists $x \in M$ such that $y \not< x$ for all $y \in M$.

3.2.1 Definition. A family $T = \langle E_i, D_i \rangle_{i \in I}$ of cell signatures is *compatible*, if $\langle E_i \sqcup D_i \rangle_{i \in I}$ is compatible. □

3.2.2 Definition. Let $T = \langle E_i, D_i \rangle_{i \in I}$ be a compatible family of cell signatures.

The *syntactic dependence relation* of T is the relation $<_T \subseteq I \times I$ defined by:

$$k <_T i \;:\Longleftrightarrow\; k \neq i \text{ and } D_k \sqcap (E_i \sqcup D_i) \not\sqsubseteq E_k.$$

Its transitive closure is written "$\ll_T$".

A *dependence relation* for T is a well-founded relation $< \subseteq I \times I$ such that $<_T \subseteq <$. The transitive closure of a dependence relation $<$ is written "$\ll$". □

The systems considered in this thesis will always have a dependence relation, and by definition, such a dependence relation is well-founded (often, the dependence relation agrees with the syntactic dependence relation). This also implies that the syntactic dependence relation is well-founded.

Systems with a well-founded dependence relation are often called "hierarchical", and this thesis deals with hierarchical systems only. In particular, circular ("recursive") dependencies between cells are excluded. The reason for this is that circular dependencies appear to be incompatible with the basic goal of modular programming, which is that the correctness of individual modules should imply the correctness of the composed system (the "Structured Correctness Argument

for a Modular System", illustrated in Figure 3-5). Why this is so will be explained in Section 3.4 below (p. 151–154).

3.2.3 Example. In Examples 3.1.5 and 3.1.15, it was shown how the interfaces of the *dictionary* program development can be organized into cells (in the institution $\langle$**ASig, AIncl, Alg**$\rangle$). Associated with the five program modules M_{DICT}, M_{INOUT}, M_{INPUT}, M_{OUTPUT}, and M_{STORE} are the five cells $\mathcal{M}_{DICT}$, $\mathcal{M}_{INOUT}$, $\mathcal{M}_{INPUT}$, $\mathcal{M}_{OUTPUT}$, and $\mathcal{M}_{STORE}$ that specify them (see Figure 3-4). The family of cell signatures of the specification cells is

$$\mathcal{T} = \langle \mathcal{E}_i, \mathcal{D}_i \rangle_{i \in \{DICT, INOUT, INPUT, OUTPUT, STORE\}},$$

where

$$\begin{aligned}
\mathcal{E}_{DICT} &= \mathrm{Sig}(I_{INOUT}), & \mathcal{D}_{DICT} &= \mathrm{Sig}(I_{DICT}),\\
\mathcal{E}_{INOUT} &= \mathrm{Sig}(I_{INPUT}) \sqcup \mathrm{Sig}(I_{OUTPUT}), & \mathcal{D}_{INOUT} &= \mathrm{Sig}(I_{INOUT}),\\
\mathcal{E}_{INPUT} &= \mathrm{Sig}(I_{INSERT}) \sqcup \mathrm{Sig}(I_{LISTITEM}), & \mathcal{D}_{INPUT} &= \mathrm{Sig}(I_{INPUT}),\\
\mathcal{E}_{OUTPUT} &= \mathrm{Sig}(I_{MIN}) \sqcup \mathrm{Sig}(I_{LISTITEM}), & \mathcal{D}_{OUTPUT} &= \mathrm{Sig}(I_{OUTPUT}),\\
\mathcal{E}_{STORE} &= \mathrm{Sig}(I_{LISTITEM}) \sqcup \mathrm{Sig}(I_{ITEM}), & \mathcal{D}_{STORE} &= \mathrm{Sig}(I_{INSERT}) \\
& & & \quad \sqcup \mathrm{Sig}(I_{MIN}).
\end{aligned}$$

All these signatures are compatible, as can be verified easily using the criterion of Theorem 2.3.6 (a): one checks that no symbol occurs both as sort and as function symbol, nor as function symbol with different types.

The syntactic dependence relation $<_{\mathcal{T}}$ of $\mathcal{T}$ is given in Figure 3-6. The dependencies in this figure are explained as follows: $\mathcal{T}_{DICT}$ and $\mathcal{T}_{INOUT}$ import *input* from $\mathcal{T}_{INPUT}$, *output* from $\mathcal{T}_{OUTPUT}$ and *store* from $\mathcal{T}_{STORE}$, $\mathcal{T}_{INPUT}$ imports *store*, *empty* and *insert* from $\mathcal{T}_{STORE}$, and $\mathcal{T}_{OUTPUT}$ imports *store*, *min* and *removemin* from $\mathcal{T}_{STORE}$.

Note that $\mathcal{T}_{INOUT}$ has no new symbols, and so no other cell signature of the system depends on it. Recall that $\mathcal{T}$ is the signature family of the system $\mathcal{M}$ of Figure 3-4, which in turn is derived from the design graph of Figure 1-20. This graph suggests an additional dependency $INOUT < DICT$, which is also shown in Figure 3-4. Adding this dependency to $<_{\mathcal{T}}$ yields a dependence relation for $\mathcal{T}$, which is shown in Figure 3-7. □

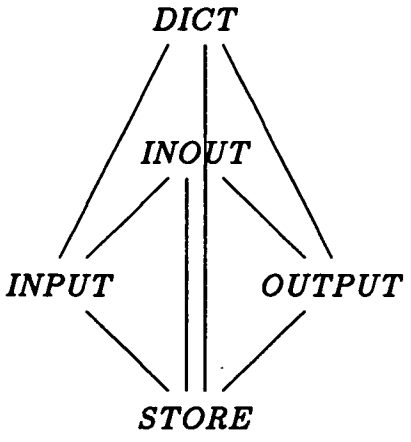


Figure 3-6: The syntactic dependence relation of $\mathcal{T}$

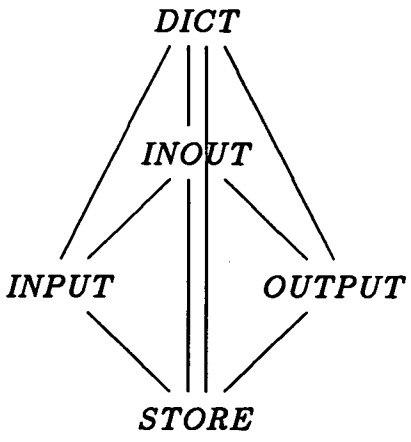


Figure 3-7: A dependence relation for $\mathcal{T}$

The following definition expresses the syntactical requirements of a modular system: the family of its cell signatures must form a "signature system".

3.2.4 Definition. Let $T = \langle E_i, D_i \rangle_{i \in I}$ be a compatible family of cell signatures, and let $< \subseteq I \times I$ be a dependence relation for T.

A *system site* for $[T, <]$ is a signature $\bar{E}$, compatible with $\bigsqcup_{i \in I} D_i$, such that for all $i \in I$:

$$\bar{E} \sqcup \bigsqcup_{k \ll i} D_k \text{ is a site for } \langle E_i, D_i \rangle.$$

The pair $[T, <]$ is an *ordered signature system*, if there exists a system site for it.

A *signature system* is a compatible family T of cell signatures, such that $[T, <_T]$ is an ordered signature system. □

According to this definition, an ordered signature system can be supplied with a "system site". Such a system site has the property that each cell signature of the system will be supplied with a site when all the cell signatures on which it (directly or indirectly) depends have been "installed" on the system site (the result of this installation is given by the expression $\bar{E} \sqcup \bigsqcup_{k \ll i} D_k$ in the definition above). One may imagine the cell signatures of the system being installed in a

"bottom-up" fashion, where the set of indices of installed cell signatures is always $<$-downward closed.

In general, a system site must contain the program entities that are used but not defined within a system. It may contain additional entities, but none that may clash with entities defined within the system.

3.2.5 Example. Consider again the cell signature family $\mathcal{T}$ of the previous example (3.2.3), and let $<$ be the dependence relation given by Figure 3-7, which is obviously well-founded.

A system site for $[\mathcal{T}, <]$ is the signature

$$\bar{\mathcal{E}} := \mathrm{Sig}(I_{ITEM}) \sqcup \mathrm{Sig}(I_{LISTITEM}).$$

To prove this, one has to show that for each $i \in \{DICT, INOUT, INPUT, OUTPUT, STORE\}$,

$$\bar{\mathcal{E}} \sqcup \bigsqcup_{k \ll i} \mathcal{D}_k \text{ is a site for } \langle \mathcal{E}_i, \mathcal{D}_i \rangle,$$

that is,

$$(\bar{\mathcal{E}} \sqcup \bigsqcup_{k \ll i} \mathcal{D}_k) \sqcap (\mathcal{E}_i \sqcup \mathcal{D}_i) = \mathcal{E}_i.$$

Since all signatures involved are compatible ($\bar{\mathcal{E}}$ contributes nothing new), according to the formulas (b) and (c) of Theorem 2.3.6 these checks reduce to the analogous equations where the symbol sets of the signatures are combined by $\cup$ and $\cap$. These equations are easily verified.

The fact that $\bar{\mathcal{E}}$ is a system site for $[\mathcal{T}, <]$ implies that $[\mathcal{T}, <]$ is an ordered signature system.

As we shall see later, in Theorem 3.5.8, the fact that $\bar{\mathcal{E}}$ is a system site for $[\mathcal{T}, <]$ implies that it is a system site for any ordered system of the form $[\mathcal{T}, <']$, where $<'$ is a dependence relation for $\mathcal{T}$, and hence that $[\mathcal{T}, <']$ is an ordered signature system. □

The next definition presents the relation that must hold between an ordered signature system $[T, <]$ and a "global" cell signature $\langle \bar{E}, \bar{D} \rangle$, which gives the syntax of a pair of external interfaces for the system.

3.2.6 Definition. Let $\langle \bar{E}, \bar{D} \rangle$ be a cell signature. A *syntactic decomposition* of $\langle \bar{E}, \bar{D} \rangle$ is an ordered signature system $[T, <]$, such that $\bar{E}$ is a system site for $[T, <]$, and (with $T = \langle E_i, D_i \rangle_{i \in I}$) $\bar{D} \sqsubseteq \bar{E} \sqcup \bigsqcup_{i \in I} D_i$. □

This definition just says that it must be possible to install T on $\bar{E}$ (the order of installation being given by $<$), and that the entities of the external result signature $\bar{D}$ must be provided by the system.

3.2.7 Example. The ordered signature system $[\mathcal{T}, <]$ of the previous example (3.2.5) is a syntactic decomposition of the cell signature $\langle \bar{\mathcal{E}}, \bar{\mathcal{D}} \rangle$, where

$$\bar{\mathcal{E}} = \mathrm{Sig}(I_{ITEM}) \sqcup \mathrm{Sig}(I_{LISTITEM}) \qquad \text{(Figs. 1-3 and 1-4)}$$

$$\bar{\mathcal{D}} = \mathrm{Sig}(I_{DICT}) \qquad \text{(Fig. 1-5).}$$

We saw in Example 3.2.5 that $\bar{\mathcal{E}}$ is a system site for $[\mathcal{T}, <]$, and it is clear that $\bar{\mathcal{D}}$ is contained in the join of $\bar{\mathcal{E}}$ with the definition signatures, because it is equal to the definition signature of $\mathcal{T}_{DICT}$. □

So far, we have dealt with the syntactic aspects of modular system design, and for this, we could restrict our attention to cell signatures. Now we begin to consider the semantical aspects, and therefore deal with cells.

3.2.8 Definition. An *ordered cell system* is a pair $[M, <]$, where M is a family of cells whose family of cell signatures T is such that $[T, <]$ is an ordered signature system.

A *cell system* is a family of cells whose family of signatures is a signature system.

The signature family of a cell system M is written "$\mathrm{Sig}(M)$". □

3.2.9 Example. In the examples 3.1.5 and 3.1.15, it was shown how the interfaces of the *dictionary* program development can be organized into cells such that for each of the five program modules there is a cell that specifies it.

These specification cells form the family

$$\mathcal{M} = \langle \mathcal{Q}_i, \mathcal{R}_i \rangle_{i \in \{DICT, INOUT, INPUT, OUTPUT, STORE\}},$$

where

$$\begin{aligned}
\mathcal{Q}_{DICT} &= I_{INOUT}, & \mathcal{R}_{DICT} &= I_{DICT}, \\
\mathcal{Q}_{INOUT} &= I_{INPUT} \wedge I_{OUTPUT}, & \mathcal{R}_{INOUT} &= I_{INOUT}, \\
\mathcal{Q}_{INPUT} &= I_{INSERT} \wedge I_{LISTITEM}, & \mathcal{R}_{INPUT} &= I_{INPUT}, \\
\mathcal{Q}_{OUTPUT} &= I_{MIN} \wedge I_{LISTITEM}, & \mathcal{R}_{OUTPUT} &= I_{OUTPUT}, \\
\mathcal{Q}_{STORE} &= I_{LISTITEM} \wedge I_{ITEM}, & \mathcal{R}_{STORE} &= I_{INSERT} \wedge I_{MIN}.
\end{aligned}$$

The family $\mathcal{T}$ of signatures of this cell family was given in Example 3.2.3. With $<$ the dependence relation as in Figure 3-7, it was shown in Example 3.2.5 that $[\mathcal{T}, <]$ is an ordered signature system; hence $[\mathcal{M}, <]$ is an ordered cell system. □

Note that the concepts "cell system" and "ordered cell system" are still syntactical in nature; whether or not a family of cells is a cell system depends only on the signatures of the cells, not on their semantics.

The semantics of cells enter the picture in the following definition, which describes the "decomposition" relation that must hold between an ordered cell system $[M, <]$ and a "global" cell $\langle \bar{Q}, \bar{R} \rangle$ that consists of the external interfaces of the system.

3.2.10 Definition. Let $\langle \bar{Q}, \bar{R} \rangle$ be a cell of signature $\langle \bar{E}, \bar{D} \rangle$. A *decomposition* of $\langle \bar{Q}, \bar{R} \rangle$ is an ordered cell system $[M, <]$ such that, with $M = \langle Q_i, R_i \rangle_{i \in I}$ of signature $T = \langle E_i, D_i \rangle_{i \in I}$, we have

(a) $[T, <]$ is a syntactic decomposition of $\langle \bar{E}, \bar{D} \rangle$,

(b) $(\bar{Q} \wedge \bigwedge_{k \ll i} R_k) /\!\!/ E_i \subseteq Q_i \qquad$ for all $i \in I$,

(c) $(\bar{Q} \wedge \bigwedge_{i \in I} R_i) /\!\!/ \bar{D} \subseteq \bar{R}$. □

Again, the idea of the definition is that the cells of M are installed "bottom-up" on a base as described by $\bar{Q}$. Clause (a) says that no syntactic problems arise, clause (b) says that the external requirement interface $\bar{Q}$ together with the result interfaces of the cells on which M_i directly or indirectly depends (*i.e.*, $\bigwedge_{k \ll i} R_k$)

guarantees that M_i will be supplied with a base in the installation process (it is a consequence of (a) that the signature of $\bar{Q} \wedge \bigwedge_{k \ll i} R_k$ is a site for $\langle E_i, D_i \rangle$). Finally, clause (c) guarantees that the result produced by the installation process (which is an element of $\bar{Q} \wedge \bigwedge_{i \in I} R_i$) matches the external result interface $\bar{R}$.

3.2.11 Example. Consider the ordered cell system $[\mathcal{M}, <]$ of the previous example, where $\mathcal{M}$ has the signature family $\mathcal{T}$ given in Example 3.2.3, and $<$ is the dependence relation given in Figure 3-7. This ordered system is a decomposition of the cell

$$\langle \bar{\mathcal{Q}}, \bar{\mathcal{R}} \rangle,$$

where

$$\bar{\mathcal{Q}} = I_{LISTITEM} \wedge I_{ITEM},$$
$$\bar{\mathcal{R}} = I_{DICT}.$$

The syntactic decomposition property was verified in Example 3.2.5. Checking the clauses (b) and (c) of the "decomposition" definition is trivial, because the cell system has been obtained as a translation of the design graph of Fig. 1-20: the requirement interface of each cell is entirely composed of interfaces that occur below it in the graph, and hence occur in either $\bar{\mathcal{Q}}$ or the result interfaces of the cells on which the cell depends; similarly, the external result interface $\bar{\mathcal{R}} = I_{DICT}$ occurs in the graph, and hence occurs in either $\bar{\mathcal{Q}}$ or the result interfaces of the cells. □

This simple argument illustrates that design graphs are useful for designing decompositions: by keeping track of "atomic" interfaces as the oval nodes of the graph (*e.g.*, I_{DICT}, I_{INPUT}, I_{MIN} etc.), and by recording which modules depend on and which modules provide these interfaces, a design graph makes it trivial to prove the semantic part of the decomposition property of its translation into an ordered cell system.

It is important for this that a design graph is translated into an ordered cell system rather than just a cell system, because the syntactic dependence relation might not be sufficient to make the decomposition property trivially

true. This point is illustrated by our example: in the design graph (Fig. 1-20), M_{DICT} depends on I_{INOUT}, hence its requirement interface is $\mathcal{Q}_{DICT} = I_{INOUT}$, whereas according to the syntactic dependence relation $<_\tau$ (Fig. 3-6), we have $INOUT \not<_\tau DICT$, and so the proof that

$$(\bar{\mathcal{Q}} \wedge \bigwedge_{k \ll_\tau DICT} \mathcal{R}_k) /\!\!/ \mathcal{E}_{DICT} \subseteq \mathcal{Q}_{DICT}$$

would not be trivial; in the dependence relation $<$, however, we have $INOUT < DICT$, so $I_{INOUT} = \mathcal{R}_{INOUT}$ appears on the left hand side of the analogous formula, which is therefore trivially true.

While design graphs are able to guarantee a semantically sound decomposition, they are not as helpful as far as syntax is concerned: in order that the translation of a design graph be syntactically correct, the program symbols of the atomic interfaces must be chosen in such a way that symbols are equal exactly when they are supposed to denote the same program entity; in general, this seems to rule out independent choice of symbols for different interfaces.

It might also be a defect in practice that an individual interface such as I_{MIN} (Figure 1-17) does not express the distinction between the symbols it is intended to define (here, *isempty*, *min*, and *removemin*), and those that are merely used (here, *bool*, *item*, *leitem*, *listitem*, and *store*).

3.3 Composition of Systems

This section introduces the "composition" operation, which, given a cell system and a signature of program entities to be exported from the system, produces a single cell as result. This cell describes the "joint effect" of the cells of the system.

In particular, the composition operation describes the joint semantics of a family of program modules that import and export program entities from and to a common name space, so that modules can import entities exported by other modules.

Before dealing with the composition operation itself, we prove an important syntactical property of cell systems: for each signature system (the "syntax" of a cell system), there is a $\sqsubseteq$-least system site (which is necessarily unique). This signature may be thought of as containing the program entities that the system needs to import, because they are used but not defined within the system.

3.3.1 Proposition. *Let $[T, <]$ be an ordered signature system, where $T = \langle E_i, D_i \rangle_{i \in I}$. The set of system sites for $[T, <]$ has a $\sqsubseteq$-least element $\bar{E}_0$, and this element satisfies the equation*

$$\bar{E}_0 = \bar{E} \sqcap \bigsqcup_{i \in I} E_i$$

for every system site $\bar{E}$ for $[T, <]$.

For the proof, we need two lemmas.

3.3.2 Lemma. *If $[T, <]$ is an ordered signature system, $T = \langle E_i, D_i \rangle_{i \in I}$, and $\bar{E}$ a system site for $[T, <]$, then $\bar{E}$ is compatible with $\bigsqcup_{i \in I}(E_i \sqcup D_i)$, and*

$$\bar{E} \sqcup \bigsqcup_{i \in I}(E_i \sqcup D_i) = \bar{E} \sqcup \bigsqcup_{i \in I} D_i.$$

Proof. All the signatures $\bar{E}$, E_i, D_i for $i \in I$ are subsignatures of $\bar{E} \sqcup \bigsqcup_{i \in I} D_i$, because for all $i \in I$:

$$\begin{aligned} E_i &\sqsubseteq \bar{E} \sqcup \bigsqcup_{k \ll i} D_k \qquad (\bar{E} \text{ is system site}) \\ &\sqsubseteq \bar{E} \sqcup \bigsqcup_{i \in I} D_i. \end{aligned}$$

□

3.3.3 Lemma. *Let $[T, <]$ be an ordered signature system, $T = \langle E_i, D_i \rangle_{i \in I}$, and let $\bar{E}$ and $\bar{E}'$ be system sites for $[T, <]$. Then for all $i \in I$:*

$$\bar{E} \sqcap (E_i \sqcup D_i) \sqsubseteq \bar{E}'.$$

Proof. Suppose the conclusion was false. Choose $j \in I$ to be a $\ll$-minimal value satisfying

$$\bar{E} \sqcap (E_j \sqcup D_j) \not\sqsubseteq \bar{E}'.$$

Because $\bar{E}$ and $\bar{E}'$ are system sites for $[T, <]$, we have

$$\begin{aligned}
\bar{E} \sqcap (E_j \sqcup D_j) &\sqsubseteq (\bar{E} \sqcup \bigsqcup_{k \ll j} D_k) \sqcap (E_j \sqcup D_j) \\
&= E_j \\
&= (\bar{E}' \sqcup \bigsqcup_{k \ll j} D_k) \sqcap (E_j \sqcup D_j) \\
&\sqsubseteq \bar{E}' \sqcup \bigsqcup_{k \ll j} D_k,
\end{aligned}$$

and hence

$$\begin{aligned}
\bar{E} \sqcap (E_j \sqcup D_j) &\sqsubseteq \bar{E} \sqcap (\bar{E}' \sqcup \bigsqcup_{k \ll j} D_k) \\
&= (\bar{E} \sqcap \bar{E}') \sqcup (\bar{E} \sqcap \bigsqcup_{k \ll j} D_k) \\
&= (\bar{E} \sqcap \bar{E}') \sqcup \bigsqcup_{k \ll j} (\bar{E} \sqcap D_k) \\
&\sqsubseteq \bar{E}' \sqcup \bar{E}' && \text{(minimality of } j\text{)} \\
&= \bar{E}'.
\end{aligned}$$

This contradicts the definition of j and concludes the proof. □

Proof of Proposition 3.3.1.

Let $[T, <]$ be an ordered signature system, where $T = \langle E_i, D_i \rangle_{i \in I}$. We shall show that whenever $\bar{E}$ is a system site for $[T, <]$, then the signature $\bar{E}_0 := \bar{E} \sqcap \bigsqcup_{i \in I} E_i$ is the least element of the set of system sites for $[T, <]$. Because the set of system sites is nonempty, this implies that it has a least element $\bar{E}_0$, and because the least element is unique while $\bar{E}$ can be any system site, it follows that the equation $\bar{E}_0 = \bar{E} \sqcap \bigsqcup_{i \in I} E_i$ holds for any system site $\bar{E}$.

Let $\bar{E}$ be any system site for $[T, <]$, and define $\bar{E}_0 := \bar{E} \sqcap \bigsqcup_{i \in I} E_i$ (by Lemma 3.3.2, $\bar{E}$ and $\bigsqcup_{i \in I} E_i$ are compatible). To show that $\bar{E}_0$ is a system site,

consider any $i \in I$. We have

$$\begin{aligned}
&(\bar{E}_0 \sqcup \bigsqcup_{k \ll i} D_k) \sqcap (E_i \sqcup D_i) \\
&\sqsubseteq (\bar{E} \sqcup \bigsqcup_{k \ll i} D_k) \sqcap (E_i \sqcup D_i) \\
&= E_i,
\end{aligned}$$

and conversely,

$$\begin{aligned}
E_i &= E_i \sqcap (\bar{E} \sqcup \bigsqcup_{k \ll i} D_k) \\
&= (E_i \sqcap \bar{E}) \sqcup (E_i \sqcap \bigsqcup_{k \ll i} D_k) \\
&\sqsubseteq \bar{E}_0 \sqcup \bigsqcup_{k \ll i} D_k,
\end{aligned}$$

so that $(\bar{E}_0 \sqcup \bigsqcup_{k \ll i} D_k) \sqcap (E_i \sqcup D_i) = E_i$. Hence $\bar{E}_0$ is a system site for $[T, <]$.

We now wish to show that $\bar{E}_0$ is the least of these system sites. Let $\bar{E}'$ be any system site for $[T, <]$. Then

$$\begin{aligned}
\bar{E}_0 &= \bar{E} \sqcap \bigsqcup_{i \in I} E_i \\
&= \bigsqcup_{i \in I} (\bar{E} \sqcap E_i) \\
&\sqsubseteq \bigsqcup_{i \in I} (\bar{E} \sqcap (\bar{E}' \sqcup \bigsqcup_{k \ll i} D_k)) \\
&= \bigsqcup_{i \in I} ((\bar{E} \sqcap \bar{E}') \sqcup \bigsqcup_{k \ll i} (\bar{E} \sqcap D_k)) \\
&\sqsubseteq (\bar{E} \sqcap \bar{E}') \sqcup \bigsqcup_{i \in I} (\bar{E} \sqcap D_i) \\
&\sqsubseteq \bar{E}' \qquad \text{(by Lemma 3.3.3).}
\end{aligned}$$

Since $\bar{E}'$ was arbitrary, it follows that $\bar{E}_0$ is a $\sqsubseteq$-lower bound of the set of system sites for $[T, <]$. Since $\bar{E}_0$ is a system site itself, it is the $\sqsubseteq$-least system site for $[T, <]$. □

The following definition describes the composition of a cell system syntactically. The signature C expresses which of the entities defined within the system are to be exported (it "confines" the list of exported entities).

3.3.4 Definition. Let $T = \langle E_i, D_i \rangle_{i \in I}$ be a signature system, and let C be a signature compatible with $\bigsqcup_{i \in I} D_i$. The *composition* of T confined by C, written "$\square_C T$", is the cell signature

$$\square_C T = \langle \bar{E}, \bar{D} \rangle,$$

where

$$\bar{E} \text{ is the } \sqsubseteq\text{-least system site for } [T, <_T],$$
$$\bar{D} = \Big(\bar{E} \sqcup \big(C \sqcap \bigsqcup_{i \in I} D_i\big)\Big). \qquad \square$$

In this definition, $\bar{E}$ is the least system site for $[T, <_T]$, that is, the least signature onto which the cells described by T can be built. The signature $\bar{E}$ therefore expresses the "minimal syntactic requirements" of a system with signature T, and is the natural candidate for the environment signature of $\square_C T$. The result signature $\bar{D}$ contains $\bar{E}$ and in addition the symbols defined by the system that are "exported" according to the "confinement" signature C.

3.3.5 Example. The final program in the *dictionary* example is obtained as the composition of the five modules M_{DICT}, M_{INOUT}, M_{INPUT}, M_{OUTPUT}, and M_{STORE}. The family T of signatures of these cells is as follows:

$$T = \langle E_i, D_i \rangle_{i \in \{DICT, INOUT, INPUT, OUTPUT, STORE\}},$$

where

$\langle E_{DICT}, D_{DICT} \rangle =$
cell signature
environment signature
listitem, *store*: sort
input: *listitem* $\rightarrow$ *store*
output: *store* $\rightarrow$ *listitem*
defined symbols
dictionary: *listitem* $\rightarrow$ *listitem*

$\langle E_{INOUT}, D_{INOUT} \rangle =$

cell signature

environment signature

⟨empty⟩

defined symbols

⟨empty⟩

$\langle E_{INPUT}, D_{INPUT} \rangle =$

cell signature

environment signature

bool, *item*, *listitem*, *store*: **sort**

isnil: *listitem* → *bool*

hd: *listitem* → *item*

tl: *listitem* → *listitem*

empty: → *store*

insert: *item store* → *store*

defined symbols

input: *listitem* → *store*

$\langle E_{OUTPUT}, D_{OUTPUT} \rangle =$

cell signature

environment signature

bool, *item*, *listitem*, *store*: **sort**

leitem: *item item* → *bool*

nil: → *listitem*

cons: *item listitem* → *listitem*

isempty: *store* → *bool*

min: *store* → *item*

removemin: *store* → *store*

defined symbols

output: *store* → *listitem*

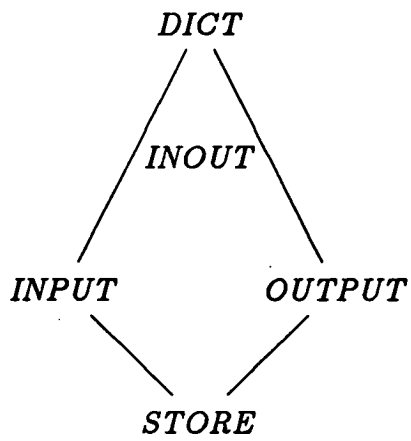


Figure 3-8: The syntactic dependence relation of T

$\langle E_{STORE}, D_{STORE} \rangle =$

cell signature

environment signature

$\mathrm{Sig}(I_{ITEM}) \sqcup \mathrm{Sig}(I_{LISTITEM})$

defined symbols

store: **sort**

empty: $\rightarrow$ *store*

insert: *item store* $\rightarrow$ *store*

isempty: *store* $\rightarrow$ *bool*

min: *store* $\rightarrow$ *item*

removemin: *store* $\rightarrow$ *store.*

The syntactic dependence relation $<_T$ of this signature system is given in Figure 3-8. Note that, in contrast to $<_\mathcal{T}$ (the syntactic dependence relation, shown in Figure 3-6, of the specification cell system $\mathcal{T}$ of Example 3.2.3), *INOUT* no longer depends on anything, because the environment signature of M_{INOUT} is empty.

It is easily checked that each cell signature of T is a syntactic refinement of the corresponding cell signature of $\mathcal{T}$. From Lemma 3.4.3 (d) below it follows that T also is a signature system.

Since the signature $\bar{\mathcal{E}}$ of Example 3.2.5 is a system site for $[\mathcal{T}, <]$, it follows from Lemma 3.4.3 (c) below that $\bar{\mathcal{E}}$ also is a system site for $[T, <_T]$. Using Proposition 3.3.1, it follows that $\bar{\mathcal{E}}$ is the least system site for $[T, <_T]$, because $\bar{\mathcal{E}}$ is contained in the join of the environment signatures of T (it equals E_{STORE}). Hence the composition of T has environment signature $\bar{E} := \bar{\mathcal{E}}$.

Since the *dictionary* problem is specified by I_{DICT}, only the program entities of $\mathrm{Sig}(I_{DICT})$ need to be exported from the system, and this can be expressed by using the confinement signature $C := \mathrm{Sig}(I_{DICT})$. This signature is included in the join of the result signatures of T (it is already included in $D_{DICT} \sqcup E_{STORE}$, which in turn is included in $D_{DICT} \sqcup D_{STORE}$). Hence one obtains

$$\Box_C T = \langle \bar{E}, \bar{D} \rangle,$$

where

$$\begin{aligned} \bar{E} = \bar{\mathcal{E}} \quad &= \mathrm{Sig}(I_{ITEM}) \sqcup \mathrm{Sig}(I_{LISTITEM}) \\ \bar{D} = \bar{E} \sqcup C &= \mathrm{Sig}(I_{ITEM}) \sqcup \mathrm{Sig}(I_{LISTITEM}) \sqcup \mathrm{Sig}(I_{DICT}). \end{aligned}$$

This cell signature will be the signature of the "composition" of the modules of the *dictionary* example, that is, the signature of the cell describing the final program. □

So far, we have considered the syntactical aspects of the composition operation, and therefore dealt with cell signatures only. Here now is the definition of the composition operation for cells.

3.3.6 Definition. Let $M = \langle Q_i, R_i \rangle_{i \in I}$ be a cell system with signature system $T = \langle E_i, D_i \rangle_{i \in I}$, and let C be a signature compatible with $\bigsqcup_{i \in I} D_i$. The *composition* of M confined by C, written "$\Box_C M$", is the cell $\Box_C M = \langle \bar{Q}, \bar{R} \rangle$ of signature $\langle \bar{E}, \bar{D} \rangle := \Box_C T$ defined by

$$\bar{Q} := \{ A \in Mod(\bar{E}) \mid (\{A\} \wedge \bigwedge_{k \ll_T i} R_k) /\!\!/ E_i \subseteq Q_i \text{ for all } i \in I \}$$

$$\bar{R} := (\bar{Q} \wedge \bigwedge_{i \in I} R_i) /\!\!/ \bar{D}.$$ □

The idea behind this definition is that $\langle \bar{Q}, \bar{R} \rangle$ describes the "joint effect" of the cells $\langle Q_i, R_i \rangle$ $(i \in I)$. The external requirement interface $\bar{Q}$ consists of those $\bar{E}$-models for which it is guaranteed that each cell M_i obtains a proper base when the cells are "installed" in a bottom-up fashion according to their syntactic dependence relation: When cell M_i is installed, at least the cells M_k with $k \ll_T i$ will be already in place. Given an $\bar{E}$-model A, the cells with $k \ll_T i$ will produce a result B in $\{A\} \wedge \bigwedge_{k \ll_T i} R_k$, and the condition

$$(\{A\} \wedge \bigwedge_{k \ll_T i} R_k) /\!\!/ E_i \subseteq Q_i$$

in the definition of $\bar{Q}$ implies that B is a base for $\langle Q_i, R_i \rangle$.

The result interface $\bar{R}$ is obtained by combining the individual result interfaces with $\bar{Q}$.

A useful alternative characterization of $\bar{Q}$ is given by the following proposition.

3.3.7 Proposition. *In the previous definition, $\bar{Q}$ is the $\subseteq$-largest interface of signature $\bar{E}$ satisfying*

$$(\bar{Q} \wedge \bigwedge_{k \ll_T i} R_k) /\!\!/ E_i \subseteq Q_i \qquad \text{for all } i \in I.$$

Proof. To show that $\bar{Q}$ satisfies the inclusion above for all $i \in I$, let $i \in I$ and $X \in \bar{Q} \wedge \bigwedge_{k \ll_T i} R_k$. Then $X \in \{X/\bar{E}\} \wedge \bigwedge_{k \ll_T i} R_k$ also, and $X/\bar{E} \in \bar{Q}$. The definition of $\bar{Q}$ thus implies that $X/E_i \in Q_i$.

To show that $\bar{Q}$ is largest, let Q' be any interface of signature $\bar{E}$ such that $(Q' \wedge \bigwedge_{k \ll_T i} R_k) /\!\!/ E_i \subseteq Q_i$ for all $i \in I$, and let $A \in Q'$. Since then $\{A\} \subseteq Q'$, we have, by monotonicity of conjunction,

$$(\{A\} \wedge \bigwedge_{k \ll_T i} R_k) /\!\!/ E_i \subseteq (Q' \wedge \bigwedge_{k \ll_T i} R_k) /\!\!/ E_i \subseteq Q_i$$

for all $i \in I$, and hence $A \in \bar{Q}$. Hence $Q' \subseteq \bar{Q}$. □

3.3.8 Example. Consider the cell family M consisting of the modules of the *dictionary* example:

$$M = \langle M_i \rangle_{i \in \{DICT, INOUT, INPUT, OUTPUT, STORE\}},$$

where

M_{DICT}	is given in Figure 1-7,
M_{INOUT}	is the empty module (Figure 1-14),
M_{INPUT}	is given in Figure 1-16,
M_{OUTPUT}	is given in Figure 1-18,
M_{STORE}	is given in Figure 1-19.

The signatures of these cells form the signature system T of Example 3.3.5; in particular, M is a cell system. In the previous example it was shown that with $C := \mathrm{Sig}(I_{DICT})$, we have $\Box_C T = \langle \bar{E}, \bar{D} \rangle$, where

$$\bar{E} = \mathrm{Sig}(I_{ITEM}) \sqcup \mathrm{Sig}(I_{LISTITEM}),$$
$$\bar{D} = \mathrm{Sig}(I_{ITEM}) \sqcup \mathrm{Sig}(I_{LISTITEM}) \sqcup \mathrm{Sig}(I_{DICT}).$$

Let $\bar{M} := \langle \bar{Q}, \bar{R} \rangle := \Box_C M$ be the composition of M. The following paragraphs aim at giving a more detailed presentation of the cell $\bar{M}$.

First, what is the requirement interface $\bar{Q}$ precisely? This interface must ensure that all five cells of M obtain proper bases. Syntactically, this is already guaranteed by the fact that $\bar{E}$, the signature of $\bar{Q}$, is a system site for $[M, <_T]$, so it remains to look at the properties required by the individual cells. These are quite simple: M_{DICT} and M_{INOUT} require no properties at all, the other three modules require that $bool = \{\mathbf{T}, \mathbf{F}\}$. Thus, every $\bar{E}$-model A in which $A_{bool} = \{\mathbf{T}, \mathbf{F}\}$ will guarantee that the requirements of all cells are satisfied. On the other hand, this condition is necessary, because the module M_{STORE} does not depend on any other cell (*i. e.*, $STORE$ is $<_T$-minimal), and so its requirement "$bool = \{\mathbf{T}, \mathbf{F}\}$" must be guaranteed by $\bar{Q}$. We thus have the following interface $\bar{Q}$:

interface

signature

$\mathrm{Sig}(I_{ITEM}) \sqcup \mathrm{Sig}(I_{LISTITEM})$

(containing *item*, *leitem*, *listitem*, *nil*, *cons*, *isnil*, *hd*, *tl*)

properties

bool = {T, F}.

Second, what is the result interface $\bar{R}$? The signature $\bar{D}$ of $\bar{R}$ is that of $\bar{Q}$ extended by the symbol *dictionary*. Semantically, however, this interface is quite complex, as it is obtained as the conjunction of $\bar{Q}$ and the result interfaces (*i. e.*, the code) of the five cells of the system. To determine $\bar{R}$ formally, one would have to use the mathematical denotations of the cells, that is, the semantics of their code, calculate their composition, and project this onto $\bar{D}$ to obtain $\bar{R}$.

The interface $\bar{R}$ thus obtained would describe the semantics of the function *dictionary* for *any* model of $\bar{Q}$, that is, for any interpretation of the symbols of $\bar{E}$ that satisfies "*bool* = {T, F}". There are no constraints on the interpretations of the other symbols of $\bar{E}$—a well-defined function *dictionary* is obtained even for interpretations that bear no relation whatsoever to the "intended" ones that have the properties stated in $\bar{Q}$.

We shall not delve any further into the calculation of $\bar{R}$ here, since the purpose of the theory of this thesis is to make consideration of $\bar{R}$ unnecessary—the correctness of the composed module is to be derived from the correctness of the modules from which it is composed.

More precisely, the goal of the theory of this thesis is to guarantee that $\langle \bar{Q}, \bar{R} \rangle$ is a refinement of $\langle \bar{\mathcal{Q}}, \bar{\mathcal{R}} \rangle$ (Example 3.2.11), and thus that $\langle \bar{Q}, \bar{R} \rangle$, the final result of the *dictionary* program development, is correct with respect to the external interfaces $\bar{\mathcal{Q}}$ and $\bar{\mathcal{R}}$, as shown in Figure 3-9. □

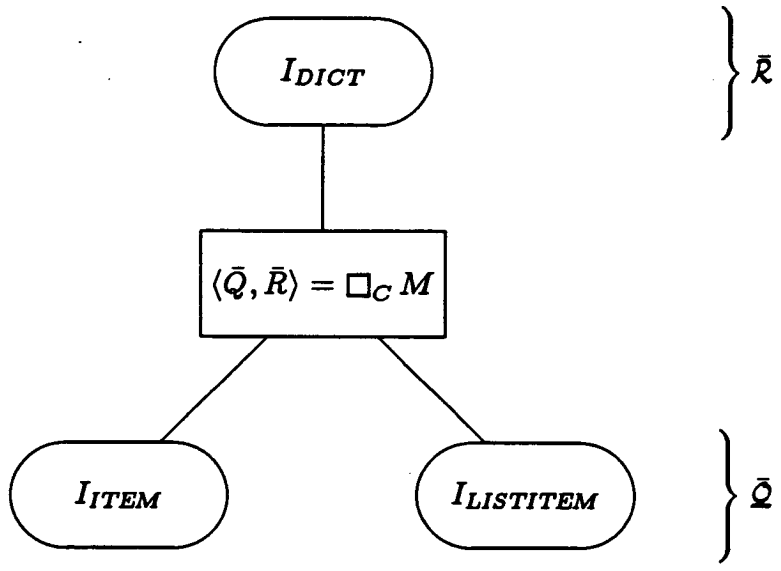


Figure 3-9: The composed module $\langle \bar{Q}, \bar{R} \rangle$ and the external interfaces $\bar{\mathcal{Q}}$ and $\bar{\mathcal{R}}$

3.4 Composability of Refinements

We have now formalized the components of the "structured correctness argument" of a modular system as shown in Figure 3-5. The specification cells of a modular design must form a decomposition (Def. 3.2.10) of the global cell that consists of the external interfaces of the system. The specification cells can then be refined (Def. 3.1.18) individually. Composing the refinements (Def. 3.3.6) yields a single cell as result; this composition operation is performed in practice by a compiler or linking loader.

The "composability theorem of refinements" now asserts that the final composed cell obtained in this manner is a refinement of the global cell that was decomposed originally; since "refinement" reflects the correctness notion of modular programming, this theorem may be regarded as asserting the soundness of the modular programming method.

Theorem 4.1.12 (Composability of Refinements).
Let $[M, <]$ *be a decomposition of the* $\langle \bar{E}, \bar{D} \rangle$*-cell* $\langle \bar{Q}, \bar{R} \rangle$*, and let* M' *be a componentwise refinement of* M*. Then* M' *is a cell system, the signature* $\bar{D}$ *is*

compatible with the join of the definition signatures of M', and

$$\square_{\bar{D}} M' \textit{ is a refinement of } \langle \bar{Q}, \bar{R} \rangle.$$

This theorem will not be proved here, because it is a corollary of the more general theorem asserting the "composability of universal implementations" (Theorem 4.1.7) to be proved in the next chapter (hence it has the number 4.1.12).

An interesting aspect of the theorem is the rôle of the dependence relation $<$. This relation appears only once in the theorem, in the phrase "Let $[M, <]$ be a decomposition ...". This means that the dependence relation $<$ is only used to verify the decomposition property of $[M, <]$, and that it is irrelevant for the remaining components of the correctness argument ("componentwise refinement" and "composition").

Note especially that the composition operation (yielding $\square_{\bar{D}} M'$) is independent of $<$; it only depends on the syntactic dependence relation of M'. This relation can be different from $<$, as the *dictionary* example shows: compare the dependence relation $<$ (Figure 3-7) of the ordered specification system with the syntactic dependence relation $<_M$ (Figure 3-8) of the concrete system.

Practical considerations make it essential that the composition operation use only the syntactic dependence relation: Composition is normally performed by compilation or linking, and it is the syntactic dependence relation that is naturally available at this stage. If composition were to use the dependence relation on which the decomposition was based, it would become necessary to specify this ordering to the compiler or linker.

With the (not-yet-proven) theorem about the composability of refinements, the present chapter has presented a theory that is similar in scope to theories developed previously by myself [Schoett 81] and by Back and Mannila [BM 84].

The theory I developed in 1981 [Schoett 81] anticipates some of the features of the present theory; in particular, the idea of analysing modularization by dealing with "modules" and "module specifications" that have specific environment and result signatures, and the idea of structuring the correctness argument for a

modular program according to Figure 3-5. Thus, concepts of "decomposition" and "composition" occur already in the theory of 1981.

Compared to the present theory, however, the theory of 1981 is limited in scope: it deals exclusively with the institution $\langle$**ASig, AIncl, Alg**$\rangle$, its "specification" concept is restricted to one particular specification language, the refinements of a specification must be modules with trivial requirement (*i. e.*, $\langle \bar{E}, \bar{D} \rangle$-modules of the form $\langle Mod(\bar{E}), \bar{R} \rangle$ rather than arbitrary cells), the composition operation is only defined for such modules (this makes it trivial to determine the requirement of the composed cell), and cell systems must be finite.

In particular, I did not realize in 1981 that a "module" can be treated as a special case of a "module specification" (*i. e.*, of a cell), and that it is possible to allow arbitrary cells in place of "modules". In Chapter 5 below, we will encounter results that could not have been formulated without the unification of "modules" and "module specifications" embodied in the cell concept (*e. g.*, Theorem 5.1.11).

It is a bit more difficult to relate the concepts of the theory of Back and Mannila [BM 84] (which will be called the "BM theory" from now on) to the concepts of this thesis. At first, one encounters the following restrictions in the BM theory:

- The theory is "typeless", that is, every symbol can refer to any semantic value. Hence, the BM theory essentially deals with the institution $\langle$**Set, SetIncl, SetMod**$\rangle$ only.
- The modules of the BM theory (called "declarations") define exactly one new symbol, and assign a value to it for every possible environment. Hence, they are cells with trivial requirement.
- A "specification" describes the value of exactly one symbol; that is, relationships between the interpretations of different symbols cannot be specified.

Nevertheless, the BM theory covers similar ground to the present theory. In particular, the concept of "locality" is analogous to the concept of the "Correct-

ness of Modular Programming": "Locality" means that the "local" correctness of a system (each module defines an entity satisfying the result specification, provided it is supplied with an environment that satisfies the specifications of the environment symbols) implies "global" correctness (the composition of the modules is correct); and so the "locality" theorem of Back and Mannila that

"Finite hierarchical modularization mechanisms are local"

[BM 84, Corollary 7] is analogous to the composability theorem for refinements.

In particular, the term "hierarchical" in the theorem of Back and Mannila means that systems must have a well-founded dependence ordering, as they must in the theory of this thesis. It would seem that the well-foundedness requirement should enable the BM theory to cope with infinite systems as well; however, the "locality" theorem of the BM theory cannot be generalized to infinite systems.

The reason is that the BM theory employs a semantic notion of "dependence" between modules: A module A depends on a module B if varying the result of B can have an effect on the result of A. This definition does not work well in infinite systems: If, for example, one defines

$$y := \begin{cases} 1, & \text{if infinitely many of the } x_i \ (i \in \mathbf{N}) \text{ are } 1 \\ 0, & \text{otherwise,} \end{cases}$$

then this definition would *not* depend on any of the x_i according to the BM theory. Hence, the following is a (BM-)hierarchical system of declarations: define for $i \in \mathbf{N}$

$$x_i := \begin{cases} 1, & \text{if infinitely many of the } x_i \ (i \in \mathbf{N}) \text{ are } 1 \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

If we "specify" the x_i by postulating that

$$x_i = 0 \qquad \text{for all } i \in \mathbf{N}, \tag{2}$$

then each of the declarations above is locally correct, because if all the x_i satisfy (2), then (1) defines a correct value for every x_i. However, the system (1) is also locally correct with respect to the specification

$$x_i = 1 \qquad \text{for all } i \in \mathbf{N}, \tag{3}$$

and since the solution of (1) cannot possibly satisfy both (2) and (3), locality fails for infinite systems in the BM theory.

A point in which the theory of Back and Mannila is more general than the theory of this thesis is that it is not restricted to systems with a well-founded dependence relation, but also allows recursive dependencies in systems.

It was my deliberate choice not to try to deal with recursive systems in this thesis, because I regard them as incompatible with the basic goal of modularization, which is to separate the correctness argument for a system into separate correctness arguments for its individual modules.

The following example illustrates that such a separation is not possible for recursive systems. Consider the result interface

$F =$ **interface**

signature

nat: **sort**

f: $nat \rightarrow nat$

properties

$nat = \mathbf{N}$

$f(0) = 0$

and assume we are given the following interface as a requirement interface:

$G =$ **interface**

signature

nat: **sort**

g: $nat \rightarrow nat$

properties

$nat = \mathbf{N}$

$g(0) = 0.$

It is then trivial to program a module that is a correct refinement of the cell $\langle G, F \rangle$:

$\mathcal{F} =$ **module**

environment signature

nat: **sort**

g: $nat \rightarrow nat$

defined symbols

f: $nat \rightarrow nat$

requirements

⟨none⟩

result

$f = g$

(some programming notations might require one to code this in a more complicated form like "**funct** $f(x$: **nat**$)$: **nat** $= g(x)$").

If recursive dependencies between modules are allowed, we can define the function g by a module that uses the function f:

$\mathcal{G}$ = **module**

environment signature

nat: **sort**

f: $nat \rightarrow nat$

defined symbols

g: $nat \rightarrow nat$

requirements

⟨none⟩

result

$g = f$.

This module $\mathcal{G}$ is a correct refinement of the cell $\langle F, G \rangle$. Hence, we have the design graph shown in Figure 3-10. Each of the modules $\mathcal{F}$ and $\mathcal{G}$ in this figure is correct with respect to its requirement and result interfaces.

However, composing the cells $\mathcal{F}$ and $\mathcal{G}$ essentially means to compose the two definitions

$$f = g$$

$$g = f.$$

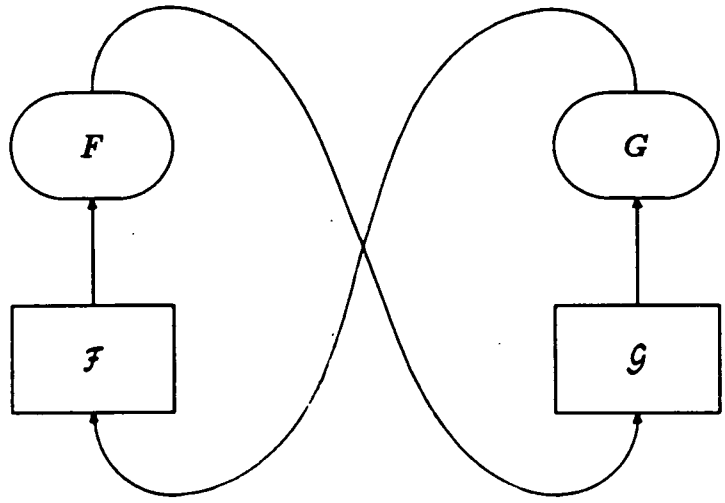


Figure 3-10: Design graph of two mutually recursive function definitions

This system of equations is satisfied by all models in which $f = g$, and there is no guarantee that the interfaces F and G will be satisfied in the solution.

In a concrete programming language, for example, the composition of the modules $\mathcal{F}$ and $\mathcal{G}$ would normally result in both f and g being undefined everywhere, so that neither of the interfaces F and G is satisfied.

Consider also that the design graph in Figure 3-10 remains correct if the interfaces F and G are replaced by

$$F' \quad \text{specifying} \quad f(0) = 1, \quad \text{and}$$
$$G' \quad \text{specifying} \quad g(0) = 1.$$

Clearly, the composition of $\mathcal{F}$ and $\mathcal{G}$ cannot both satisfy the interfaces F and G and the interfaces F' and G'.

The problems of recursion also manifest themselves in the BM theory. In order to obtain a locality result for non-hierarchical (in particular, recursive) systems, the theory imposes the requirement that specifications (*i. e.*, interfaces) must be "continuous" with respect to the partial order between semantic values that is employed in solving recursive systems of definitions. This continuity requirement severely restricts the expressive power of specifications. For example, in

the standard case of the recursive definition of partial functions, where the partial ordering is graph inclusion, continuous specifications must always admit the function that is undefined everywhere. As a consequence, continuous specifications are unable to express that a function should yield a result (*i. e.*, terminate).

This continuity requirement rules out the example presented above, because the interfaces F and G are not continuous—if they were continuous, they would at least have to admit the totally undefined functions, and the solution produced in concrete programming notations (f and g totally undefined) would then be correct.

I feel, however, that specifications that always admit the totally undefined function are too weak to be practically useful. For example, any system specified in this way could simply be realized by declaring all functions to be undefined everywhere. Even the simplest form of termination requirement in a specification, however, seems incompatible with recursive dependencies between modules, as illustrated by the example above.

Hence, I hope that the reader does not feel too uncomfortable with the fact that the theory of this thesis deals only with systems that have a well-founded dependence relation. It should perhaps be pointed out that this does *not* preclude recursively defined functions in modules, as long as the cycles of the call graph do not cross module boundaries. Functions that call each other (or themselves) recursively within a module do not cause problems, as these recursive definitions can always be solved locally to determine the semantics of the module in the form of a cell (this was discussed in Example 3.1.14).

In the remainder of this section, the following theorem is proved, which states the "syntactical component" of the theorem of the composability of refinements.

3.4.1 Theorem (Syntactic Correctness of Modular Programming).
Let $[T, <]$ *be a syntactic decomposition of the cell signature* $\langle \bar{E}, \bar{D} \rangle$, *and let* T' *be a componentwise syntactic refinement of* T. *Then* T' *is a signature system, the signature* $\bar{D}$ *is compatible with the join of the result signatures of* T', *and* $\square_{\bar{D}} T'$ *is a syntactic refinement of* $\langle \bar{E}, \bar{D} \rangle$.

For the proof, two lemmas are required.

3.4.2 Lemma. *Let $[T, <]$ be an ordered signature system, and let T' be a componentwise syntactic refinement of T, where $T = \langle E_i, D_i\rangle_{i\in I}$ and $T' = \langle E'_i, D'_i\rangle_{i\in I}$. Whenever $\bar{E}$ is a system site for $[T, <]$ and $K \subseteq I$ is $<$-downward closed, then*

$$\bar{E} \sqcup \bigsqcup_{k\in K} D_k = \bar{E} \sqcup \bigsqcup_{k\in K} D'_k.$$

This lemma states that the "result signature" obtained by installing a $<$-downward closed subset of the cell signatures of T on a system site $\bar{E}$ is the same as the one obtained by installing the analogous subset of the cell signatures of T'.

The proof of this lemma uses "Zorn's lemma" from set theory (see [Barwise 77, p. 355], [Levy 79, p. 161]):

> A partial order in which every chain has an upper bound has a maximal element.

Recall that a *chain* in a partial order $\langle M, \sqsubseteq\rangle$ is a subset of M that is totally ordered by $\sqsubseteq$, and a *maximal* element is one for which no strictly larger element exists, *i. e.*, $x \in M$ is maximal iff for all $y \in M$: $y \sqsupseteq x \Rightarrow y = x$.

Zorn's lemma will be used frequently in the remainder of this thesis; it is equivalent to the Axiom of Choice in the presence of the other axioms of ZFC (*e. g.*, [Levy 79, p. 162], [Barwise 77, p. 355]).

Proof of Lemma 3.4.2.
Let $T = \langle E_i, D_i\rangle_{i\in I}$, $T' = \langle E'_i, D'_i\rangle_{i\in I}$, and $<$ be defined as in the lemma; let $\bar{E}$ be a system site for $[T, <]$, and let $K \subseteq I$ be $<$-downward closed.

Note first that for all $i \in I$,

$$E'_i \sqcup D'_i \sqsubseteq E_i \sqcup D_i \sqsubseteq \bar{E} \sqcup \bigsqcup_{i\in I}(E_i \sqcup D_i)$$

by Proposition 3.1.17 (b), and so $\langle E'_i \sqcup D'_i\rangle_{i\in I}$ is compatible, and the join of this family is compatible with $\bar{E}$.

Consider now the set $\mathcal{U}$ which consists of all sets $L \subseteq K$ such that L is $<$-downward closed and $\bar{E} \sqcup \bigsqcup_{k \in L} D_k = \bar{E} \sqcup \bigsqcup_{k \in L} D'_k$. Let $\mathcal{U}$ be partially ordered by $\subseteq$.

Every chain in $\mathcal{U}$ has an upper bound in $\mathcal{U}$, for if $\langle L_m \rangle_{m \in M}$ is a family of elements of $\mathcal{U}$ (in particular, a chain), then $\bar{L} := \bigcup_{m \in M} L_m$ is in $\mathcal{U}$, because $\bar{L}$ is obviously $<$-downward closed, and

$$\begin{aligned}
\bar{E} \sqcup \bigsqcup_{k \in \bar{L}} D_k &= \bar{E} \sqcup \bigsqcup_{m \in M} \left(\bar{E} \sqcup \bigsqcup_{k \in L_m} D_k \right) \\
&= \bar{E} \sqcup \bigsqcup_{m \in M} \left(\bar{E} \sqcup \bigsqcup_{k \in L_m} D'_k \right) \\
&= \bar{E} \sqcup \bigsqcup_{k \in \bar{L}} D'_k .
\end{aligned}$$

According to Zorn's lemma, we can therefore pick a $\subseteq$-maximal element L of $\mathcal{U}$. It will now be shown that the assumption that $L \neq K$ leads to a contradiction. Hence L must be equal to K, which implies that $K \in \mathcal{U}$, and hence verifies the lemma.

Suppose that $L \neq K$. Choose a $<$-minimal element j in $L \setminus K$ ($<$ is well-founded). Then $L^+ := L + \{j\}$ also is in $\mathcal{U}$: It is easy to see that L^+ is $<$-downward closed. Since $\bar{E}$ is a system site for $[T, <]$, $\bar{E} \sqcup \bigsqcup_{k \ll j} D_k$ is a site for $\langle E_j, D_j \rangle$. Since $\langle E'_j, D'_j \rangle$ is a syntactic refinement of $\langle E_j, D_j \rangle$, we have

$$\left(\bar{E} \sqcup \bigsqcup_{k \ll j} D_j \right) \sqcup D_j = \left(\bar{E} \sqcup \bigsqcup_{k \ll j} D_k \right) \sqcup D'_j .$$

Hence,

$$\begin{aligned}
\bar{E} \sqcup \bigsqcup_{k \in L^+} D_k &= \left(\bar{E} \sqcup \bigsqcup_{k \ll j} D_k \sqcup D_j \right) \sqcup \bigsqcup_{k \in L} D_k \\
&= \left(\bar{E} \sqcup \bigsqcup_{k \ll j} D_k \sqcup D'_j \right) \sqcup \bigsqcup_{k \in L} D_k \\
&= \left(\bar{E} \sqcup \bigsqcup_{k \in L} D_k \right) \sqcup D'_j \\
&= \left(\bar{E} \sqcup \bigsqcup_{k \in L} D'_k \right) \sqcup D'_j \qquad (L \in \mathcal{U}) \\
&= \bar{E} \sqcup \bigsqcup_{k \in L^+} D'_k .
\end{aligned}$$

Hence $L^+ \in \mathcal{U}$, which contradicts the definition of L as a maximal element of $\mathcal{U}$. □

The next lemma describes in more detail the relation between an ordered signature system $[T, <]$ and a componentwise syntactical refinement T' of T.

3.4.3 Lemma. *Let $[T, <]$ be an ordered signature system, and let T' be a componentwise syntactic refinement of T. Then*

(a) *T' is a compatible family of cell signatures,*

(b) $<_{T'} \subseteq <_T \subseteq <$,

(c) *every system site for $[T, <]$ is a system site for $[T', <_{T'}]$,*

(d) *T' is a signature system.*

Proof. Let $[T, <]$ be an ordered signature system where $T = \langle E_i, D_i \rangle_{i \in I}$, and let $T' = \langle E'_i, D'_i \rangle_{i \in I}$ be a componentwise syntactic refinement of T.

Clause (a): By Proposition 3.1.17 (b), we have $E'_i \sqcup D'_i \sqsubseteq E_i \sqcup D_i$ for all $i \in I$, and since the family $\langle E_i \sqcup D_i \rangle_{i \in I}$ is compatible, so is $\langle E'_i \sqcup D'_i \rangle_{i \in I}$.

Clause (b): Since $<$ is a dependence relation for T, we have $<_T \subseteq <$.

Suppose now that $<_{T'}$ is not included in $<_T$. Then we can choose k and i in I such that $k <_{T'} i$ and $k \not<_T i$. From $k <_{T'} i$ it follows that $k \neq i$, and hence $k \not<_T i$ implies that

$$D_k \sqcap (E_i \sqcup D_i) \sqsubseteq E_k.$$

Hence, we have

$$\begin{aligned} D'_k \sqcap (E'_i \sqcup D'_i) &\sqsubseteq (E'_k \sqcup D'_k) \sqcap (E'_i \sqcup D'_i) \\ &\sqsubseteq (E_k \sqcup D_k) \sqcap (E_i \sqcup D_i) \qquad \text{(Prop. 3.1.17 (b))} \\ &= (E_k \sqcap (E_i \sqcup D_i)) \sqcup (D_k \sqcap (E_i \sqcup D_i)) \\ &\sqsubseteq E_k. \end{aligned}$$

Therefore,

$$\begin{aligned} D'_k \sqcap (E'_i \sqcup D'_i) &= D'_k \sqcap (E'_i \sqcup D'_i) \sqcap E_k \\ &\sqsubseteq D'_k \sqcap E_k \end{aligned}$$

$$\sqsubseteq (E'_k \sqcup D'_k) \sqcap E_k$$
$$= E'_k$$
(E_k is a site for $\langle E'_k, D'_k \rangle$ by Prop. 3.1.17 (c)).

This implies that $k \not<_{T'} i$, which is a contradiction.

Clause (c): Let $\bar{E}$ be a system site for $[T, <]$. Because $\bar{E}$ is compatible with $\bigsqcup_{i \in I}(E_i \sqcup D_i)$ by Lemma 3.3.2, and $D'_i \sqsubseteq E_i \sqcup D_i$ for all $i \in I$ by Proposition 3.1.17 (b), the signatures $\bar{E}$ and $\bigsqcup_{i \in I} D'_i$ are compatible.

For every $i \in I$, $\bar{E} \sqcup \bigsqcup_{k \ll i} D_k$ is a site for $\langle E_i, D_i \rangle$ and hence for $\langle E'_i, D'_i \rangle$. From Lemma 3.4.2 it follows that

$$\bar{E} \sqcup \bigsqcup_{k \ll i} D'_k \text{ is a site for } \langle E'_i, D'_i \rangle. \tag{1}$$

To show that $\bar{E}$ is a system site for $[T', <_{T'}]$, we have to show that for all $i \in I$,

$$\bar{E} \sqcup \bigsqcup_{k \ll_{T'} i} D'_k \text{ is a site for } \langle E'_i, D'_i \rangle \tag{2}$$

(compare this with (1)! This is one place where the different dependence relations of a system and its refinement complicate matters).

Suppose (2) was false for some $i \in I$. Then we can choose j to be $\ll$-minimal among the values of i for which (2) fails.

From (1) it follows that

$$(\bar{E} \sqcup \bigsqcup_{k \ll_{T'} j} D'_k) \sqcap (E'_j \sqcup D'_j)$$
$$\sqsubseteq (\bar{E} \sqcup \bigsqcup_{k \ll j} D'_k) \sqcap (E'_j \sqcup D'_j) \qquad (\text{as } <_{T'} \subseteq <)$$
$$= E'_j,$$

and it remains to show the converse of this inclusion. For this, it is sufficient to show that

$$E'_j \sqsubseteq \bar{E} \sqcup \bigsqcup_{k \ll_{T'} j} D'_k.$$

It will be shown below that for every $m \ll j$,

$$D'_m \sqcap E'_j \sqsubseteq \bar{E} \sqcup \bigsqcup_{k \ll_{T'} j} D'_k. \tag{3}$$

From this, it follows that

$$\begin{aligned} E'_j &= (\bar{E} \sqcup \bigsqcup_{m \ll j} D'_m) \sqcap E'_j \\ &= (\bar{E} \sqcap E'_j) \sqcup \bigsqcup_{m \ll j} (D'_m \sqcap E'_j) \\ &\sqsubseteq \bar{E} \sqcup \bigsqcup_{k \ll_{T'} j} D'_k. \end{aligned}$$

This proves that $\bar{E} \sqcup \bigsqcup_{k \ll_{T'} j} D'_k$ is a site for $\langle E'_j, D'_j \rangle$ and contradicts the definition of j. Hence (2) is proved and thus that $\bar{E}$ is a system site for $[T', <_{T'}]$.

It remains to prove (3). Suppose that (3) is false. Then we can choose n to be $\ll_{T'}$-minimal among the values of m that falsify (3). It follows that $n \not<_{T'} j$, because otherwise the right hand side of (3) would include D'_n. Since $n \ll j$, we have $n \neq j$, and hence $n \not<_{T'} j$ implies that

$$D'_n \sqcap E'_j \sqsubseteq D'_n \sqcap (E'_j \sqcup D'_j) \sqsubseteq E'_n. \tag{4}$$

Also, since $n \ll j$, the minimality of j implies that n satisfies (2) when substituted for i, *i. e.*,

$$(\bar{E} \sqcup \bigsqcup_{k \ll_{T'} n} D'_k) \text{ is a site for } \langle E'_n, D'_n \rangle.$$

In particular,

$$E'_n \sqsubseteq \bar{E} \sqcup \bigsqcup_{k \ll_{T'} n} D'_k. \tag{5}$$

Hence,

$$\begin{aligned} D'_n \sqcap E'_j &= D'_n \sqcap E'_j \sqcap E'_n && \text{[by (4)]} \\ &\sqsubseteq E'_j \sqcap E'_n \\ &\sqsubseteq E'_j \sqcap (\bar{E} \sqcup \bigsqcup_{k \ll_{T'} n} D'_k) && \text{[by (5)]} \\ &= (E'_j \sqcap \bar{E}) \sqcup \bigsqcup_{k \ll_{T'} n} (E'_j \sqcap D'_k) \end{aligned}$$

Because each value of k in this expression satisfies (3) when substituted for m (by minimality of n), this signature is included in $\bar{E} \sqcup \bigsqcup_{k \ll_{T'} j} D'_k$. Thus, n is a value for m that satisfies (3), which contradicts the definition of n. This proves (3) and concludes the proof of clause (c) of the lemma.

Clause (d): This follows trivially from (a), (b) and (c): $<_{T'}$ is well-founded because it is included in the well-founded relation $<$. Since $[T, <]$ is a signature system, there exists a system site for it; by (c), that system site is also a system site for $[T', <_{T'}]$. □

Proof of Theorem 3.4.1 (the Syntactic Correctness Theorem).
Let $[T, <]$ be a syntactic decomposition of the cell signature $\langle \bar{E}, \bar{D} \rangle$, where $T = \langle E_i, D_i \rangle_{i \in I}$, and let $T' = \langle E'_i, D'_i \rangle_{i \in I}$ be a componentwise syntactic refinement of T.

By Lemma 3.4.3 (d), T' is a signature system. The signatures $\bar{D}$ and $\bigsqcup_{i \in I} D'_i$ are compatible, because they are included in $\bar{E} \sqcup \bigsqcup_{i \in I} D_i$: We have $\bar{D} \sqsubseteq \bar{E} \sqcup \bigsqcup_{i \in I} D_i$, because $[T, <]$ is a syntactic decomposition of $\langle \bar{E}, \bar{D} \rangle$, and for all $i \in I$, we have $D'_i \sqsubseteq \bar{E} \sqcup \bigsqcup_{i \in I} D_i$, because $D'_i \sqsubseteq E_i \sqcup D_i$ by Proposition 3.1.17 (b), and $E_i \sqcup D_i \sqsubseteq \bar{E} \sqcup \bigsqcup_{i \in I} D_i$ by Lemma 3.3.2.

It remains to show that $\square_{\bar{D}} T'$ is a syntactic refinement of $\langle \bar{E}, \bar{D} \rangle$. Let $\langle \bar{E}', \bar{D}' \rangle := \square_{\bar{D}} T'$, so that

$$\bar{E}' \text{ is the smallest system site for } [T', <_{T'}], \text{ and}$$
$$\bar{D}' = \bar{E}' \sqcup (\bar{D} \sqcap \bigsqcup_{i \in I} D'_i).$$

To show that $\langle \bar{E}', \bar{D}' \rangle$ is a syntactic refinement of $\langle \bar{E}, \bar{D} \rangle$, we shall use Proposition 3.1.17 (c), which requires us to verify that $\bar{E}$ is a site for $\langle \bar{E}', \bar{D}' \rangle$, and that $\bar{E} \sqcup \bar{D}' = \bar{E} \sqcup \bar{D}$.

Note that $\bar{E}' \sqsubseteq \bar{E}$, because $\bar{E}$ is a system site for $[T', <_{T'}]$ by Lemma 3.4.3 (c), and $\bar{E}'$ is the smallest such system site. Therefore,

$$\begin{aligned} \bar{E} \sqcap (\bar{E}' \sqcup \bar{D}') &= \bar{E} \sqcap (\bar{E}' \sqcup (\bar{D} \sqcap \bigsqcup_{i \in I} D'_i)) \\ &= (\bar{E} \sqcap \bar{E}') \sqcup (\bar{E} \sqcap \bar{D} \sqcap \bigsqcup_{i \in I} D'_i) \end{aligned}$$

$$= \bar{E}' \sqcup (\bar{E} \sqcap \bar{D} \sqcap \bigsqcup_{i \in I} (\bar{E} \sqcap D'_i)).$$

This expression equals $\bar{E}'$, because $\bar{E} \sqcap D'_i \sqsubseteq \bar{E}'$ for all $i \in I$ by Lemma 3.3.3. Hence, $\bar{E}$ is a site for $\langle \bar{E}', \bar{D}' \rangle$.

Finally,

$$\begin{aligned}
\bar{E} \sqcup \bar{D}' &= \bar{E} \sqcup \bar{E}' \sqcup (\bar{D} \sqcap \bigsqcup_{i \in I} D'_i) \\
&= \bar{E} \sqcup (\bar{D} \sqcap \bigsqcup_{i \in I} D'_i) \\
&= \bar{E} \sqcup (\bar{D} \sqcap \bar{E}) \sqcup (\bar{D} \sqcap \bigsqcup_{i \in I} D'_i) \\
&= \bar{E} \sqcup (\bar{D} \sqcap (\bar{E} \sqcup \bigsqcup_{i \in I} D'_i)) \\
&= \bar{E} \sqcup (\bar{D} \sqcap (\bar{E} \sqcup \bigsqcup_{i \in I} D_i)) \qquad \text{(Lemma 3.4.2)},
\end{aligned}$$

and this is equal to $\bar{E} \sqcup \bar{D}$, because $[T, <]$ is a syntactic refinement of $\langle \bar{E}, \bar{D} \rangle$, and hence $\bar{D} \sqsubseteq \bar{E} \sqcup \bigsqcup_{i \in I} D_i$.

This verifies the conditions of Proposition 3.1.17 (c), and thus shows that $\Box_{\bar{D}} T' = \langle \bar{E}', \bar{D}' \rangle$ is a syntactic refinement of $\langle \bar{E}, \bar{D} \rangle$. □

3.5 The Relation between Decomposition and Composition

The composability theorem for refinements (Theorem 4.1.12, discussed in the previous section) asserts that the "structured correctness argument" for a modular system shown in Figure 3-5 is sound if "refinement" is used as the correctness notion: a componentwise refinement of a decomposition of a global cell yields a refinement of the global cell when composed. This section proves a converse theorem, which asserts that under certain syntactic restrictions, every system whose composition is a refinement of some global cell is a decomposition of that global cell.

Another theorem to be proved here is that composition of cells preserves single-valuedness and consistency (and hence the property of being a module).

Finally, it is shown that for ordered systems whose cells are consistent, the decomposition notion is independent of the dependence ordering.

An example illustrates the effects of inconsistent cells.

Many of the theorems of this section depend on the Composability Theorem for Refinements, which has not yet been proven. This does not lead to logical circularity, because the theorems of this section will not be used in Section 4.1, where the Composability Theorem is proved.

The following theorem characterizes the relation between the syntactic decomposition and composition notions.

3.5.1 Theorem. *An ordered signature system* $[T, <]$, *where* $T = \langle E_i, D_i \rangle_{i \in I}$, *is a syntactic decomposition of a cell signature* $\langle \bar{E}, \bar{D} \rangle$, *if and only if*

(a) $\bar{E} \sqcup \bar{D}$ *is compatible with* $\bigsqcup_{i \in I} D_i$,

(b) $\bar{E} \sqcap D_i \sqsubseteq E_i$ *for all* $i \in I$, *and*

(c) $\square_{\bar{D}} T$ *is a syntactic refinement of* $\langle \bar{E}, \bar{D} \rangle$.

The proof is based on the following lemma.

3.5.2 Lemma. *Let* $T = \langle E_i, D_i \rangle_{i \in I}$ *be a signature system, and let* $\bar{E}$ *be a system site for* $[T, <_T]$. *If* $K \subseteq I$ *is* $<_T$*-downward closed, and* j *is* $<_T$*-minimal in* $I \setminus K$, *then*

$$\bar{E} \sqcup \bigsqcup_{k \in K} D_k \text{ is a site for } \langle E_j, D_j \rangle.$$

Proof. Let $K \subseteq I$ be $<_T$-downward closed, and let j be $<_T$-minimal in $I \setminus K$. The signatures $\bar{E} \sqcup \bigsqcup_{k \in K} D_k$ and $E_j \sqcup D_j$ are compatible, because $\bar{E}$ and $\bigsqcup_{i \in I}(E_i \sqcup D_i)$ are compatible according to Lemma 3.3.2. Since K is

$<_T$-downward closed, we have $K \supseteq \{ k \mid k \ll_T j \}$, and hence

$$E_j = (\bar{E} \sqcup \bigsqcup_{k \ll_T j} D_k) \sqcap (E_j \sqcup D_j) \tag{1}$$
$$\sqsubseteq (\bar{E} \sqcup \bigsqcup_{k \in K} D_k) \sqcap (E_j \sqcup D_j).$$

It remains to show the converse inclusion. From the equation (1), it follows that

$$\bar{E} \sqcap (E_j \sqcup D_j) \sqsubseteq E_j, \tag{2}$$

and hence it is sufficient to show that for all $k \in K$,

$$D_k \sqcap (E_j \sqcup D_j) \sqsubseteq E_j. \tag{3}$$

Suppose that (3) is false. Then we can choose m to be a $\ll_T$-minimal element of K such that

$$D_m \sqcap (E_j \sqcup D_j) \not\sqsubseteq E_j.$$

Due to (1), it cannot be the case that $m \ll_T j$, hence $m \not\ll_T j$. In particular, $m \not<_T j$, and since $m \in K$ and hence $m \neq j$, this implies that

$$D_m \sqcap (E_j \sqcup D_j) \sqsubseteq E_m \sqsubseteq \bar{E} \sqcup \bigsqcup_{k \ll_T m} D_k.$$

Hence

$$\begin{aligned} D_m \sqcap (E_j \sqcup D_j) &\sqsubseteq (\bar{E} \sqcup \bigsqcup_{k \ll_T m} D_k) \sqcap (E_j \sqcup D_j) \\ &= (\bar{E} \sqcap (E_j \sqcup D_j)) \sqcup \bigsqcup_{k \ll_T m} (D_k \sqcap (E_j \sqcup D_j)) \\ &\sqsubseteq E_j \sqcup \bigsqcup_{k \ll_T m} (D_k \sqcap (E_j \sqcup D_j)) \qquad \text{[by (2)]} \\ &= E_j, \end{aligned}$$

since for $k \ll_T m$, we have $k \in K$ (K is $<_T$-downward closed) and hence $D_k \sqcap (E_j \sqcup D_j) \sqsubseteq E_j$ by minimality of m. This contradicts the definition of m, and hence proves (3). It follows that

$$E_j = (\bar{E} \sqcup \bigsqcup_{k \in K} D_k) \sqcap (E_j \sqcup D_j). \qquad \square$$

Proof of Theorem 3.5.1.

Let $[T, <]$ be an ordered signature system, where $T = \langle E_i, D_i\rangle_{i\in I}$, and let $\langle \bar{E}, \bar{D}\rangle$ be a cell signature

Suppose first that $[T, <]$ is a syntactic decomposition of $\langle \bar{E}, \bar{D}\rangle$. Then $\bar{E}$ is a system site for $[T, <]$ and hence compatible with $\bigsqcup_{i\in I} D_i$, and $\bar{D} \sqsubseteq \bar{E} \sqcup \bigsqcup_{i\in I} D_i$, which proves (a). Also, we have for all $i \in I$:

$$\bar{E} \sqcap D_i \sqsubseteq \Big(\bar{E} \sqcup \bigsqcup_{k\ll i} D_k\Big) \sqcap (E_i \sqcup D_i) = E_i,$$

and thus (b) holds. Finally, Theorem 3.4.1 (applied with $T' = T$) yields (c).

Conversely, suppose that (a), (b) and (c) hold. Let $\langle \bar{E}', \bar{D}'\rangle := \square_{\bar{D}} T$. Then

$$\bar{E}' \text{ is the smallest system site for } [T, <_T], \text{ and}$$
$$\bar{D}' = \bar{E}' \sqcup \Big(\bar{D} \sqcap \bigsqcup_{i\in I} D_i\Big),$$

and, according to (c):

$$\bar{E} \text{ is a site for } \langle \bar{E}', \bar{D}'\rangle$$
$$\text{(in particular, } \bar{E}' \sqsubseteq \bar{E}\text{), and}$$
$$\bar{E} \sqcup \bar{D}' = \bar{E} \sqcup \bar{D}.$$

To show that $[T, <]$ is a syntactic decomposition of $\langle \bar{E}, \bar{D}\rangle$, we have to show that $\bar{E}$ is a system site for $[T, <]$, and that $\bar{D} \sqsubseteq \bar{E} \sqcup \bigsqcup_{i\in I} D_i$.

From (a) it follows that $\bar{E}$ and $\bigsqcup_{i\in I} D_i$ are compatible, and by Lemma 3.3.2, we have

$$\bigsqcup_{i\in I}(E_i \sqcup D_i) \sqsubseteq \bar{E}' \sqcup \bigsqcup_{i\in I} D_i \sqsubseteq \bar{E} \sqcup \bigsqcup_{i\in I} D_i,$$

hence $\bar{E}$ and $\bigsqcup_{i\in I}(E_i \sqcup D_i)$ are compatible.

For every $i \in I$, we have

$$\begin{aligned}
\Big(\bar{E} \sqcup \bigsqcup_{k\ll i} D_k\Big) \sqcap (E_i \sqcup D_i) &= \Big(\bar{E} \sqcup \bar{E}' \sqcup \bigsqcup_{k\ll i} D_k\Big) \sqcap (E_i \sqcup D_i) \\
&= (\bar{E} \sqcap (E_i \sqcup D_i)) \sqcup \Big(\big(\bar{E}' \sqcup \bigsqcup_{k\ll i} D_k\big) \sqcap (E_i \sqcup D_i)\Big) \\
&= (\bar{E} \sqcap (E_i \sqcup D_i)) \sqcup E_i \qquad \text{(Lemma 3.5.2)} \\
&= E_i \qquad \text{(by (b))}.
\end{aligned}$$

Hence $\bar{E}$ is a system site for $[T, <]$.

Finally,

$$\begin{aligned}\bar{D} &\sqsubseteq \bar{E} \sqcup \bar{D} \\ &= \bar{E} \sqcup \bar{D}' \\ &= \bar{E} \sqcup \bar{E}' \sqcup (\bar{D} \sqcap \bigsqcup_{i \in I} D_i) \\ &= \bar{E} \sqcup (\bar{D} \sqcap \bigsqcup_{i \in I} D_i) \\ &\sqsubseteq \bar{E} \sqcup \bigsqcup_{i \in I} D_i.\end{aligned}$$

Hence, $[T, <]$ is a syntactic decomposition of $\langle \bar{E}, \bar{D} \rangle$. □

Theorem 3.5.1 shows that every signature system whose composition is correct with respect to a global cell signature (clause (c)) and that satisfies the additional conditions (a) and (b) is a syntactic decomposition of the global cell signature. The additional condition (a) says that the system's internal symbols ($\bigsqcup_{i \in I} D_i$) must be compatible with the global cell signature, condition (b) says that no part of the global environment signature $\bar{E}$ may be redefined inside the system.

These additional conditions are needed, because the decomposition notion is formulated using just a single "name space", that is, a single signature which contains both the external and the internal symbols of the system. A more complex "decomposition" concept, where the external and internal signatures are connected by signature morphisms, might remedy the need for such additional conditions.

As a corollary of Theorem 3.5.1, we obtain that every system is a syntactic decomposition of its composition.

3.5.3 Theorem. *Let $[T, <]$ be an ordered signature system, and let C be a signature compatible with the join of the definition signatures of T. Then $[T, <]$ is a syntactic decomposition of $\square_C T$.*

Proof. Let $\langle \bar{E}, \bar{D} \rangle := \square_C T$. We verify the three conditions of Theorem 3.5.1.

(a) $\bar{E} \sqcup \bar{D} = \bar{D} = \bar{E} \sqcup (C \sqcap \bigsqcup_{i \in I} D_i) \sqsubseteq \bar{E} \sqcup \bigsqcup_{i \in I} D_i$,

hence $\bar{E} \sqcup \bar{D}$ is compatible with $\bigsqcup_{i \in I} D_i$;

(b) for $i \in I$, we have

$$\bar{E} \sqcap D_i \sqsubseteq (\bar{E} \sqcup \bigsqcup_{j \ll_T i} D_j) \sqcap (E_i \sqcup D_i) = E_i,$$

as $\bar{E}$ is a system site for $[T, <_T]$;

(c) the environment signature of $\square_{\bar{D}} T$ is the smallest system site for $[T, <_T]$, *i.e.*, $\bar{E}$; the result signature of $\square_{\bar{D}} T$ is

$$\begin{aligned}
\bar{E} \sqcup (\bar{D} \sqcap \bigsqcup_{i \in I} D_i) &= \bar{E} \sqcup \Big((\bar{E} \sqcup (C \sqcap \bigsqcup_{i \in I} D_i)) \sqcap \bigsqcup_{i \in I} D_i \Big) \\
&= \bar{E} \sqcup (\bar{E} \sqcap \bigsqcup_{i \in I} D_i) \sqcup ((C \sqcap \bigsqcup_{i \in I} D_i) \sqcap \bigsqcup_{i \in I} D_i) \\
&= \bar{E} \sqcup (C \sqcap \bigsqcup_{i \in I} D_i) \\
&= \bar{D}.
\end{aligned}$$

Hence $\square_{\bar{D}} T = \langle \bar{E}, \bar{D} \rangle$, which is a syntactic refinement of itself by Proposition 3.1.17 (a).

Thus, Theorem 3.5.1 is applicable and yields that $[T, <]$ is a syntactic decomposition of $\square_C T$. □

The following theorem treats the relation between "decomposition" and "correct composition" on the semantic level; the syntactic level is "filtered out" by considering only systems whose signature is a syntactical decomposition of the global cell signature.

3.5.4 Theorem. *Let $\langle \bar{Q}, \bar{R} \rangle$ be a cell of signature $\langle \bar{E}, \bar{D} \rangle$, and let $[M, <]$ be an ordered cell system whose ordered signature system is a syntactic decomposition of $\langle \bar{E}, \bar{D} \rangle$.*

If $\square_{\bar{D}} M$ is a refinement of $\langle \bar{Q}, \bar{R} \rangle$,
then $[M, <]$ is a decomposition of $\langle \bar{Q}, \bar{R} \rangle$.

If the cells of M are consistent, the converse implication also holds.

Proof. Let $M = \langle Q_i, R_i\rangle_{i\in I}$, and let the signature system of M be $T = \langle E_i, D_i\rangle_{i\in I}$. Note that $\bar{D} \sqsubseteq \bar{E} \sqcup \bigsqcup_{i\in I} D_i$, since $[T, <]$ is a syntactic decomposition of $\langle \bar{E}, \bar{D}\rangle$ by assumption, and so $\bar{D}$ and $\bigsqcup_{i\in I} D_i$ are compatible. Let $\langle \bar{Q}', \bar{R}'\rangle := \square_{\bar{D}} M$, and let $\langle \bar{E}', \bar{D}'\rangle$ be the cell signature of $\langle \bar{Q}', \bar{R}'\rangle$, that is, $\langle \bar{E}', \bar{D}'\rangle = \square_{\bar{D}} T$. We then have by definition

$$\bar{Q}' = \{\, A \in Mod(\bar{E}') \mid (\{A\} \wedge \bigwedge_{k \ll_T i} R_k) /\!\!/ E_i \subseteq Q_i \text{ for all } i \in I \,\} \qquad (1)$$

$$\bar{R}' = (\bar{Q}' \wedge \bigwedge_{i\in I} R_i) /\!\!/ \bar{D}'.$$

Now suppose that $\langle \bar{Q}', \bar{R}'\rangle$ is a refinement of $\langle \bar{Q}, \bar{R}\rangle$, so that, according to Proposition 3.1.19 (e),

$$\langle \bar{E}', \bar{D}'\rangle \text{ is a syntactic refinement of } \langle \bar{E}, \bar{D}\rangle,$$

$$(\text{in particular, } \bar{E}' \sqsubseteq \bar{E}), \qquad (2)$$

$$\bar{Q} /\!\!/ \bar{E}' \subseteq (\bar{Q}' \wedge \bar{R}') /\!\!/ \bar{E}', \qquad (3)$$

$$\text{and } (\bar{Q} \wedge \bar{R}') /\!\!/ \bar{D} \subseteq \bar{R}. \qquad (4)$$

We wish to show that $[M, <]$ is a decomposition of $\langle \bar{Q}, \bar{R}\rangle$. By assumption, the ordered signature system $[T, <]$ is a decomposition of $\langle \bar{E}, \bar{D}\rangle$.

Let $i \in I$, and consider $B \in (\bar{Q} \wedge \bigwedge_{k\ll i} R_k)$. Then

$$\begin{aligned} B/\bar{E}' &= (B/\bar{E})/\bar{E}' && \text{(by (2))} \\ &\in \bar{Q} /\!\!/ \bar{E}' \\ &\subseteq (\bar{Q}' \wedge \bar{R}') /\!\!/ \bar{E}' && \text{(by (3))} \\ &\subseteq \bar{Q}' /\!\!/ \bar{E}' && \text{(Proposition 3.1.8)} \\ &= \bar{Q}'. \end{aligned}$$

By (1), this implies that

$$(\{B/\bar{E}'\} \wedge \bigwedge_{k \ll_T i} R_k) /\!\!/ E_i \subseteq Q_i.$$

Since $B \in (\bar{Q} \wedge \bigwedge_{k\ll i} R_k)$ and $<_T \subseteq <$, hence $\ll_T \subseteq \ll$, we have $B/D_i \in R_i$ for $k \ll_T i$, and hence

$$B/E_i = B/(\bar{E}' \sqcup \bigsqcup_{k \ll_T i} D_k)/E_i$$

$$\begin{aligned} &\in (\{B/\bar{E}'\} \wedge \bigwedge_{k \ll_T i} R_i) /\!\!/ E_i \\ &\subseteq Q_i. \end{aligned}$$

Hence $(\bar{Q} \wedge \bigwedge_{k \ll i} R_k) /\!\!/ E_i \subseteq Q_i$ for all $i \in I$.

Finally, consider $B \in (\bar{Q} \wedge \bigwedge_{i \in I} R_k)$. As before, $B/\bar{E}' \in \bar{Q}'$, and thus

$$\begin{aligned} B/\bar{D}' &= B/(\bar{E}' \sqcup \bigsqcup_{i \in I} D_i)/\bar{D}' \qquad \text{(definition of } \bar{D}') \\ &\in (\bar{Q}' \wedge \bigwedge_{i \in I} R_i) /\!\!/ \bar{D}' \\ &= \bar{R}'. \end{aligned}$$

Hence

$$\begin{aligned} B/\bar{D} &= B/(\bar{E} \sqcup \bar{D}')/\bar{D} \\ &\in (\bar{Q} \wedge \bar{R}') /\!\!/ \bar{D} \\ &\subseteq \bar{R}. \qquad \text{(by (4))} \end{aligned}$$

This shows that $(\bar{Q} \wedge \bigwedge_{i \in I} R_k) /\!\!/ \bar{D} \subseteq \bar{R}$, and hence that $[M, <]$ is a decomposition of $\langle \bar{Q}, \bar{R} \rangle$.

Now assume that the cells of M are consistent, and that $[M, <]$ is a decomposition of $\langle \bar{Q}, \bar{R} \rangle$. By Lemma 3.1.19 (c), M is a componentwise refinement of itself. Hence the composability theorem of refinements can be applied with $M' = M$, and this yields that $\square_{\bar{D}} M$ is a refinement of $\langle \bar{Q}, \bar{R} \rangle$. □

For more explanation of the consistency condition in this theorem, see Example 3.5.10 below.

The following theorem characterizes the relation between decomposition and composition on both the syntactic and the semantic level. It is a simple combination of Theorems 3.5.1 and 3.5.4.

3.5.5 Theorem. *Let $\langle \bar{Q}, \bar{R} \rangle$ be a cell of signature $\langle \bar{E}, \bar{D} \rangle$, let $[M, <]$ be an ordered cell system, and let $T = \langle E_i, D_i \rangle_{i \in I}$ be the signature system of M.*
If

(a) *$\bar{E} \sqcup \bar{D}$ is compatible with $\bigsqcup_{i \in I} D_i$,*

(b) *$\bar{E} \sqcap D_i \sqsubseteq E_i$ for all $i \in I$, and*

(c) *$\square_{\bar{D}} M$ is a refinement of $\langle \bar{Q}, \bar{R} \rangle$,*

then $[M, <]$ is a decomposition of $\langle \bar{Q}, \bar{R} \rangle$.

If all cells of M are consistent, the converse implication also holds.

Proof. Suppose first that the clauses (a)–(c) hold. Clause (c) implies that $\square_{\bar{D}} T$ is a syntactic refinement of $\langle \bar{E}, \bar{D} \rangle$, hence Theorem 3.5.1 is applicable and yields that $[T, <]$ is a syntactic decomposition of $\langle \bar{E}, \bar{D} \rangle$. By the previous theorem (Thm. 3.5.4), $[M, <]$ is a decomposition of $\langle \bar{Q}, \bar{R} \rangle$.

For the converse, suppose that all cells of M are consistent, and that $[M, <]$ is a decomposition of $\langle \bar{Q}, \bar{R} \rangle$. Then by definition, $[T, <]$ is a decomposition of $\langle \bar{E}, \bar{D} \rangle$, and so (a) and (b) follow from Theorem 3.5.1; clause (c) follows from Theorem 3.5.4. □

The semantical analogue to Theorem 3.5.3 is that every cell system is a decomposition of its composition.

3.5.6 Theorem. *Let $[M, <]$ be an ordered cell system, and let C be a signature compatible with the join of the definition signatures of M. Then $[M, <]$ is a decomposition of $\square_C M$.*

Proof. Write M in the form $M = \langle Q_i, R_i \rangle_{i \in I}$, let $T = \langle E_i, D_i \rangle_{i \in I}$ be the signature family of M, let $\langle \bar{Q}, \bar{R} \rangle := \square_C M$, and let $\langle \bar{E}, \bar{D} \rangle$ be the signature of this cell, *i.e.*, $\langle \bar{E}, \bar{D} \rangle := \square_C T$. We verify the conditions of the "decomposition" definition (Def. 3.2.10).

(a) By Theorem 3.5.3, $[T, <]$ is a syntactic decomposition of $\langle \bar{E}, \bar{D} \rangle$.

(b) For $i \in I$, we have

$$(\bar{Q} \wedge \bigwedge_{k \ll i} R_k) /\!\!/ E_i \subseteq (\bar{Q} \wedge \bigwedge_{k \ll_T i} R_k) /\!\!/ E_i \qquad \text{(Prop. 3.1.8, and } \ll_T \subseteq \ll\text{)}$$
$$\subseteq Q_i. \qquad \text{(Prop. 3.3.7)}$$

(c) $(\bar{Q} \wedge \bigwedge_{i \in I} R_i) /\!\!/ \bar{D} = \bar{R}$ by definition.

Hence $[M, <]$ is a decomposition of $\langle \bar{Q}, \bar{R} \rangle = \square_C M$. □

We can now verify that the composition operation preserves the "type" of the cells involved.

3.5.7 Theorem ("Type" Preservation).
Let M be a cell system, and let C be a signature compatible with the join of the definition signatures of M.

If all cells of M are consistent, so is $\square_C M$.

If all cells of M are single-valued, so is $\square_C M$.

If all cells of M are modules, so is $\square_C M$.

Proof. Suppose first that all cells of M are consistent; in other words (Proposition 3.1.19 (c)), that M is a componentwise refinement of itself.

By Theorem 3.5.6, $[M, <]$ is a decomposition of $\square_C M$. By the Composability Theorem of Refinements, it follows that $\square_C M$ is a refinement of $\square_C M$, and hence that $\square_C M$ is consistent.

Now suppose that the cells of M are single-valued. Let M be of the form $M = \langle Q_i, R_i \rangle_{i \in I}$, and let $T = \langle E_i, D_i \rangle_{i \in I}$ be the signature system of M. Let $\langle \bar{Q}, \bar{R} \rangle := \square_C M$, and let $\langle \bar{E}, \bar{D} \rangle$ be the signature of $\langle \bar{Q}, \bar{R} \rangle$, *i.e.*, $\langle \bar{E}, \bar{D} \rangle = \square_C T$.

Let A be a base for $\square_C M$, and let F be the signature of A. Suppose that there are two different results B and B' of $\square_C M$ on A. Both B and B' are models of signature $F \sqcup \bar{D}$. Since $B/F = A = B'/F$, it follows that $B/\bar{D} \neq B'/\bar{D}$. Since B and B' are results of $\square_C M$ on A, both $B/\bar{D}$ and $B'/\bar{D}$ are elements of $\bar{R} = (\bar{Q} \wedge \bigwedge_{i \in I} R_i) /\!\!/ \bar{D}$. Hence there exist $C, C' \in (\bar{Q} \wedge \bigwedge_{i \in I} R_i)$ such that $C/\bar{D} = B/\bar{D}$ and $C'/\bar{D} = B'/\bar{D}$; in particular, $C \neq C'$.

Both C and C' are of signature $\bar{E} \sqcup \bigsqcup_{i \in I} D_i$. Since

$$\begin{aligned} C/\bar{E} &= (C/\bar{D})/\bar{E} = (B/\bar{D})\bar{E} = B/\bar{E} \\ &= (B/F)/\bar{E} = A/\bar{E} = (B'/F)/\bar{E} = B'/\bar{E} \\ &= (B'/\bar{D})/\bar{E} = (C'/\bar{D})/\bar{E} = C'/\bar{E}, \end{aligned}$$

it follows that $C/(\bigsqcup_{i \in I} D_i) \neq C'/(\bigsqcup_{i \in I} D_i)$, which means that the set $\mathcal{L} := \{ i \in I \mid C/D_i \neq C'/D_i \}$ is nonempty.

Let j be a $\ll_T$-minimal member of $\mathcal{L}$. Then

$$X := C/(\bar{E} \sqcup \bigsqcup_{i \ll_T j} D_i) = C'/(\bar{E} \sqcup \bigsqcup_{i \ll_T j} D_i).$$

This model X is a base for $\langle Q_j, R_j \rangle$:

- $\bar{E} \sqcup \bigsqcup_{i \ll_T j} D_i$ is a site for $\langle E_j, D_j \rangle$, because $\bar{E}$ is a system site for $[T, <_T]$,
- $$\begin{aligned} X/E_j &= C/E_j \\ &\in (\bar{Q} \wedge \bigwedge_{i \in I} R_i) /\!\!/ E_j \\ &\subseteq (\bar{Q} \wedge \bigwedge_{i \ll_T j} R_i) /\!\!/ E_j \qquad \text{(Prop. 3.1.8)} \\ &\subseteq Q_j. \qquad \text{(Prop. 3.3.7)} \end{aligned}$$

Now $Y := C/(\bar{E} \sqcup \bigsqcup_{i \ll_T j} D_i \sqcup D_j)$ and $Y' := C'/(\bar{E} \sqcup \bigsqcup_{i \ll_T j} D_i \sqcup D_j)$ are results of $\langle Q_j, R_j \rangle$ on the base X:

- their signature is the result signature of $\langle E_j, D_j \rangle$ on the site $\bar{E} \sqcup \bigsqcup_{i \ll_T j} D_i$,
- $Y/(\bar{E} \sqcup \bigsqcup_{i \ll_T j} D_i) = C/(\bar{E} \sqcup \bigsqcup_{i \ll_T j} D_i) = X$, and $Y'/(\bar{E} \sqcup \bigsqcup_{i \ll_T j} D_i) = X$ analogously,
- $Y/D_j = C/D_j \in R_j$ and $Y'/D_j = C'/D_j \in R_j$.

But Y and Y' are different, because by definition of j, we have

$$Y/D_j = C/D_j \neq C'/D_j = Y'/D_j.$$

This contradicts the single-valuedness of M_j. Hence the assumption that there are two different results B and B' of $\Box_C M$ on A is refuted, which proves that $\Box_C M$ is single-valued.

If all cells of M are modules, *i.e.*, both consistent and single-valued, then by the implications just proven, $\square_C M$ is consistent and single-valued, *i.e.*, a module. □

The following theorems show that the dependence relation of an ordered system is nearly irrelevant, because (except for systems with inconsistent cells) the syntactic and semantic decomposition notions are independent of it (nevertheless, it does not seem advisable to dispense with nonsyntactical dependence relations, since these arise naturally from design graphs, as was discussed on p. 135 f.).

3.5.8 Theorem. *Let $[T, <]$ be an ordered signature system, let E be a signature, and let $\langle \bar{E}, \bar{D} \rangle$ be a cell signature. Then the predicates*

$$\text{"}E \text{ is a system site for } [T, <]\text{"}$$

and

$$\text{"}[T, <] \text{ is a syntactic decomposition of } \langle \bar{E}, \bar{D} \rangle\text{"}$$

do not depend on $<$.

Proof. Consider first the predicate "E is a system site for $[T, <]$". We show that this predicate is equivalent to "E is a system site for $[T, <_T]$", which is independent of $<$.

If E is a system site for $[T, <]$, then E also is a system site for $[T, <_T]$ by Lemma 3.4.3 (c) (put $T' := T$ in the lemma). Conversely, if E is a system site for $[T, <_T]$, where $T = \langle E_i, D_i \rangle_{i \in I}$, then for each $i \in I$, Lemma 3.5.2 implies that $E \sqcup \bigsqcup_{k \ll i} D_k$ is a site for $\langle E_i, D_i \rangle$, because $<_T \subseteq <$, and hence $\{ k \mid k \ll i \}$ is $<_T$-downward closed. Hence E is a system site for $[T, <]$.

Consider now the predicate "$[T, <]$ is a syntactic decomposition of $\langle \bar{E}, \bar{D} \rangle$". This predicate is equivalent to the composition of the clauses (a), (b), and (c) of Theorem 3.5.1, and these clauses are independent of $<$. □

3.5.9 Theorem. *Let $[M, <]$ be an ordered cell system whose cells are consistent, and let $\langle \bar{Q}, \bar{R} \rangle$ be a cell. Then the relation*

$$\text{"}[M, <] \text{ is a decomposition of } \langle \bar{Q}, \bar{R} \rangle\text{"}$$

is independent of $<$.

Proof. The three clauses (a), (b), and (c) of Theorem 3.5.5 are independent of $<$. □

It will now be shown by an example that inconsistent cells have the following unpleasant properties:

(a) in Theorem 3.5.4, the consistency requirement cannot be dropped—there are decompositions of a global cell which contain inconsistent cells and whose composition is not a refinement of the global cell,

(b) yet the requirement is not mathematically "necessary" either—there are decompositions of a global cell which contain inconsistent cells and whose composition *is* a refinement of the global cell,

(c) in Theorem 3.5.9, the consistency requirement cannot be dropped—for ordered systems with inconsistent cells, the "decomposition" property can depend on the dependence relation.

3.5.10 Example. In the institution ⟨**ASig**, **AIncl**, **Alg**⟩, the empty signature has exactly one model, the empty algebra $\emptyset$, which is a mapping whose domain is the symbol set of the empty signature, namely $\emptyset$. There are two interfaces of the empty signature:

$$TRUE := \{\emptyset\}, \qquad FALSE := \emptyset$$

(the interface $TRUE$ is satisfied by every model, the interface $FALSE$ by none). Define the cell system $M = \langle M_1, M_2 \rangle$ by

$$M_1 := \langle TRUE, FALSE \rangle,$$
$$M_2 := \langle FALSE, TRUE \rangle.$$

The signature system T of M is given by $T_1 = T_2 = \langle \emptyset, \emptyset \rangle$. The syntactic dependence relation of T is empty, and so $< := \{(1,2)\}$ is a dependence relation for T.

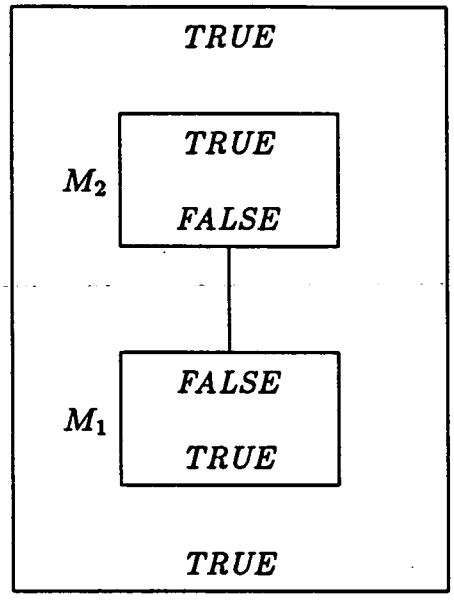


Figure 3-11: $[M, <]$ is a decomposition of $\langle TRUE, TRUE \rangle$

The ordered cell system $[M, <]$ is a decomposition of the cell $\langle TRUE, TRUE \rangle$, because the requirement of M_2 is implied by the result interface of M_1 (see Figure 3-11).

However, the ordered cell system $[M, <_T] = [M, \emptyset]$ is not a decomposition of $\langle TRUE, TRUE \rangle$, because the requirement of M_2 is not implied by the external requirement $TRUE$ alone (see Figure 3-12). This illustrates that for ordered systems with inconsistent cells, the dependence relation can influence the decomposition property (point (c) above).

Figure 3-13 illustrates that the composition of M is $\langle FALSE, FALSE \rangle$: The requirement of the composition must be $FALSE$ in order to imply the requirement of M_2; the result is $FALSE$, because the requirement is (the result interface of M_1 is another reason for the result of the composition to be $FALSE$).

We saw above that $[M, <]$ is a decomposition of the cell $\langle TRUE, TRUE \rangle$ (Figure 3-11). But the composition of M is $\langle FALSE, FALSE \rangle$, and this is not a refinement of $\langle TRUE, TRUE \rangle$, because the empty algebra $\emptyset$ is a base for $\langle TRUE, TRUE \rangle$ but not for $\langle FALSE, FALSE \rangle$. This shows that for systems

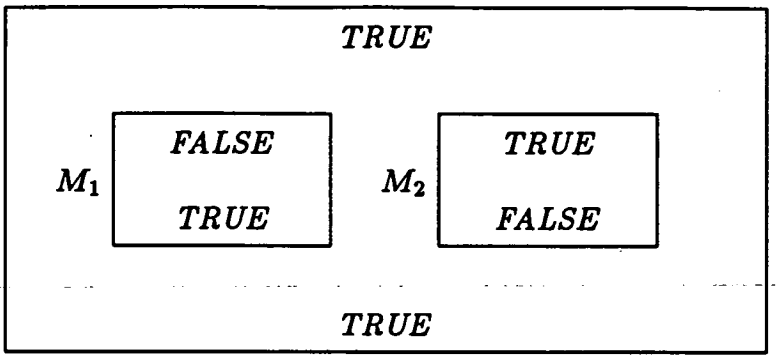


Figure 3-12: $[M, \emptyset]$ is not a decomposition of $\langle TRUE, TRUE \rangle$

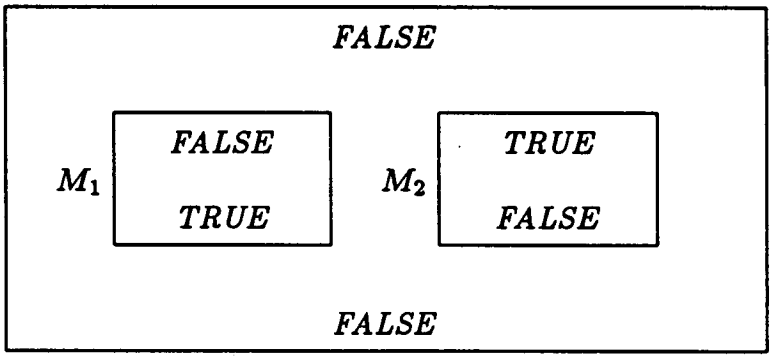


Figure 3-13: The composition of M is $\langle FALSE, FALSE \rangle$

with inconsistent cells, the decomposition property does not necessarily imply that the system's composition is a refinement of the global cell (point (a) above).

Finally, observe that Figure 3-13 also shows a correct decomposition: $[M, \emptyset]$ is a decomposition of $\langle FALSE, FALSE \rangle$. The composition of M is the same cell, and this cell is consistent(!), hence a refinement of itself (Proposition 3.1.19 (c)). Thus we have a decomposition into a system containing inconsistent cells, whose composition is a refinement of the global cell—this illustrates point (b) above. □

The general observation in this example is that an inconsistent cell (here, M_1) can help establish the requirements of another cell (here, M_2) that is not syntactically dependent on the inconsistent cell (this makes Figure 3-11 a decomposition). This is so because the result interface of an inconsistent cell can make

stronger statements about the cell's environment than the requirement interface (if the requirement of M_1 was made *FALSE*, so that M_1 would be consistent, Figure 3-11 would no longer be a decomposition, because the new interface was not implied by the external requirement interface *TRUE*).

These phenomena do not affect the composability of refinements (Theorem 4.1.12), because an inconsistent cell cannot be refined by another cell (Proposition 3.1.19 (b)), and hence the composability theorem can only be applied to decompositions whose cells are consistent. In other words, if a decomposition contains inconsistent cells, then the bottom left corner of Figure 3-5 (page 123) cannot be completed, and so one will always detect inconsistencies when trying to establish the structured correctness argument for a modular system.

Chapter 4

Data Abstraction

THIS CHAPTER introduces data abstraction into the theory. The basic idea of data abstraction is that the program entities (in other words, the semantic model) defined by a program need not satisfy the specification; rather, it is sufficient that they "represent" a model satisfying the specification. Accordingly, data abstraction is based on a relation of "representation" between models.

The first section of the chapter deals with data abstraction in an abstract setting, namely in an arbitrary institution. A "representation" relation between the models of an institution can be any relation that satisfies a few simple axioms. For such a representation relation, a "universal implementation" relation between cells is defined; in data abstraction, every cell must be a universal implementation of its specification. The main theorem of Section 4.1 and the central theorem of this thesis is the theorem asserting the "composability of universal implementations". According to this theorem, "universal implementation" can be used as the correctness notion in the structured correctness argument for a modular system (Figure 3-5): If a global cell is decomposed into a cell system, then a componentwise universal implementation of the system will yield a universal implementation of the global cell when composed. Since "refinement" is just a special case of the "universal implementation" notion, the composability theorem of refinements (which has been cited and used in Section 3.4 already) is obtained as a corollary of this theorem.

The remaining sections of the chapter deal with representation relations between partial algebras. Section 4.2 introduces a new institution $\langle$**TSig, TIncl, TAlg**$\rangle$ for this purpose; it is a slight variant of $\langle$**ASig, AIncl, Alg**$\rangle$, in which

each signature has a distinguished subset of "visible" sorts. The following sections deal with the three representation relations "behavioural inclusion", "behavioural equivalence", and the "standard representation" relation based on abstraction functions.

The reader is warned that the "universal implementation" concept presented in this chapter is not meant to be used in practice, as it would be too difficult to verify. Chapter 5 to follow will decompose "universal implementation" into "simple implementation" and "stability", where the "simple implementation" concept formalizes a practical method of proving implementations correct.

4.1 Data Abstraction in an Institution

This section deals with data abstraction on an abstract level: we shall consider an arbitrary "representation relation" in a arbitrary institution. Hence the following convention:

Convention. Throughout this section, the triple $\langle Sig, Incl, Mod\rangle$ is assumed to be an institution. The concepts that depend on an institution (such as "signature", "inclusion", or "model") are implicitly assumed to refer to the institution $\langle Sig, Incl, Mod\rangle$. □

Here are the axioms for a "representation relation".

4.1.1 Definition. A *representation relation* is a $|Sig|$-indexed relation $\rightsquigarrow = \langle \underset{S}{\rightsquigarrow} \rangle_{S\in|Sig|}$, such that

(a) for each $S \in |Sig|$, $\underset{S}{\rightsquigarrow}$ is a preordering on $Mod(S)$ (*i. e.*, $\underset{S}{\rightsquigarrow} \subseteq Mod(S) \times Mod(S)$ is reflexive and transitive),

(b) If $\sigma\colon S \to T$ in Sig and $A' \underset{T}{\rightsquigarrow} A$, then $\overline{\sigma}\, A' \underset{S}{\rightsquigarrow} \overline{\sigma}\, A$.

If $\rightsquigarrow$ is a representation relation, S a signature, and $A' \underset{S}{\rightsquigarrow} A$, say that A' is a *representation* of A according to $\rightsquigarrow$ (or just "A' is a $\rightsquigarrow$-representation

of A"). The subscript in "$\underset{S}{\rightsquigarrow}$" will be dropped when the signature S is obvious from the context.

A representation relation $\rightsquigarrow$ is *chain-closed*, if in addition

(c) Whenever $\langle S_i \rangle_{i \in I}$ is a nonempty compatible chain of signatures, and A, A' are models of signature $\bigsqcup_{i \in I} S_i$ satisfying $A'/S_i \underset{S_i}{\rightsquigarrow} A/S_i$ for all $i \in I$, then $A' \underset{\bigsqcup S_i}{\rightsquigarrow} A$. □

It is natural to require reflexivity and transitivity of a representation relation: every model should at least be representable by itself, and a representation of a representation should also be a representation. Clause (b) says that "reducing" models along signature morphisms preserves the representation relation. The chain-closedness property of a representation relation will allow us to compose infinite systems (see Theorem 4.1.7).

Three examples of representation relations in the institution ⟨**TSig**, **TIncl**, **TAlg**⟩ (a slight variant of ⟨**ASig**, **AIncl**, **Alg**⟩) will be studied in later sections of this chapter.

A simple and useful representation relation is equality.

4.1.2 Proposition. *Equality is a chain-closed representation relation.*

Proof. Define $\rightsquigarrow = \langle \underset{S}{\rightsquigarrow} \rangle_{S \in |Sig|}$ by $A' \underset{S}{\rightsquigarrow} A \ :\Longleftrightarrow \ A' = A$. Clauses (a) and (b) of the "representation relation" definition are then trivial. Chain-closedness follows from the completeness property of an institution (uniqueness of joins). □

The following two trivial propositions allow the construction of new representation relations.

4.1.3 Proposition. *The converse of a (chain-closed) representation relation is a (chain-closed) representation relation.*

To be precise: if $\rightsquigarrow = \langle \underset{S}{\rightsquigarrow} \rangle_{S \in |Sig|}$ *is a representation relation, then the relation* $\succsim = \langle \underset{S}{\succsim} \rangle_{S \in |Sig|}$ *defined by* $A \underset{S}{\succsim} B \ :\Longleftrightarrow \ B \underset{S}{\rightsquigarrow} A$ *is a representation relation. If* $\rightsquigarrow$ *is chain-closed, so is* $\succsim$. □

4.1.4 Proposition. *The intersection of a family of (chain-closed) representation relations is a (chain-closed) representation relation.*

To be precise, if $\overset{i}{\rightsquigarrow} = (\underset{S}{\overset{i}{\rightsquigarrow}})_{S \in |Sig|}$ *is a representation relation for each* $i \in I$, *then the relation* $\overset{*}{\rightsquigarrow} = (\underset{S}{\overset{*}{\rightsquigarrow}})_{S \in |Sig|}$ *defined by*

$$A' \underset{S}{\overset{*}{\rightsquigarrow}} A \quad :\Longleftrightarrow \quad A' \underset{S}{\overset{i}{\rightsquigarrow}} A \text{ for all } i \in I$$

is a representation relation. If all $\overset{i}{\rightsquigarrow}$ *are chain-closed, so is* $\overset{*}{\rightsquigarrow}$. □

As a corollary of this proposition, one obtains that the representation relations in an institution form a complete lattice under the componentwise inclusion ordering (using the well-known theorem that a partial order in which every set has a least upper bound is a complete lattice [Birkhoff 67, p. 112] [Cohn 81, p. 21]). The "bottom" element of this lattice (the most restrictive representation relation) is equality, the "top" element (the most general representation relation) is the "total" representation relation obtained as the intersection of the empty family of representation relations.

The next concept, "universal implementation", is the correctness notion for data abstraction for which we shall derive a composability theorem.

4.1.5 Definition. Let $\rightsquigarrow = (\underset{S}{\rightsquigarrow})_{S \in |Sig|}$ be a representation relation, and let $\langle Q, R \rangle$ be a cell of signature $\langle E, D \rangle$. A *universal implementation* of $\langle Q, R \rangle$ (with respect to $\rightsquigarrow$) is a cell $\langle Q', R' \rangle$ of signature $\langle E', D' \rangle$ such that $\langle E', D' \rangle$ is a syntactic refinement of $\langle E, D \rangle$ and whenever $A \in Mod(F)$ is a base for $\langle Q, R \rangle$ and $A' \underset{F}{\rightsquigarrow} A$, then

A' is a base for $\langle Q', R' \rangle$,
there exists a result of $\langle Q', R' \rangle$ on A', and
whenever B' is a result of $\langle Q', R' \rangle$ on A'
then there exists a result B of $\langle Q, R \rangle$ on A
such that $B' \underset{F \sqcup D}{\rightsquigarrow} B$. □

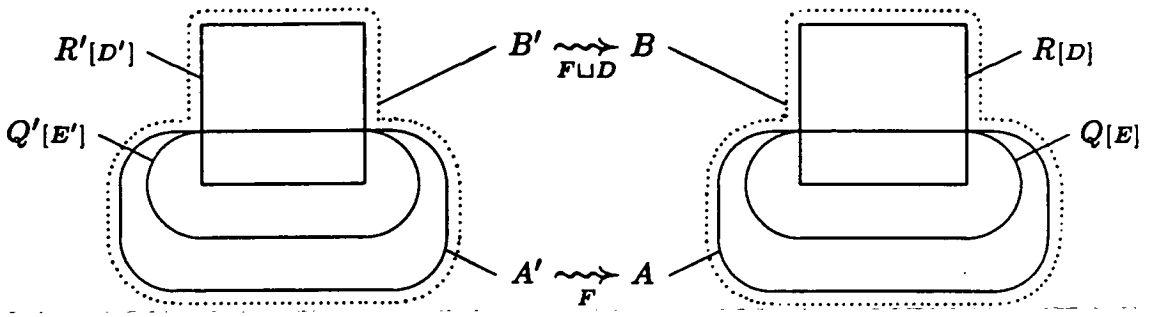


Figure 4-1: Universal Implementation

This definition is illustrated in Figure 4-1. The figure shows on the right hand side the cell $\langle Q, R\rangle$ with a base A and a result B, and on the left hand side the cell $\langle Q', R'\rangle$ with a base A' and a result B' (signatures of the interfaces and models are given in small type).

The idea behind the definition of "universal implementation" is that the specification cell and the implementation cell can be seen as operating side by side; while the specification cell on the right operates on an "abstract" base A, the implementation cell on the left operates on an arbitrary representation A' of A (hence if A is a base for $\langle Q, R\rangle$ and $A' \overset{F}{\rightsquigarrow} A$, then A' must be a base for $\langle Q', R'\rangle$ and $\langle Q', R'\rangle$ must have a result on A'). To ensure that the representation relation between left and right side remains valid, every result B' of $\langle Q', R'\rangle$ on A' must represent some result B of $\langle Q, R\rangle$ on A (the model B may be regarded as the "user view" of B').

In a sense, the universal implementation relation is "necessary" for modular programming with data abstraction: For, if $\langle Q, R\rangle$ is *not* a universal implementation of $\langle Q, R\rangle$, then there exists a base A of $\langle Q, R\rangle$ with a representation A', such that $\langle Q', R'\rangle$ does not function properly on A'—either by not yielding a result at all, or by being able to yield an "incorrect" result, namely a result that cannot be interpreted as the representation of a result on the "abstract" side.

There is one way the "universal implementation" notion could be generalized, namely by exploiting the fact that not all the possible representations A' of an "abstract" base A might actually be definable by a program. Thus, one

might distinguish a subinstitution of "concrete" models and postulate that the implementation cell $\langle Q', R' \rangle$ need only function properly on representations A' that belong to the "concrete" models. It is not clear, however, whether this generalization would yield any additional insights.

In the remainder of this section, a composability theorem for universal implementations will be proved which shows that the universal implementation concept is not just "necessary", but also "sufficient" for modular programming with data abstraction.

A special case of the "universal implementation" notion is the "refinement" notion that was discussed in the previous chapter. It is obtained by using equality as the representation relation.

4.1.6 Proposition. *A cell $\langle Q', R' \rangle$ is universal implementation of a cell $\langle Q, R \rangle$ with respect to equality, if and only if $\langle Q', R' \rangle$ is a refinement of $\langle Q, R \rangle$.*

Proof. With "equality" the representation relation, we can put $A' = A$ and $B' = B$ in the definition of "universal implementation". The definition then becomes equivalent to the definition of "refinement". □

It is instructive to compare the universal implementation concept for a "general" representation relation $\rightsquigarrow$ with the refinement notion, that is, universal implementation with respect to equality.

By definition, $\rightsquigarrow$ includes equality, *i. e.*, it is a more general relation. This does not, however, imply any direct relationship between universal implementation with respect to $\rightsquigarrow$ and refinement:

Consider two cells $\langle Q', R' \rangle$ and $\langle Q, R \rangle$ as in the definitions of universal implementation and of refinement. On the one hand, universal implementation requires that all representations of bases for $\langle Q, R \rangle$ are bases for $\langle Q', R' \rangle$, rather than only the bases for $\langle Q, R \rangle$, hence on this point, universal implementation is more restrictive than refinement. On the other hand, the result of $\langle Q', R' \rangle$ need only represent a result of $\langle Q, R \rangle$ rather than equal such a result, hence on this point, universal implementation is more general than refinement.

Since, intuitively, data abstraction should make the programmer's job easier, one might be puzzled by the fact that universal implementation is not more general than refinement—after all, universal implementation is supposed to be the correctness relation for cells in data abstraction.

This problem will be resolved in Chapter 5, where a different implementation concept, called "simple implementation", will be introduced, which corresponds to practical implementation correctness proofs, and which is more general than refinement. This "simple implementation" property implies the "universal implementation" property in case the implementation cell is "stable", a property that can be guaranteed by defining the implementation cell in a suitable programming notation (a "data abstraction language"). Hence programmers need not concern themselves directly with the "universal implementation" concept, but only with "simple implementation"—the "universal implementation" concept only acts as a theoretical intermediary.

From now on until the end of this section, we shall deal with an arbitrary, but fixed, representation relation. Hence the following convention.

Convention. Throughout the remainder of this section, the $|Sig|$-indexed relation $\rightsquigarrow = \langle \underset{S}{\rightsquigarrow} \rangle_{S \in |Sig|}$ is a representation relation in the institution $\langle Sig, Incl, Mod \rangle$. The term "universal implementation" is understood to imply "with respect to $\rightsquigarrow$". □

The remainder of this section is devoted to proving the following theorem, which is the central theorem of the thesis.

4.1.7 Theorem (Composability of Universal Implementations).
Let $[M, <]$ be a decomposition of the $\langle \bar{E}, \bar{D} \rangle$-cell $\langle \bar{Q}, \bar{R} \rangle$ such that $\rightsquigarrow$ is chain-closed or M is finite, and let M' be a componentwise universal implementation of M. Then M' is a cell system, the signature $\bar{D}$ is compatible with the join of the definition signatures of M', and

$$\square_{\bar{D}} M' \text{ is a universal implementation of } \langle \bar{Q}, \bar{R} \rangle.$$

For the proof, some auxiliary concepts and lemmas are needed. The first two lemmas deal with possible conflicts between internal symbols of a system and symbols that might be present in a site for the system's composition. The signature H in these lemmas can be thought of as containing the internal symbols of a system whose composition is $\langle Q', R' \rangle$. Lemma 4.1.9 below says that in determining whether $\langle Q', R' \rangle$ is a universal implementation of the $\langle E, D \rangle$-cell $\langle Q, R \rangle$, we can restrict our attention to sites for $\langle E, H \rangle$, that is, sites for $\langle E, D \rangle$ that avoid clashes with internal symbols of the system. The proof uses the "renaming" axiom of an institution syntax (Axiom (c) of Def. 2.3.5) to "rename" an arbitrary site F for $\langle E, D \rangle$ so that it becomes a site for $\langle E, H \rangle$.

First, however, a simple preliminary lemma.

4.1.8 Lemma. *Let $\langle E, D \rangle$ be a cell signature, let H be a signature such that $H \sim E$ and $E \sqcup H \sqsupseteq E \sqcup D$. Then every site for $\langle E, H \rangle$ is a site for $\langle E, D \rangle$.*

Proof. Let F be a site for $\langle E, H \rangle$, *i.e.*, $F \sim E \sqcup H$ and $F \sqcap (E \sqcup H) = E$. Then also $F \sim E \sqcup D$, and because $E \sqsubseteq F$, we have

$$E = F \sqcap E \sqsubseteq F \sqcap (E \sqcup D) \sqsubseteq F \sqcap (E \sqcup H) = E,$$

hence $E = F \sqcap (E \sqcup D)$. Thus, F is a site for $\langle E, D \rangle$. □

4.1.9 Lemma (Renaming).
Let $\langle Q, R \rangle$ be a cell of signature $\langle E, D \rangle$, let $\langle Q', R' \rangle$ be a cell of signature $\langle E', D' \rangle$, and let H be a signature such that $H \sim E$ and $E \sqcup H \sqsupseteq E \sqcup D$. Then $\langle Q', R' \rangle$ is a universal implementation of $\langle Q, R \rangle$, if and only if $\langle E', D' \rangle$ is a syntactic refinement of $\langle E, D \rangle$ and whenever F is a site for $\langle E, H \rangle$, $A \in Mod(F)$ is a base for $\langle Q, R \rangle$ and $A' \underset{F}{\rightsquigarrow} A$, then

A' is a base for $\langle Q', R' \rangle$,
there exists a result of $\langle Q', R' \rangle$ on A', and
whenever B' is a result of $\langle Q', R' \rangle$ on A'
then there exists a result B of $\langle Q, R \rangle$ on A
such that $B' \underset{F \sqcup D}{\rightsquigarrow} B$.

Proof. The only difference between the criterion of this lemma and the definition of "universal implementation" (Def. 4.1.5) is that here, F is required to be a site for $\langle E, H\rangle$ rather than only a site for $\langle E, D\rangle$.

By the previous lemma, it follows that the criterion of the present lemma is necessary for the universal implementation property.

To see that it is sufficient, let $\langle Q, R\rangle$ be a cell of signature $\langle E, D\rangle$, let $\langle Q', R'\rangle$ be a cell of signature $\langle E', D'\rangle$, let H be a signature satisfying

$$H \sim E \quad \text{and} \quad E \sqcup H \sqsupseteq E \sqcup D, \tag{1}$$

and assume that the criterion of the present lemma is satisfied. We have to show that $\langle Q', R'\rangle$ is a universal implementation of $\langle Q, R\rangle$.

Let F be an arbitrary site for $\langle E, D\rangle$, and define $G := F \sqcup D$. Then the "renaming" axiom of an institution syntax (Def. 2.3.5 (c)) is applicable with $S := E \sqcup D$, $T := H$, and $U := F$, because $E \sqcup D \sim H$ as $E \sqcup D \sqsubseteq E \sqcup H$ and $F \sim E \sqcup D$ as F is a site for $\langle E, D\rangle$. By the axiom, we can choose a signature $\hat{F}$ such that

$$\hat{F} \sim E \sqcup D \sqcup H \quad \text{and} \quad \hat{F} \sqcap (E \sqcup D \sqcup H) = F \sqcap (E \sqcup D) = E \tag{2}$$

and isomorphisms

$$\begin{aligned} j&\colon \hat{F} \to F, \\ k&\colon \hat{F} \sqcup (E \sqcup D) \to F \sqcup (E \sqcup D), \end{aligned}$$

such that, with $\hat{G} := \hat{F} \sqcup D$ (*i.e.*, $k\colon \hat{G} \to G$):

$$\begin{aligned} (\hat{F} \sqsubseteq \hat{G})\,;k &= j\,;(F \sqsubseteq G) \\ (E \sqcup D \sqsubseteq \hat{G})\,;k &= (E \sqcup D \sqsubseteq G) \\ (E \sqsubseteq \hat{F})\,;j &= (E \sqsubseteq F). \end{aligned}$$

The signature $\hat{F}$ is a site for $\langle E, H\rangle$, since from (2) it follows that $\hat{F} \sim (E \sqcup H)$ and

$$\begin{aligned} \hat{F} \sqcap (E \sqcup H) &= \hat{F} \sqcap (E \sqcup D \sqcup H) \sqcap (E \sqcup H) \\ &= \hat{F} \sqcap (E \sqcup D \sqcup H) && \text{(by (1))} \\ &= E && \text{(by (2))}. \end{aligned}$$

By the previous lemma, $\hat{F}$ also is a site for $\langle E, D\rangle$.

We now need two simple lemmas relating models of the signatures F and $\hat{F}$ and of the signatures G and $\hat{G}$. The first lemma is:

$$\begin{array}{ll} \text{Whenever } \langle Q^\bullet, R^\bullet\rangle \text{ is a cell whose signature} & \\ \qquad \langle E^\bullet, D^\bullet\rangle \text{ is a syntactical refinement of } \langle E, D\rangle, & \\ \qquad A \in \mathit{Mod}(F), \text{ and } \hat{A} = \overline{j}\, A \in \mathit{Mod}(\hat{F}), & (3) \\ \text{then} \quad A \text{ is a base for } \langle Q^\bullet, R^\bullet\rangle \text{ iff } \hat{A} \text{ is a base for } \langle Q^\bullet, R^\bullet\rangle. & \end{array}$$

We already know that F and $\hat{F}$ are sites for $\langle E, D\rangle$ and hence for $\langle E^\bullet, D^\bullet\rangle$. Now

$$\begin{aligned} \hat{A}/E^\bullet &= \overline{(E^\bullet \sqsubseteq E)\,;(E \sqsubseteq \hat{F})}\,\hat{A} \\ &= \overline{(E^\bullet \sqsubseteq E)\,;(E \sqsubseteq \hat{F})}(\overline{j}\, A) \\ &= \overline{(E^\bullet \sqsubseteq E)\,;(E \sqsubseteq \hat{F})\,;j}\, A \\ &= \overline{(E^\bullet \sqsubseteq E)\,;(E \sqsubseteq F)}\, A \\ &= A/E^\bullet. \end{aligned}$$

Hence $A/E^\bullet \in Q^\bullet$ iff $\hat{A}/E^\bullet \in Q^\bullet$, which proves (3).

The second lemma is:

$$\begin{array}{ll} \text{Whenever } \langle Q^\bullet, R^\bullet\rangle \text{ is a cell whose signature} & \\ \qquad \langle E^\bullet, D^\bullet\rangle \text{ is a syntactical refinement of } \langle E, D\rangle, & \\ \qquad A \in \mathit{Mod}(F) \text{ is a base for } \langle Q^\bullet, R^\bullet\rangle, & \\ \qquad \hat{A} = \overline{j}\, A \in \mathit{Mod}(\hat{F}), & (4) \\ \qquad B \in \mathit{Mod}(G) \text{ and } \hat{B} = \overline{k}\, B \in \mathit{Mod}(\hat{G}), & \\ \text{then} \quad B \text{ is a result of } \langle Q^\bullet, R^\bullet\rangle \text{ on } A & \\ \qquad \text{iff } \hat{B} \text{ is a result of } \langle Q^\bullet, R^\bullet\rangle \text{ on } \hat{A}. & \end{array}$$

Let the assumptions of this lemma be in force. By the previous lemma, the algebra $\hat{A}$ is a base for $\langle Q^\bullet, R^\bullet\rangle$. Since the situation is symmetrical (we can swap the rôles of $\hat{F}$ and F, $\hat{G}$ and G, j and j^{-1}, k and k^{-1}, $\hat{A}$ and A, and $\hat{B}$ and B), it suffices to prove one direction of the conclusion.

Assume that B is a result of $\langle Q^\bullet, R^\bullet \rangle$ on A. Then

$$\begin{aligned}
\hat{B}/\hat{F} &= \overline{(\hat{F} \sqsubseteq \hat{G})}(\overline{k}\, B) \\
&= \overline{(\hat{F} \sqsubseteq \hat{G})\,;k}\, B \\
&= \overline{j\,;(F \sqsubseteq G)}\, B \\
&= \overline{j}(B/F) \\
&= \overline{j}\, A \\
&= \hat{A},
\end{aligned}$$

and

$$\begin{aligned}
\hat{B}/D^\bullet &= \overline{(D^\bullet \sqsubseteq \hat{G})}(\overline{k}\, B) \\
&= \overline{(D^\bullet \sqsubseteq \hat{G})\,;k}\, B \\
&= \overline{(D^\bullet \sqsubseteq E \sqcup D)\,;(E \sqcup D \sqsubseteq \hat{G})\,;k}\, B \\
&\qquad (D^\bullet \sqsubseteq E \sqcup D \text{ by syntactic refinement}) \\
&= \overline{(D^\bullet \sqsubseteq E \sqcup D)\,;(E \sqcup D \sqsubseteq G)}\, B \\
&= B/D^\bullet \\
&\in R^\bullet.
\end{aligned}$$

Thus, $\hat{B}$ is a result of $\langle Q^\bullet, R^\bullet \rangle$ on $\hat{A}$, and (4) is proved.

Recall that we have to show that $\langle Q', R' \rangle$ is a universal implementation of $\langle Q, R \rangle$. Let $A \in Mod(F)$ be a base for $\langle Q, R \rangle$, and $A' \underset{F}{\rightsquigarrow} A$. Define $\hat{F}$-models $\hat{A} := \overline{j}\, A$ and $\hat{A}' := \overline{j}\, A'$.

By (3), applied to $\langle Q, R \rangle$, $\hat{A}$ is a base for $\langle Q, R \rangle$. By the functorality property of $\rightsquigarrow$ (Def. 4.1.1 (b)), we have $\hat{A}' \underset{\hat{F}}{\rightsquigarrow} \hat{A}$. Thus, the criterion of the lemma can be applied to $\hat{F}$, $\hat{A}'$, and $\hat{A}$.

First, this yields that $\hat{A}'$ is a base for $\langle Q', R' \rangle$. By (3), applied to $\langle Q', R' \rangle$, this implies that A' is a base for $\langle Q', R' \rangle$.

By the criterion of the lemma, there exists a result of $\langle Q', R' \rangle$ on $\hat{A}'$. Applying (4) to $\langle Q', R' \rangle$ yields that there exists a result of $\langle Q', R' \rangle$ on A'.

Finally, let B' be any result of $\langle Q', R' \rangle$ on A'. We have to show that there exists a result B of $\langle Q, R \rangle$ on A such that $B' \underset{G}{\rightsquigarrow} B$. Define $\hat{B}' := \overline{k}\, B'$. By (4),

applied to $\langle Q', R' \rangle$, $\hat{B}'$ is a result of $\langle Q', R' \rangle$ on $\hat{A}'$. Applying the condition of the lemma yields that there exists a result $\hat{B}$ of $\langle Q, R \rangle$ on $\hat{A}$ such that $\hat{B}' \underset{\hat{G}}{\rightsquigarrow} \hat{B}$. Define $B := \overline{k^{-1}}\,\hat{B}$. Since then $\hat{B} = \overline{k}\,B$, we obtain from (4), applied to $\langle Q, R \rangle$, that B is a result of $\langle Q, R \rangle$ on A. As $B' = \overline{k^{-1}}\,\hat{B}'$, we have $B' \underset{G}{\rightsquigarrow} B$ by functorality.

It has been proved that $\langle Q', R' \rangle$ is a universal implementation of $\langle Q, R \rangle$. □

The following "approximation" concept will be used in the proof of the Composability Theorem.

4.1.10 Definition (Approximation).
Let $[M, <]$ be an ordered cell system, $M = \langle Q_i, R_i \rangle_{i \in I}$, let F be a system site for $[\mathrm{Sig}(M), <]$, and let $A \in Mod(F)$. An *approximation* for $[M, <]$ over A is a pair $\langle K, X \rangle$ such that

$$K \subseteq I \text{ is } <\text{-downward closed}, \quad \text{and}$$
$$X \in \{A\} \wedge \bigwedge_{i \in K} R_i.$$

If $\langle K, X \rangle$ and $\langle K', X' \rangle$ are approximations for $[M, <]$ over A, say that $\langle K, X \rangle$ is *included* in $\langle K', X' \rangle$ (written "$\langle K, X \rangle \sqsubseteq \langle K', X' \rangle$"), if $K \subseteq K'$ and $X \sqsubseteq X'$. □

4.1.11 Lemma (least upper bounds of approximation chains).
Let $[M, <]$ be an ordered cell system, let F be a system site for $[\mathrm{Sig}(M), <]$, let $A \in Mod(F)$, and let $\langle K_j, X_j \rangle_{j \in J}$ be a $\sqsubseteq$-chain of approximations for $[M, <]$ over A. Then

$$\left\langle \bigcup_{j \in J} K_j,\ A \sqcup \bigsqcup_{j \in J} X_j \right\rangle$$

is an approximation for $[M, <]$ over A, and it is the $\sqsubseteq$-least upper bound of the approximation chain $\langle K_j, X_j \rangle_{j \in J}$.

Proof. It will be proved that $\langle \bigcup_{j \in J} K_j,\ A \sqcup \bigsqcup_{j \in J} X_j \rangle$ is an approximation for $[M, <]$ over A. It is then clear that it is the $\sqsubseteq$-least upper bound of the $\langle K_j, X_j \rangle_{j \in J}$.

Write M as $\langle Q_i, R_i\rangle_{i\in I}$, and let $T = \langle E_i, D_i\rangle_{i\in I}$ be the signature family of M.

Clearly, $\bar{K} := \bigcup_{j\in J} K_j$ is a $<$-downward closed subset of I, because all the K_j are $<$-downward closed.

The family obtained by adjoining A to $\langle X_j\rangle_{j\in J}$ is a nonempty $\sqsubseteq$-chain of models, because $\langle X_j\rangle_{j\in J}$ is a chain, and $A_j \sqsubseteq X_j$ for all $j \in J$. Hence the completeness property of an institution allows to define $\bar{X} := A \sqcup \bigsqcup_{j\in J} X_j$, which is a model of signature

$$F \sqcup \bigsqcup_{j\in J}(F \sqcup \bigsqcup_{i\in K_j} D_j) = F \sqcup \bigsqcup_{j\in \bar{K}} D_j,$$

which is the signature of the interface

$$\{A\} \wedge \bigwedge_{i\in\bar{K}} R_j.$$

It remains to show that $\bar{X}$ satisfies this interface. Clearly, $X/\bar{F} = A$, because A is a component of the join. For $i \in \bar{K}$, we can choose $j \in J$ such that $i \in K_j$ and hence $D_i \sqsubseteq \mathrm{Sig}(X_j)$. Since $X_j \in \{A\} \sqcup \bigsqcup_{i\in K_j} R_i$, we have $\bar{X}/D_i = X_j/D_i \in R_i$. □

Proof of the Composability Theorem (Thm. 4.1.7).
Let $[M, <]$ be a decomposition of the $\langle \bar{E}, \bar{D}\rangle$-cell $\langle \bar{Q}, \bar{R}\rangle$, where $M = \langle Q_i, R_i\rangle_{i\in I}$ is a system of signature $T = \langle E_i, D_i\rangle_{i\in I}$, and assume that $\rightsquigarrow$ is chain-closed or M is finite. Let $M' = \langle Q'_i, R'_i\rangle_{i\in I}$ be a componentwise universal implementation of M, and let $T' = \langle E'_i, D'_i\rangle_{i\in I}$ be the signature family of M'.

We first verify that M' is a cell system, and that $\bar{D}$ is compatible with $\bigsqcup_{i\in I} D'_i$ (which implies that $\square_{\bar{D}} M'$ is well-defined).

As $[M, <]$ is a decomposition of $\langle \bar{Q}, \bar{R}\rangle$, the ordered signature system $[T, <]$ is a syntactic decomposition of $\langle \bar{E}, \bar{D}\rangle$. For all $i \in I$, the cell signature T'_i is a syntactic refinement of T_i, and so Lemma 3.4.3 yields that

T' is a signature system,

$<_{T'} \subseteq <_T \subseteq <$,

every system site for $[T, <]$ is a system site for $[T', <_{T'}]$, (1)

and Theorem 3.4.1 yields that

$$\bar{D} \text{ is compatible with } \bigsqcup_{i\in I} D'_i, \quad \text{and}$$
$$\bigsqcap_{\bar{D}} T' \text{ is a syntactic refinement of } \langle \bar{E}, \bar{D}\rangle.$$

Write $\langle \bar{Q}', \bar{R}'\rangle := \square_{\bar{D}} M'$ and $\langle \bar{E}', \bar{D}'\rangle := \square_{\bar{D}} T'$.

To show that $\langle \bar{Q}', \bar{R}'\rangle$ is a universal implementation of $\langle \bar{Q}, \bar{R}\rangle$, we use the criterion of Lemma 4.1.9 with $H := \bigsqcup_{i\in I} D_i$ (the conditions $H \sim \bar{E}$ and $\bar{E} \sqcup H \sqsupseteq \bar{E} \sqcup \bar{D}$ follow from the fact that T is a syntactic decomposition of $\langle \bar{E}, \bar{D}\rangle$).

Thus, let F be any site for $\langle \bar{E}, H\rangle$, that is,

$$F \sim \bar{E} \sqcup H \quad \text{and} \quad F \sqcap (\bar{E} \sqcup H) = \bar{E}, \tag{2}$$

and let $A, A' \in Mod(F)$ be such that A is a base for $\langle \bar{Q}, \bar{R}\rangle$ and $A' \underset{F}{\rightsquigarrow} A$.

Note that F is a system site for $[T, <]$, because for every $i \in I$, we have

$$\begin{aligned} F \sqcap (E_i \sqcup D_i) &\sqsubseteq F \sqcap \Big(\bar{E} \sqcup \bigsqcup_{i\in I} (E_i \sqcup D_i)\Big) \\ &= F \sqcap \Big(\bar{E} \sqcup \bigsqcup_{i\in I} D_i\Big) && \text{(Lemma 3.3.2)} \\ &= \bar{E} && \text{(by (2))}, \end{aligned}$$

and therefore

$$\begin{aligned} E_i &= \Big(\bar{E} \sqcup \bigsqcup_{j\ll i} D_j\Big) \sqcap (E_i \sqcup D_i) && (\bar{E} \text{ is system site for } [T, <]) \\ &\sqsubseteq \Big(F \sqcup \bigsqcup_{j\ll i} D_j\Big) \sqcap (E_i \sqcup D_i) \\ &= \Big(F \sqcup \bar{E} \sqcup \bigsqcup_{j\ll i} D_j\Big) \sqcap (E_i \sqcup D_i) \\ &= \Big(F \sqcap (E_i \sqcup D_i)\Big) \sqcup \Big(\Big(\bar{E} \sqcup \bigsqcup_{j\ll i} D_j\Big) \sqcap (E_i \sqcup D_i)\Big) \\ &= \Big(F \sqcap (E_i \sqcup D_i) \sqcap (E_i \sqcup D_i)\Big) \sqcup \Big(\Big(\bar{E} \sqcup \bigsqcup_{j\ll i} D_j\Big) \sqcap (E_i \sqcup D_i)\Big) \\ &\sqsubseteq \Big(\bar{E} \sqcap (E_i \sqcup D_i)\Big) \sqcup \Big(\Big(\bar{E} \sqcup \bigsqcup_{j\ll i} D_j\Big) \sqcap (E_i \sqcup D_i)\Big) && \text{(as just shown)} \\ &= \Big(\bar{E} \sqcup \bigsqcup_{j\ll i} D_j\Big) \sqcap (E_i \sqcup D_i) \\ &= E_i, \end{aligned}$$

which means that

$$E_i = \left(F \cup \bigsqcup_{j \ll i} D_j\right) \sqcap (E_i \cup D_i).$$

By (1), the signature F also is a system site for $[T', <_{T'}]$.

We now need a definition and two lemmas.

Definition.

For $K \subseteq I$ $<$-downward closed, let

$$S_K := F \cup \bigsqcup_{i \in K} D_i.$$

For $K \subseteq I$ $<_{T'}$-downward closed, let

$$S'_K := F \cup \bigsqcup_{i \in K} D'_i.$$

A *joint approximation* is a quadruple

$$\langle K', X', K, X \rangle$$

such that

$\langle K', X' \rangle$ is an aproximation for $[M', <_{T'}]$ on A',
$\langle K, X \rangle$ is an approximation for $[M, <]$ on A,
$K' \supseteq K$, and
$X'/S_K \overset{S_K}{\rightsquigarrow} X$.

If $\langle K', X', K, X \rangle$ and $\langle L', Y', L, Y \rangle$ are joint approximations, say that $\langle K', X', K, X \rangle$ is *included* in $\langle L', Y', L, Y \rangle$ (written "$\langle K', X', K, X \rangle \sqsubseteq \langle L', Y', L, Y \rangle$"), if

$$\langle K', X' \rangle \sqsubseteq \langle L', Y' \rangle \quad \text{and} \quad \langle K, X \rangle \sqsubseteq \langle L, Y \rangle.$$

Observe that if $\langle K', X', K, X \rangle$ is a joint approximation, then $X' \in Mod(S'_{K'})$ and $X \in Mod(S_K)$. Since $K' \supseteq K$, we have

$$\begin{aligned} S'_{K'} &= F \cup \bigsqcup_{i \in K'} D'_i \\ &\sqsupseteq F \cup \bigsqcup_{i \in K} D'_i \\ &= F \cup \bigsqcup_{i \in K} D_i \qquad \text{(Lemma 3.4.2)} \\ &= S_K, \end{aligned}$$

and so the requirement "$X'/S_K \underset{S_K}{\rightsquigarrow} X$" for a joint approximation is syntactically well-formed.

Note also that all signatures of the form S_K or S'_K are compatible, because they are included in $\bar{F} \cup \bigsqcup_{i \in I} D_i$ or $F \cup \bigsqcup_{i \in I} D'_i$, and these signatures are equal by Lemma 3.4.2.

Lemma 1 (Chain-completeness).
Let $\langle K', X', K, X \rangle$ be a joint approximation. Then in the set of all joint approximations that include $\langle K', X', K, X \rangle$, every $\sqsubseteq$-chain has an upper bound.

Proof. Let $\langle L'_j, Y'_j, L_j, Y_j \rangle_{j \in J}$ be a $\sqsubseteq$-chain of joint approximations that include $\langle K', X', K, X \rangle$. Fix $\bullet \notin J$, let $J^\bullet := J + \{\bullet\}$, and define $\langle L'_\bullet, Y'_\bullet, L_\bullet, Y_\bullet \rangle := \langle K', X', K, X \rangle$. This makes $\langle L'_j, Y'_j, L_j, Y_j \rangle_{j \in J^\bullet}$ a nonempty $\sqsubseteq$-chain of joint approximations.

Since $\langle L'_j, Y'_j \rangle_{j \in J^\bullet}$ is a $\sqsubseteq$-chain of approximations for $[M', <_{T'}]$ over A', by Lemma 4.1.11 we can choose an upper bound $\langle N', Z' \rangle$ for this chain. By the same lemma, since $\langle L_j, Y_j \rangle_{j \in J^\bullet}$ is a $\sqsubseteq$-chain of approximations for $[M, <]$ over A, the pair $\langle N, Z \rangle$ is an upper bound for this chain, where

$$N = \bigcup_{j \in J^\bullet} L_j \quad \text{and}$$
$$Z = A \cup \bigsqcup_{j \in J^\bullet} L_j = \bigsqcup_{j \in J^\bullet} L_j \qquad (\text{as } J^\bullet \neq \emptyset).$$

We shall now show that $\langle N', Z', N, Z \rangle$ is a joint approximation. From its definition, it is clear that it is then an upper bound of $\langle L'_j, Y'_j, L_j, Y_j \rangle_{j \in J^\bullet}$, hence of the original J-indexed family, and that it includes $\langle K', X', K, X \rangle$.

By construction, $\langle N', Z' \rangle$ is an approximation for $[M', <_{T'}]$ on A', and $\langle N, Z \rangle$ is an approximation for $[M, <]$ on A. Also,

$$N = \bigcup_{j \in J^\bullet} L_j = K \cup \bigcup_{j \in J} L_j$$
$$\subseteq K' \cup \bigcup_{j \in J} L'_j = \bigcup_{j \in J^\bullet} L'_j \subseteq N'.$$

It remains to show that $Z'/S_N \underset{S_N}{\rightsquigarrow} Z$. Here we use the assumption that $\rightsquigarrow$ is chain-closed or M is finite.

If M is finite (*i. e.*, I is finite), we can choose $k \in J^\bullet$ such that $N = L_k$, because $N = \bigcup_{j \in J^\bullet} L_j = \bigcup\{ L_j \mid j \in J^\bullet \}$, and the set $\{ L_j \mid j \in J^\bullet \}$ is a nonempty finite $\subseteq$-chain (the L_j $(j \in J)$ form a $\subseteq$-chain, and are subsets of the finite set I). Hence

$$Z'/S_N = Z'/S_{L_k} = (Z'/S'_{L'_k})/S_{L_k} = Y'_k/S_{L_k}$$
$$\underset{S_{L_k}=S_N}{\rightsquigarrow} Y_k = Z/S_{L_k} = Z/S_N = Z.$$

If $\rightsquigarrow$ is chain-closed, consider the signature chain $\langle S_{L_j} \rangle_{j \in J^\bullet}$. This chain is compatible, because

$$S_{L_j} = F \sqcup \bigsqcup_{i \in L_j} D_i \sqsubseteq F \sqcup \bigsqcup_{i \in I} D_i \qquad \text{for all } j \in J^\bullet.$$

The join of the chain is

$$\begin{aligned} \bigsqcup_{j \in J^\bullet} S_{L_j} &= \bigsqcup_{j \in J^\bullet} (F \sqcup \bigsqcup_{i \in L_j} D_i) \\ &= F \sqcup \bigsqcup_{i \in \bigcup_{j \in J^\bullet} L_j} D_i \qquad (\text{as } J^\bullet \neq \emptyset) \\ &= F \sqcup \bigsqcup_{i \in N} D_i \\ &= S_N. \end{aligned}$$

Because for all $j \in J^\bullet$, we have

$$(Z'/S_N)/S_{L_j} = Z'/S_{L_j} = Y'_j/S_{L_j} \underset{S_{L_j}}{\rightsquigarrow} Y_j = Z/S_{L_j},$$

the chain-completeness of $\rightsquigarrow$ yields that $Z'/S_N \underset{S_N}{\rightsquigarrow} Z$.

In either case, $Z'/S_N \underset{S_N}{\rightsquigarrow} Z$, and hence $\langle N', Z', N, Z \rangle$ is a joint approximation. Thus, Lemma 1 is proved.

Lemma 2 (Completion of joint approximations).
For every joint approximation $\langle K', X', K, X \rangle$, there exist $Y', Y \in Mod(S_I)$ such that $\langle I, Y', I, Y \rangle$ is a joint approximation and

$$\langle K', X', K, X \rangle \sqsubseteq \langle I, Y', I, Y \rangle.$$

Proof. Let $\langle K', X', K, X\rangle$ be a joint approximation. Consider the set $\mathcal{U}$ of joint approximations that include $\langle K', X', K, X\rangle$. By the previous lemma, every $\sqsubseteq$-chain in $\mathcal{U}$ has an upper bound in $\mathcal{U}$. By Zorn's lemma, $\mathcal{U}$ has a $\sqsubseteq$-maximal element.

Let $\langle L', Y', L, Y\rangle$ be a $\sqsubseteq$-maximal element of $\mathcal{U}$. We shall prove below that $L = I$. Since $L' \supseteq L$, it is then clear that $L' = I$ also, and that Y' and Y are the models postulated by the Lemma.

To prove $L = I$, suppose that $L \neq I$, *i.e.*, that L is a proper subset of I. Then we can choose j so that it is $\ll$-minimal in $I \setminus L$.

Note first that Y is a base for $\langle Q_j, R_j\rangle$:

Y is a model of signature $S_L = F \sqcup \bigsqcup_{i\in L} D_i$. We saw above that F is a system site for $[T, <]$; by Lemma 3.4.3 (c), F also is a system site for $[T, <_T]$; since L is $<$-downward closed and hence $<_T$-downward closed, Lemma 3.5.2 applies and yields that S_L is a site for $\langle E_j, D_j\rangle$.

Since $\langle L, Y\rangle$ is an approximation for $[M, <]$ over A, we have $Y \in \{A\} \wedge \bigwedge_{i\in L} R_i$, and hence

$$
\begin{aligned}
Y/E_j &\in \big(\{A\} \wedge \bigwedge_{i\in L} R_i\big)/\!\!/E_j \\
&\subseteq \big(\{A\}/\bar{E} \wedge \bigwedge_{i\in L} R_i\big)/\!\!/E_j && \text{(Prop. 3.1.9 and } \bar{E} \sqcup \textstyle\bigsqcup_{i\in L} D_i \sqsupseteq E_j\text{)} \\
&= \big(\{A/\bar{E}\} \wedge \bigwedge_{i\in L} R_i\big)/\!\!/E_j \\
&\subseteq \big(\{A/\bar{E}\} \wedge \bigwedge_{i\ll j} R_i\big)/\!\!/E_j && \text{(Prop. 3.1.8 and } \ll\text{-minimality of } j\text{)} \\
&\subseteq \big(\bar{Q} \wedge \bigwedge_{i\ll j} R_i\big)/\!\!/E_j && \text{(Prop. 3.1.9 and } A/\bar{E} \in \bar{Q}\text{)} \\
&\subseteq Q_j && \text{(by decomposition).}
\end{aligned}
$$

Thus, Y is a base for $\langle Q_j, R_j\rangle$.

We now distinguish the two cases $j \notin L'$ and $j \in L'$. In either case we shall construct a joint approximation that strictly includes $\langle L', Y', L, Y\rangle$.

Case I: Suppose that $j \notin L'$. Since Y is a base for $\langle Q_j, R_j \rangle$, $Y'/S_L \underset{S_L}{\rightsquigarrow} Y$, and $\langle Q'_j, R'_j \rangle$ is a universal implementation of $\langle Q_j, R_j \rangle$, it follows that Y'/S_L is a base for $\langle Q'_j, R'_j \rangle$, and that we can choose a result Z of $\langle Q'_j, R'_j \rangle$ on Y'/S_L.

Note that $S'_L = S_L$ by Lemma 3.4.2, because L is $<$-downward closed. Also, the signatures $S'_{L'}$ and S'_L are sites for $\langle E'_j, D'_j \rangle$ by Lemma 3.5.2, since L' and L are $<_{T'}$-downward closed and F is a system site for $[T', <_{T'}]$.

The signature of Y' is $S'_{L'}$, and the signature of Z is $S_L \sqcup D'_j$. These signatures are compatible, because they are included in $\bar{F} \sqcup \bigsqcup_{i \in I} D_i = \bar{F} \sqcup \bigsqcup_{i \in I} D'_i$. Their intersection is

$$\begin{aligned}
S'_{L'} \sqcap (S_L \sqcup D'_j) &= S'_{L'} \sqcap (S'_L \sqcup D'_j) \\
&= (S'_{L'} \sqcap S'_L) \sqcup (S'_{L'} \sqcap D'_j) \\
&= S'_L \sqcup (S'_{L'} \sqcap D'_j) && (L \subseteq L') \\
&\sqsubseteq S'_L \sqcup E'_j && (S'_{L'} \text{ is a site for } \langle E'_j, D'_j \rangle) \\
&= S'_L && (S'_L \text{ is a site for } \langle E'_j, D'_j \rangle) \\
&= S_L.
\end{aligned}$$

By definition of Z, we have $Z/S_L = Y'/S_L$, and so we can form the join $Y' \sqcup Z$.

We now show that

$$\langle L' + \{j\},\ Y' \sqcup Z,\ L,\ Y \rangle$$

is a joint approximation.

The pair $\langle L' + \{j\},\ Y' \sqcup Z \rangle$ is an approximation for $[M', <_{T'}]$ over A: the set $L' + \{j\}$ is $<_{T'}$-downward closed, because L' is $<_{T'}$-downward closed, and because $i <_{T'} j$ implies $i < j$, hence $i \in L \subseteq L'$. The signature of $Y' \sqcup Z$ is $S'_{L'} \sqcup D'_j = S'_{L'+\{j\}}$, and $(Y' \sqcup Z) \in \{A'\} \wedge \bigwedge_{i \in L'+\{j\}} R'_i$, because

- $(Y' \sqcup Z)/F = Y'/F = A'$,
- $(Y' \sqcup Z)/D'_j = Z/D'_j \in R'_j$, and
- for $i \in L'$: $(Y' \sqcup Z)/D'_i = Y'/D'_i \in R'_i$.

Since $(Y' \sqcup Z)/S_L = Y'/S_L \underset{S_L}{\rightsquigarrow} Y$, the quadruple $\langle L' + \{j\},\ Y' \sqcup Z,\ L,\ Y \rangle$ is a joint approximation, and, obviously, it strictly includes $\langle L', Y', L, Y \rangle$.

Case II: Suppose that $j \in L'$. Since Y is a base for $\langle Q_j, R_j \rangle$, $Y'/S_L \underset{S_L}{\rightsquigarrow} Y$, and $\langle Q'_j, R'_j \rangle$ is a universal implementation of $\langle Q_j, R_j \rangle$, it follows that Y'/S_L is a base for $\langle Q'_j, R'_j \rangle$. Now $Y'/S_{L+\{j\}}$ is a result of $\langle Q'_j, R'_j \rangle$ on Y'/S_L: since S_L is a site for $\langle E_j, D_j \rangle$ and $\langle E'_j, D'_j \rangle$ is a syntactic refinement of $\langle E_j, D_j \rangle$, we have

$$S_{L+\{j\}} = S_L \sqcup D_j = S_L \sqcup D'_j,$$

which is the result signature of $\langle E'_j, D'_j \rangle$ on S_L, and $(Y'/S_{L+\{j\}})/D'_j = Y'/D'_j \in R'_j$ because $\langle L', Y' \rangle$ is an approximation and $j \in L'$.

As $\langle Q'_j, R'_j \rangle$ is a universal implementation of $\langle Q_j, R_j \rangle$, we can choose a result Z of $\langle Q_j, R_j \rangle$ on Y such that $Y'/S_{L+\{j\}} \underset{S_{L+\{j\}}}{\rightsquigarrow} Z$.

We now show that

$$\langle L',\, Y',\, L+\{j\},\, Z \rangle$$

is a joint approximation.

The pair $\langle L+\{j\}, Z \rangle$ is an approximation for $[M, <]$ over A: the set $L+\{j\}$ is $<$-downward closed, since L is $<$-downward closed and j is $\ll$-minimal (hence $<$-minimal) in $I \setminus L$. The model Z is of signature $S_{L+\{j\}}$, and $Z \in \{A\} \wedge \bigwedge_{i \in L+\{j\}} R_i$, because

- $Z/F = (Z/S_L)/F = Y/F = A$,
- $Z/D_j \in R_j$ (Z is defined to be a result of $\langle Q_j, R_j \rangle$ on Y), and
- for $i \in L$: $Z/D_i = (Z/S_L)/D_i = Y/D_i \in R_i$.

Now $L+\{j\} \subseteq L'$, as $L \subseteq L'$ and $j \in L'$, and $Y'/S_{L+\{j\}} \underset{S_{L+\{j\}}}{\rightsquigarrow} Z$ by definition of Z. Hence $\langle L', Y', L+\{j\}, Z \rangle$ is a joint approximation, and, obviously, it strictly includes $\langle L', Y', L, Y \rangle$.

In either of the cases $j \notin L'$ and $j \in L'$, we have constructed a joint approximation that strictly includes $\langle L', Y', L, Y \rangle$. This contradicts the $\sqsubseteq$-maximality of $\langle L', Y', L, Y \rangle$ in $\mathcal{U}$. Hence the assumption that $L \neq I$ must be false, and thus we have proved that $L = L' = I$, which concludes the proof of Lemma 2.

To prove the composability theorem, we now show that $\langle \bar{Q}', \bar{R}' \rangle$ is a universal implementation of $\langle \bar{Q}, \bar{R} \rangle$ by verifying the condition of the renaming lemma for the models A and A' of signature F that have already been introduced.

First, we show that A' is a base for $\langle \bar{Q}', \bar{R}' \rangle$. As F is a site for $\langle \bar{E}, H \rangle$, it is a site for $\langle \bar{E}, \bar{D} \rangle$ by Lemma 4.1.8, and since $\langle \bar{E}', \bar{D}' \rangle$ is a syntactic refinement of $\langle \bar{E}, \bar{D} \rangle$, F is a site for $\langle \bar{E}', \bar{D}' \rangle$ also. Recall that $\bar{Q}'$ is defined as

$$\bar{Q}' = \{ B \in Mod(\bar{E}') \mid (\{B\} \wedge \bigwedge_{k \ll_{T'} i} R'_k) /\!\!/ E'_i \subseteq Q'_i \text{ for all } i \in I \}.$$

To show that $A'/\bar{E}' \in \bar{Q}'$, consider any $i \in I$ and suppose that

$$(\{A'/\bar{E}'\} \wedge \bigwedge_{k \ll_{T'} i} R'_k) /\!\!/ E'_i \not\subseteq Q'_i,$$

which means that we can choose $X \in \{A'/\bar{E}'\} \wedge \bigwedge_{k \ll_{T'} i} R'_k$ such that $X/E'_i \notin Q_i$.

The meet of the signatures of A' and of X is $F \sqcap (\bar{E}' \sqcup \bigsqcup_{k \ll_{T'} i} D'_k)$, which is included in $\bar{E}'$ by Lemma 3.3.3 (F is a system site for $[T', <_{T'}]$). Since $X/\bar{E}' \in \{A/\bar{E}'\}$, A' and X have the same reduct on the meet of their signatures, and we can form $Y := A' \sqcup X$.

Let $K' := \{ k \mid k \ll_{T'} i \}$. Then

$$\langle K', Y, \emptyset, A \rangle$$

is a joint approximation, as is easily checked. By Lemma 2, there exist Z' and Z in $Mod(S_I)$ such that

$$\langle I, Z', I, Z \rangle$$

is a joint approximation which includes $\langle K', Y, \emptyset, A \rangle$.

Let $K := \{ k \mid k \ll i \}$. Then Z/S_K is a base for $\langle Q_i, R_i \rangle$, because S_K is a site for $\langle E_i, D_i \rangle$ by Lemma 3.5.2, and because

$$\begin{aligned}
(Z/S_K)/E_i &= Z/E_i \\
&\in (\{A\} \wedge \bigwedge_{k \in I} R_k) /\!\!/ E_i \\
&\subseteq (\bar{Q} \wedge \bigwedge_{k \ll i} R_k) /\!\!/ E_i && \text{(Prop. 3.1.8 and 3.1.9)} \\
&\subseteq Q_i && ([M, <] \text{ is a decomposition of } \langle \bar{Q}, \bar{R} \rangle).
\end{aligned}$$

As $Z' \rightsquigarrow Z$, hence $Z'/S_K \rightsquigarrow Z/S_K$, and $\langle Q_i', R_i' \rangle$ is a universal implementation of $\langle Q_i, R_i \rangle$, it follows that Z'/S_K is a base for $\langle Q_i', R_i' \rangle$. But this implies that

$$\begin{aligned} X/E_i' &= \big(Y/(\bar{E}' \sqcup \bigsqcup_{k \ll_T i} D_i')\big)/E_i' = Y/E_i' \\ &= (Z'/S_{K'}')/E_i' = Z'/E_i' = (Z'/S_K)/E_i' \in Q_i', \end{aligned}$$

which contradicts the definition of X. Hence $A'/\bar{E}' \in \bar{Q}'$, and thus A' is a base for $\langle \bar{Q}', \bar{R}' \rangle$.

Next, we show that there exists a result of $\langle \bar{Q}', \bar{R}' \rangle$ on A'. The quadruple

$$\langle \emptyset, A', \emptyset, A \rangle$$

is a joint approximation, and by Lemma 2, there exists a joint approximation

$$\langle I, Y', I, Y \rangle.$$

But then $Y'/(F \sqcup \bar{D}')$ is a result of $\langle \bar{Q}', \bar{R}' \rangle$ on A': We have

$$Y'/(F \sqcup \bar{D}')/F = Y'/F = A',$$

and

$$\begin{aligned} Y'/(F \sqcup \bar{D}')/\bar{D}' &= Y'/\bar{D}' \\ &\in \big(\{A'\} \wedge \bigwedge_{i \in I} R_i'\big) /\!\!/ \bar{D}' \\ &\subseteq \big(\bar{Q}' \wedge \bigwedge_{i \in I} R_i'\big) /\!\!/ \bar{D}' \qquad \text{(Prop. 3.1.9 and } \{A'\}/\bar{E}' \subseteq \bar{Q}') \\ &= \bar{R}'. \end{aligned}$$

Finally, let B' be any result of $\langle \bar{Q}', \bar{R}' \rangle$ on A'. We show that B' represents a result of $\langle Q, R \rangle$ on A.

Since $B'/\bar{D}' \in \bar{R}' = \big(\bar{Q}' \wedge \bigwedge_{i \in I} R_i'\big) /\!\!/ \bar{D}'$, we can choose $C' \in \big(\bar{Q}' \wedge \bigwedge_{i \in I} R_i'\big)$ such that $C'/\bar{D}' = B'/\bar{D}'$. The meet of the signatures of B' and C' is

$$\begin{aligned} (F \sqcup \bar{D}') \sqcap \big(\bar{E}' \sqcup \bigsqcup_{i \in I} D_i'\big) &\sqsubseteq F \sqcap \big(\bar{E}' \sqcup \bigsqcup_{i \in I} D_i'\big) \sqcup \bar{D}' \\ &\sqsubseteq \bar{E}' \sqcup \bar{D}' \qquad \text{(Lemma 3.3.3)} \\ &= \bar{D}' \qquad (\langle \bar{E}', \bar{D}' \rangle \text{ is a syntactic composition}), \end{aligned}$$

hence the reducts of B' and C' to this signature agree, and we can form $X' := B' \sqcup C'$, which is a model of signature

$$F \sqcup \bar{D}' \sqcup \bar{E}' \sqcup \bigsqcup_{i \in I} D'_i = F \sqcup \bigsqcup_{i \in I} D'_i \qquad (\text{as } \bar{D}' \sqsubseteq \bar{E}' \sqcup \bigsqcup_{i \in I} D'_i \text{ and } \bar{E}' \sqsubseteq F)$$
$$= S'_I.$$

The pair $\langle I, X' \rangle$ is an approximation for $[M', <_{T'}]$ on A', because

- $X'/F = B'/F = A'$, and
- $X'/D'_i = C'/D'_i \in R'_i$ for all $i \in I$.

Since $X'/F = A' \underset{F}{\rightsquigarrow} A$, the quadruple

$$\langle I, X', \emptyset, A \rangle$$

is a joint approximation. By Lemma 2, there exist $Y, Y' \in Mod(S_I)$ such that

$$\langle I, Y', I, Y \rangle$$

is a joint approximation that includes $\langle I, X', \emptyset, A \rangle$. In particular, this means that $Y' = X'$.

Define $B := Y/(F \sqcup \bar{D})$. We show that B is the desired result of $\langle \bar{Q}, \bar{R} \rangle$ on A:

$$\begin{aligned}
B/F &= Y/F = A, \\
B/\bar{D} &= Y/\bar{D} \\
&\in (\{A\} \wedge \bigwedge_{i \in I} R_i) /\!\!/ \bar{D} \\
&\subseteq (\bar{Q} \wedge \bigwedge_{i \in I} R_i) /\!\!/ \bar{D} && \text{(Prop. 3.1.9, using } \{A\} /\!\!/ \bar{E} \subseteq \bar{Q}) \\
&\subseteq \bar{R} && ([M, <] \text{ is a decomposition of } \langle \bar{Q}, \bar{R} \rangle).
\end{aligned}$$

Hence B is a result of $\langle \bar{Q}, \bar{R} \rangle$ on A.

Finally, since $Y' \underset{S_I}{\rightsquigarrow} Y$, we have

$$\begin{aligned}
B' &= X'/(F \sqcup \bar{D}') \\
&= X'/(F \sqcup \bar{D}) && (\langle \bar{E}', \bar{D}' \rangle \text{ is a syntactic refinement of } \langle \bar{E}, \bar{D} \rangle) \\
&= Y'/(F \sqcup \bar{D}) \\
&\underset{F \sqcup \bar{D}}{\rightsquigarrow} Y/(F \sqcup \bar{D}) && \text{(Functorality)} \\
&= B.
\end{aligned}$$

Hence B' represents the result B of $\langle \bar{Q}, \bar{R} \rangle$ on A.

We have verified the condition of the renaming lemma (4.1.9), from which it follows that $\langle \bar{Q}', \bar{R}' \rangle$ is a universal implementation of $\langle \bar{Q}, \bar{R} \rangle$. This concludes the proof of the composability theorem for universal implementations. □

As a corollary of the composability theorem, we obtain the "composability of refinements", which has already been cited and used in Chapter 3.

4.1.12 Theorem (Composability of Refinements).
Let $[M, <]$ be a decomposition of the $\langle \bar{E}, \bar{D} \rangle$-cell $\langle \bar{Q}, \bar{R} \rangle$, and let M' be a componentwise refinement of M. Then M' is a cell system, the signature $\bar{D}$ is compatible with the join of the definition signatures of M', and

$$\square_{\bar{D}} M' \text{ is a refinement of } \langle \bar{Q}, \bar{R} \rangle.$$

Proof. Apply the Composability Theorem to the chain-closed representation relation "Equality" (Proposition 4.1.2). By Proposition 4.1.6, the "universal implementation" concept for this representation relation is just refinement, and so the Composition Theorem turns into the theorem above. □

Since the Composability Theorem of Refinements has now been proved, the theorems of Section 3.5, in whose proof this theorem was employed occasionally, have now become proper theorems and may be used in proofs.

4.2 Introducing Visible Sorts

The three sections to follow will deal with representation relations between partial algebras. However, the relations to be studied are not simply relations between partial algebras of a signature Σ, but depend on an additional parameter V, a subset of the sorts of Σ, which contains "visible" sorts that are to be "preserved" by the representation. The present section explains the need for this parameter and constructs a new institution syntax of "tagged algebraic signatures", that is, algebraic signatures with a distinguished subset of visible sorts.

Why is it necessary to distinguish visible sorts? The answer is that the "standard" representation relation between algebras, which treats all sorts alike, is too general to be practically useful, because for every signature Σ, there exists a computable Σ-algebra that represents every total Σ-algebra with countable carriers. Such a "universal representation" is easy to implement, but cannot be regarded as useful. But then that would be too much to expect, because it would make the implementation task trivial.

In the following, this point will be made more precise. Only total algebras will be considered, because the majority of the literature deals with total algebras only, and only for these can a "standard" representation concept be said to exist.

The following definition presents a rather restrictive representation relation between total algebras, the "inverse image" relation.

4.2.1 Definition. Let $\Sigma = \langle S, \alpha\colon F \to S^+ \rangle$ be an algebraic signature. A *homomorphism* $h\colon A \to B$ between total Σ-algebras A and B is an S-sorted total function from A/S to B/S such that whenever $f\colon s_1 \ldots s_n \to r$ in Σ and $x_i \in A_{s_i}$ for $i \in \{1, \ldots, n\}$, then

$$h_r A_f(x_1, \ldots, x_n) = B_f(h_{s_1} x_1, \ldots, h_{s_n} x_n).$$

The homomorphism is called *surjective* (Notation: "$h\colon A \twoheadrightarrow B$"), if all its components are surjective functions. If there exists a surjective homomorphism

h: $A \twoheadrightarrow B$, call B a *homomorphic image* of A, and A an *inverse homomorphic image* (short "inverse image") of B. □

The "inverse image" relation is included in almost all the representation relations (excepting the relation "isomorphic") that have been proposed in the literature, for example in [SW 82, p. 13], [Lipeck 83, p. 52] and [KA 84, p. 322]. Yet it is still too general to be practically useful. This can be seen using the well-known "term algebra" $T_\Sigma(X)$ (also called "free Σ-algebra generated by X"), which consists of terms constructed from the operators of the algebraic signature $\Sigma = \langle S, \alpha: F \to S^+ \rangle$ and from values in the S-sorted set X (this "term" notion is formally defined in Definition 4.3.1 below; the operations of the term algebra are the evident term constructor functions). This algebra, together with the embedding η: $X \to (T_\Sigma(X))/S$, which maps each element x of X to the term consisting of just x (to be precise, if $x \in X_s$, then $\eta_s(x) = \langle s, x \rangle$), has the following "universality" property:

> "For every S-sorted function f: $X \to A/S$, where A is a Σ-algebra, there exists a unique homomorphism $f^\sharp$: $T_\Sigma(X) \to A$ that satisfies $\eta \,;\, f^\sharp = f$."

Let $\{\mathbf{N}\}^S$ be the S-sorted set $\langle \mathbf{N} \rangle_{s \in S}$, all of whose components are equal to $\mathbf{N}$, the set of natural numbers. We then have the following theorem.

4.2.2 Theorem. *Let $\Sigma = \langle S, \alpha: F \to S^+ \rangle$ be an algebraic signature. Then the algebra $T_\Sigma(\{\mathbf{N}\}^S)$ is an inverse image of every total Σ-algebra whose carriers are nonempty and finite or countable.*

Proof. Let A be a total Σ-algebra with nonempty and finite or countable carriers. Then for each sort $s \in S$ there exists an enumeration of A_s, that is, a surjective function from $\mathbf{N}$ to A_s. By the axiom of choice, there exists a surjective S-sorted function f: $\{\mathbf{N}\}^S \twoheadrightarrow A/S$. The universality property of $T_\Sigma(\{\mathbf{N}\}^S)$ yields that there exists a homomorphism $f^\sharp$: $T_\Sigma(\{\mathbf{N}\}^S) \to A$. Since $\eta \,;\, f^\sharp = f$ and f is surjective, $f^\sharp$ also is surjective. Hence $T_\Sigma(\{\mathbf{N}\}^S)$ is an inverse image of A. □

According to this theorem, $T_\Sigma(\{N\}^S)$ is an inverse image, and hence a "representation" in the sense of the papers quoted above, of every Σ-algebra with nonempty and finite or countable carriers. Also, if S is finite, $T_\Sigma(\{N\}^S)$ is computable [MG 85, Theorem 30] and trivial to realize in programming notations such as HOPE [BMS 81] and ML [HMM 86]. But it is clear that this representation is useless: if we want to determine, say, the result of the expression "$f(x)$" in A, where f is a function of the algebra A and x is a data value, we would have to encode x as a natural number, n say, the representation would yield the term "$f(n)$", and we would then have to decode this term by decoding n and applying f to the result (*i. e.*, to x). The "real work" of determining the value of f on x is not done by the representation, but by the decoding.

But the example points to a solution of the problem. What is wrong is that the output we get (the term "$f(n)$") is in a form determined by the representation and has to be decoded to arrive at the desired result (the value $f(x)$). As we have seen, this allows one to design representations in such a way that the decoding, rather than the representation, performs the actual computation.

A solution to the problem is obtained by demanding that the encoding and decoding of input and output be trivial to perform. The most extreme interpretation of the term "trivial" here is that the sorts used for input and output are represented by themselves and that the homomorphism must be the identity. It appears that even slightly more general interpretations of "trivial" (*e. g.*, bijections between input/output sorts and their representations) admit representations like the one above, in which the essential work is performed in the encoding and decoding steps.

If we were to demand trivial representation of all the sorts of an algebra, then an algebra could only be represented by itself. However, an algebra represents the data types and functions of a program, and typically only a subset of the types of a program is used for input and output of data values. Since the purpose of requiring trivial representation is to avoid complex encoding and decoding of input and output, it is only necessary to require trivial representation of the sorts used for input and output, which will be called the "visible" sorts of an algebra. Since data values of the other, "hidden" sorts of an algebra are not the

subject of input and output, no restriction on the representation of hidden sorts is necessary.

A more precise concept of "visible" sorts defines them to be the sorts which can be accessed in ways beyond those represented by the functions of the algebra; this includes the sorts accessed by input/output operations, but also sorts that play special rôles in a programming notation—for example, the type of booleans is special with respect to the **if** construct, the type of integers often plays a special rôle in **array** data type definitions.

As we shall see in Sections 4.3 and 4.4 below, the distinction between visible and hidden sorts allows to generalize the conventional data type representation concepts that are based on homomorphisms: it is no longer necessary to have a *function* from the representation algebra to the algebra it represents, rather, a special kind of relation, called a "correspondence", is sufficient.

It might appear at first that by declaring sorts to be "visible" and hence allowing only identical representations for these sorts, the freedom is lost to select a representation for input and output data of a program. This freedom can be regained as follows.

Suppose that a Σ-algebra A is given, and that the sorts in the sets I and O are to be used for input and output. Construct a signature Σ^+ by duplicating the sorts in I and O, as shown in Figure 4-2. For each input sort s, we have a new sort s' and a new function $\hat{s}$: $s' \rightarrow s$, and for each output sort u, we have a new sort u'' and a new function symbol $\check{u}$: $u \rightarrow u''$. The new sorts are declared visible, while the original sorts of Σ are hidden.

We construct a Σ^+-algebra A^+ from A as shown in Figure 4-3: the interpretations of the new sort symbols are the same as those of their originals, and the new functions are identities.

A representation B^+ of A^+ is shown in Figure 4-4. The algebra B^+ contains arbitrary representations B_x^+, B_y^+, B_s^+, B_t^+, and B_u^+ of the sorts of A. However, and this is the crucial difference to the representation by $\mathrm{T}_\Sigma(\{\mathrm{N}\}^S)$ shown above, the algebra B^+ also contains explicit conversion functions $B_{\hat{s}}^+$ and $B_{\hat{t}}^+$ that allow to encode the original input values from A_s and A_t, and conversion functions

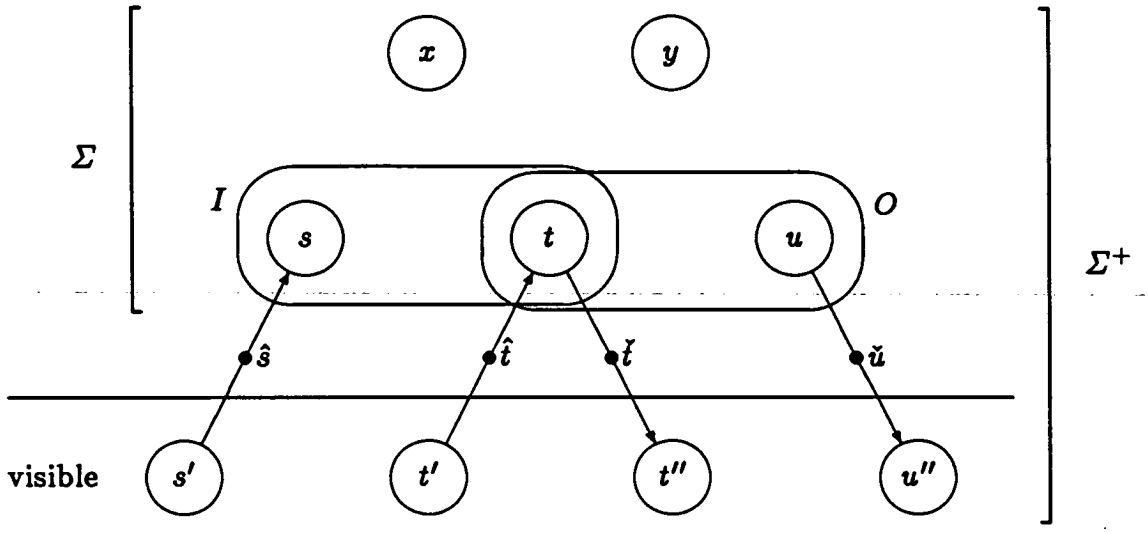


Figure 4-2: The signature Σ^+

$B^+_{\check{t}}$ and $B^+_{\check{u}}$ that allow to decode the output values of B^+_t and B^+_u into values of A_t and A_u. Hence the algebra B^+ allows to perform computations with input and output values in the original, "abstract" carriers of A. In the representation by $T_\Sigma(\{N\}^S)$, no encoding and decoding operations are provided, and it is in the decoding step that the essential computation is performed.

We have seen that for a representation concept to be useful, it is necessary to distinguish a set of "visible" sorts which must be preserved in the representation. The "visible sorts" idea is well known in data abstraction: in a large number of papers, including early ones, we find that data abstraction is treated in the context of "data type extensions", where some fixed, given ("primitive") sorts are to be enriched by one or more "types of interest" ([LZ 75], [GH 78], [Wand 79], [BW 82], and papers dealing with behavioural representation concepts, which are cited in Sections 4.3 and 4.4 below).

In order to give representation concepts with visible sorts the "functorality" property of a representation relation (Axiom (b) of Def. 4.1.1), it is necessary to work in an institution of "tagged algebraic signatures", that is, of algebraic

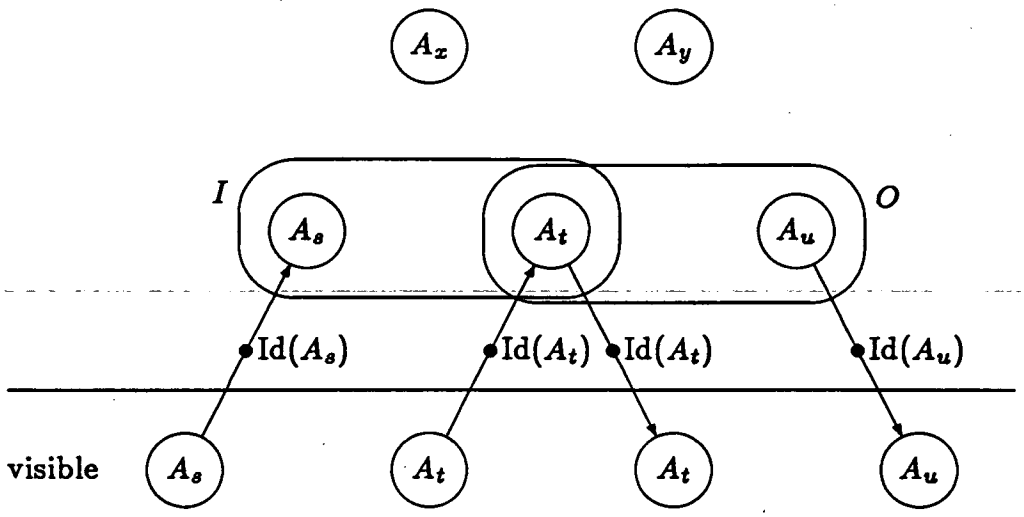


Figure 4-3: The Σ^+-algebra A^+

signatures with a distinguished subset of visible sorts. The remainder of this section is devoted to constructing this institution.

4.2.3 Definition. A *tagged algebraic signature* (short "tagged signature") is a pair

$$\langle \Sigma, V \rangle,$$

where Σ is an algebraic signature, and V is a subset of the sort set of Σ.

A *signature morphism* σ: $\langle \Sigma, V \rangle \to \langle \Sigma', V' \rangle$ between tagged algebraic signatures $\langle \Sigma, V \rangle$ and $\langle \Sigma', V' \rangle$ is a signature morphism from Σ to Σ' such that $\sigma(\!| V |\!) \subseteq V'$.

A signature morphism σ: $\langle \Sigma, V \rangle \to \langle \Sigma', V' \rangle$ is an *inclusion*, if $\sigma = (\Sigma \sqsubseteq \Sigma')$ is an inclusion of algebraic signatures, and $V = V' \cap S$, where S is the sort set of Σ. □

4.2.4 Proposition. *The signature morphisms and the inclusions between tagged signatures form categories when composition and identities are defined as for algebraic signature morphisms.*

Proof. It is trivial to check that the signature morphisms between tagged signatures form a category.

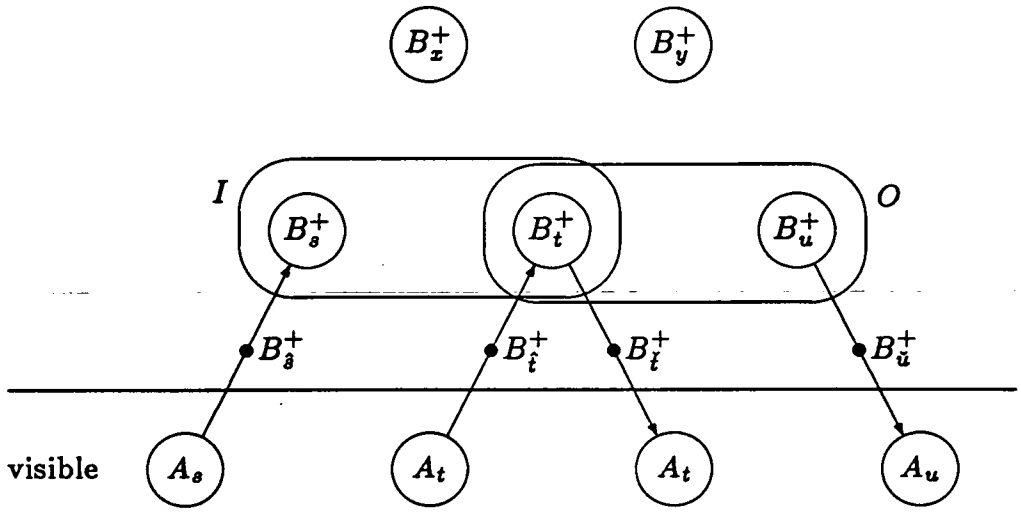


Figure 4-4: A representation B^+ of A^+

Concerning the inclusions, we have to show that the composition of two inclusions

$$\sigma\colon \langle \Sigma, V\rangle \to \langle \Sigma', V'\rangle \quad \text{and} \quad \sigma'\colon \langle \Sigma', V'\rangle \to \langle \Sigma'', V''\rangle$$

is again an inclusion. The composition $\sigma;\sigma'$ is an inclusion of algebraic signatures, and if S is the sort set of Σ and S' the sort set of Σ', we have

$$V = V' \cap S = V'' \cap S' \cap S = V'' \cap S. \qquad \square$$

4.2.5 Definition. Let **TSig** be the category whose arrows are the signature morphisms between small tagged signatures (with composition and identities as for algebraic signature morphisms); let **TIncl** be the subcategory of **TSig** whose arrows are the signature inclusions. $\square$

4.2.6 Theorem. *The pair* $\langle$**TSig, TIncl**$\rangle$ *is an institution syntax. In particular,*

(a) *If* $\langle \Sigma, V\rangle$ *and* $\langle \Sigma', V'\rangle$ *are small tagged signatures, where* $\Sigma = \langle S, \alpha\rangle$, *then*

$$\langle \Sigma, V\rangle \sqsubseteq \langle \Sigma', V'\rangle \qquad \textit{iff} \qquad \Sigma \sqsubseteq \Sigma' \quad \textit{and} \quad V = V' \cap S.$$

(b) *A family* $\langle \Sigma_i, V_i \rangle_{i \in I}$ *of small tagged signatures* ($\Sigma_i = \langle S_i, \alpha_i \rangle$) *is compatible, if and only if*

$\langle \Sigma_i \rangle_{i \in I}$ *is compatible in* $\langle \mathbf{ASig}, \mathbf{AIncl} \rangle$ *and*
for all $i, j \in I$: $V_i \cap S_j \subseteq V_j$
(equivalently, for all $i, j \in I$: $V_i \cap S_j = S_i \cap V_j$*).*

(c) *If* $\langle \Sigma_i, V_i \rangle_{i \in I}$ *is a compatible family of small tagged signatures, then*

$$\bigsqcup_{i \in I} \langle \Sigma_i, V_i \rangle = \Big\langle \bigsqcup_{i \in I} \Sigma_i, \bigcup_{i \in I} V_i \Big\rangle.$$

(d) *If* $\langle \Sigma_i, V_i \rangle_{i \in I}$ *is a nonempty compatible family of small tagged signatures, then*

$$\bigsqcap_{i \in I} \langle \Sigma_i, V_i \rangle = \Big\langle \bigsqcap_{i \in I} \Sigma_i, \bigcap_{i \in I} V_i \Big\rangle.$$

Proof. The category **TIncl** is a partial order category, because there is at most one morphism between two objects, and because the existence of inclusion morphisms from $\langle \Sigma, V \rangle$ to $\langle \Sigma', V' \rangle$ and *vice versa* implies that $\Sigma = \Sigma'$, $V \subseteq V'$, and $V' \subseteq V$, hence that $\langle \Sigma, V \rangle = \langle \Sigma', V' \rangle$. Thus, $\langle$**TSig, TIncl**$\rangle$ is a partially ordered category.

Before proving that $\langle$**TSig, TIncl**$\rangle$ is an institution syntax, we verify the four formulas (a) to (d) of the theorem.

Formula (a) is trivial from the definition of an inclusion morphism.

To verify (b), assume first that the family $\langle \Sigma_i, V_i \rangle_{i \in I}$ of small tagged signatures ($\Sigma_i = \langle S_i, \alpha_i \rangle$) is compatible, so that we can choose an upper bound $\langle \hat{\Sigma}, \hat{V} \rangle$ for the family. Then $\Sigma_i \sqsubseteq \hat{\Sigma}$ for all i, hence $\langle \Sigma_i \rangle_{i \in I}$ is compatible, and for $i, j \in I$, we have

$$V_i \cap S_j = (\hat{V} \cap S_i) \cap S_j = (\hat{V} \cap S_j) \cap S_i = V_j \cap S_i \subseteq V_j.$$

Conversely, suppose that $\langle \Sigma_i \rangle_{i \in I}$ is compatible and that $V_i \cap S_j \subseteq V_j$ for all $i, j \in I$. Let $\bar{\Sigma} := \bigsqcup_{i \in I} \Sigma_i$ and $\bar{V} := \bigcup_{i \in I} V_i$. Obviously, $\bar{V}$ is a subset of the sort set of $\bar{\Sigma}$, hence $\langle \bar{\Sigma}, \bar{V} \rangle$ is a small tagged signature. This signature is an

upper bound of the family $\langle \Sigma_i, V_i \rangle_{i \in I}$, because for all $i \in I$, we have $\Sigma_i \sqsubseteq \bar{\Sigma}$ and

$$V_i \subseteq \bar{V} \cap S_i = (\bigcup_{j \in I} V_j) \cap S_i = \bigcup_{j \in I} (V_j \cap S_i) \subseteq V_i,$$

hence $V_i = \bar{V} \cap S_i$.

The second formula given in (b) is equivalent to the first, because from

$$V_i \cap S_j \subseteq V_j \quad \text{and} \quad V_j \cap S_i \subseteq V_i,$$

it follows that

$$V_i \cap S_j = V_i \cap V_j = V_j \cap S_i.$$

The converse implication is trivial.

To verify (c), assume that $\langle \Sigma_i, V_i \rangle_{i \in I}$ is a compatible family of small tagged signatures, and consider again the small tagged signature $\langle \bar{\Sigma}, \bar{V} \rangle$ defined as just before in the proof of (b). It was shown there that $\langle \bar{\Sigma}, \bar{V} \rangle$ is an upper bound of the family, so it remains to show that $\langle \bar{\Sigma}, \bar{V} \rangle$ is least among the upper bounds of $\langle \Sigma_i, V_i \rangle_{i \in I}$. If $\langle \hat{\Sigma}, \hat{V} \rangle$ is an upper bound of this family, then $\bar{\Sigma} = \bigsqcup_{i \in I} \Sigma_i \sqsubseteq \hat{\Sigma}$, and writing $\bar{\Sigma} = \langle \bar{S}, \bar{\alpha} \rangle$, such that $\bar{S} = \bigcup_{i \in I} S_i$ by Theorem 2.3.6 (b), we have

$$\begin{aligned} \hat{V} \cap \bar{S} &= \hat{V} \cap (\bigcup_{i \in I} S_i) = \bigcup_{i \in I} (\hat{V} \cap S_i) \\ &= \bigcup_{i \in I} V_i = \bar{V}, \end{aligned}$$

so that $\langle \bar{\Sigma}, \bar{V} \rangle \sqsubseteq \langle \hat{\Sigma}, \hat{V} \rangle$.

To verify (d), let $\langle \Sigma_i, V_i \rangle_{i \in I}$ be a nonempty compatible family of small tagged signatures where $\Sigma_i = \langle S_i, \alpha_i \rangle$, and define $\bar{\Sigma} := \prod_{i \in I} \Sigma_i$ and $\bar{V} := \bigcap_{i \in I} V_i$. By Theorem 2.3.6 (c), $\bar{\Sigma} = \langle \bar{S}, \bar{\alpha} \rangle$ where $\bar{S} = \bigcap_{i \in I} S_i$. Now for all $i \in I$, we have $\bar{\Sigma} \sqsubseteq \Sigma_i$, and

$$\begin{aligned} \bar{V} = \bigcap_{j \in I} V_j \subseteq V_i \cap \bigcap_{j \in I} S_j &= \bigcap_{j \in I} (V_i \cap S_j) && (I \neq \emptyset) \\ &\subseteq \bigcap_{j \in I} V_j && \text{(compatibility and (b))} \\ &= \bar{V}, \end{aligned}$$

hence $\bar{V} = V_i \cap \bigcap_{j \in I} S_j = V_i \cap \bar{S}$, and thus $\langle \bar{\Sigma}, \bar{V} \rangle \sqsubseteq \langle \Sigma_i, V_i \rangle$. Hence, $\langle \bar{\Sigma}, \bar{V} \rangle$ is a lower bound of the family $\langle \Sigma_i, V_i \rangle_{i \in I}$.

If $\langle \check{\Sigma}, \check{V} \rangle$ is any lower bound of the family where $\check{\Sigma} = \langle \check{S}, \check{\alpha} \rangle$, then $\check{\Sigma} \sqsubseteq \Sigma_i$ for all $i \in I$, hence $\check{\Sigma} \sqsubseteq \bar{\Sigma}$, and

$$\begin{aligned} \bar{V} \cap \check{S} = (\bigcap_{i \in I} V_i) \cap \check{S} &= \bigcap_{i \in I} (V_i \cap \check{S}) && (I \neq \emptyset) \\ &= \bigcap_{i \in I} \check{V} && (\langle \check{\Sigma}, \check{V} \rangle \sqsubseteq \langle \Sigma_i, V_i \rangle \text{ for all } i \in I) \\ &= \check{V} && (I \neq \emptyset), \end{aligned}$$

hence $\langle \check{\Sigma}, \check{V} \rangle \sqsubseteq \langle \bar{\Sigma}, \bar{V} \rangle$, and thus $\langle \bar{\Sigma}, \bar{V} \rangle$ is the greatest lower bound of the family $\langle \Sigma_i, V_i \rangle_{i \in I}$.

Having proved the formulas (a) to (d) of the theorem, it remains to prove that the partially ordered category $\langle$**TSig**, **TIncl**$\rangle$ is an institution syntax.

From (c), it immediately follows that **TIncl** is compatibly complete. The formulas of (c) and (d), together with distributivity in **AIncl**, immediately yield distributivity in **TIncl**.

It remains to verify the "renaming" property of an institution syntax. Let small tagged signatures $\langle \Sigma_i, V_i \rangle$ with $\Sigma_i = \langle S_i, \alpha_i \rangle$ be given for $i \in \{0, 1, 2\}$ such that $\langle \Sigma_0, V_0 \rangle \sim \langle \Sigma_1, V_1 \rangle$ and $\langle \Sigma_0, V_0 \rangle \sim \langle \Sigma_2, V_2 \rangle$ (they play the rôles of S, T and U in the renaming axiom). Fix $\hat{\Sigma}_2 = \langle \hat{S}_2, \hat{\alpha}_2 \rangle$ together with signature isomorphisms

$$j\colon \hat{\Sigma}_2 \to \Sigma_2, \qquad k\colon \hat{\Sigma}_2 \sqcup \Sigma_0 \to \Sigma_2 \sqcup \Sigma_0$$

according to the renaming property of $\langle$**ASig**, **AIncl**$\rangle$, so that

$$\hat{\Sigma}_2 \sim \Sigma_0 \sqcup \Sigma_1, \tag{1}$$

$$\hat{\Sigma}_2 \sqcap (\Sigma_0 \sqcup \Sigma_1) = \Sigma_2 \sqcap \Sigma_0 \tag{2}$$

and j and k satisfy

$$(\hat{\Sigma}_2 \sqsubseteq \hat{\Sigma}_2 \sqcup \Sigma_0)\,;k = j\,;(\Sigma_2 \sqsubseteq \Sigma_2 \sqcup \Sigma_0) \tag{3}$$

$$(\Sigma_0 \sqsubseteq \hat{\Sigma}_2 \sqcup \Sigma_0)\,;k = (\Sigma_0 \sqsubseteq \Sigma_2 \sqcup \Sigma_0) \tag{4}$$

$$(\Sigma_2 \sqcap \Sigma_0 \sqsubseteq \hat{\Sigma}_2)\,;j = (\Sigma_2 \sqcap \Sigma_0 \sqsubseteq \Sigma_2). \tag{5}$$

Let $\hat{V}_2 := j^{-1}(\!|V_2|\!)$. Then $\langle \hat{\Sigma}_2, \hat{V}_2 \rangle$ is a small tagged signature, and since $j\colon \hat{\Sigma}_2 \to \Sigma_2$ is a signature isomorphism, j and j^{-1} are inverse tagged signature isomorphisms between $\langle \hat{\Sigma}_2, \hat{V}_2 \rangle$ and $\langle \Sigma_2, V_2 \rangle$.

We verify that $\langle \hat{\Sigma}_2, \hat{V}_2 \rangle \sim \langle \Sigma_0, V_0 \rangle \sqcup \langle \Sigma_1, V_1 \rangle$ using formula (b) of the present theorem. Due to (1), it remains to show that $\hat{V}_2 \cap (S_0 \cup S_1) = \hat{S}_2 \cap (V_0 \cup V_1)$. Recall from Theorem 2.3.6 (b) and (c), that (2) implies the analogous equation between the sort sets of the signatures involved. In particular, $\hat{S}_2 \cap S_0 \subseteq S_2 \cap S_0 = (S_2 \cap S_0) \cap S_0 \subseteq \hat{S}_2 \cap S_0$, so that

$$\hat{S}_2 \cap S_0 = S_2 \cap S_0 \tag{6}$$

Now if $x \in j^{-1}(\!|V_2 \setminus S_0|\!) \cap S_0$, then $x \in j^{-1}(\!|V_2|\!) = \hat{V}_2 \subseteq \hat{S}_2$ and $x \in S_0$, hence $x \in S_2 \cap S_0$, and (5) implies that $j(x) = x \in S_0$. This contradicts $x \in j^{-1}(\!|V_2 \setminus S_0|\!)$, and so

$$j^{-1}(\!|V_2 \setminus S_0|\!) \cap S_0 = \emptyset. \tag{7}$$

Thus,

$$\begin{aligned}
\hat{V}_2 \cap S_0 &= j^{-1}(\!|V_2|\!) \cap S_0 \\
&= (j^{-1}(\!|V_2 \cap S_0|\!) + j^{-1}(\!|V_2 \setminus S_0|\!)) \cap S_0 \\
&= j^{-1}(\!|V_2 \cap S_0|\!) \cap S_0 + j^{-1}(\!|V_2 \setminus S_0|\!) \cap S_0 \\
&= j^{-1}(\!|V_2 \cap S_0|\!) \cap S_0 && \text{(by (7))} \\
&= (V_2 \cap S_0) \cap S_0 && (j^{-1} \text{ is the identity on } S_2 \cap S_0) \\
&= V_2 \cap S_0 \\
&= V_0 \cap S_2 && (\langle \Sigma_0, V_0 \rangle \sim \langle \Sigma_2, V_2 \rangle) \\
&= V_0 \cap S_0 \cap S_2 \\
&= V_0 \cap S_0 \cap \hat{S}_2 && \text{(by (6))} \\
&= V_0 \cap \hat{S}_2. && (8)
\end{aligned}$$

Since

$$\hat{S}_2 \cap (S_0 \cup S_1) = S_2 \cap S_0 \subseteq S_0, \tag{9}$$

it follows that

$$\begin{aligned}
\hat{V}_2 \cap (S_0 \cup S_1) &= \hat{V}_2 \cap (S_0 \cup S_1) \cap S_0 && (\hat{V}_2 \subseteq \hat{S}_2 \text{ and (9)}) \\
&= \hat{V}_2 \cap S_0
\end{aligned}$$

$$
\begin{aligned}
&= \hat{S}_2 \cap V_0 && \text{(by (8))} \\
&= \hat{S}_2 \cap (V_0 \cup (V_1 \cap S_0)) && (V_1 \cap S_0 \subseteq V_0, \text{ because} \\
&&& \quad \langle \Sigma_0, V_0 \rangle \sim \langle \Sigma_1, V_1 \rangle) \\
&= \hat{S}_2 \cap (V_0 \cup V_1) \cap S_0 && (V_0 \subseteq S_0) \\
&= \hat{S}_2 \cap (V_0 \cup V_1). && (V_0 \cup V_1 \subseteq S_0 \cup S_1 \text{ and (9)})
\end{aligned}
$$

Hence

$$\langle \hat{\Sigma}_2, V_2 \rangle \sim \langle \Sigma_0, V_0 \rangle \sqcup \langle \Sigma_1, V_1 \rangle. \tag{10}$$

Next, we verify that $\langle \hat{\Sigma}_2, \hat{V}_2 \rangle \sqcap (\langle \Sigma_0, V_0 \rangle \sqcup \langle \Sigma_1, V_1 \rangle) = \langle \Sigma_2, V_2 \rangle \sqcap \langle \Sigma_0, V_0 \rangle$ using formulas (c) and (d). Due to (2), it remains to show that $\hat{V}_2 \cap (V_0 \cup V_1) = V_2 \cap V_0$. From (10), using formula (b), we have

$$
\begin{aligned}
\hat{V}_2 \cap (S_0 \cup S_1) &= \hat{S}_2 \cap (V_0 \cup V_1) \\
&= \hat{V}_2 \cap (S_0 \cup S_1) \cap \hat{S}_2 \cap (V_0 \cup V_1) && \text{(intersection of previous} \\
&&& \quad \text{two expressions)} \\
&= \hat{V}_2 \cap (V_0 \cup V_1),
\end{aligned}
$$

and hence

$$
\begin{aligned}
\hat{V}_2 \cap (V_0 \cup V_1) &= \hat{V}_2 \cap (S_0 \cup S_1) \\
&= \hat{V}_2 \cap (S_0 \cup S_1) \cap S_0 && \text{(by (9))} \\
&= \hat{V}_2 \cap S_0 \\
&= V_2 \cap S_0 && \text{(see (8))} \\
&= V_0 \cap S_2 && \text{(see (8))} \\
&= V_2 \cap S_0 \cap V_0 \cap S_2 && \text{(intersect previous lines)} \\
&= V_2 \cap V_0.
\end{aligned}
$$

We know that k and k^{-1} are inverse isomorphisms between the algebraic signatures $\hat{\Sigma}_2 \sqcup \Sigma_0$ and $\Sigma_2 \sqcup \Sigma_0$. They are also inverse isomorphisms between the tagged signatures $\langle \hat{\Sigma}_2, \hat{V}_2 \rangle \sqcup \langle \Sigma_0, V_0 \rangle = \langle \hat{\Sigma}_2 \sqcup \Sigma_0, \hat{V}_2 \cup V_0 \rangle$ and $\langle \Sigma_2, V_2 \rangle \sqcup \langle \Sigma_0, V_0 \rangle = \langle \Sigma_2 \sqcup \Sigma_0,\ V_2 \cup V_0 \rangle$, because

$$k(\!|\hat{V}_2 \cup V_0|\!) = k(\!|\hat{V}_2|\!) \cup k(\!|V_0|\!)$$

$$= k(|\hat{V}_2|) \cup V_0 \qquad \text{(by (4))}$$
$$= j(|\hat{V}_2|) \cup V_0 \qquad \text{(by (3))}$$
$$= V_2 \cup V_0,$$

hence also $k^{-1}(|V_2 \cup V_0|) = \hat{V}_2 \cup V_0$, and thus k and k^{-1} are tagged signature morphisms.

Finally, since the algebraic signature morphisms j and k satisfy the three equations (3), (4), and (5), so do the tagged signature morphisms j and k.

This concludes the proof of the renaming axiom and hence the proof that $\langle$**TSig, TIncl**$\rangle$ is an institution syntax. □

An important consequence of formula (a) of the theorem just proved is that visibility of sorts is a *global* attribute for every compatible family of tagged signatures. For, each signature of the family is included in the join of the family (call it "$\langle \bar{\Sigma}, \bar{V} \rangle$"), and by formula (a), a sort of a tagged signature in the family is visible in that signature iff it is visible in $\langle \bar{\Sigma}, \bar{V} \rangle$ (*i. e.*, a member of $\bar{V}$). In particular, since all the signatures involved in a structured correctness argument are compatible (see Theorem 3.4.1 and Lemma 3.4.2), each sort is either visible in all signatures or invisible in all signatures involved.

As we shall see in Chapter 5, the situation is simpler still: From a practical point of view, it is unnecessary to distinguish visible sorts at all in most modular program developments.

To obtain an institution, the models of a tagged signature $\langle \Sigma, V \rangle$ are defined to be just the small Σ-algebras.

4.2.7 Definition. Let **TAlg**: $\mathbf{TSig}^{op} \to \mathbf{Cls}$ be the functor whose object function maps $\langle \Sigma, V \rangle \in |\mathbf{TSig}|$ to $\mathbf{Alg}(\Sigma)$, and whose arrow function maps σ: $\langle \Sigma, V \rangle \to \langle \Sigma', V' \rangle$ in **TSig** to $\bar{\sigma} := \mathbf{Alg}(\sigma^{op})$: $\mathbf{TAlg}\langle \Sigma', V' \rangle \to \mathbf{TAlg}\langle \Sigma, V \rangle$ (note that the tagged signature morphism σ also is an algebraic signature morphism σ: $\Sigma \to \Sigma'$). In other words, **TAlg** = U ; **Alg**, where U is the evident forgetful functor from $\mathbf{TSig}^{op}$ to $\mathbf{ASig}^{op}$. □

4.2.8 Theorem. *The triple* $\langle$**TSig, TIncl, TAlg**$\rangle$ *is an institution. In particular,*

(a) *If* $\langle\Sigma,V\rangle \sqsubseteq \langle\Sigma',V'\rangle$ *and* $A \in \mathbf{TAlg}\langle\Sigma',V'\rangle$, *then* $A \in \mathbf{Alg}(\Sigma')$ *and* $A/\langle\Sigma,V\rangle = A/\Sigma$.

(b) *If* $\langle\Sigma_i,V_i\rangle_{i\in I}$ *is a nonempty compatible family of small tagged signatures, and* $A_i \in \mathbf{TAlg}\langle\Sigma_i,V_i\rangle$ *such that for all* $i,j \in I$: $A_i/(\langle\Sigma_i,V_i\rangle \sqcap \langle\Sigma_j,V_j\rangle) = A_j/(\langle\Sigma_i,V_i\rangle \sqcap \langle\Sigma_j,V_j\rangle)$, *then there exists a unique* $\bar{A} \in \mathbf{TAlg}(\bigsqcup_{i\in I}\langle\Sigma_i,V_i\rangle)$ *that satisfies* $\bar{A}/\langle\Sigma_i,V_i\rangle = A_i$ *for all* $i \in I$. *The algebra* $\bar{A}$ *equals the join of the* A_i *in the institution* $\langle$**ASig, AIncl, Alg**$\rangle$.

Proof. By Theorem 4.2.6, $\langle$**TSig, TIncl**$\rangle$ is an institution syntax, and by definition, **TAlg** is a functor from $\mathbf{TSig}^{\mathrm{op}}$ to **Cls**, which is a category of sets. Hence $\langle$**TSig, TIncl, TAlg**$\rangle$ is a preinstitution.

We shall now verify the clauses (a) and (b) of the theorem. Clause (b) implies the completeness property, and hence that $\langle$**TSig, TIncl, TAlg**$\rangle$ is an institution.

Clause (a): If $\langle\Sigma,V\rangle \sqsubseteq \langle\Sigma',V'\rangle$ and $A \in \mathbf{TAlg}\langle\Sigma',V'\rangle$, then $A \in \mathbf{Alg}(\Sigma')$, and because the inclusion morphism $(\langle\Sigma,V\rangle \sqsubseteq \langle\Sigma',V'\rangle)$ in **TIncl** equals the inclusion morphism $(\Sigma \sqsubseteq \Sigma')$ in **AIncl**, we have

$$A/\langle\Sigma,V\rangle = \mathbf{TAlg}((\langle\Sigma,V\rangle \sqsubseteq \langle\Sigma',V'\rangle)^{\mathrm{op}})A = \mathbf{Alg}((\Sigma \sqsubseteq \Sigma')^{\mathrm{op}})A = A/\Sigma.$$

Clause (b): If $\langle\Sigma_i,V_i\rangle_{i\in I}$ and $\langle A_i\rangle_{i\in I}$ are as described in clause (b), then $\langle\Sigma_i\rangle_{i\in I}$ and $\langle A_i\rangle_{i\in I}$ satisfy the assumptions of the completeness axiom for $\langle$**ASig, AIncl, Alg**$\rangle$. Let $\bar{A} \in \mathbf{Alg}(\bigsqcup_{i\in I}\Sigma_i)$ be the join of the A_i in $\langle$**ASig, AIncl, Alg**$\rangle$. Now

$$\begin{aligned}\mathbf{TAlg}\Big(\bigsqcup_{i\in I}\langle\Sigma_i,V_i\rangle\Big) &= \mathbf{TAlg}\Big(\Big\langle\bigsqcup_{i\in I}\Sigma_i, \bigcup_{i\in I}V_i\Big\rangle\Big)\\ &= \mathbf{Alg}\Big(\bigsqcup_{i\in I}\Sigma_i\Big) \ni \bar{A},\end{aligned}$$

and, due to (a), we have $\bar{A}/\langle\Sigma_i,V_i\rangle = \bar{A}/\Sigma_i = A_i$ for all $i \in I$.

The algebra $\bar{A}$ is unique with this property, because whenever an algebra $B \in \mathbf{TAlg}(\bigsqcup_{i\in I}\langle\Sigma_i,V_i\rangle)$ satisfies $B/\langle\Sigma_i,V_i\rangle = A_i$ for all $i \in I$, then $B/\Sigma_i = A_i$

for all $i \in I$, and the uniqueness of joins in $\langle$**ASig, AIncl, Alg**$\rangle$ implies that $B = \bar{A}$. □

The institution $\langle$**TSig, TIncl, TAlg**$\rangle$ is not very different from $\langle$**ASig, AIncl, Alg**$\rangle$: The only difference is that the sorts of signatures are now "tagged" as being either "visible" (the sorts in V) or "hidden" (the sorts not in V), and this may affect the compatibility of signatures (see Theorem 4.2.6 (b)). The operations on the models are the same (Theorem 4.2.8).

In practical programming, the visible sorts of a signature would generally be those that can be accessed in ways not covered by the algebraic model. Hence, data types whose values can be input or output of a program would be modelled as visible sorts; the type of truth values ("*bool*") would have to be modelled as a visible sort also, because the **if** construct is not part of the algebraic model (unless higher order types are considered, cf. page 116 f.).

Note also that normally modules cannot define new visible sorts, because it is characteristic for a module that its new sorts can only be accessed by means of the operations defined in it, which do appear in the algebraic model.

Since the example institution we shall deal with in the remainder of the thesis is $\langle$**TSig, TIncl, TAlg**$\rangle$ rather than $\langle$**ASig, AIncl, Alg**$\rangle$, it is necessary to explain how the interfaces and modules of the *dictionary* program development are rendered in $\langle$**TSig, TIncl, TAlg**$\rangle$. Strictly speaking, a set of visible sorts would have to be distinguished in each of the signatures occurring there. As remarked earlier (after the proof of Theorem 4.2.6), each of the sorts *bool*, *listitem* and *store* is either visible in all signatures in which it occurs or invisible in all of them. Since *store* is defined by a module in the development, it cannot be visible, and since the program code that was designed uses the **if** construct, the sort *bool* must be visible. This leaves only the status of *listitem* to be decided. As we shall see in Chapter 5, the correctness proofs of the development are unaffected by whether *listitem* is visible or not (the same holds true for *bool*, incidentally), and so we shall leave this decision open.

4.3 Behavioural Inclusion

The present and the next two sections discuss three representation relations in the institution ⟨**TSig**, **TIncl**, **TAlg**⟩: "behavioural inclusion", "behavioural equivalence", and "standard representation". It would be ideal to present these concepts starting with abstraction function representation and ending with behavioural inclusion: the abstraction function representation concept, based on [Hoare 72], has become very well known in Computer Science and in programming practice; behavioural representation concepts, although implied by the proof method of [Milner 71], have only more recently received attention in abstract data type theory, beginning with [GGM 76]; behavioural inclusion, related to the "covering" relation between automata [Ginzburg 68, p. 97] is new—it combines the behaviour idea with the "partial implementation" idea of Kamin and Archer [KA 84].

For technical reasons, however, it is better to begin with behavioural inclusion, then treat behavioural equivalence, and finally abstraction function representations. In this order, each concept is more restrictive than its predecessors, and it is possible to apply previous theorems instead of having to prove successive generalizations.

Hence, the present section treats the most general of the three representation relations: behavioural inclusion.

To speak about the "behaviour" of a program, it must be clarified how observations about a program can be made. In this thesis, programs are modelled by partial algebras, and the most general form of observation here is to apply an operation to some data values and to observe whether a result value is produced, and if so, which one.

The idea underlying behavioural representation concepts is that not all of the observations just described are practically meaningful: in concrete programs, not all data can be generated and observed directly, rather we have a distinction between "hidden" and "visible" sorts, as explained in the previous section. The

characteristic property of hidden sorts is that values in these sorts cannot be directly generated or inspected; rather, the only way to access these values is by means of the operations of the algebra. The visible data types, on the other hand, are accessible in other ways than those described in the algebra, so that these values must preserved in a representation in order to preserve program behaviour.

In the algebraic model, this suggests the following concept of "observation": Rather than considering single operations with arbitrary input values, one considers terms constructed from function symbols and values of the visible sorts. An algebra defines an "evaluation" of such terms, and it can be observed whether the evaluation succeeds or fails, and, if the result value is of a visible sort, this value can be observed as well.

This concept of "observation" will now be formalised. The set V in the definitions below is to be understood as the set of visible sorts. Recall that if S is a set, an S-sorted set is an S-indexed family of sets; analogously, an S-sorted relation (S-sorted partial function, S-sorted mapping) between S-sorted sets X and Y is a family $\langle R_s \rangle_{s \in S}$ such that each R_s is a relation (partial function, mapping) from X_s to Y_s. The notations "$f\colon X \to Y$" and "$f\colon X \nrightarrow Y$" are used to express that f is an S-sorted mapping or partial function from X to Y.

4.3.1 Definition (Terms).
Let $\Sigma = \langle S, \alpha\colon F \to S^+ \rangle$ be an algebraic signature and let X be a V-sorted set such that $V \subseteq S$. The S-sorted set $\mathrm{T}_\Sigma(X)$ is the componentwise smallest family of subsets of

$$((S + F) \cup \bigcup_{v \in V} X_v)^*$$

that satisfies

(a) $v \in V,\ x \in X_v \Longrightarrow \langle v, x \rangle \in \mathrm{T}_\Sigma(X)_v$,

(b) $f\colon s_1 \ldots s_n \to r$ in Σ, $t_i \in \mathrm{T}_\Sigma(X)_{s_i}$ for $i \in \{1, \ldots, n\}$
$\Longrightarrow \langle f \rangle \circ t_1 \circ \cdots \circ t_n \in \mathrm{T}_\Sigma(X)_r$.

For $s \in S$, the elements of $\mathrm{T}_\Sigma(X)_s$ are called the *Σ-terms* over X of sort s. □

The essential property of this definition is that every term is obtained in a unique way by a finite number of the construction steps indicated under (a) and (b). This allows to apply the familiar techniques of "structural induction" to prove properties of terms and of "structural recursion" to define functions on terms.

4.3.2 Definition (Evaluation of Terms).
Let $\Sigma = \langle S, \alpha\colon F \to S^+\rangle$ be an algebraic signature, let A be a Σ-algebra, and let $V \subseteq S$. Then "A/V" denotes the V-sorted set $\langle A_v\rangle_{v\in V}$. The *evaluation function* from $T_\Sigma(A/V)$ to A/S is the minimal (w.r.t. componentwise graph inclusion) S-sorted partial function ϕ that satisfies

(a) $t = \langle v, x\rangle$ with $v \in V$ and $x \in A_v \Longrightarrow \phi_v(t) = x$,

(b) $t = \langle f\rangle \circ u_1 \circ \cdots \circ u_n$ with $f\colon s_1 \ldots s_n \to r$ in Σ, $u_i \in \operatorname{dom}\phi_{s_i}$ for $i \in \{1,\ldots,n\}$ and $\langle \phi_{s_1}u_1, \ldots, \phi_{s_n}u_n\rangle \in \operatorname{dom} A_f$
$\Longrightarrow \phi_r(t) = A_f(\phi_{s_1}u_1, \ldots, \phi_{s_n}u_n)$. □

In this definition of "evaluation", a term can only be evaluated (*i. e.*, is in the domain of the evaluation function), if all its subterms can. This is the standard mathematical definition of evaluation of terms over partial algebras (cf. [Grätzer 79, p. 84], [Burmeister 82, p. 313]).

The interpretation of a partial Σ-algebra A as a model of data types and operations in a functional program was explained in Chapter 2. Terms in $T_\Sigma(A/V)$ may be regarded as "symbolic computations" with input data from A/V, that is, as descriptions of computations to be performed using the functions listed in Σ, beginning with data values from A/V. This corresponds to the view that the "visible sorts" in V play the rôle of external data types whose elements may be used freely as input to computations. The evaluation of terms corresponds to the evaluation of expressions in a programming notation with call-by-value semantics.

A second aspect of the "observation" idea still has to be considered, namely that only values of visible sort can be directly inspected. This is done in the following definition of the "behavioural inclusion" relation between partial algebras.

4.3.3 Definition (Behavioural Inclusion).
Let $\Sigma = \langle S, \alpha\colon F \to S^+\rangle$ be an algebraic signature, and let $V \subseteq S$. Let A and B be Σ-algebras with evaluation functions

$$\phi\colon \mathrm{T}_\Sigma(A/V) \nrightarrow A/S, \qquad \psi\colon \mathrm{T}_\Sigma(B/V) \nrightarrow B/S.$$

The algebra A is *V-behaviourally included* in B (written "$A \lesssim_V B$"), iff

(a) $\forall v \in V\colon A_v \subseteq B_v$,

(b) $\forall s \in S\colon \mathrm{dom}\,\phi_s \subseteq \mathrm{dom}\,\psi_s$, and

(c) $\forall v \in V\colon \phi_v \subseteq \psi_v$. □

Clause (a) of this definition requires that the visible values of A are also visible values of B. This implies that every Σ-term over A/V also is a Σ-term over B/V, *i. e.*, that $\mathrm{T}_\Sigma(A/V)$ is componentwise included in $\mathrm{T}_\Sigma(B/V)$. Clause (b) says that every computation that "succeeds" in A (*i. e.*, has a value under evaluation), succeeds in B also. Clause (c), which in the presence of (b) may also be written

$$\forall v \in V,\ t \in \mathrm{dom}\,\phi_v\colon\ \phi_v(t) = \psi_v(t),$$

says that succeeding computations of visible sort yield the same result in B that they yield in A.

It follows trivially from the definition that for fixed Σ and V, V-behavioural inclusion is a preordering between Σ-algebras.

The behavioural inclusion relation formalizes the idea of a "partial" or "restricted" representation of a data type, where computations may fail more often in the representation than in the algebra represented, yet computations that succeed in the representation yield the same observable results as in the algebra represented. Such representations occur frequently in practice, when size constraints are imposed on the representation of an abstract type [KA 84]. For example, integers are often implemented as words of fixed size; representations of lists and sets often impose bounds on their length or cardinality. One might even argue that, because all real machines and storage units are finite, every concrete representation of a type with infinitely many values must be partial.

The partial representation idea is similar to the "partial correctness" concept for programs (cf. [Hoare 69]) and to the "covering" relation between automata [Ginzburg 68, p. 97]. It has been introduced to data type theory by Kamin and Archer [KA 84].

However, the concept of Kamin and Archer is based on "abstraction functions", and the concept presented here is strictly more general, as will be shown in Theorem 4.3.11 below.

Before presenting examples of behavioural inclusion, we shall develop a characterization that provides a proof method for behavioural inclusion. This characterization and proof method are based on the concept of a "correspondence".

4.3.4 Definition (Correspondence).
Let $\Sigma = \langle S, \alpha\colon F \to S^+\rangle$ be an algebraic signature, and let A and B be Σ-algebras. A *correspondence* from A to B is an S-sorted relation $G = \langle G_s\rangle_{s\in S}$, where $G_s \subseteq A_s \times B_s$ for all $s \in S$, such that all $f \in F$ are *compatible* with G, *i.e.*, if $f\colon s_1 \dots s_n \to r$ in Σ, then

$$\begin{array}{ll} \text{whenever } (x_i, y_i) \in G_{s_i} \text{ for } i \in \{1,\dots,n\} & \\ \qquad \text{and } \langle x_1,\dots,x_n\rangle \in \operatorname{dom} A_f & \\ \text{then} \quad \langle y_1,\dots,y_n\rangle \in \operatorname{dom} B_f & \\ \qquad \text{and } \big(A_f(x_1,\dots,x_n),\ B_f(y_1,\dots,y_n)\big) \in G_r. & \end{array}$$

The fact that G is a correspondence from A to B is written "$G\colon A \mathrel{-\!\!\times} B$". If all components of G are partial functions, then G is a *partial homomorphism* ("$G\colon A \mathrel{+\!\!\to} B$"); if they are total functions, then G is a *homomorphism* from A to B ("$G\colon A \longrightarrow B$").

For $V \subseteq S$, a correspondence G (partial homomorphism, homomorphism) from A to B is a *V-correspondence* (*partial V-homomorphism*, *V-homomorphism*), if for all $v \in V$

$$A_v \subseteq B_v \quad \text{and} \quad G_v \text{ is the inclusion function;}$$

this fact is denoted by writing $G\colon A \mathrel{\underset{V}{-\!\!\times}} B$ ($G\colon A \mathrel{\underset{V}{+\!\!\to}} B$, $G\colon A \underset{V}{\longrightarrow} B$). □

In other words, a relation is a correspondence from an algebra A to an algebra B, if every function f that yields a result for some arguments in A yields a related result when applied to related arguments in B.

The correspondence concept has been developed independently at about the same time by Nipkow [personal communication, April 1985] and by myself [Schoett 85], in both cases as a generalization of the "correspondences" of [Schoett 83] (which are strong correspondences in the sense of this thesis). The concept is similar in spirit to the "weak homomorphisms" used by Ginzburg [Ginzburg 68] to characterize coverings of automata. The homomorphism concept agrees with the "homomorphisms" of [Grätzer 79, p. 81] and [Burmeister 82, p. 310], and with the "partial homomorphisms" of [KA 84, p. 321 f.]. A family of partial functions whose converse is a correspondence is called a "conformism" in [Burmeister 82, p. 347]. Using notational ideas from that paper, the requirement for a relation G between the carriers of the Σ-algebras A and B to be a correspondence can be written as follows:

$$\text{for every } f\colon s \to r \text{ in } \Sigma\colon \quad (G^{\cup})_{(s)} \,;\, A_f \subseteq B_f \,;\, G^{\cup}_r \,.$$

Here $G^{\cup}$ is the componentwise converse of G, and $G_{(s)}$ for $s = s_1 \ldots s_n$ is the relation between $\prod\langle A_{s_1}, \ldots, A_{s_n}\rangle$ and $\prod\langle B_{s_1}, \ldots, B_{s_n}\rangle$ defined by

$$\big(\langle x_1, \ldots, x_n\rangle, \langle y_1, \ldots, y_n\rangle\big) \in G_{(s)} \;:\Longleftrightarrow\; (x_i, y_i) \in G_{s_i} \text{ for } i \in \{1, \ldots, n\}.$$

The "weak subalgebra" relation of [Grätzer 79, p. 81] is also characterized by correspondences, as follows.

4.3.5 Definition. Let $\Sigma = \langle S,\ \alpha\colon F \to S^+\rangle$ be an algebraic signature, and let A and B be Σ-algebras. The algebra A is a *weak subalgebra* of B (notation: "$A \hookrightarrow B$"), if there exists a homomorphism from A to B consisting entirely of inclusions (this homomorphism is unique, and it is called the *inclusion homomorphism* from A to B). □

It is easily checked that the correspondences between Σ-algebras form a category:

4.3.6 Proposition. *Let $\Sigma = \langle S,\ \alpha\colon F \to S^+\rangle$ be an algebraic signature, and let $V \subseteq S$. The following components form a category:*

- *objects: a set of Σ-algebras,*
- *arrows: the V-correspondences (partial V-homomorphisms, V-homomorphisms) between the algebras,*
- *identity arrow for an object A: the S-sorted identity map from A/S to itself,*
- *composition of arrows: componentwise relational composition.*

Note that since every correspondence is a $\emptyset$-correspondence, the proposition remains true when the prefix "V-" is deleted.

Proof. It is easy to see that for every Σ-algebra A, the S-sorted identity map from A/S to A/S is a V-homomorphism from A to itself. Clearly, it is the identity under componentwise relational composition.

Next, we prove that the componentwise relational composition of V-correspondences is again a V-correspondence. Let $G\colon A \underset{V}{\rightarrowtail} B$ and $H\colon B \underset{V}{\rightarrowtail} C$ be V-correspondences. Let $f\colon s_1 \ldots s_n \to r$ in Σ, and let $(x_i, z_i) \in (G\,;H)_{s_i}$ for $i \in \{1,\ldots,n\}$ be such that $\langle x_1,\ldots,x_n\rangle \in \operatorname{dom} A_f$. By the definition of relational composition, we can pick $y_1,\ldots,y_n$ such that $(x_i,y_i) \in G_{s_i}$ and $(y_i,z_i) \in H_{s_i}$ for $i \in \{1,\ldots,n\}$. Since G is a correspondence, we have $\langle y_1,\ldots,y_n\rangle \in \operatorname{dom} B_f$, and $(A_f(x_1,\ldots,x_n),\ B_f(y_1,\ldots,y_n)) \in G_r$. Since H is a correspondence, we have $\langle z_1,\ldots,z_n\rangle \in \operatorname{dom} C_f$ and $(B_f(y_1,\ldots,y_n),\ C_f(z_1,\ldots,z_n)) \in H_r$. But this means that $(A_f(x_1,\ldots,x_n),\ C_f(z_1,\ldots,z_n)) \in (G\,;H)_r$, and hence f is compatible with $G\,;H$. It follows that $G\,;H$ is a correspondence from A to C, and it is trivial that $G\,;H$ is again a V-correspondence. Thus, the V-correspondences form a category.

That the partial V-homomorphisms and the V-homomorphisms also form categories now follows trivially from the fact that the relational composition of partial or total functions is again a partial or total function. □

The following theorem characterizes behavioural inclusion by means of correspondences.

4.3.7 Theorem. *Let $\Sigma = \langle S, \alpha\colon F \to S^+\rangle$ be an algebraic signature, let $V \subseteq S$, and let A and B be Σ-algebras. Then A is V-behaviourally included in B, if and only if there exists a V-correspondence from A to B.*

The proof requires two lemmas.

4.3.8 Lemma. *Let $\Sigma = \langle S, \alpha\colon F \to S^+\rangle$ be an algebraic signature, let $V \subseteq S$, let A and B be Σ-algebras, let $G\colon A \xrightarrow[V]{} B$, and let*

$$\phi\colon \mathrm{T}_\Sigma(A/V) \twoheadrightarrow A/S \quad \text{and} \quad \psi\colon \mathrm{T}_\Sigma(B/V) \twoheadrightarrow B/S$$

be the evaluation functions. If $s \in S$ and $t \in \operatorname{dom}\phi_s$, then $t \in \operatorname{dom}\psi_s$ and $(\phi_s(t), \psi_s(t)) \in G_s$.

Proof. We prove that $t \in \operatorname{dom}\phi_s$ implies $t \in \operatorname{dom}\psi_s$ and $(\phi_s(t), \psi_s(t)) \in G_s$ by structural induction on t.

(a) $t = \langle s, x\rangle$ with $s \in V$ and $x \in A_s$:
As G is a V-correspondence, we have $x \in B_s$ and $(\phi_s(t), \psi_s(t)) = (x, x) \in G_s$.

(b) $t = \langle f\rangle \circ u_1 \circ \cdots \circ u_n$ with $f\colon r_1 \ldots r_n \to s$ in Σ, $u_i \in \mathrm{T}_\Sigma(A/V)_{r_i}$ for $i \in \{1, \ldots, n\}$:
If $t \in \operatorname{dom}\phi_s$, then by definition of ϕ:

$$\begin{aligned} &u_i \in \operatorname{dom}\phi_{r_i} \text{ for } i \in \{1, \ldots, n\},\\ &\langle \phi_{r_1}(u_1), \ldots, \phi_{r_n}(u_n)\rangle \in \operatorname{dom} A_f, \text{ and}\\ &\phi_s(t) = A_f(\phi_{r_1}(u_1), \ldots, \phi_{r_n}(u_n)). \end{aligned}$$

By the inductive hypothesis, $u_i \in \operatorname{dom}\psi_{r_i}$ and $(\phi_{r_i}(u_i), \psi_{r_i}(u_i)) \in G_{r_i}$ for $i \in \{1, \ldots, n\}$. As G is a correspondence, it follows that $\langle \psi_{r_1}(u_1), \ldots, \psi_{r_n}(u_n)\rangle \in \operatorname{dom} B_f$, and

$$\begin{aligned} (\phi_s(t), \psi_s(t)) &= \big(A_f(\phi_{r_1}(u_1), \ldots, \phi_{r_n}(u_n)),\ B_f(\psi_{r_1}(u_1), \ldots, \psi_{r_n}(u_n))\big)\\ &\in G_s. \end{aligned}$$

□

4.3.9 Lemma. *Let A and B be algebras of signature $\Sigma = \langle S, \alpha\colon F \to S^+\rangle$, let $V \subseteq S$, and let $\phi\colon \mathrm{T}_\Sigma(A/V) \twoheadrightarrow A/S$ and $\psi\colon \mathrm{T}_\Sigma(B/V) \twoheadrightarrow B/S$ be the evaluation functions. If $A \lesssim_V B$, then the S-sorted relation G defined by $G_s = \phi_s^\cup ; \psi_s$ is the least V-correspondence from A to B with respect to the componentwise subset ordering.*

Proof. First, we show that G is a correspondence from A to B. For $s \in S$, we have $G_s \subseteq A_s \times B_s$, because $\operatorname{dom}\phi_s^\cup = \operatorname{ran}\phi_s \subseteq A_s$ and $\operatorname{ran}\psi_s \subseteq B_s$. Now let $f\colon s_1 \dots s_n \to r$ in Σ, and let $(x_i, y_i) \in G_{s_i}$ for $i \in \{1, \dots, n\}$ be such that $\langle x_1, \dots, x_n\rangle \in \operatorname{dom} A_f$. Then for $i \in \{1, \dots, n\}$ we can choose $u_i \in \operatorname{dom}\phi_{s_i} \cap \operatorname{dom}\psi_{s_i} \subseteq \mathrm{T}_\Sigma(A/V)_{s_i}$ such that $\phi_{s_i}(u_i) = x_i$ and $\psi_{s_i}(u_i) = y_i$. Let $t := \langle f\rangle \circ u_1 \circ \cdots \circ u_n$. This is a term in $\mathrm{T}_\Sigma(A/V)_r$, and because $\langle x_1, \dots, x_n\rangle \in \operatorname{dom} A_f$, it follows that $t \in \operatorname{dom}\phi_r$. Since $A \lesssim_V B$, we have $t \in \operatorname{dom}\psi_r$; in particular, $\langle y_1, \dots, y_n\rangle \in \operatorname{dom} B_f$. It follows that

$$\begin{aligned}\big(A_f(\phi_{s_1}(u_1), \dots, \phi_{s_n}(u_n)),\, B_f(\psi_{s_1}(u_1), \dots, \psi_{s_n}(u_n))\big) &= \big(\phi_r(t), \psi_r(t)\big)\\ &\in \phi_r^\cup ; \psi_r = G_r.\end{aligned}$$

This shows that G is a correspondence from A to B.

To see that G is a V-correspondence, note first that for $v \in V$, we have $A_v \subseteq B_v$, because $A \lesssim_V B$. Furthermore, for $x \in A_v$, we have $x = \phi_v(\langle v, x\rangle) = \psi_v(\langle v, x\rangle)$, hence $(x, x) \in G_v$. Hence $\{\,(x, x) \mid x \in A_v\,\} \subseteq G_v$.

Conversely, if $v \in V$ and $(x, y) \in G_v = \phi_v^\cup ; \psi_v$, there exists $t \in \operatorname{dom}\phi_v \cap \operatorname{dom}\psi_v$ such that $\phi_v(t) = x$ and $\psi_v(t) = y$. By clause (c) of Definition 4.3.3, this implies that $x = y$. Hence $G_v \subseteq \{\,(x, x) \mid x \in A_v\,\}$. Together with the converse inclusion just proved, this shows that G_v is the inclusion map from A_v to B_v, and hence that G is a V-correspondence.

Finally, to see that G is the least V-correspondence from A to B, suppose that $H\colon A \xrightarrow[V]{} B$, and let $s \in S$ and $(x, y) \in G_s$. Then we can choose $t \in \operatorname{dom}\phi_s \cap \operatorname{dom}\psi_s$ such that $\phi_s(t) = x$ and $\psi_s(t) = y$. By the previous lemma, applied to the correspondence H, we have $(x, y) = (\phi_s(t), \psi_s(t)) \in H_s$. Thus, $G_s \subseteq H_s$ for all $s \in S$. □

Proof of Theorem 4.3.7.

Let $\Sigma = \langle S, \alpha: F \to S^+ \rangle$ be an algebraic signature, let $V \subseteq S$, and let A and B be Σ-algebras.

If $A \lesssim_V B$, there exists a V-correspondence from A to B by Lemma 4.3.9.

Conversely, suppose that there exists a V-correspondence from A to B, *i.e.*, that we can choose $G: A \xrightarrow[V]{\times} B$. By definition, $A_v \subseteq B_v$ for $v \in V$. Let $\phi: T_\Sigma(A/V) \twoheadrightarrow A/S$ and $\psi: T_\Sigma(B/V) \twoheadrightarrow B/S$ be the evaluation functions. From Lemma 4.3.8 it follows that $\mathrm{dom}\,\phi_s \subseteq \mathrm{dom}\,\psi_s$ for all $s \in S$, and that if $v \in V$ and $t \in \mathrm{dom}\,\phi_v$, then $(\phi_v(t), \psi_v(t)) \in G_v = \{(x,x) \mid x \in A_v\}$, and hence $\phi_v(t) = \psi_v(t)$. This means that $\phi_v \subseteq \psi_v$ for $v \in V$, and hence that A is V-behaviourally included in B. □

In the following example, the theorem just proved is used to show that an algebra is behaviourally included in another one. As we shall see later (in Example 4.3.12), this example can not be handled using the "partial implementation" notion of Kamin and Archer [KA 84].

4.3.10 Example. Recall that if $s = \langle s_1, \ldots, s_n \rangle$ is a sequence, then $\mathrm{ran}\, s = \{s_1, \ldots, s_n\}$ is the set of elements occurring in it.

Consider the following signature Σ.

signature

bool, *char*, *string*: **sort**

single: *char* → *string*

occurs: *char string* → *bool*

join: *string string* → *string*

Let C be a set, which will serve as the set of values of type *char*, and let $K \geq 1$ be a natural number, which will be a size constraint on the *string* values handled by the algebra B. Consider the Σ-algebras A and B defined by

$$A_{bool} = B_{bool} = \{\mathbf{T}, \mathbf{F}\}$$

$$A_{char} = B_{char} = C$$

$$A_{string} = B_{string} = C^*$$

$$A_{single}(x) \quad = B_{single}(x) \quad = \langle x \rangle$$

$$A_{occurs}(x,s) = B_{occurs}(x,s) = \begin{cases} \mathbf{T}, & \text{if } x \in \operatorname{ran} s \\ \mathbf{F}, & \text{if } x \notin \operatorname{ran} s \end{cases}$$

$$A_{join}(s,t) = s \circ t$$

$B_{join}(s,t)$ is defined by the recursive code

$$\begin{aligned} join(s,t) = \textbf{if} \quad & length(s) = 0 \ \textbf{then} \ t \\ \textbf{else if} \ & occurs(hd\, s, t) \ \textbf{then} \ join(tl\, s, t) \\ \textbf{else if} \ & length(t) \geq K \ \textbf{then} \ Error() \\ \textbf{else} \quad & join(tl\, s, cons(hd\, s, t)). \end{aligned}$$

In the code for B_{join}, the following familiar operations on sequences are used:

$$nil() = \langle\rangle$$

$$cons(x, \langle y_1, \ldots, y_n \rangle) = \langle x, y_1, \ldots, y_n \rangle$$

$$length(\langle x_1, \ldots, x_n \rangle) = n$$

$$\text{for } n \geq 1: \begin{cases} hd(\langle x_1, \ldots, x_n \rangle) = x_1 \\ tl(\langle x_1, \ldots, x_n \rangle) = \langle x_2, \ldots, x_n \rangle. \end{cases}$$

The algebra A can be seen as a specification of a "string" data type. The algebra B may be regarded as a partial representation of the type, written in a functional programming notation. The functions B_{single} and B_{occurs} could be coded as follows:

$$\begin{aligned} single(x) \quad &= cons(x, nil()) \\ occurs(x,s) &= \textbf{if} \quad length(s) = 0 \ \textbf{then} \ \mathbf{F} \\ &\quad \textbf{else if} \ x = hd\, s \quad \textbf{then} \ \mathbf{T} \\ &\quad \textbf{else} \quad occurs(x, tl\, s). \end{aligned}$$

The code for B_{join}, which was given above, is based on the assumption that the three functions *single*, *join* and *occurs* are the only means of manipulating values of type *string*; in other words, that *string* is encapsulated by these three access functions. Under this assumption, information about a value of type *string* can only be gained by means of the *occurs* operation. Since the output of this

operation is completely determined by the *range* of its second argument, that is, the *set* of *char* values occurring in the *string*, it is unnecessary to preserve sequence information or store elements repeatedly in a *string*. The code for B_{join} exploits this observation by ignoring elements of the first argument s that already occur in the second argument t, so that memory is saved because the result sequence is shorter. Also, the sequence information is not preserved (if x, y and z are different, then $join(\langle x, y\rangle, \langle z\rangle) = \langle y, x, z\rangle$). Finally, the operators of B do not allow to generate a sequence of length more than K, due to the fact that B_{join} aborts (by calling the nullary function *Error* that does not yield a result value) if necessary. This means that computations may fail in B, although no computation can fail in A.

We now show that B is V-behaviourally included in A, where $V := \{bool, char\}$. For this purpose, we construct a V-correspondence G: $B \xrightarrow[V]{\times} A$. Let

$$\begin{aligned}
&G_{bool} \text{ be the identity relation on } \{\mathbf{T}, \mathbf{F}\},\\
&G_{char} \text{ be the identity relation on } C,\\
&G_{string} := \{\, (s', s) \mid \operatorname{ran} s' = \operatorname{ran} s \,\} \subseteq C^* \times C^*.
\end{aligned}$$

To show that G is a correspondence, we have to show that each of the operations is compatible with G. It is then clear that G is a V-correspondence also.

single: Let $(x', x) \in G_{char}$ such that $x' \in \operatorname{dom} B_{single}$ (which is always true). It follows that $x' = x$, and hence

$$\begin{aligned}
(B_{single}(x'),\ A_{single}(x)) &= (\langle x'\rangle, \langle x\rangle)\\
&= (\langle x\rangle, \langle x\rangle)\\
&\in G_{string}.
\end{aligned}$$

occurs: Let $(x', x) \in G_{char}$ and $(s', s) \in G_{string}$ such that $\langle x', s'\rangle \in \operatorname{dom} B_{occurs}$ (which is always true). Then $x = x'$ and $\operatorname{ran} s = \operatorname{ran} s'$, and hence

$$B_{occurs}(x', s') = \begin{cases} \mathbf{T}, & \text{if } x' \in \operatorname{ran} s'\\ \mathbf{F}, & \text{if } x' \notin \operatorname{ran} s' \end{cases}$$

$$= \begin{cases} \mathbf{T}, & \text{if } x \in \operatorname{ran} s \\ \mathbf{F}, & \text{if } x \notin \operatorname{ran} s \end{cases}$$

$$= A_{occurs}(x, s).$$

join: We shall prove below that

$$\text{for all } \langle s,t\rangle \in \operatorname{dom} B_{join}: \quad \operatorname{ran} B_{join}(s,t) = \operatorname{ran} s \cup \operatorname{ran} t \qquad (*)$$

From this it easily follows that *join* is compatible with G: if (s',s) and (t',t) are elements of G_{string} (*i.* e., $\operatorname{ran} s' = \operatorname{ran} s$ and $\operatorname{ran} t' = \operatorname{ran} t$) such that $\langle s',t'\rangle \in \operatorname{dom} B_{join}$, then $\langle s,t\rangle \in \operatorname{dom} A_{join}$ (since this is always true), and

$$\begin{aligned} \operatorname{ran} B_{join}(s',t') &= \operatorname{ran} s' \cup \operatorname{ran} t' \qquad \text{(by } (*)\text{)} \\ &= \operatorname{ran} s \cup \operatorname{ran} t \\ &= \operatorname{ran} A_{join}(s,t), \end{aligned}$$

and hence $\big(B_{join}(s',t'),\ A_{join}(s,t)\big) \in G_{string}$.

To prove $(*)$, we show by induction on $n \in \mathbf{N}$, that

$$\begin{aligned} &\text{if} \quad \langle s,t\rangle \in \operatorname{dom} B_{join} \text{ and } \mathit{length}(s) = n \\ &\text{then } \operatorname{ran} B_{join}(s,t) = \operatorname{ran} s \cup \operatorname{ran} t. \end{aligned}$$

Note that for $\langle s,t\rangle \in \operatorname{dom} B_{join}$, the code of B_{join} yields the equation

$$B_{join}(s,t) = \begin{cases} t, & \text{if } \mathit{length}(s) = 0 \\ B_{join}(\mathit{tl}\, s, t), & \text{if } \mathit{length}(s) \neq 0 \text{ and } B_{occurs}(\mathit{hd}\, s, t) = \mathbf{T} \\ B_{join}(\mathit{tl}\, s, \mathit{cons}(\mathit{hd}\, s, t)), & \\ & \text{if } \mathit{length}(s) \neq 0 \text{ and } B_{occurs}(\mathit{hd}\, s, t) = \mathbf{F}, \end{cases}$$

because in the third case we must have $\mathit{length}(t) < K$, since otherwise $B_{join}(s,t)$ would be undefined.

As the base of the induction, let $n = 0$, and suppose that $\langle s,t\rangle \in \operatorname{dom} B_{join}$ and $\mathit{length}(s) = n = 0$. Then

$$\begin{aligned} \operatorname{ran} B_{join}(s,t) &= \operatorname{ran} t \\ &= \emptyset \cup \operatorname{ran} t \\ &= \operatorname{ran} s \cup \operatorname{ran} t. \end{aligned}$$

For the inductive step, let $n > 0$, and suppose that $\langle s,t\rangle \in \operatorname{dom} B_{join}$ and $\mathit{length}(s) = n$.

- In case $B_{occurs}(\mathit{hd}\, s,t) = \mathbf{T}$ (*i.e.*, $\mathit{hd}\, s \in \operatorname{ran} t$), we have

$$\begin{aligned}\operatorname{ran} B_{join}(s,t) &= \operatorname{ran} B_{join}(\mathit{tl}\, s,t)\\ &= \operatorname{ran}(\mathit{tl}\, s) \cup \operatorname{ran} t \qquad \text{(induction hypothesis)}\\ &= \operatorname{ran}(\mathit{tl}\, s) \cup \{\mathit{hd}\, s\} \cup \operatorname{ran} t\\ &= \operatorname{ran} s \cup \operatorname{ran} t.\end{aligned}$$

- In case $B_{occurs}(\mathit{hd}\, s,t) = \mathbf{F}$, we have

$$\begin{aligned}\operatorname{ran} B_{join}(s,t) &= \operatorname{ran} B_{join}(\mathit{tl}\, s, \mathit{cons}(\mathit{hd}\, s,t))\\ &= \operatorname{ran}(\mathit{tl}\, s) \cup \operatorname{ran} \mathit{cons}(\mathit{hd}\, s,t)\\ &= \operatorname{ran}(\mathit{tl}\, s) \cup \{\mathit{hd}\, s\} \cup \operatorname{ran} t\\ &= \operatorname{ran} s \cup \operatorname{ran} t.\end{aligned}$$

This concludes the proof of $(*)$.

Thus, *join* is compatible with G, and it has been proved that $G\colon B \xrightarrow[V]{\times} A$ is a correspondence.

Theorem 4.3.7 yields that $B \lesssim_V A$. In practice, this means that the code defining B provides a correct partial representation of the abstract data type *string* specified by A. □

The example illustrates the use of correspondences to prove practical data representations correct. In general, correspondences seem to be more convenient for this purpose than a direct structural induction over computations as suggested by the definition of behavioural inclusion. In particular, a straightforward structural induction to prove $\phi_v(t) = \psi_v(t)$ for terms t of visible sort v will not normally succeed, because no suitable induction hypothesis is available for the subterms of t of hidden sort. A correspondence can be regarded in this context as providing the induction hypothesis

$$t \in \operatorname{dom} \phi_s \implies (\phi_s(t), \psi_s(t)) \in G_s$$

for terms t of *arbitrary* sort s (cf. Lemma 4.3.8).

So far, the present section has presented a representation relation "behavioural inclusion" between partial algebras that captures the idea of "partial" or "restricted" representations of data types, where the representation may abort when its capacity is exceeded, but otherwise must exhibit observable behaviour that agrees with that of the model. Furthermore, it has been shown that behavioural inclusion between algebras is characterized by the existence of a correspondence between them, and the preceding example has illustrated that correspondences provide a practical method for proving data representations correct.

The idea of partial data representation is not new, but was introduced by Kamin and Archer [KA 84].[1] However, the definition of Kamin and Archer is different from mine. They define an algebra A to "implement" an algebra B,

---

[1] On page 322 f. of their paper, Kamin and Archer ascribe the partial representation idea to Hoare's 1972 paper [Hoare 72], because "... Hoare's paper allowed for ... pre-conditions on implementations of operations" [KA 84, p. 322]. However, this interpretation of Hoare's paper appears to be incorrect. Hoare does indeed consider preconditions on the operations in his example representation of sets by arrays:

> "For example, the correctness of the insert procedure depends on the fact that the size of the resulting set is not greater than 100 .... This precondition ... must accordingly be proved to hold before every call of the procedure." [Hoare 72, p. 276]

But the quotation shows that the precondition is stated in the *abstract* terminology (sets rather than arrays), and is to be proved when the procedure *insert* is called; that is, in the proof of the *abstract* program. Hence the condition that sets can not have more than 100 elements is not just introduced by the representation; it is to be regarded as part of the abstract model to be represented. Hoare does not consider a partial representation of "sets of integers", but a full representation of "sets of integers with no more than 100 elements":

> "... consider an abstract program which operates on several small sets of integers. It is known that none of these sets ever has more than a hundred members." [Hoare 72, p. 272]

iff there exists a homomorphism $h\colon [\![A]\!] \longrightarrow [\![B]\!]$, where $[\![A]\!]$ and $[\![B]\!]$ are certain subalgebras of A and B, called their "reachable parts" [KA 84, p. 322].

Of course, h may be regarded as a partial homomorphism $h\colon A \mathrel{+\!\!\!\rightarrow} B$. But there remain two differences to the concept of this thesis.

The first difference is that Kamin and Archer do not consider visible sorts in their definition. This could be matched in my definition by using an empty set of visible sorts (*i. e.*, by considering the relation $\lesssim_\emptyset$). This would admit more representations than a nonempty set of visible sorts. However, Section 4.2 has shown that this generality is harmful in practice, because "representations" become possible in which the output values are delivered in a useless form, *e. g.*, simply as a formal term that is independent of the operations to be implemented.

The second difference is that Kamin and Archer require h to be a homomorphism, that is, a functional correspondence, whereas our notion of behavioural inclusion allows the representing and the represented algebra to be connected by an arbitrary correspondence. The following theorem shows that this makes the representation relation of Kamin and Archer more restrictive.

4.3.11 Theorem. *There exist an algebraic signature Σ and Σ-algebras A and B such that $A \lesssim_\emptyset B$ and $B \lesssim_\emptyset A$, but no partial homomorphism from A to B nor from B to A exists.*

Proof. Let Σ have sort set $S := \{s\}$ and three operations $a, b, c\colon \rightarrow s$, and define the Σ-algebras A and B by

$$\begin{array}{lllll} A_s & = \{0,1\} & = B_s & = \{0,1\} \\ A_a() & = 0 & = B_a() & = 0 \\ A_b() & = 0 & B_b() & = 1 \\ A_c() & = 1 & = B_c() & = 1. \end{array}$$

Both $A/\emptyset$ and $B/\emptyset$ are $\emptyset$-sorted sets and thus identical; we have

$$\mathrm{T}_\Sigma(A/\emptyset)_s = \mathrm{T}_\Sigma(B/\emptyset)_s = \{\langle a\rangle, \langle b\rangle, \langle c\rangle\}.$$

The evaluation functions

$$\phi\colon \mathrm{T}_\Sigma(A/\emptyset) \mathrel{+\!\!\!\rightarrow} A/S \quad \text{and} \quad \psi\colon \mathrm{T}_\Sigma(B/\emptyset) \mathrel{+\!\!\!\rightarrow} B/S$$

are given by

$$\begin{aligned} \phi_s(\langle a \rangle) &= 0, & \psi_s(\langle a \rangle) &= 0, \\ \phi_s(\langle b \rangle) &= 0, & \psi_s(\langle b \rangle) &= 1, \\ \phi_s(\langle c \rangle) &= 1, & \psi_s(\langle c \rangle) &= 1. \end{aligned}$$

The propositions "$A \lesssim_\emptyset B$" and "$B \lesssim_\emptyset A$" by definition reduce to "$\mathrm{dom}\,\phi_s \subseteq \mathrm{dom}\,\psi_s$" and "$\mathrm{dom}\,\psi_s \subseteq \mathrm{dom}\,\phi_s$", and these are obviously true.

From Lemma 4.3.9, it follows that every correspondence $H\colon A \longrightarrow\!\!\!\!\!\times\ B$ must satisfy

$$\phi_s^{\cup}\,;\psi_s = \{(0,0),(0,1),(1,1)\} \subseteq H_s,$$

therefore H_s cannot be a partial function and H cannot be a partial homomorphism. Symmetrically, if $H\colon B \longrightarrow\!\!\!\!\!\times\ A$, the lemma yields that

$$\psi_s^{\cup}\,;\phi_s = \{(0,0),(1,0),(1,1)\} \subseteq H_s,$$

and hence that H cannot be a partial homomorphism. □

The theorem shows that $\emptyset$-behavioural inclusion is strictly more general than the "partial implementation" notion of Kamin and Archer. Of course, the example used in the proof is of purely theoretical interest. A more significant example is obtained from Example 4.3.10.

4.3.12 Example. Let $\Sigma = \langle S,\ \alpha\colon F \to S^+ \rangle$ be the signature

signature

bool, *char*, *string*: **sort**

a, *b*, *c*: $\to$ *char*

single: *char* $\to$ *string*

occurs: *char string* $\to$ *bool*

join: *string string* $\to$ *string*

(this is just the signature Σ of the previous example (4.3.10) with the operators *a*, *b*, and *c* added), and let $V := \{bool, char\}$ as in the previous example. Assume that C, the set of *char* values, contains at least three different elements p, q, and r,

and assume that the capacity K of the representation is at least 3 (*i.e.*, $K \geq 3$). Let the algebras B and A be obtained from the algebras B and A of the previous example by adding the interpretations

$$B_a() = A_a() = p, \qquad B_b() = A_b() = q, \quad \text{and} \quad B_c() = A_c() = r.$$

For the S-sorted relation G of the previous example it was shown there that the operations *single*, *join* and *occurs* are compatible with it, and obviously the three operations a, b and c are compatible with it also. Hence $G\colon B \xrightarrow[V]{} A$ is a V-correspondence from B to A, and thus $B \lesssim_V A$, that is, B is V-behaviourally included in A. Obviously, this implies that $B \lesssim_\emptyset A$.

However, no partial homomorphism can exist from B to A (we need not even consider any visible sorts, which would restrict the homomorphism even further). To see this, let

$$\phi\colon \mathrm{T}_\Sigma(A/\emptyset) \twoheadrightarrow A/S \quad \text{and} \quad \psi\colon \mathrm{T}_\Sigma(B/\emptyset) \twoheadrightarrow B/S$$

be the evaluation functions, and suppose that $H\colon B \longrightarrow\!\!\!\!\!\times\ A$ is a correspondence. By Lemma 4.3.9, applied with $V = \emptyset$, it follows that $\psi^{\cup}_{string}\,;\phi_{string} \subseteq H_{string}$. Consider now the following two terms in $\mathrm{T}_\Sigma(A/\emptyset)_{string}$ and their evaluations in B and A:

term t	$\psi_{string}(t)$	$\phi_{string}(t)$
$\langle$*join*, *single*, *a*, *join*, *single*, *b*, *single*, *c*$\rangle$ (*i.e.*, *join*($\langle$*p*$\rangle$, *join*($\langle$*q*$\rangle$, $\langle$*r*$\rangle$)))	$\langle p,q,r\rangle$	$\langle p,q,r\rangle$
$\langle$*join*, *join*, *single*, *b*, *single*, *a*, *single*, *c*$\rangle$ (*i.e.*, *join*(*join*($\langle$*q*$\rangle$, $\langle$*p*$\rangle$), $\langle$*r*$\rangle$)).	$\langle p,q,r\rangle$	$\langle q,p,r\rangle$

Since $\psi^{\cup}_{string}\,;\phi_{string} \subseteq H_{string}$, the two pairs of the form $(\psi_{string}(t), \phi_{string}(t))$ must be elements of H_{string}, so that H_{string} contains two pairs with the same left component $\langle p,q,r\rangle$ but two different right components $\langle p,q,r\rangle$ and $\langle q,p,r\rangle$. Hence H_{string} cannot be a partial function. Since H was an arbitrary correspondence, no partial homomorphism from B to A exists. □

We here have a more practical example of a representation that is correct according to the behavioural inclusion criterion, but not correct according to the

"partial implementation" concept of Kamin and Archer. The example shows that the representation concept of Kamin and Archer is affected by "representation bias" in a specification, whereas behavioral inclusion is not. This phenomenon and its significance is discussed in detail on page 258 f. below, where "behavioral equivalence" is compared with "standard representation". The relation between these two representation concepts is precisely analogous to the relation between behavioural inclusion and the "partial implementation" concept of Kamin and Archer: "standard representation" and "partial implementation" insist on a function from representation values to the represented values, while the behavioural concepts allow a relation instead.

The following theorem states that behavioural inclusion is a representation relation in the institution $\langle$**TSig, TIncl, TAlg**$\rangle$. As a consequence, the general theory of Section 4.1 applies to behavioural inclusion.

4.3.13 Theorem. *Behavioural inclusion is a chain-closed representation relation in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$.

To be precise, one obtains a chain-closed representation relation $\rightsquigarrow = \langle \underset{\langle \Sigma, V \rangle}{\rightsquigarrow} \rangle_{\langle \Sigma, V \rangle \in |\mathbf{TSig}|}$ *by defining*

$$A \underset{\langle \Sigma, V \rangle}{\rightsquigarrow} B \;:\Longleftrightarrow\; A \lesssim_V B \qquad \text{for } A, B \in \mathbf{Alg}(\Sigma) = \mathbf{TAlg}\langle \Sigma, V \rangle.$$

In the proof of this theorem, the following proposition about correspondences will be used.

4.3.14 Proposition. *If* $\sigma\colon \langle \Sigma, V \rangle \to \langle \Sigma', V' \rangle$ *is a tagged signature morphism where* $\Sigma = \langle S, \alpha \rangle$, *and* $G\colon A \underset{V'}{\rightarrowtail\!\!\!\!\times} B$ *is a* V'*-correspondence between* Σ'*-algebras* A *and* B, *then the* S*-sorted relation* $\overline{\sigma}\, G$ *defined by* $(\overline{\sigma}\, G)_s = G_{\sigma s}$ *is a* V*-correspondence* $\overline{\sigma}\, G\colon \overline{\sigma}\, A \underset{V}{\rightarrowtail\!\!\!\!\times} \overline{\sigma}\, B$.

Proof. Consider $f\colon s_1 \dots s_n \to r$ in Σ, and

$$(x_i, y_i) \in (\overline{\sigma}\, G)_{s_i} = G_{\sigma s_i} \qquad \text{for } i \in \{1, \dots, n\}$$

such that $\langle x_1,\ldots,x_n\rangle \in \mathrm{dom}(\overline{\sigma}\,A)_f = \mathrm{dom}\,A_{\sigma f}$. Then $\sigma f\colon \sigma s_1\ldots\sigma s_n \to \sigma r$ in Σ'. Since G is a correspondence, it follows that

$$\langle y_1,\ldots,y_n\rangle \in \mathrm{dom}\,B_{\sigma f} = \mathrm{dom}(\overline{\sigma}\,B)_f$$

and

$$\begin{aligned}\big((\overline{\sigma}\,A)_f(x_1,\ldots,x_n),\,(\overline{\sigma}\,B)_f(y_1,\ldots,y_n)\big) &= \big(A_{\sigma f}(x_1,\ldots,x_n),\,B_{\sigma f}(y_1,\ldots,y_n)\big)\\ &\in G_{\sigma r}\\ &= (\overline{\sigma}\,G)_r\,.\end{aligned}$$

Hence $(\overline{\sigma}\,G)\colon(\overline{\sigma}\,A) \longrightarrow\!\!\!\!\times\, (\overline{\sigma}\,B)$.

To see that $\overline{\sigma}\,G$ is V-constant, observe that for $v \in V$: $\sigma v \in V'$ (as σ is a tagged signature morphism) and hence, as G is a V'-correspondence:

$$(\overline{\sigma}\,A)_v = A_{\sigma v} \subseteq B_{\sigma v} = (\overline{\sigma}\,B)_v,$$

and

$$(\overline{\sigma}\,G)_v = G_{\sigma v} \text{ is the inclusion function from } A_{\sigma v} \text{ to } B_{\sigma v},$$

i. e., $(\overline{\sigma}\,G)_v$ is the inclusion function from $(\overline{\sigma}\,A)_v$ to $(\overline{\sigma}\,B)_v$. Hence $\overline{\sigma}\,G\colon \overline{\sigma}\,A \underset{V}{\longrightarrow\!\!\!\!\times}\, \overline{\sigma}\,B$ is a V-correspondence. □

Furthermore, two lemmas about terms and evaluation functions are required.

4.3.15 Lemma. *Let $\langle\Sigma,V\rangle \sqsubseteq \langle\Sigma',V'\rangle$ be tagged signatures where $\Sigma = \langle S,\alpha\rangle$ and $\Sigma' = \langle S',\alpha'\rangle$, let $A \in \mathbf{Alg}(\Sigma')$, and let*

$$\begin{aligned}&\phi'\colon \mathrm{T}_{\Sigma'}(A/V') \twoheadrightarrow A/S' \text{ be the evaluation function defined by } A,\\ &\phi\colon\ \mathrm{T}_{\Sigma}(A/V) \twoheadrightarrow A/S \text{ be the evaluation function defined by } A/\Sigma.\end{aligned}$$

Then for all $s \in S$:

$$\begin{aligned}&\mathrm{T}_{\Sigma}(A/V)_s \subseteq \mathrm{T}_{\Sigma'}(A/V')_s \quad \text{and}\\ &\phi_s = \phi'_s/(\mathrm{T}_{\Sigma}(A/V)_s).\end{aligned}$$

Proof. We prove the conclusion by structural induction on terms in $\mathrm{T}_\Sigma(A/V)$.

(a) $t = \langle v, x\rangle$ with $v \in V$, $x \in A_v$.

Then $v \in V'$ by Theorem 4.2.6 (a), hence $\langle v, x\rangle \in \mathrm{T}_{\Sigma'}(A/V')$, and

$$\phi_s(\langle v, x\rangle) = x = \phi'_s(\langle v, x\rangle).$$

(b) $t = \langle f\rangle \circ u_1 \circ \cdots \circ u_n$ with $f\colon s_1 \ldots s_n \to r$ in Σ, $u_i \in \mathrm{T}_\Sigma(A/V)_{s_i}$ for $i \in \{1, \ldots, n\}$.

By the induction hypothesis,

$$u_i \in \mathrm{T}_{\Sigma'}(A/V')_{s_i} \qquad \text{for } i \in \{1, \ldots, n\}.$$

Since $f\colon s_1 \ldots s_n \to r$ in Σ' also, it follows that $t \in \mathrm{T}_{\Sigma'}(A/V')_{s_i}$. Now

$$\begin{aligned} t \in \operatorname{dom}\phi_r &\iff u_i \in \operatorname{dom}\phi_{s_i} \text{ for } i \in \{1, \ldots, n\} \\ &\qquad \text{and } \langle \phi_1 u_1, \ldots, \phi_n u_n\rangle \in \operatorname{dom} A_f \\ &\iff u_i \in \operatorname{dom}\phi'_{s_i} \text{ for } i \in \{1, \ldots, n\} \\ &\qquad \text{and } \langle \phi'_1 u_1, \ldots, \phi'_n u_n\rangle \in \operatorname{dom} A_f \\ &\qquad \text{(by the inductive hypothesis)} \\ &\iff t \in \operatorname{dom}\phi'_r. \end{aligned}$$

If $t \in \operatorname{dom}\phi_r$, we have

$$\begin{aligned} \phi_r(t) &= A_f(\phi_1 u_1, \ldots, \phi_n u_n) \\ &= A_f(\phi'_1 u_1, \ldots, \phi'_n u_n) \\ &\qquad \text{(by the inductive hypothesis)} \\ &= \phi'_r(t). \end{aligned}$$

Hence ϕ and ϕ' agree on t. □

4.3.16 Lemma. *Let $\langle \Sigma_i, V_i\rangle_{i\in I}$ be a compatible chain of small tagged signatures, and $A \in \mathbf{Alg}(\bigsqcup_{i\in I} \Sigma_i)$. Defining $\langle \bar\Sigma, \bar V\rangle := \bigsqcup_{i\in I} \langle \Sigma_i, V_i\rangle$, and writing $\Sigma_i = \langle S_i, \alpha_i\rangle$ for $i \in I$ and $\langle \bar\Sigma, \bar V\rangle = \langle \bar S, \bar\alpha\rangle$, we have for all $s \in \bar S$:*

$$\mathrm{T}_{\bar\Sigma}(A/\bar V)_s = \bigcup_{\{i \mid s \in S_i\}} \mathrm{T}_{\Sigma_i}(A/V_i)_s.$$

Proof. By the previous lemma, $\mathrm{T}_{\Sigma_i}(A/V_i)_s$ is included in $\mathrm{T}_{\bar{\Sigma}}(A/\bar{V})_s$ whenever $s \in S_i$.

For the converse inclusion, we prove by structural induction over terms t in $\mathrm{T}_{\bar{\Sigma}}(A/\bar{V})$, that if $t \in \mathrm{T}_{\bar{\Sigma}}(A/\bar{V})_s$, then there exists $i \in I$ such that $s \in S_i$ and $t \in \mathrm{T}_{\Sigma_i}(A/V_i)_s$.

(a) $t = \langle v, x \rangle$ with $v \in \bar{V}$ and $x \in A_v$.

Pick i such that $v \in V_i$. Then $x \in A_v = (A/V_i)_v$, and so

$$t = \langle v, x \rangle \in \mathrm{T}_{\Sigma_i}(A/V_i).$$

(b) $t = \langle f \rangle \circ u_1 \circ \cdots \circ u_n$ with $f\colon s_1 \ldots s_n \to r$ in $\bar{\Sigma}$, $u_i \in \mathrm{T}_{\Sigma}(A/V)_{s_i}$ for $i \in \{1, \ldots, n\}$.

By the inductive hypothesis, for each $i \in \{1, \ldots, n\}$ we can pick $k_i \in I$ such that $u_i \in \mathrm{T}_{\Sigma_{k_i}}(A/V_{k_i})_{s_i}$. Let k_0 be such that $f\colon s_1 \ldots s_n \to r$ in Σ_{k_0}. The set of tagged signatures $\{\langle \Sigma_{k_0}, V_{k_0} \rangle, \ldots, \langle \Sigma_{k_n}, V_{k_n} \rangle\}$ is finite and totally ordered by $\sqsubseteq$, hence has a greatest element. Define j to be such that $\langle \Sigma_j, V_j \rangle$ is this greatest element. We then have $f\colon s_1 \ldots s_n \to r$ in Σ_j, and by the previous lemma, $u_i \in \mathrm{T}_{\Sigma_j}(A/V_j)_{s_i}$ for $i \in \{1, \ldots, n\}$. Hence $\langle f \rangle \circ u_1 \circ \cdots \circ u_n \in \mathrm{T}_{\Sigma_j}(A/V_j)_r$. □

Proof of Theorem 4.3.13.

We verify the three axioms of Definition 4.1.1.

Axiom (a): For every $\langle \Sigma, V \rangle \in |\mathbf{TSig}|$, the relation $\underset{\langle \Sigma, V \rangle}{\rightsquigarrow}$ is a preordering on $\mathbf{TAlg}\langle \Sigma, V \rangle = \mathbf{Alg}(\Sigma)$, because V-behavioural inclusion is a preordering on $\mathbf{Alg}(\Sigma)$.

Axiom (b): If $\sigma\colon \langle \Sigma, V \rangle \to \langle \Sigma', V' \rangle$ is morphism in $\mathbf{TSig}$ and $A \underset{\langle \Sigma', V' \rangle}{\rightsquigarrow} B$, then by Theorem 4.3.7 there exists a V'-correspondence $G\colon A \underset{V'}{\multimap} B$. By Proposition 4.3.14, $\overline{\sigma} G\colon \overline{\sigma} A \underset{V}{\multimap} \overline{\sigma} B$. Applying Theorem 4.3.7 again yields that $\overline{\sigma} A \underset{\langle \Sigma, V \rangle}{\rightsquigarrow} \overline{\sigma} B$.

Axiom (c): Let $\langle \Sigma_i, V_i \rangle_{i \in I}$ be a compatible chain of small tagged signatures, where $\Sigma_i = \langle S_i, \alpha_i \rangle$ for $i \in I$, and define $\bar{\Sigma} = \langle \bar{S}, \bar{\alpha} \rangle := \bigsqcup_{i \in I} \Sigma_i$, $\bar{V} := \bigcup_{i \in I} V_i$, such that $\langle \bar{\Sigma}, \bar{V} \rangle = \bigsqcup_{i \in I} \langle \Sigma_i, V_i \rangle$.

Let A and B be $\bar{\Sigma}$-algebras such that $A/\langle \Sigma_i, V_i\rangle \underset{\langle \Sigma_i, V_i\rangle}{\rightsquigarrow} B/\langle \Sigma_i, V_i\rangle$ (*i. e.*, $A/\Sigma_i \lesssim_{V_i} B/\Sigma_i$) for all $i \in I$, and let

ϕ_i: $\mathrm{T}_{\Sigma_i}(A/V_i) \twoheadrightarrow A/S_i$ be the evaluation function of A/Σ_i for $i \in I$,
ψ_i: $\mathrm{T}_{\Sigma_i}(B/V_i) \twoheadrightarrow B/S_i$ be the evaluation function of B/Σ_i for $i \in I$,
$\bar{\phi}$: $\mathrm{T}_{\bar{\Sigma}}(A/\bar{V}) \twoheadrightarrow A/\bar{S}$ be the evaluation function of A, and
$\bar{\psi}$: $\mathrm{T}_{\bar{\Sigma}}(B/\bar{V}) \twoheadrightarrow B/\bar{S}$ be the evaluation function of B.

We show that $A \lesssim_{\bar{V}} B$.

To verify Clause (a) of the definition of behavioural inclusion (Def. 4.3.3), consider $v \in \bar{V}$. Choose $i \in I$ such that $v \in V_i$. Since $A/\Sigma_i \lesssim_{V_i} B/\Sigma_i$, we have

$$A_v = (A/\Sigma_i)_v \subseteq (B/\Sigma_i)_v = B_v.$$

To verify Clause (b) of the definition, consider $s \in S$ and $t \in \mathrm{dom}\,\bar{\phi}_s \subseteq \mathrm{T}_{\bar{\Sigma}}(A/\bar{V})_s$. By Lemma 4.3.16, we can pick $i \in I$ such that $s \in S_i$ and $t \in \mathrm{T}_{\Sigma_i}(A/V_i)_s$. Since $A/\Sigma_i \lesssim_{V_i} B/\Sigma_i$, the V_i-sorted set $A/V_i = (A/\Sigma_i)/V_i$ is componentwise included in $(B/\Sigma_i)/V_i = B/V_i$, hence $\mathrm{T}_{\Sigma_i}(A/V_i)$ is componentwise included in $\mathrm{T}_{\Sigma_i}(B/V_i)$, and so $t \in \mathrm{T}_{\Sigma_i}(B/V_i)_s$ also. Lemma 4.3.15 yields that $t \in \mathrm{dom}\,\phi_{i,s}$, and because $A/\Sigma_i \lesssim_{V_i} B/\Sigma_i$, it follows that $t \in \mathrm{dom}\,\psi_{i,s}$. By Lemma 4.3.15 again, it follows that $t \in \mathrm{dom}\,\bar{\psi}_s$.

To very Clause (c) of the definition, assume that in addition, we have $s \in \bar{V}$. Since $\langle \Sigma_i, V_i\rangle \sqsubseteq \langle \bar{\Sigma}, \bar{V}\rangle$, we have $s \in V_i = \bar{V} \cap S_i$ also, and hence

$$\begin{aligned}
\bar{\phi}_s(t) &= \phi_{i,s}(t) && \text{(by Lemma 4.3.15)} \\
&= \psi_{i,s}(t) && \text{(as } s \in V_i \text{ and } (A/\Sigma_i) \lesssim_{V_i} (B/\Sigma_i)\text{)} \\
&= \bar{\psi}_s(t) && \text{(by Lemma 4.3.15).}
\end{aligned}$$

This concludes the proof that $A \lesssim_{\bar{V}} B$, and hence that behavioural inclusion is a chain-closed representation relation. □

With the theorem that behavioural inclusion is a representation relation, we now have available a universal implementation concept for cells in $\langle$**TSig**, **TIncl**, **TAlg**$\rangle$ that could be used in program development. The concept will be further analysed in Section 5.2, where a simplified way of proving the universal implementation property is developed and an example is given.

4.4 Behavioural Equivalence

We now turn to the representation relation between partial algebras that is perhaps the most important for practical programming: behavioural equivalence.

4.4.1 Definition. Let Σ be an algebraic signature, and let V be a subset of the sorts of Σ. Two Σ-algebras A and B are *V-behaviourally equivalent* ("$A \simeq_V B$"), if A is V-behaviourally included in B and *vice versa*, *i. e.*,

$$A \simeq_V B \ :\Longleftrightarrow\ A \lesssim_V B \wedge B \lesssim_V A. \qquad \square$$

This means that V-behavioural equivalence is just the equivalence relation induced by V-behavioural inclusion. From the definition of V-behavioural inclusion, we immediately obtain a characterization of behavioural equivalence in terms of the evaluation functions.

4.4.2 Proposition. *Let $\Sigma = \langle S, \alpha \rangle$ be an algebraic signature, and let $V \subseteq S$. Let A and B be Σ-algebras with evaluation functions*

$$\phi\colon \mathrm{T}_\Sigma(A/V) \twoheadrightarrow A/S, \qquad \psi\colon \mathrm{T}_\Sigma(B/V) \twoheadrightarrow B/S.$$

Then A is V-behaviourally equivalent to B, if and only if

(a) $\forall v \in V\colon\ A_v = B_v$,

(b) $\forall s \in S\colon\ \mathrm{dom}\,\phi_s = \mathrm{dom}\,\psi_s$, *and*

(c) $\forall v \in V\colon\ \phi_v = \psi_v$. $\square$

Viewing terms in $\mathrm{T}_\Sigma(A/V)$ as computations starting with the visible values in A/V, clause (a) says that the visible carriers of A and B must be equal, clause (b) says that the same computations succeed in A and in B, and clause (c) says that for succeeding computations of visible sort, the result values are the same.

Together, these clauses mean that A and B have the same observable behaviour under the constraint that values in "hidden" sorts can only be generated

and inspected by means of the operations, and it seems natural to use "behavioural equivalence" as the formal criterion for a data structure to be a "correct representation" of another one.

The idea of regarding "behaviour" as the relevant aspect of a data structure is implicit in the "simulation" method proposed by Milner [Milner 71] to prove a program correct with respect to another one, which allows for different internal data representations (cf. Example 2, p. 486 f. and 485 of [Milner 71]). An explicit behaviour concept for algebras first appears in [GGM 76, p. 580] (called "semantics" of an algebra).

The exact behavioural equivalence relation defined here has been studied before by Bothe [Bothe 81], myself [Schoett 81, Section 5.1], and by Meseguer and Goguen [GM 82, p. 269][1]. Comparisons of this notion with other behavioural equivalence notions can be found in [HR 83] and [ST 84a, p. 17].

Just as behavioural inclusion is characterized by the existence of a correspondence, behavioural equivalence is characterized by the existence of a "strong correspondence".

4.4.3 Definition. Let $\Sigma = \langle S, \alpha \rangle$ be an algebraic signature, and let A and B be Σ-algebras. A *strong correspondence* G from A to B (Notation: "$G\colon A \Longrightarrow\!\!\!\!\times\ B$") is a correspondence from A to B whose converse is a correspondence from B to A, *i. e.*,

$$G\colon A \Longrightarrow\!\!\!\!\times\ B \ :\Longleftrightarrow\ G\colon A \longrightarrow\!\!\!\!\times\ B \ \wedge\ G^{\cup}\colon B \longrightarrow\!\!\!\!\times\ A.$$

A *strong partial homomorphism* $G\colon A \Longrightarrow\!\!\!\!\!\!+\ B$ is a strong correspondence that is a partial homomorphism; a *strong homomorphism* $G\colon A \Longrightarrow B$ is a strong correspondence that is a homomorphism.

---

[1] Note that only "having the same behaviour" in the sense of [GM 82, p. 269] matches behavioural equivalence; the "behavioural equivalence" concept of the same paper [GM 82, Section 4.2] does not.

For $V \subseteq S$, a strong correspondence (strong partial homomorphism, strong homomorphism) G from A to B is a *strong V-correspondence* (*strong partial V-homomorphism*, *strong V-homomorphism*), if for all $v \in V$:

$$A_v = B_v \quad \text{and} \quad G_v \text{ is the identity map on this set}$$

(*i.e.*, if both G and $G^\cup$ are V-correspondences); this fact is written "$G\colon A \underset{V}{\Longrightarrow\!\!\!\!\times} B$" ("$G\colon A \underset{V}{\Longleftrightarrow} B$", "$G\colon A \underset{V}{\Longrightarrow} B$").

The following proposition gives a direct characterization of strong correspondences as relations with which the functions of A and B are "strongly compatible".

4.4.4 Proposition. *Let $\Sigma = \langle S, \alpha\colon F \to S^+\rangle$ be an algebraic signature, and let A and B be Σ-algebras. An S-sorted relation $G = \langle G_s\rangle_{s\in S}$ from A/S to B/S is a strong correspondence, if and only if all $f \in F$ are "strongly compatible" with G, i.e., if $f\colon s_1 \dots s_n \to r$ in Σ, then*

$$\begin{aligned} &\textit{whenever } (x_i, y_i) \in G_{s_i} \textit{ for } i \in \{1, \dots, n\},\\ &\textit{then} \quad \langle x_1, \dots, x_n\rangle \in \operatorname{dom} A_f \iff \langle y_1, \dots, y_n\rangle \in \operatorname{dom} B_f,\\ &\qquad \textit{and if both sides of this equivalence are true, then}\\ &\qquad (A_f(x_1, \dots, x_n),\ B_f(y_1, \dots, y_n)) \in G_r. \end{aligned}$$

Proof. The condition given here is symmetric under exchange of A and B plus substitution of $G^\cup$ for G. Since it clearly implies $G\colon A \longrightarrow\!\!\!\!\times\ B$, it also implies $G^\cup\colon B \longrightarrow\!\!\!\!\times\ A$. Conversely, if $G\colon A \Longrightarrow\!\!\!\!\times\ B$, then from $G^\cup\colon B \longrightarrow\!\!\!\!\times\ A$, it follows that if $f\colon s_1 \dots s_n \to r$ in Σ and $(x_i, y_i) \in G_{s_i}$ (*i.e.*, $(y_i, x_i) \in G^\cup_{s_i}$) for $i \in \{1, \dots, n\}$, we have

$$\langle y_1, \dots, y_n\rangle \in \operatorname{dom} B_f \implies \langle x_1, \dots, x_n\rangle \in \operatorname{dom} A_f.$$

Combining this with the definition of "$G\colon A \longrightarrow\!\!\!\!\times\ B$", we get the condition of the proposition. □

We have seen that a correspondence is a strong correspondence, if for related argument tuples $\langle x_1, \ldots, x_n \rangle$ and $\langle y_1, \ldots, y_n \rangle$, we have

$$\langle x_1, \ldots, x_n \rangle \in \operatorname{dom} A_f \iff \langle y_1, \ldots, y_n \rangle \in \operatorname{dom} B_f,$$

rather than just the implication from left to right, as in the definition of a correspondence (Def. 4.3.4). This means that A_f and B_f either both succeed or both fail on related arguments, and, as for correspondences, their results are related if they succeed.

Strong homomorphisms are well known in the literature on partial algebras; *e. g.*, they appear in [Grätzer 79, p. 81] and [BW 82, p. 51], and under the name "closed homomorphism" in [Burmeister 82, p. 311] and [Reichel 84, p. 76]. The strong partial homomorphism concept seems to appear first in [Schoett 81, p. 109] under the name "homomorphism".[1] It can be seen as combining the "representation invariant" (as the domain of the homomorphism) and the "abstraction function" of [Hoare 72] into a single concept—this connection will be exploited in the next section.

The strong correspondence concept seems to have been first presented in [Schoett 83, p. 22], later in [Schoett 85, p. 8] under the name "correspondence". It has recently been generalized by Nipkow to a "simulation" concept between "structures" that may contain nondeterministic operations [Nipkow 86, p. 633]. Partial many-sorted algebras are special "structures", and the "simulations" between partial many-sorted algebras are just the strong correspondences.

Just like correspondences, the strong V-correspondences between Σ-algebras form a category.

---

[1] The "homomorphism" definition on p. 109 in this report contains a crucial printing error: The defining relation must hold

$$\text{for all } \langle x_1, \ldots, x_n \rangle \in \textstyle\prod_{i \in [n]} \operatorname{dom} h_{s_i};$$

this line is missing from the definition, as can be seen in later proofs on the pages 111, 121, and 132.

4.4.5 Proposition. *Let $\Sigma = \langle S, \alpha\colon F \to S^+\rangle$ be an algebraic signature, and let $V \subseteq S$. The following components form a category:*

- *objects: a set of Σ-algebras,*
- *arrows: the strong V-correspondences (strong partial V-homomorphisms, strong V-homomorphisms) between the algebras,*
- *identity arrow for an object A: the S-sorted identity map from A/S to itself,*
- *composition of arrows: componentwise relational composition.*

Proof. Every strong V-correspondence is a V-correspondence, and these form a category with the same identities and composition as in the definition above (Proposition 4.3.6).

We show that the composition of strong V-correspondences is again a strong V-correspondence. Let $G\colon A \underset{V}{\Longrightarrow\!\!\!\!\times} B$ and $H\colon B \underset{V}{\Longrightarrow\!\!\!\!\times} C$ be strong V-correspondences. Since $G\colon A \underset{V}{\longrightarrow\!\!\!\!\times} B$ and $H\colon B \underset{V}{\longrightarrow\!\!\!\!\times} C$, it follows that $G;H\colon A \underset{V}{\longrightarrow\!\!\!\!\times} C$ as the V-correspondences form a category, and since $G^{\cup}\colon B \underset{V}{\longrightarrow\!\!\!\!\times} A$ and $H^{\cup}\colon C \underset{V}{\longrightarrow\!\!\!\!\times} B$, it follows that $(G\,;H)^{\cup} = (H^{\cup}\,;G^{\cup})\colon C \underset{V}{\longrightarrow\!\!\!\!\times} A$ for the same reason. Hence $(G\,;H)\colon A \underset{V}{\Longrightarrow\!\!\!\!\times} B$ is a strong V-correspondence, and it follows that the strong V-correspondences form a subcategory of the V-correspondences.

Since the property of being a partial or total function is preserved under composition, it follows that the strong partial V-homomorphisms and the strong V-homomorphisms also form categories. □

Just as behavioural inclusion is characterized by correspondences, behavioural equivalence is characterized by strong correspondences.

4.4.6 Theorem. *Let Σ be an algebraic signature, let V be a subset of its sorts, and let A and B be Σ-algebras. Then A is V-behaviourally equivalent to B, if and only if there exists a strong V-correspondence from A to B.*

Proof. Suppose first that there exists a strong V-correspondence G: $A \underset{V}{\rightleftharpoons\!\!\times} B$. Then G: $A \underset{V}{\longrightarrow\!\!\!\!\times} B$ and $G^{\cup}$: $B \underset{V}{\longrightarrow\!\!\!\!\times} A$, and Theorem 4.3.7 yields that $A \lesssim_V B$ and $B \lesssim_V A$, hence $A \simeq_V B$.

Conversely, suppose that $A \simeq_V B$, *i.e.*, that $A \lesssim_V B$ and $B \lesssim_V A$. Write $\Sigma = \langle S, \alpha \rangle$, and let

$$\phi\colon \mathrm{T}_\Sigma(A/V) \twoheadrightarrow A/S \quad \text{and} \quad \psi\colon \mathrm{T}_\Sigma(B/V) \twoheadrightarrow B/S$$

be the evaluation functions of A and B. Lemma 4.3.9 yields that $(\phi^{\cup}\,;\psi)$: $A \underset{V}{\longrightarrow\!\!\!\!\times} B$, and, applying the lemma with A and B, ϕ and ψ interchanged, we get that $(\psi^{\cup}\,;\phi)$: $B \underset{V}{\longrightarrow\!\!\!\!\times} A$. Since $\psi^{\cup}\,;\phi$ is the converse of $\phi^{\cup}\,;\psi$, it follows that $(\phi^{\cup}\,;\psi)$: $A \underset{V}{\rightleftharpoons\!\!\times} B$. □

The following example illustrates how strong correspondences can be used to prove two algebras behaviourally equivalent.

4.4.7 Example. Let the signature Σ and the Σ-algebra A specifying "strings" be given as in Example 4.3.10. Define the "representation" algebra B to be like A, except that this time, B_{join} is defined by the code

$$\begin{aligned} join(s,t) = \text{if}\quad & length(s) = 0 \ \text{ then } t \\ \text{else if } & occurs(hd\,s, t) \text{ then } join(tl\,s, t) \\ \text{else} \quad & join(tl\,s, cons(hd\,s, t)). \end{aligned}$$

This is just the code for B_{join} of Example 4.3.10, except that *join* no longer imposes a restriction on the length of the list it produces.

To show that B is behaviorally equivalent to A over $V := \{bool, char\}$, we construct a strong V-correspondence G: $B \underset{V}{\rightleftharpoons\!\!\times} A$. Define G as in Example 4.3.10 (recall that $C = A_{char} = B_{char}$):

$$\begin{aligned} &G_{bool} \text{ is the identity relation on } \{\mathbf{T}, \mathbf{F}\}, \\ &G_{char} \text{ is the identity relation on } C, \\ &G_{string} = \{\, (s', s) \mid \operatorname{ran} s' = \operatorname{ran} s \,\} \subseteq C^* \times C^*. \end{aligned}$$

By a proof precisely along the lines of the proof in Example 4.3.10, one proves that G is a correspondence. Since B and A are total, Proposition 4.4.4 (in which

the two statements of the form "$\langle \ldots \rangle \in \mathrm{dom} \ldots$" are uniformly true) yields that G is a strong correspondence. Since $B_{bool} = A_{bool}$, $B_{char} = A_{char}$, and G_{bool} and G_{char} are the identities, G is a strong V-correspondence from B to A. By Theorem 4.4.6, it follows that $A \simeq_V B$. □

As a corollary of the theorem that behavioural inclusion is a representation relation in the institution $\langle$**TSig, TIncl, TAlg**$\rangle$ (Theorem 4.3.13), we obtain the same result for behavioural equivalence.

4.4.8 Theorem. *Behavioural equivalence is a chain-closed representation relation in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$.

To be precise, one obtains a chain-closed representation relation $\rightsquigarrow = \langle \underset{\langle \Sigma, V \rangle}{\rightsquigarrow} \rangle_{\langle \Sigma, V \rangle \in |\mathbf{TSig}|}$ *by defining*

$$A \underset{\langle \Sigma, V \rangle}{\rightsquigarrow} B \ :\Longleftrightarrow\ A \simeq_V B \qquad \text{for } A, B \in \mathbf{Alg}(\Sigma) = \mathbf{TAlg}\langle \Sigma, V \rangle.$$

Proof. For every $\langle \Sigma, V \rangle \in |\mathbf{TSig}|$, the relation $\simeq_V$ is the intersection of the relation $\lesssim_V$ with its converse. Hence the $|\mathbf{TSig}|$-indexed behavioural equivalence relation is the componentwise intersection of the $|\mathbf{TSig}|$-indexed behavioural inclusion relation with its componentwise converse. Since behavioural inclusion is a chain-closed representation relation (Theorem 4.3.13), Propositions 4.1.3 and 4.1.4 yield that behavioural equivalence also is a chain-closed representation relation. □

This theorem makes the theory of Section 4.1 applicable to behavioural equivalence; we obtain a universal implementation concept and a composability theorem. In principle, this provides a program development method. Section 5.3 will present a more practical version of this method.

4.5 Standard Representation

This section discusses the "standard" notion of representation of an algebra by another, which is due to Hoare [Hoare 72]. This representation concept requires that there be a mapping from the carriers of the representation algebra to the algebra represented, usually called the "abstraction function" (written "$\mathcal{A}$" in [Hoare 72, p. 275]). The operations of the algebras must be compatible with the abstraction function (to be precise, "strongly compatible" in the sense of Prop. 4.4.4). The abstraction function need only be defined on a subset of the values of the representation; this subset is characterized by a predicate usually called the "representation invariant" (written "I" in [Hoare 72, p. 275]). This proof method has already been sketched informally in Section 1.4, on page 54–57.

Numerous formal representation relations, stated in varying terminology, are based on Hoare's concept; for example, the ones in [GHM 78] (note that an abstraction function is considered part of the representation: "$SYMT$" in Fig. 5), [Jones 80, Ch. 11], [Ehrich 82, p. 216] (note that here we have an abstraction function f followed by a restriction step, rather than *vice versa*), [SW 82, p. 13], [Lipeck 83, p. 52], and [BW 84, p. 269]. Hoare's concept also appears disguised as a pair "representation invariant" plus "congruence" (every abstraction function gives rise to a congruence, while the canonical projection associated with a congruence is an abstraction function); for example, in [GTW 78, p. 138] and [EKMP 82, p. 228].

In the terminology of this thesis, the pair "representation invariant" plus "abstraction function" is just a strong partial homomorphism; the domain of the homomorphism is the representation invariant, and the homomorphism itself is the abstraction function on this set.

In most of the papers mentioned so far, the strong partial homomorphism from the representation to the algebra represented is also required to be surjective; that is, every value in the algebra represented must have at least one "representation" (a value that is mapped to it by the abstraction function). However, surjectivity was not required in Hoare's original paper [Hoare 72].

Sometimes surjectivity results automatically from the fact that only "reachable" algebras are considered; that is, algebras in which all elements can be generated by a finite number of operation applications, for example in [BW 84, p. 269]. Surjectivity is also implicit in representation relations based on congruences or quotients ([GTW 78, p. 138], [EKMP 82, p. 228]).

Since all the papers cited—except Hoare's—do require surjectivity of the abstraction function, this property will be regarded part of the standard representation concept.

A set of visible sorts will also be used in our version of the standard representation concept, although this is rarely done in the literature. The reason for this was explained in Section 4.2: Without visible sorts, the relation would admit trivial representations by term algebras.

We arrive at the following definition of the standard representation concept.

4.5.1 Definition. Let A and B be Σ-algebras, and let V be a subset of the sorts of Σ. An *abstraction function* from A to B is a surjective strong partial homomorphism from A to B (Notation: "$h\colon A \twoheadrightarrow\!\!\!\!\!\rightarrow B$"); a *$V$-abstraction function* from A to B is a surjective strong partial V-homomorphism from A to B (Notation: "$h\colon A \underset{V}{\twoheadrightarrow\!\!\!\!\!\rightarrow} B$").

The algebra A is a *standard V-representation* of B (written "$A \backsimeq_V B$"), iff there exists a V-abstraction function from A to B. □

To illustrate the meaning of this definition, here is an explicit description of what an abstraction function is.

4.5.2 Proposition. *Let A and B be Σ-algebras. An S-sorted partial function $h\colon A/S \twoheadrightarrow B/S$ is an abstraction function from A to B, iff all the h_s $(s \in S)$ are surjective, and whenever $f\colon s_1 \dots s_n \to r$ in Σ and $x_i \in \operatorname{dom} h_{s_i}$ for $i \in \{1,\dots,n\}$, then*

$$\langle x_1,\dots,x_n\rangle \in \operatorname{dom} A_f \iff \langle h_{s_1}x_1,\dots,h_{s_n}x_n\rangle \in \operatorname{dom} B_f,$$

and if both sides of this equivalence are true, then

$$h_r A_f(x_1, \ldots, x_n) = B_f(h_{s_1} x_1, \ldots, h_{s_n} x_n).$$

For V a subset of the sorts of Σ, h is a V-abstraction function, if in addition for all $v \in V$

$$A_v = B_v \quad \text{and} \quad h_v \text{ is the identity function on this set.}$$

Proof. This proposition is easily derived from the previous definition and Proposition 4.4.4, noting that for $i \in \{1, \ldots, n\}$,

$$(x_i, y_i) \in h_{s_i} \iff x_i \in \operatorname{dom} h_{s_i} \text{ and } y_i = h_{s_i} x_i. \qquad \square$$

4.5.3 Example. We prove the correctness of the module M_{STORE} of Figure 1-19, which is the proof that was omitted from Section 1.4. The method of proof we are using was already sketched informally in that section (page 54–57), and we can now perform the proof formally, using the standard representation notion just defined. In Section 5.4 below, it will be shown that this proof is the correctness proof of M_{STORE} in the *dictionary* program development that is required by the theory.

Precisely, we prove that whenever C is a model of the interface $I_{ITEM} \wedge I_{LISTITEM}$, then the algebra B defined on C by the module M_{STORE} is a standard $\{bool, item, listitem\}$-representation of a result A of the specification cell M_{STORE} on C. Recall from Figure 3-4 that M_{STORE} consists of the interfaces

$$\mathcal{Q}_{STORE} = I_{ITEM} \wedge I_{LISTITEM}$$
$$\mathcal{R}_{STORE} = I_{INSERT} \wedge I_{MIN}.$$

The desired "abstract" algebra A therefore must satisfy $I_{INSERT} \wedge I_{MIN}$. This almost fully defines A, except for the behaviour of *min* and *removemin* on an empty *store*, which this interface does not prescribe. Since in the representation B, these functions abort on the representation of the empty *store* (*i. e.*, the empty list), we must choose A so that these functions abort on the empty *store* as well.

Let C be a model of the interface $I_{ITEM} \wedge I_{LISTITEM}$. We shall not attempt to write down the result B of M_{STORE} on C directly, but rather proceed as in Examples 4.3.10 and 4.4.7 and verify the desired properties of B from the code for the operation B_{insert}. We shall make use of the fact that $B_{store} = B_{listitem}$.

The "abstract model" A that was just described is the following algebra.

$$A_{store} = \mathbf{F}(A_{item}) = \mathbf{F}(C_{item})$$

$$A_{empty}() = \emptyset$$

$$A_{insert}(x, s) = \{x\} \cup s$$

$$A_{isempty}(s) = \begin{cases} \mathbf{T}, & \text{if } s = \emptyset \\ \mathbf{F}, & \text{if } s \neq \emptyset \end{cases}$$

$$\operatorname{dom} A_{min} = \operatorname{dom} A_{removemin} = A_{store} \setminus \{\emptyset\}$$

for all $s \in A_{store} \setminus \{\emptyset\}$:

$$A_{min}(s) = \min s$$

$$A_{removemin}(s) = s \setminus \{\min s\}$$

The interpretations in A of the other symbols (*bool*, *item*, *leitem*, *listitem*, *leitem*, *nil*, *cons*, *isnil*, *hd*, and *tl*) are the same as in the given algebra C.

Note that all sorts and operations of the given algebra C are the same in the algebras A, B and C. To simplify the notation, we shall therefore write just "*item*" instead of "A_{item}", "B_{item}" or "C_{item}," and analogously for the other components of C. As in Section 1.4, we let $\leq$ and $<$ be the total ordering and strict total ordering on *item* that are defined by the operation *leitem*. Also from Section 1.4, recall the predicate "Ascending" for lists in $item^*$, which asserts that a list is strictly monotonic according to $<$:

$$\text{Ascending}(l) \iff \text{whenever } 1 \leq i < j \leq |l|\text{, then } l_i < l_j.$$

Trivial consequences of Ascending(l) are that the elements of l are different from each other, and that $l_1 = \min(\operatorname{ran} l)$.

We now present a V-abstraction function h: $B \overset{V}{\twoheadrightarrow} A$, where $V = \{bool, item, listitem\}$:

h_{bool}, h_{item}, and $h_{listitem}$ are the identity maps,

$$h_{store}\colon B_{store} \nrightarrow A_{store} \ (\text{i.e., } h_{store}\colon item^* \nrightarrow \mathbf{F}(item)),$$
$$\operatorname{dom} h_{store} = \{\, l \in B_{store} \mid \text{Ascending}(l) \,\},$$
$$\text{for all } l \in \operatorname{dom} h_{store}\colon \ h_{store}(l) = \operatorname{ran} l.$$

We show that h is a strong partial homomorphism using Proposition 4.5.2. For this, the operations will be considered in turn.

leitem, nil, cons, isnil, hd, tl: These operations trivially satisfy the criterion of Proposition 4.5.2, since they are the same in A, B and C, and since h is the identity on the sorts of C.

empty: We have $\langle\rangle \in \operatorname{dom} B_{empty}$ and $\langle\rangle \in \operatorname{dom} A_{empty}$, and $h_{store}(B_{empty}()) = h_{store}(\langle\rangle) = \emptyset = A_{empty}() = A_{empty}(h_{store}(\langle\rangle))$.

insert: We shall prove below that for all $x \in item$ and $l \in item^*$ such that Ascending(l), we have

$$\begin{aligned} &\langle x, l\rangle \in \operatorname{dom} B_{insert}, \\ &\text{Ascending}(B_{insert}(x,l)), \text{ and} \qquad (*) \\ &\operatorname{ran} B_{insert}(x,l) = \{x\} \cup \operatorname{ran} l. \end{aligned}$$

From this it easily follows that *insert* is strongly compatible with h: If $x \in \operatorname{dom} h_{item}$ and $l \in \operatorname{dom} h_{store}$, that is, if $x \in item$ and $l \in item^*$ such that Ascending(l), then $\langle x, l\rangle \in \operatorname{dom} B_{insert}$ by $(*)$, and since A_{insert} is total, we have $\langle h_{item}(x), h_{store}(l)\rangle \in \operatorname{dom} A_{insert}$. Finally, using $(*)$ we have

$$\begin{aligned} h_{store}(B_{insert}(x,l)) &= \operatorname{ran} B_{insert}(x,l) \\ &= \{x\} \cup \operatorname{ran} l \\ &= A_{insert}(x, \operatorname{ran} l) \\ &= A_{insert}(h_{item}(x), h_{store}(l)). \end{aligned}$$

It remains to prove $(*)$. This we do by induction on the length of l. To be precise, we prove by induction on $n \in N$ that

$$\text{if} \quad x \in item, \ l \in item^* \text{ such that Ascending}(l) \text{ and } |l| = n,$$

then $\langle x, l\rangle \in \operatorname{dom} B_{insert}$,

$\text{Ascending}(B_{insert}(x,l))$, and

$\operatorname{ran} B_{insert}(x,l) = \{x\} \cup \operatorname{ran} l$.

For the base of the induction, assume that $n = 0$, and suppose that $x \in item$, $l \in item^*$ such that $\text{Ascending}(l)$ and $|l| = n$. Then $l = \langle\rangle$, $B_{isnil}(l) = \mathbf{T}$, and hence

$$\begin{aligned} B_{insert}(x,l) &= cons(x, nil()) \\ &= cons(x, \langle\rangle) \\ &= \langle x\rangle. \end{aligned}$$

$\text{Ascending}(\langle x\rangle)$ is vacuously true, and

$$\operatorname{ran}(\langle x\rangle) = \{x\} = \{x\} \cup \emptyset = \{x\} \cup \operatorname{ran} l.$$

For the inductive step, assume that $n > 0$, and suppose that $x \in item$, $l \in item^*$ such that $\text{Ascending}(l)$ and $|l| = n$. Then $isnil(l) = \mathbf{F}$. There remain three cases in the code of B_{insert}, which we now investigate in turn.

First, assume that $x = hd\, l$. This means that $leitem(x, hd\, l) = \mathbf{T}$ and $leitem(hd\, l, x) = \mathbf{T}$, so that $B_{insert}(x,l) = l$. We have $\text{Ascending}(B_{insert}(x,l))$, because $\text{Ascending}(l)$ is true by assumption. Since $x = hd\, l = l_1 \in \operatorname{ran} l$, we have

$$\operatorname{ran} B_{insert}(x,l) = \operatorname{ran} l = \{x\} \cup \operatorname{ran} l.$$

Second, assume that $x < hd\, l$. This means that $leitem(x, hd\, l) = \mathbf{T}$ and $leitem(hd\, l, x) = \mathbf{F}$, so that $B_{insert}(x,l) = cons(x,l)$. Write $l' := cons(x,l) = \langle x, l_1, \ldots, l_n\rangle$. We have $\text{Ascending}(l')$, because whenever $1 \le i < j \le |l|$, then

- if $i = 1$, then $l'_i = l'_1 = x < hd\, l = l_1 \le l_{j-1} = l'_j$;
- if $i > 1$, then $l'_i = l_{i-1} < l_{j-1} = l'_j$,

so that in each case $l'_i < l'_j$. Furthermore,

$$\begin{aligned} \operatorname{ran} B_{insert}(x,l) &= \operatorname{ran}\langle x, l_1, \ldots, l_n\rangle \\ &= \{x, l_1, \ldots, l_n\} \\ &= \{x\} \cup \{l_1, \ldots, l_n\} \\ &= \{x\} \cup \operatorname{ran} l. \end{aligned}$$

Third, assume that $x > hd\,l$. This means that $leitem(x, hd\,l) = \mathbf{F}$, so that $B_{insert}(x,l)$ is defined by the expression "$cons(hd\,l, insert(x, tl\,l))$". By the inductive hypothesis, $\langle x, tl\,l\rangle \in \operatorname{dom} B_{insert}$, and so

$$B_{insert}(x,l) = cons(hd\,l, B_{insert}(x, tl\,l)).$$

Write $l' = \langle l'_1, \ldots, l'_n\rangle := B_{insert}(x, tl\,l)$, so that, by the inductive hypothesis, Ascending(l') and $\operatorname{ran} l' = \{x\} \cup \operatorname{ran}(tl\,l)$. Let $l'' := B_{insert}(x,l) = cons(hd\,l, l') = \langle l_1, l'_1, \ldots, l'_n\rangle$. To show Ascending($l''$), consider $1 \le i < j \le |l''|$. Note that $l''_j \in \operatorname{ran} l' = \{x\} \cup \operatorname{ran}(tl\,l)$, so that the following three cases are exhaustive.

- If $i > 1$, then $l''_i = l'_{i-1} < l'_{j-1} = l''_j$;
- if $i = 1$ and $l''_j = x$, then $l''_i = l''_1 = l_1 = hd\,l < x = l''_j$;
- if $i = 1$ and $l''_j \in \operatorname{ran}(tl\,l) = \{l_2, \ldots, l_n\}$, then $l''_i = l''_1 = l_1 < l''_j$, because Ascending($l$) by assumption.

Furthermore,

$$\begin{aligned}
\operatorname{ran} B_{insert}(x,l) &= \operatorname{ran} l'' \\
&= \{l_1\} \cup \operatorname{ran} l' \\
&= \{l_1\} \cup \{x\} \cup \operatorname{ran}(tl\,l) \qquad \text{(Inductive Hypothesis)} \\
&= \{x\} \cup \operatorname{ran} l.
\end{aligned}$$

This concludes the induction step, hence (*) is proved, and thus *insert* is strongly compatible with h.

isempty: Let $l \in \operatorname{dom} h_{store}$. Then $l \in \operatorname{dom} B_{isempty}$ and $h_{store}(l) \in \operatorname{dom} A_{isempty}$, since both $B_{isempty}$ and $A_{isempty}$ are total. Furthermore,

$$\begin{aligned}
h_{bool}(B_{isempty}(l)) &= B_{isempty}(l) \\
&= \begin{cases} \mathbf{T}, & \text{if } l = \langle\rangle \\ \mathbf{F}, & \text{if } l \neq \langle\rangle \end{cases} \\
&= \begin{cases} \mathbf{T}, & \text{if } \operatorname{ran} l = \emptyset \\ \mathbf{F}, & \text{if } \operatorname{ran} l \neq \emptyset \end{cases}
\end{aligned}$$

$$= \begin{cases} \mathbf{T}, & \text{if } h_{store}(l) = \emptyset \\ \mathbf{F}, & \text{if } h_{store}(l) \neq \emptyset \end{cases}$$

$$= A_{isempty}(h_{store}(l)).$$

min: Let $l \in \operatorname{dom} h_{store}$, *i. e.*, $l \in item^*$ and Ascending(l). Then

$$\begin{aligned} l \in \operatorname{dom} B_{min} &\iff l \neq \langle\rangle \\ &\iff \operatorname{ran} l \neq \emptyset \\ &\iff h_{store}(l) \neq \emptyset \\ &\iff h_{store}(l) \in \operatorname{dom} A_{min}, \end{aligned}$$

and if $l \in \operatorname{dom} B_{min}$, *i. e.*, $l \neq \langle\rangle$, then $B_{min}(l) = hd(l) = l_1$. Since Ascending(l) by assumption, we have

$$\begin{aligned} B_{min}(l) &= l_1 \\ &= \min\{l_1, \ldots, l_n\} \\ &= \min(\operatorname{ran} l) \\ &= \min(h_{store}(l)) \\ &= A_{min}(h_{store}(l)). \end{aligned}$$

removemin: Let $l \in \operatorname{dom} h_{store}$, *i. e.*, $l \in item^*$ and Ascending(l). As before for *min*, one shows

$$l \in \operatorname{dom} B_{removemin} \iff h_{store}(l) \in \operatorname{dom} A_{removemin}.$$

Now suppose $l \in \operatorname{dom} B_{removemin}$, *i. e.*, $l \neq \langle\rangle$. Then

$$\begin{aligned} h_{store}(B_{removemin}(l)) &= h_{store}(tl\, l) \\ &= h_{store}\langle l_2, \ldots, l_n\rangle \\ &= \operatorname{ran}\langle l_2, \ldots, l_n\rangle \\ &= \{l_2, \ldots, l_n\} \\ &= \{l_1, \ldots, l_n\} \setminus \{l_1\} \qquad \text{(Ascending}(l)\text{)} \end{aligned}$$

$$= \operatorname{ran} l \setminus \{\min(\operatorname{ran} l)\} \qquad (\text{Ascending}(l))$$
$$= A_{removemin}(\operatorname{ran} l)$$
$$= A_{removemin}(h_{store}(l)).$$

We have shown that all the operations are strongly compatible with h, and hence that h is a strong partial homomorphism. It is obvious that for $v \in V$, $A_v = B_v = C_v$ and h_v is the identity map from B_v to A_v. Hence $h\colon B \overset{\Longrightarrow}{\underset{V}{\rightrightarrows}} A$.

To show that h is surjective, it remains to show that h_{store} is surjective. For this, we show by induction on $n \in N$, that

> if $s \in A_{store} = \mathbf{F}(item)$ is such that $\operatorname{card}(s) = n$,
> then there exists $l \in \operatorname{dom} h_{store}$ such that $h_{store}(l) = s$.

If $n = 0$, then $s = \emptyset$, and we can choose $l = \langle\rangle$. If $n > 0$, let $x := \min s$ and $s' := s \setminus \min s$. By the inductive hypothesis, we can choose $l' \in \operatorname{dom} h_{store}$ such that $h_{store}(l') = s'$. In particular, Ascending(l'). Write $l' = \langle l'_1, \ldots, l'_k \rangle$, and define $l := \langle x, l'_1, \ldots, l'_k \rangle$. We have Ascending($l$), because whenever $1 \le i < j \le |l|$, then

- if $i = 1$, then $l_j = l'_{j-1} \in \operatorname{ran} l' = h_{store}(l') = s'$, and by the definition of x and s', this implies that $l_i = l_1 = x < l_j$,
- if $i > 1$, then $l_i = l'_{i-1} < l'_{j-1} = l_j$, because Ascending($l'$).

Hence Ascending(l) and so $l \in \operatorname{dom} h_{store}$. Since

$$h_{store}(l) = \operatorname{ran} l = \operatorname{ran}\langle x, l'_1, \ldots, l'_k \rangle$$
$$= \{x\} \cup \operatorname{ran} l' = \{x\} \cup s' = s,$$

the inductive step is complete.

It has thus been shown that h is surjective, and thus that $h\colon B \overset{\Longrightarrow}{\underset{V}{\twoheadrightarrow}} A$ is a strong surjective partial homomorphism, *i. e.*, an abstraction function. It follows that B is a standard V-representation of A. □

The reader will recognize that the proof performed in this example is a data representation correctness proof in the traditional manner based on Hoare's paper [Hoare 72]: The "representation invariant" is the predicate "Ascending(l)", since it defines the domain of h_{store}, and h_{store} itself is the "abstraction function". Thus, the example supplies the correctness proof of the module M_{STORE} that was only described informally in Section 1.4.

Note that this proof is not sufficient to prove that M_{STORE} is a universal implementation of its specification $\mathcal{M}_{STORE}$ with respect to standard representation. In such a proof, one could not assume, for example, that the basic types and operations of the algebra C have the properties ascribed to them in $I_{ITEM} \wedge I_{LISTITEM}$; rather, one would have to deal with an arbitrary representation of such an algebra.

Nevertheless, Chapter 5 will explain that the proof just performed may be regarded as "*the* correctness proof of M_{STORE}", because if M_{STORE} is coded in a suitable "data abstraction" programming language, the proof will allow one to infer that M_{STORE} is a universal implementation of $\mathcal{M}_{STORE}$ with respect to behavioural inclusion, behavioural equivalence, and standard representation (cf. Example 5.4.3).

The following theorem shows that standard representation is a more restrictive representation relation than behavioural equivalence.

4.5.4 Theorem. *There exist an algebraic signature Σ and Σ-algebras A and B such that $A \simeq_\emptyset B$, but no partial homomorphism from A to B nor from B to A exists. In particular, neither of the two algebras is a standard representation of the other.*

Proof. Choose Σ, A, and B according to Theorem 4.3.11. Since $A \lesssim_\emptyset B$ and $B \lesssim_\emptyset A$, we have $A \simeq_\emptyset B$, but no partial homomorphism from A to B nor from B to A exists. Since standard representation requires the existence of a surjective strong partial homomorphism between the algebras, neither of the algebras is a standard representation of the other. □

A practical example illustrating the difference between behavioural equivalence and standard representation is obtained from Example 4.4.7.

4.5.5 Example. Let Σ be the signature of Example 4.3.12:

signature
bool, *char*, *string*: sort
a, b, c: $\rightarrow$ *char*
single: *char* $\rightarrow$ *string*
occurs: *char string* $\rightarrow$ *bool*
join: *string string* $\rightarrow$ *string*

This is just the signature of Example 4.4.7, with a, b, c: $\rightarrow$ *char* added. Let $S := \{bool, char, string\}$ be the sort set of this signature. Let C, which will be the set of *char* values, be a set containing at least three different elements p, q, and r. Let the algebras A and B be defined by:

$$
\begin{aligned}
A_{bool} &= B_{bool} = \{\mathbf{T}, \mathbf{F}\} \\
A_{char} &= B_{char} = C \\
A_{string} &= B_{string} = C^* \\
A_a() &= B_a() = p \\
A_b() &= B_b() = q \\
A_c() &= B_c() = r \\
A_{single}(x) &= B_{single}(x) = \langle x \rangle \\
A_{occurs}(x, s) = B_{occurs}(x, s) &= \begin{cases} \mathbf{T}, & \text{if } x \in \operatorname{ran} s \\ \mathbf{F}, & \text{if } x \notin \operatorname{ran} s \end{cases} \\
A_{join}(s, t) &= s \circ t
\end{aligned}
$$

$B_{join}(s, t)$ is defined by the recursive program

$$
\begin{aligned}
join(s, t) = \text{if } & \quad length(s) = 0 \text{ then } t \\
& \text{else if } occurs(hd\ s, t) \text{ then } join(tl\ s, t) \\
& \text{else } join(tl\ s, cons(hd\ s, t)).
\end{aligned}
$$

These are just the algebras A and B of Example 4.4.7 (letting C be the same set there as here), with interpretations of the symbols a, b, and c added. Recall that the algebra A can be seen as the ("abstract model") specification of the

data type *string* with the access operations *single*, *occurs* and *join*, and that the algebra B can be seen as a representation of the data type, written in a concrete programming notation that provides a "sequence" data type constructor (code for B_{single} and B_{occurs} was given in Example 4.3.10).

The algebra B is behaviourally equivalent to A over $V := \{bool, char\}$. To prove this, let G be the S-sorted relation defined by

$$G_{bool} \text{ is the identity relation on } \{\mathbf{T}, \mathbf{F}\},$$
$$G_{char} \text{ is the identity relation on } C,$$
$$G_{string} := \{\, (s', s) \in C^* \times C^* \mid \operatorname{ran} s' = \operatorname{ran} s \,\}.$$

This is just the relation G of Example 4.4.7. In that example, G was proved to be a strong V-correspondence: it was proved that the operations *single*, *join*, and *occurs* are compatible with both G and $G^{\cup}$. Obviously, the new operations a, b, and c also are compatible with G and $G^{\cup}$. Thus, $G\colon B \underset{V}{\Longrightarrow\!\!\!\!\times} A$ is a strong V-correspondence, and hence $B \simeq_V A$.

To see that no partial homomorphism exists from B to A nor *vice versa* (even without considering visible sorts, which would only restrict the class of homomorphisms), let

$$\phi\colon \mathrm{T}_\Sigma(A/\emptyset) \twoheadrightarrow A/S \quad \text{and} \quad \psi\colon \mathrm{T}_\Sigma(B/\emptyset) \twoheadrightarrow B/S$$

be the evaluation functions, and consider the following three terms and their evaluations in B and A:

term t	$\psi_{string}(t)$	$\phi_{string}(t)$
$\langle join, single, a, join, single, b, single, c\rangle$ (*i. e.*, $join(\langle p\rangle, join(\langle q\rangle, \langle r\rangle))$)	$\langle p, q, r\rangle$	$\langle p, q, r\rangle$
$\langle join, join, single, b, single, a, single, c\rangle$ (*i. e.*, $join(join(\langle q\rangle, \langle p\rangle), \langle r\rangle)$)	$\langle p, q, r\rangle$	$\langle q, p, r\rangle$
$\langle join, join, single, a, single, b, single, c\rangle$ (*i. e.*, $join(join(\langle p\rangle, \langle q\rangle), \langle r\rangle)$).	$\langle q, p, r\rangle$	$\langle p, q, r\rangle$

If we write $u := \langle p, q, r\rangle$ and $v := \langle q, p, r\rangle$, we thus have

$$\{(u, u), (u, v), (v, u)\} \subseteq \psi^{\cup}_{string} \,;\, \phi_{string}.$$

Now if H: $B \longrightarrow\!\!\!\!\times\ A$ is a correspondence, by Lemma 4.3.9, applied with $V = \emptyset$, it follows that

$$H_{string} \supseteq \psi^{\cup}_{string} \,;\, \phi_{string} \supseteq \{(u,u),\,(u,v),\,(v,u)\},$$

and thus H_{string} cannot be a partial function, hence H cannot be a partial homomorphism.

Conversely, if H: $A \longrightarrow\!\!\!\!\times\ B$ is a correspondence, Lemma 4.3.9 yields that

$$\begin{aligned} H_{string} &\supseteq \phi^{\cup}_{string} \,;\, \psi_{string} \\ &= (\psi^{\cup}_{string} \,;\, \phi_{string})^{\cup} \\ &\supseteq \{(u,u),\,(u,v),\,(v,u)\}^{\cup} \\ &= \{(u,u),\,(v,u),\,(u,v)\}, \end{aligned}$$

and thus H_{string} cannot be a partial function, hence H cannot be a partial homomorphism. □

The example shows an "abstract model" data type specification A and a representation B of it that is correct with respect to behavioural equivalence, but that is not correct according to standard representation.

The problem that a specification may have behaviourally correct representations that cannot be proved correct using the standard representation criterion has been ascribed to "extraneous detail" [LZ 75, p. 11] [Guttag 77, p. 398] or an "implementation bias" [Jones 80, Ch. 15] in the specification. An abstract model specification of an encapsulated type is biased, if some values of the encapsulated type cannot occur in computations, or if there are distinct values that produce the same visible results in all computations. Conversely, a specification is unbiased, if all values of the encapsulated type can be generated by means of the access operations, and if—the "bias test" of [Jones 80, Ch. 15]—the operations allow one to distinguish the values of the encapsulated type.

In the example, the algebra A is a biased specification of the *string* data type, because the only means of distinguishing *string* values is by means of the *occurs* operation, which produces identical results for sequences containing the same set of *char* values. Thus, a *string* in the algebra A essentially represents a

set of *char* values, and it is possible to give an unbiased specification of the data type as follows:

$$
\begin{aligned}
\mathit{string} &= \mathbf{F}(C) \\
\mathit{single}(x) &= \{x\} \\
\mathit{occurs}(x,s) &= \begin{cases} \mathbf{T}, & \text{if } x \in S \\ \mathbf{F}, & \text{if } x \notin S \end{cases} \\
\mathit{join}(s,t) &= s \cup t
\end{aligned}
$$

Jones suggests that specifications might be subjected to his bias test and rewritten if bias is found [Jones 80, p. 264]; for example, he would suggest to replace A by the specification just given. This may be costly, however, if the specification has already been used in proofs, *e. g.*, the correctness proofs of modules using the encapsulated type. For other authors, the problem of "extraneous detail" or "bias" in specifications is an important argument against abstract model specification techniques [Guttag 77, p 398] [Parnas 79, p. 367] [Denert 79, p. 206].

However, biased specifications restrict the class of possible representations of an encapsulated data type only when the standard representation concept is used. With behavioural equivalence as the correctness concept and the proof method based on strong correspondences, biased specifications do not cause this problem, because only the observable behaviour of the specification and the representation are relevant.

While thus the "classical" problem associated with biased specifications is eliminated by the behavioural representation concept, this does not mean that biased specifications are completely harmless. In Section 1.4, it was shown that the process of access operation refinement may be guided towards an inefficient program if based on a biased specification. Thus, the criteria to detect bias and the methods to avoid it remain useful.

In order to apply the theory of Section 4.1 to standard representation, we verify that standard representation is a representation relation in the institution $\langle$**TSig, TIncl, TAlg**$\rangle$.

4.5.6 Theorem. *Standard representation is a representation relation in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$.

To be precise, we obtain a representation relation $\rightsquigarrow = \langle \underset{\langle \Sigma,V\rangle}{\rightsquigarrow} \rangle_{\langle \Sigma,V\rangle \in |\mathbf{TSig}|}$ *by defining*

$$A \underset{\langle \Sigma,V\rangle}{\rightsquigarrow} B \ :\Longleftrightarrow\ A \backsimeq_V B \qquad \text{for } A,B \in \mathbf{Alg}(\Sigma) = \mathbf{TAlg}\langle \Sigma,V\rangle.$$

Standard representation is not chain-closed.

In the proof of the theorem, the following proposition about strong correspondences will be used.

4.5.7 Proposition. *If* $\sigma\colon \langle \Sigma,V\rangle \to \langle \Sigma',V'\rangle$ *is a tagged signature morphism where* $\Sigma = \langle S,\alpha\rangle$, *and* $G\colon A \underset{V'}{\rightrightarrows\!\!\times} B$ *is a strong* V'*-correspondence (strong partial* V'*-homomorphism, strong* V'*-homomorphism,* V'*-abstraction function) between* Σ'*-algebras* A *and* B, *then the correspondence* $\overline{\sigma}\,G\colon \overline{\sigma}\,A \underset{V}{\longrightarrow\!\!\times} \overline{\sigma}\,B$ *defined by* $(\overline{\sigma}\,G)_s = G_{\sigma s}$ *for* $s \in S$ *(see Proposition 4.3.14) is a strong* V*-correspondence (strong partial* V*-homomorphism, strong* V*-homomorphism,* V*-abstraction function) between* Σ*-algebras.*

Proof. If $G\colon A \underset{V'}{\rightrightarrows\!\!\times} B$, then by Proposition 4.3.14, we have $\overline{\sigma}\,G\colon \overline{\sigma}\,A \underset{V}{\longrightarrow\!\!\times} \overline{\sigma}\,B$ and $\overline{\sigma}\,G^{\cup}\colon \overline{\sigma}\,B \underset{V}{\longrightarrow\!\!\times} \overline{\sigma}\,A$. Since $(\overline{\sigma}(G^{\cup}))_s = (G^{\cup})_{\sigma s} = (G_{\sigma s})^{\cup} = ((\overline{\sigma}\,G)_s)^{\cup}$, the correspondence $\overline{\sigma}(G^{\cup})$ is the componentwise converse of $\overline{\sigma}\,G$, and hence $\overline{\sigma}\,G\colon \overline{\sigma}\,A \underset{V}{\rightrightarrows\!\!\times} \overline{\sigma}\,B$. It is easy to see that if in addition, G consists of partial functions, total functions, or surjective partial functions, then $\overline{\sigma}\,G$ does so too; hence $\overline{\sigma}$ maps strong partial homomorphisms, strong homomorphisms and abstraction functions to strong correspondences of the same type. □

Proof of Theorem 4.5.6.

We verify the axioms (a) and (b) of Definition 4.1.1.

Axiom (a): For $\langle \Sigma,V\rangle \in \mathbf{TSig}$, the relation $\backsimeq_V$ is a preordering on $\mathbf{TAlg}(\Sigma,V)$, because the V-abstraction functions between Σ-algebras form a category (Proposition 4.4.5 together with the fact that the composition of surjective functions is again surjective).

Axiom (b): If σ: $\langle \Sigma, V \rangle \to \langle \Sigma', V' \rangle$ is a tagged signature morphism and $A \underset{\langle \Sigma', V' \rangle}{\rightsquigarrow} B$, choose an abstraction function h: $A \underset{V'}{\twoheadrightarrow} B$. By Proposition 4.5.7, we have $\overline{\sigma} h$: $\overline{\sigma} A \underset{V}{\twoheadrightarrow} \overline{\sigma} B$, hence $\overline{\sigma} A \backsimeq_V \overline{\sigma} B$.

Thus, standard representation is a representation relation in the institution $\langle$**TSig, TIncl, TAlg**$\rangle$.

To see that standard representation is not chain-closed, consider the nonempty compatible chain of tagged signatures $\langle \Sigma_n, \emptyset \rangle_{n \in \mathbf{N}}$, where

$$\Sigma_n = \textbf{signature}$$
$$s\textbf{: sort}$$
$$c_0, \ldots, c_{n-1} : \to s.$$

We then have

$$\bar{\Sigma} := \bigsqcup_{n \in \mathbf{N}} \Sigma_n = \textbf{signature}$$
$$s\textbf{: sort}$$
$$c_0, c_1, c_2, \ldots : \to s,$$

so that $\bigsqcup_{n \in \mathbf{N}} \langle \Sigma_n, \emptyset \rangle = \langle \bar{\Sigma}, \emptyset \rangle$.

Consider the $\bar{\Sigma}$-algebras A' and A given by

$$A'_s = \mathbf{N} \qquad A_s = \{0, 1\}$$
$$A'_{c_i}() = i \qquad A_{c_i}() = 0 \qquad \text{for } i \in \mathbf{N}.$$

Now for every $n \in \mathbf{N}$, there exists an abstraction function h from A'/Σ_n to A/Σ_n, defined by

$$h_s(i) = \begin{cases} 0, & \text{if } i < n \\ 1, & \text{if } i \geq n \end{cases} \qquad \text{for } i \in \mathbf{N}.$$

However, there does not exist a surjective partial homomorphism (whether strong or not) from A' to A, for if h is a partial homomorphism from A' to A, then for every $i \in \mathbf{N}$, we have $(A'_{c_i}(), A_{c_i}()) = (i, 0) \in h_s$ by the correspondence property of h, hence $h_s(i) = 0$ for all $i \in \mathbf{N}$, and so h_s cannot be surjective. □

Theorem 4.5.6 now makes the theory of Section 4.1 applicable to standard representation; we obtain a universal implementation concept and a composability theorem. In contrast to the representation relations equality, behavioural inclusion, and behavioural equivalence, standard representation is not chain-closed, and so the composability theorem applies only to finite systems.

I conjecture that the composability theorem still holds for infinite systems, for the following reason: Since abstraction functions are unique in suitably extended algebras (Lemma 5.2.5), a universal implementation must "extend" them in a way similar to Definition 5.1.14. Since nonempty "chains" of abstraction functions (defined in a manner to become clear in the next chapter) do have least upper bounds, it should be possible to modify the proof of the composability theorem by replacing joint approximations $\langle K', X', K, X\rangle$ by quintuples $\langle K', X', K, X, h\rangle$, where h is an abstraction function from $X'/\operatorname{Sig}(X)$ to X.

Chapter 5

Stability

THE PREVIOUS CHAPTER has introduced the "universal implementation" concept and shown that it forms the basis of a sound modular programming method. The present chapter aims at making this method practical. To this end, the universal implementation concept is decomposed into two parts: "simple implementation" and "stability". The "simple implementation" concept corresponds to practical data representation correctness proofs. It implies universal implementation if the implementing cell is "stable". If a programming language ensures that all its modules are stable, proofs of the simple implementation property suffice as correctness proofs in a structured correctness argument, and hence a sound, practical method for modular programming with data abstraction is established.

In the first section below, the "simple implementation" and "stability" concepts are developed in the context of an arbitrary institution. The three remaining sections investigate the stability concepts associated with the three representation relations for partial algebras that have been introduced in the previous chapter.

5.1 Simple Implementation and Stability in an Institution

Section 4.1 has presented a sound modular programming method based on the "universal implementation" relation between cells as the correctness concept. This method, however, cannot be said to be practical, or to reflect practical data abstraction.

The problem is that universal implementation is a very restrictive relation that is difficult to establish. Recall that universal implementation is the correctness relation that must hold between a cell $\langle Q', R' \rangle$ and its specification $\langle Q, R \rangle$ in a structured correctness argument. The interfaces of $\langle Q, R \rangle$ can be thought of as the import and export interfaces of $\langle Q', R' \rangle$ in a design graph, as shown in Figure 5-1. Naturally, we would like to be able to use the properties given in Q in the correctness proof of $\langle Q', R' \rangle$. However, this is not possible if we let "correctness" mean "universal implementation": in order to prove that $\langle Q', R' \rangle$ is a universal implementation of $\langle Q, R \rangle$, one has to consider the behaviour of $\langle Q', R' \rangle$ on any base A' that represents a base of $\langle Q, R \rangle$. The space of such models A' is large:

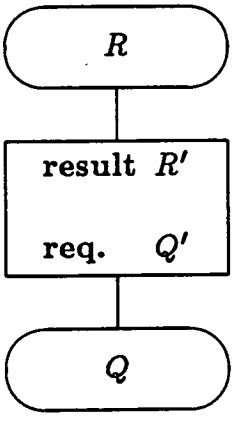


Figure 5-1: A section from a design graph

- the signature of A' is a site for $\mathrm{Sig}\langle Q, R\rangle$, and thus an almost arbitrary extension of $\mathrm{Sig}(Q)$,
- A' need only represent a model A such that $A/\mathrm{Sig}(Q) \in Q$; we need not have $A'/\mathrm{Sig}(Q) \in Q$, which means that the properties given in Q can not in general be assumed to hold in A'.

The goal of the present section is to establish a "correctness" concept in which only models of Q need to be considered as bases for $\langle Q', R'\rangle$, so that the properties given in Q may be assumed to be valid, and no program entities beyond those in Q and R need to be considered. This concept is called "simple implementation".

Of course, we do not get something for nothing, and "simple implementation" does not imply "universal implementation". However, we will call those cells "stable" that are universal implementations of a specification whenever they are simple implementations of it. As we shall see in the course of this chapter, the stability property is not unduly strong, rather it is a natural condition to impose on program modules.

As in Chapter 3 and Section 4.1, we shall work in an arbitrary, but fixed, institution.

Convention. Throughout this section, the triple $\langle Sig, Incl, Mod\rangle$ is assumed to be an institution. The concepts that depend on an institution (such as "signature", "inclusion", or "model") are implicitly assumed to refer to the institution $\langle Sig, Incl, Mod\rangle$. □

In order to be able to formulate the "simple implementation" concept, we need to modify the "representation" notion. Rather that just regarding it as a relation between models, we now consider a "representation category", whose objects are the models of a certain signature, and whose morphisms establish the representation relation between the models they connect.

5.1.1 Definition. A *representation functor* is a functor *Rep* from Sig^{op} to a category of categories that satisfies the following axioms.

(a) for each $S \in |Sig|$, the category $Rep(S)$ has object set $Mod(S)$.

$Rep(S)$ is called the *representation category* of signature S of *Rep*. If J is a morphism from A to B in $Rep(S)$, we write "$J\colon A \underset{S}{\rightsquigarrow} B$" and call J a *representation morphism* from A to B.

(b) for each $\sigma\colon S \to T$ in *Sig*, the object function of the functor $\overline{\sigma} := Rep(\sigma^{\mathrm{op}})\colon Rep(T) \to Rep(S)$ agrees with the function $Mod(\sigma^{\mathrm{op}})\colon Mod(T) \to Mod(S)$.

If $S \sqsubseteq T$, and $J\colon A \underset{T}{\rightsquigarrow} B$ is a representation morphism, let the *reduct* of J to S be the representation morphism $J/S := \overline{(S \sqsubseteq T)^{\mathrm{op}}}(J)\colon A/S \underset{S}{\rightsquigarrow} B/S$.

(c) (Pair-Completeness)
whenever S and T are compatible signatures and $J\colon A \underset{S}{\rightsquigarrow} B$ and $K\colon C \underset{T}{\rightsquigarrow} D$ are representation morphisms such that

$$J/(S \sqcap T) = K/(S \sqcap T),$$

then there exists a representation morphism $L\colon A \sqcup C \underset{S \sqcup T}{\rightsquigarrow} B \sqcup D$ such that

$$L/S = J \quad \text{and} \quad L/T = K.$$

The *representation relation* of a representation functor *Rep* is the $|Sig|$-indexed relation $\rightsquigarrow = \langle \underset{S}{\rightsquigarrow} \rangle_{S \in |Sig|}$ defined by

$$A \underset{S}{\rightsquigarrow} B \ :\Longleftrightarrow\ \exists J\colon A \underset{S}{\rightsquigarrow} B. \qquad \square$$

The axioms (a) and (b) of this definition obviously imply that the "representation relation" of a representation functor is indeed a representation relation according to Definition 4.1.1. A more concise way of stating these two axioms is to say that $Rep\ ;\ \mathbf{Obj} = Mod$, where **Obj** is the "forgetful" functor mapping categories to their object sets and functors to their object functions (this is somewhat loose, because **Obj** is not a proper mathematical object—it should be qualified to go from a certain category of categories to a certain category of sets).

Every representation relation gives rise to a functor satisfying (a) and (b) if one expresses the representation relation for each signature as a preorder category. However, this functor is not necessarily a representation functor, as it need not satisfy the pair-completeness axiom (c). This axiom is the distinguishing feature of a representation functor; as will be seen in the proofs of this section, it enables us to restrict our attention to the signatures of the cells under consideration instead of having to take arbitrary sites into account.

The representation relations between partial algebras we studied in the previous chapter all have associated representation functors—the representation morphisms are correspondences for behavioural inclusion, strong correspondences for behavioural equivalence, and abstraction functions for standard representation. These representation functors will be presented and analysed in the three sections to follow.

For the present, it will just be shown that for the two extreme representation relations, namely eqality and the total relation, the functor that maps each signature to the associated preorder category is a representation functor.

5.1.2 Proposition. *If* $\rightsquigarrow = \langle \underset{S}{\rightsquigarrow} \rangle_{S \in |Sig|}$ *is equality or the total relation, then the functor which maps* $S \in |Sig|$ *to the preorder category defined by* $\underset{S}{\rightsquigarrow}$ *and which maps a signature morphism* σ^{op} *to the unique functor with object function* $Mod(\sigma^{\mathrm{op}})$ *is a representation functor whose representation relation is* $\rightsquigarrow$.

Proof. Axioms (a) and (b) of the definition above follow immediately from the properties of a representation relation, so it remains only to show pair-completeness.

If $\rightsquigarrow$ is equality, the axiom reduces to:

> "if $A \in Mod(S)$ and $C \in Mod(T)$ are such that $A/(S \sqcap T) = C/(S \sqcap T)$, then $(A \sqcup C)/S = A$ and $(A \sqcup C)/T = C$",

which is true by definition of $\sqcup$.

If $\rightsquigarrow$ is the total relation, the axiom reduces to

"if $A, B \in Mod(S)$ and $C, D \in Mod(T)$ are such that $A/(S \sqcap T) = C/(S \sqcap T)$ and $B/(S \sqcap T) = D/(S \sqcap T)$, then $(A \sqcup C)/S = A$, $(A \sqcup C)/T = C$, $(B \sqcup D)/S = B$ and $(B \sqcup D)/T = B$",

which also is true by definition of $\sqcup$. □

Thus, we can define "*REqual*" to be the representation functor whose representation categories are the preorder categories associated with equality (commonly called "discrete categories").

5.1.3 Definition. Let *REqual* be the functor on Sig^{op} which maps $S \in |Sig|$ to the discrete category with object set $Mod(S)$, and σ^{op} (where σ: $S \to T$ in *Sig*) to the functor from $REqual(T)$ to $REqual(S)$ whose object function is $Mod(\sigma^{\mathrm{op}})$. □

Convention. For the remainder of this section, it will be assumed that *Rep* is a representation functor in $\langle Sig, Incl, Mod \rangle$, and that $\rightsquigarrow$ is its representation relation. Concepts that depend on a representation functor or representation relation, such as "universal implementation" or "simple implementation" (to be defined later), implicitly refer to *Rep* and $\rightsquigarrow$. □

The pair-completeness property of a representation functor allows us to derive a sufficient condition for "universal implementation" in which one need not consider models whose signature is an arbitrary site for the specification cell signature $\langle E, D \rangle$, but only models of signature E. In other words, the possible presence of arbitrary additional program entities beyond those of E need no longer be taken into account explicitly.

5.1.4 Theorem. *Let $\langle Q, R \rangle$ be a cell of signature $\langle E, D \rangle$. A cell $\langle Q', R' \rangle$ of signature $\langle E', D' \rangle$ is a universal implementation of $\langle Q, R \rangle$, if $\langle E', D' \rangle$ is a syntactic refinement of $\langle E, D \rangle$ and whenever J: $A' \underset{E}{\rightsquigarrow} A$ is a representation morphism such that $A \in Q$, then*

$$A' \text{ is a base for } \langle Q', R' \rangle,$$

there exists a result of $\langle Q', R' \rangle$ *on* A', *and*

whenever B' *is a result of* $\langle Q', R' \rangle$ *on* A',

then *there exists a representation morphism*

$$K\colon B' \underset{E \sqcup D}{\rightsquigarrow} B$$

such that B *is a result of* $\langle Q, R \rangle$ *on* A

and $K/E = J$.

Proof. Let $\langle Q, R \rangle$ be a cell of signature $\langle E, D \rangle$ and $\langle Q', R' \rangle$ be a cell of signature $\langle E', D' \rangle$ such that the criterion of the theorem is satisfied. We prove that $\langle Q', R' \rangle$ is a universal implementation of $\langle Q, R \rangle$.

Let F be a site for $\langle E, D \rangle$, let $A \in Mod(F)$ be a base for $\langle Q, R \rangle$, and let $A' \underset{F}{\rightsquigarrow} A$. We can pick a representation morphism $J\colon A' \underset{F}{\rightsquigarrow} A$. Since $A/E \in Q$ and $J/E\colon A'/E \underset{E}{\rightsquigarrow} A/E$, the criterion of the theorem applies to J/E.

The model $A' \in Mod(F)$ is a base for $\langle Q', R' \rangle$, because F is a site for $\langle E', D' \rangle$ ($\langle E', D' \rangle$ is a syntactic refinement of $\langle E, D \rangle$), and because A'/E is a base for $\langle Q', R' \rangle$ by the criterion of the theorem and hence

$$A'/E' = (A'/E)/E' \in Q'.$$

Also by the criterion, there exists a result C of $\langle Q', R' \rangle$ on A'/E. The models A' and C can be joined, because the meet of their signatures is

$$F \sqcap (E \sqcup D') = F \sqcap (E \sqcup D) = E,$$

and $C/E = A'/E$. The model $A' \sqcup C$ of signature $F \sqcup D' = F \sqcup D$ is a result of $\langle Q', R' \rangle$ on A', because $(A' \sqcup C)/F = A'$ and $(A' \sqcup C)/D = C/D \in R'$. Thus, there exists a result of $\langle Q', R' \rangle$ on A'.

Finally, let B' be a result of $\langle Q', R' \rangle$ on A'. Then $B'/(E \sqcup D')$ is a result of $\langle Q', R' \rangle$ on A'/E, because $(B'/(E \sqcup D'))/E = B'/E = (B'/F)/E = A'/E$, and $(B'/(E \sqcup D'))/D' = B'/D' \in R'$. By the criterion of the theorem, there exists a representation morphism $L\colon B'/(E \sqcup D') \underset{E \sqcup D}{\rightsquigarrow} C$ such that C is a result of $\langle Q, R \rangle$ on A/E, and $L/E = J/E$. The intersection of the signatures of L and J is $(E \sqcup D') \sqcap F = (E \sqcup D) \sqcap F = E$, and hence the pair-completeness axiom can be applied to J and L. By the axiom, we can pick a representation morphism

$$K\colon A' \sqcup (B'/(E \sqcup D')) \underset{F \sqcup D'}{\rightsquigarrow} A \sqcup C.$$

The model on the left is B', which follows from the uniqueness of joins and the facts that B' is of signature $F \sqcup D'$ and $B'/F = A'$. The model $A \sqcup C$ is a result of $\langle Q, R \rangle$ on A, because $(A \sqcup C)/F = A$ and $(A \sqcup C)/D = C/D \in R$. Thus, B represents a result of $\langle Q, R \rangle$ on A, which completes the proof that $\langle Q', R' \rangle$ is a universal implementation of $\langle Q, R \rangle$. □

The sufficient condition for "universal implementation" provided by this theorem is quite similar to a theorem by Mitchell dealing with "observational equivalence" of data type representations in the second-order typed λ-calculus [Mitchell 86, Theorem 6] (in fact, I only wrote down the present theorem explicitly after seeing Mitchell's theorem). Mitchell deals with "logical relations" that play a rôle similar to representation morphisms. Mitchell's theorem states that when a given logical relation $\mathcal{R}$ relating the environments of two data type representations can be extended to a relation $\mathcal{R}^+$ relating the results of the two representations as well, then any term of the second-order language SOL [MP 84] will produce $\mathcal{R}$-related results over the two representations. If, for example, $\mathcal{R}$ is the identity relation on "program types" that are considered observable, this means that every program will have the same observable results over the two representations.

It appears, however, that Mitchell's theorem is not strong enough to deal with several data type representation in a program, because the criterion only deals with a fixed logical relation $\mathcal{R}$. When a data type representation imports entities from another, "lower level", representation, it must be capable of extending the logical relation that is the result of the extension process of the lower level representation, and this relation will in general be different for different representations of the lower level data type. Hence it appears that to make Mitchell's theorem applicable to multiple data type representation within a program, a universal quantification over logical relations is necessary rather like in Theorem 5.1.4 above.

A second criticism, which applies to Mitchell's theorem as well as to Theorem 5.1.4 above, is that the criterion is still too strong to require programmers to prove it, because they would have to consider the behaviour of the implementation cell $\langle Q', R' \rangle$ on bases that only represent members of the abstract require-

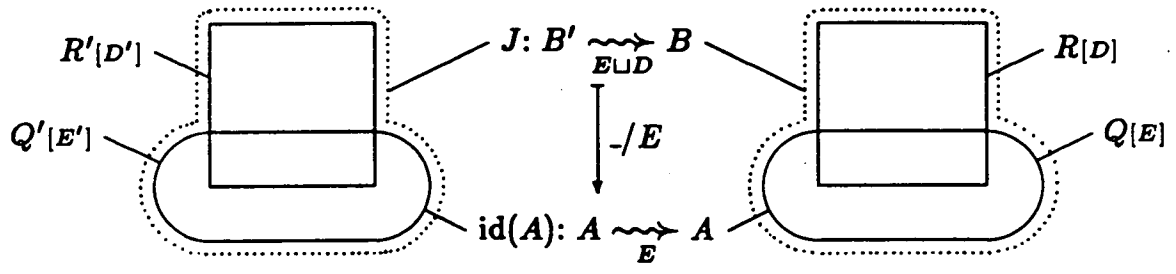


Figure 5-2: Simple Implementation

ment interface Q without satisfying this interface themselves. I criticized that already in the development method of Sannella and Tarlecki (cf. page 10–15), because it means that the designer of a module can not take the abstract import interface Q for granted when designing and verifying the implementation, and thus an essential aspect of data abstraction is missing.

In order to solve this problem, a correctness criterion called "simple implementation" will now be introduced. This criterion is to be proved by the designer of an implementation; it reflects the way data type representations are proved correct in practice, and it allows the designer to use the abstract import interface in the proof, in accordance with the view of data abstraction explained in Section 1.1.

5.1.5 Definition (Simple Implementation).
Let $\langle Q, R\rangle$ be a cell of signature $\langle E, D\rangle$. A *simple implementation* of $\langle Q, R\rangle$ (with respect to *Rep*) is a cell $\langle Q', R'\rangle$ of signature $\langle E', D'\rangle$ such that $\langle E', D'\rangle$ is a syntactic refinement of $\langle E, D\rangle$ and whenever $A \in Q$, then

A is a base for $\langle Q', R'\rangle$, and
whenever B' is a result of $\langle Q', R'\rangle$ on A
then there exists a representation morphism
$J: B' \underset{E \sqcup D}{\rightsquigarrow} B$
such that B is a result of $\langle Q, R\rangle$ on A
and $J/E = \mathrm{id}(A): A \underset{E}{\rightsquigarrow} A$ □

This definition is illustrated in Figure 5-2.

Comparing this figure with the one illustrating "universal implementation" (Figure 4-1), we find that the simple implementation concept is simpler in three ways.

- the bases under consideration are of signature E rather than an arbitrary site F for $\langle E, D\rangle$;
- the behaviour of $\langle Q', R'\rangle$ is investigated on A itself, rather than on an arbitrary representation A' of A. This means that the properties stated in Q may be taken for granted when $\langle Q', R'\rangle$ is designed or verified;
- there is no need to verify explicitly that $\langle Q', R'\rangle$ has a result on A; it is sufficient to verify that A is a base for $\langle Q', R'\rangle$ and then to establish a representation morphism for an arbitrary result B'.

However, "simple implementation" also imposes an additional requirement compared to "universal implementation":

- given a result B' of $\langle Q', R'\rangle$ on A, a representation morphism to some result B of $\langle Q, R\rangle$ must be constructed that reduces to the identity, whereas "universal implementation" required only that B' was a representation of B (*i. e.*, any representation morphism from B' to B would suffice).

5.1.6 Proposition. *The simple implementation relation is a preordering on the set of cells.*

Proof. To show transitivity, suppose that $\langle Q, R\rangle$ is an $\langle E, D\rangle$-cell, $\langle Q', R'\rangle$ is an $\langle E', D'\rangle$-cell, and $\langle Q'', R''\rangle$ is an $\langle E'', D''\rangle$-cell such that $\langle Q'', R''\rangle$ is a simple implementation of $\langle Q', R'\rangle$ and $\langle Q', R'\rangle$ is a simple implementation of $\langle Q, R\rangle$. By Proposition 3.1.17 (a), $\langle E'', D''\rangle$ is a syntactic refinement of $\langle E, D\rangle$. Now suppose that $A \in Q$. The signature of A is E, and this is a site for $\langle E, D\rangle$, hence for $\langle E', D'\rangle$ and $\langle E'', D''\rangle$. By simple implementation, A is a base for $\langle Q', R'\rangle$, hence $A/E' \in Q'$. By simple implementation, A/E' is a base for $\langle Q'', R''\rangle$, hence $A/E'' = (A/E')/E'' \in Q''$. Thus, A is a base for $\langle Q'', R''\rangle$.

Now let B'' be a result of $\langle Q'', R''\rangle$ on A. Then $B''/(E' \sqcup D'')$ is a result of $\langle Q'', R''\rangle$ on A/E', and by simple implementation, there exists a representation

morphism J: $B''/(E' \sqcup D'') \underset{E' \sqcup D''}{\rightsquigarrow} B'$, where B' is a result of $\langle Q', R' \rangle$ on A/E' and $J/E' = \mathrm{id}(A/E')$.

The representation morphisms $\mathrm{id}(A)$ and J satisfy the assumptions of the pair-completeness axiom, because the intersection of their signatures is

$$\begin{aligned} E \sqcap (E' \sqcup D'') &= E \sqcap (E' \sqcup D') && \text{(Proposition 3.1.17 (c))} \\ &= E' && (E \text{ is a site for } \langle E', D' \rangle), \end{aligned}$$

and because $\mathrm{id}(A)/E' = \mathrm{id}(A/E') = J/E'$.

Hence we can pick K: $A \sqcup B''/(E' \sqcup D'') \underset{E \sqcup D''}{\rightsquigarrow} A \sqcup B'$ such that $K/E = \mathrm{id}(A)$. Note that $A \sqcup B''/(E' \sqcup D'') = B''$ by the uniqueness of joins, hence K: $B'' \underset{E \sqcup D''}{\rightsquigarrow} A \sqcup B'$. Now $A \sqcup B'$ is a result of $\langle Q', R' \rangle$ on A, and by simple implementation, there exists L: $A \sqcup B' \underset{E \sqcup D'}{\rightsquigarrow} B$, where B is a result of $\langle Q, R \rangle$ on A and $L/E = \mathrm{id}(A)$. The composition of K and L is the desired representation morphism $(K\,;L)$: $B'' \underset{E \sqcup D''}{\rightsquigarrow} B$, because $(K\,;L)/E = (K/E)\,;(L/E) = \mathrm{id}(A)\,;\ \mathrm{id}(A) = \mathrm{id}(A)$.

To show reflexivity, let $\langle Q, R \rangle$ be an $\langle E, D \rangle$-cell. The cell signature $\langle E, D \rangle$ is a syntactic refinement of itself (Prop. 3.1.17 (a)), and if $A \in Q$, then A is a base for $\langle Q, R \rangle$, and if B' is a result of $\langle Q, R \rangle$ on A, then $\mathrm{id}(B')$ is the desired representation morphism, because $\mathrm{id}(B')/E = \mathrm{id}(A)$ by functorality. □

Another interesting property of "simple implementation" is its "monotonicity" in the sense that a larger representation category makes it more general.

5.1.7 Proposition. *Let Rep and Rep′ be two representation functors such that for all $S \in |Sig|$, $Rep(S)$ is a subcategory of $Rep'(S)$ and if σ: $S \to T$ in $|Sig|$, then the functor $Rep'(\sigma^{\mathrm{op}})$ agrees with $Rep(\sigma^{\mathrm{op}})$ on $Rep(T)$.*[1] *If a cell $\langle Q', R' \rangle$*

[1] In more advanced categorial language, the $|Sig|$-indexed family of inclusion functors from $Rep(S)$ to $Rep'(S)$ is a natural transformation from *Rep* to *Rep′*; the proposition easily generalizes to an arbitrary natural transformation from *Rep* to *Rep′* whose component functors have identities as object functions.

is a simple implementation of a cell $\langle Q, R\rangle$ *with respect to Rep, then* $\langle Q', R'\rangle$ *is a simple implementation of* $\langle Q, R\rangle$ *with respect to* Rep'.

Proof. Let $\langle Q', R'\rangle$ be a simple implementation of $\langle Q, R\rangle$ with respect to *Rep*. Then the signature of $\langle Q', R'\rangle$ is syntactic refinement of the signature of $\langle Q, R\rangle$. If $A \in Q$, then A is a base for $\langle Q', R'\rangle$, and if B' is a result of $\langle Q', R'\rangle$ on A, then there exists a representation morphism $J\colon B' \underset{E \sqcup D}{\rightsquigarrow} B$ in *Rep* such that B is a result of $\langle Q, R\rangle$ on A and $J/E = \mathrm{id}(A)$. But then J is also a representation morphism in Rep', and

$$J/E = Rep'((E \sqsubseteq E \sqcup D)^{\mathrm{op}})(J) = Rep((E \sqsubseteq E \sqcup D)^{\mathrm{op}})(J) = \mathrm{id}(A),$$

as the identity morphisms are the same in $Rep(E)$ and $Rep'(E)$. Hence $\langle Q', R'\rangle$ is a simple implementation of $\langle Q, R\rangle$ with respect to Rep'. □

This proposition indicates that it is generally better to work with richer representation categories, since this simplifies the task of proving the "simple implementation" relation between cells.

As an example, let us determine the simple implementation relation associated with *REqual*.

5.1.8 Proposition. *A cell* $\langle Q', R'\rangle$ *is a simple implementation of a cell* $\langle Q, R\rangle$ *with respect to REqual if and only if the signature of* $\langle Q', R'\rangle$ *is a syntactic refinement of the signature of* $\langle Q, R\rangle$, *and whenever* $A \in Q$, *then*

> *A is a base for* $\langle Q', R'\rangle$, *and*
> *every result of* $\langle Q', R'\rangle$ *on A is a result of* $\langle Q, R\rangle$ *on A.*

Proof. Rewrite Definition 5.1.5 using the fact that "$J\colon B' \rightsquigarrow B$" is equivalent to "$B = B'$ and $J = \mathrm{id}(B)$". □

This relation is weaker than "universal implementation with respect to equality", that is, weaker than refinement (Prop. 4.1.6), in that the cell $\langle Q', R'\rangle$ is not required to have a result on A. This is just the simplified refinement notion that was discussed briefly at the end of Section 3.1.

We now begin to investigate the relationship between universal implementation and simple implementation. The crucial observation (in fact, the main idea underlying the "simple implementation" notion) is the following.

5.1.9 Proposition. *A universal implementation of a simple implementation of a cell is a universal implementation of that cell.*

Proof. Let $\langle Q, R\rangle$ be an $\langle E, D\rangle$-cell, let $\langle Q', R'\rangle$ be an $\langle E', D'\rangle$-cell, and let $\langle Q'', R''\rangle$ be an $\langle E'', D''\rangle$-cell, and suppose that $\langle Q'', R''\rangle$ is a universal implementation of $\langle Q', R'\rangle$ and $\langle Q', R'\rangle$ is a simple implementation of $\langle Q, R\rangle$.

Since $\langle E'', D''\rangle$ is a syntactic refinement of $\langle E', D'\rangle$ by universal implementation and $\langle E', D'\rangle$ is a syntactic refinement of $\langle E, D\rangle$ by simple implementation, it follows that $\langle E'', D''\rangle$ is a syntactic refinement of $\langle E, D\rangle$.

Now let $A \in Mod(F)$ be a base for $\langle Q, R\rangle$ and $A' \underset{F}{\rightsquigarrow} A$. By syntactic refinement, F is a site for $\langle E', D'\rangle$ and $\langle E'', D''\rangle$, and $G := F \sqcup D = F \sqcup D' = F \sqcup D''$. Now $A/E \in Q$, hence by simple implementation, A/E is a base for $\langle Q', R'\rangle$, hence $A/E' = (A/E)/E' \in Q'$, and so A is a base for $\langle Q', R'\rangle$. By universal implementation, A' is a base for $\langle Q'', R''\rangle$, and there exists a result of $\langle Q'', R''\rangle$ on A'.

Now let B'' be a result of $\langle Q'', R''\rangle$ on A'. By universal implementation, there exists a result B' of $\langle Q', R'\rangle$ on A such that $B'' \underset{G}{\rightsquigarrow} B'$. This means that we can pick a representation morphism $J\colon B'' \underset{G}{\rightsquigarrow} B'$.

Recall that $A/E \in Q$. Now $C' := B'/(E \sqcup D')$ is a result of $\langle Q', R'\rangle$ on A/E, as $C'/E = B'/E = (B'/F)/E = A/E$, and $C'/D' = B'/D' \in R'$. By simple implementation, there exists a representation morphism $L\colon C' \underset{E \sqcup D'}{\rightsquigarrow} C$ such that C is a result of $\langle Q, R\rangle$ on A/E and $L/E = \mathrm{id}(A/E)$.

Now L and $\mathrm{id}(A)$ satisfy the assumptions of the pair-completeness axiom for representation morphisms (Axiom (c) of Def. 5.1.1), since the intersection of their signatures is

$$\begin{aligned}(E \sqcup D') \sqcap F &= (E \sqcup D) \sqcap F && \text{(syntactic refinement, cf. Prop. 3.1.17 (c))}\\ &= E && (F \text{ is a site for } \langle E, D\rangle),\end{aligned}$$

and $L/E = \mathrm{id}(A/E) = \mathrm{id}(A)/E$. Hence we can pick a representation morphism $K\colon A \sqcup C' \underset{G}{\rightsquigarrow} A \sqcup C$. Now $A \sqcup C' = B'$ by uniqueness of joins (since $B'/F = A$ and $B'/(E \sqcup D') = C'$). Hence $K\colon B' \underset{G}{\rightsquigarrow} A \sqcup C$, and $(J\,;K)\colon B'' \underset{G}{\rightsquigarrow} A \sqcup C$, which means that $B'' \underset{G}{\rightsquigarrow} A \sqcup C$.

Finally, $A \sqcup C$ is a result of $\langle Q, R\rangle$ on A, because $(A \sqcup C)/F = A/F = A$ and $(A \sqcup C)/D = C/D \in R$.

Hence $\langle Q'', R''\rangle$ is a universal implementation of $\langle Q, R\rangle$. □

A first consequence of this proposition is that if a universal implementation $\langle Q', R'\rangle$ of a cell $\langle Q, R\rangle$ is developed in several steps:

$$\langle Q', R'\rangle = \langle Q^{(n)}, R^{(n)}\rangle \text{ impl. of } \ldots$$
$$\ldots \text{ impl. of } \langle Q^{(1)}, R^{(1)}\rangle \text{ impl. of } \langle Q^{(0)}, R^{(0)}\rangle = \langle Q, R\rangle \qquad (n \geq 1),$$

the universal implementation property must be verified only at the last step ($\langle Q^{(n)}, R^{(n)}\rangle$ universal implementation of $\langle Q^{(n-1)}, R^{(n-1)}\rangle$), while in all the other steps, only the simple implementation property needs to be verified.

We shall go further, however, and make it unnecessary for a programmer to verify the universal implementation property even at the last step.

Note that while generally, cells may be defined by arbitrary mathematical predicates, the final cell of a chain of implementations will have to be coded in a programming notation to be useful, and a programming notation is a rather restrictive formal language. This restriction may actually be beneficial, for by appropriately designing the programming notation one may ensure useful semantic properties of the final cell.

It is well known that programming languages intended to support data abstraction should be designed such that access to encapsulated data type is possible only by means of the access functions explicitly provided by the encapsulation. It is difficult to see, however, what the semantic propertis are that this "limited access" to encapsulated data type entails.

Rather than trying to determine these properties from existing languages, we shall postulate a property of cells, called "stability", on the basis of the theoretical notions "universal implementation" and "simple implementation".

5.1.10 Definition. A cell is *stable*, if it is a universal implementation of every cell of which it is a simple implementation. □

It is clear that if the final cell of a sequence of implementation steps is stable, then at each step only the simple implementation property needs to be proved—since the final cell is stable, it is then a universal implementation of the cell it simply implements, and hence (by Proposition 5.1.9) a universal implementation of the cell with which the development began.

From the definition, however, stability appears to be a very strong requirement on cells. We shall now develop some simpler characterizations of stability, and in the sections to follow investigate the stability notions for representation functors in $\langle$**TSig**, **TIncl**, **TAlg**$\rangle$. As we shall see, stability is a property we may reasonably expect the modules of a data abstraction programming language to have.

From Proposition 5.1.9, one easily obtains the following elegant characterization of stability.

5.1.11 Theorem. *A cell is stable, if and only if it is a universal implementation of itself.*

Proof. Let $\langle Q, R\rangle$ be a cell.

Suppose first that $\langle Q, R\rangle$ is stable. By Proposition 5.1.6, $\langle Q, R\rangle$ is a simple implementation of itself. By stability, $\langle Q, R\rangle$ is a universal implementation of itself.

Conversely, suppose that $\langle Q, R\rangle$ is a universal implementation of itself. If $\langle Q', R'\rangle$ is a cell of which $\langle Q, R\rangle$ is a simple implementation, then $\langle Q, R\rangle$ is a universal implementation of the simple implementation $\langle Q, R\rangle$ of $\langle Q', R'\rangle$, hence a universal implementation of $\langle Q', R'\rangle$ by Proposition 5.1.9. Thus, $\langle Q, R\rangle$ is stable. □

This characterization is simpler than the definition of stability, because no quantification over cells is involved any more.

An interesting observation is that the right hand side of this characterization, "universal implementation of itself", depends only on the representation relation of *Rep* and is independent of the way this representation relation is characterized by representation morphisms. This is remarkable because "stability" is defined using "simple implementation", which does depend on the representation morphisms. In particular, this observation allows us to associate the stability notion with a representation relation rather than with a representation functor. So do, for example, the headings of the remaining three sections of this chapter.

As an example of the application of Theorem 5.1.11, we now determine the stability notion associated with the representation relation "equality".

5.1.12 Proposition. *A cell is stable for equality, if and only if it is consistent.*

Proof. By Theorem 5.1.11, a cell is stable for equality, if and only if it is a universal implementation of itself with respect to equality. By Proposition 4.1.6, this is equivalent to the cell's being a refinement of itself, and by Proposition 3.1.19 (c), this is equivalent to the cell's being consistent. □

In particular, this means that the simplified refinement notion in which one does not check whether the refinement has a result (*i. e.*, simple implementation with respect to equality as just characterized in Prop. 5.1.12), can be used in modular programming, provided that the cells developed are guaranteed to be consistent. This will of course be the case for well-defined programming notations, as already remarked after Example 3.1.14. Thus, we have encountered the first (simple) instance of a stability notion that is both reasonable to assume of the modules of a programming notation and useful to simplify the programmer's correctness arguments.

As another application of Theorem 5.1.11, we can prove that the composition of cells preserves stability.

5.1.13 Theorem. *Let M be a system of stable cells, and let C be a signature compatible with the join of the definition signatures of M. Then $\square_C M$ is stable.*

Proof. Let $<$ be the syntactic dependence ordering of the signature of M. By Theorem 3.5.6, $[M, <]$ is a decomposition of $\square_C M$. Since the cells of M are universal implementation of themselves, the composability theorem yields that $\square_C M$ is a universal implementation of $\square_C M$. Hence $\square_C M$ is stable. □

This theorem implies that if all cells that can be defined in a programming notation are stable, then all the cells composed from such cells are also stable. As a consequence, modular programming with data abstraction can be used recursively: each of the implementation cells in a structured correctness argument may be designed as the composition of another modular system. If the implementation cells at the bottom of such a hierarchy of modular systems are stable, then all the composed cells arising at higher levels will also be stable, and only the simple implementation property needs to be verified throughout the development.

Modular programming with data abstraction has now been formalized as consisting of development steps of two types:

- "decomposition" of a cell into a cell system (Def. 3.2.10), which can be performed by means of a design graph (cf. the discussion at the end of Section 3.2),
- "simple implementation" of a cell by another cell (Def. 5.1.5).

The theorems of this thesis show that if the final implementation cells (*i. e.*, those that are not themselves realized by a modular system) are stable, then the cell obtained by composing these cells is a universal implementation of the global cell with which the development began. In particular, the model produced by the implementation on any (representation of a) model of the external requirement interface is guaranteed to be a representation of a model of the external result interface.

The following definition presents a useful sufficient criterion for the stability of a cell.

5.1.14 Definition. A cell $\langle Q, R\rangle$ of signature $\langle E, D\rangle$ *extends representation morphisms*, if whenever $A \in Q$ and $J\colon A' \underset{E}{\rightsquigarrow} A$ is a representation morphism, then

$A' \in Q$,
there exists a result of $\langle Q, R\rangle$ on A', and
whenever B' is a result of $\langle Q, R\rangle$ on A',
then there exists a representation morphism
$K\colon B' \underset{E \sqcup D}{\rightsquigarrow} B$
such that B is a result of $\langle Q, R\rangle$ on A and
$K/E = J$. □

5.1.15 Theorem. *A cell that extends representation morphisms is stable.*

Proof. Let $\langle Q, R\rangle$ be a cell that extends representation morphisms. By Theorem 5.1.4, applied with $\langle Q', R'\rangle = \langle Q, R\rangle$, it follows that $\langle Q, R\rangle$ is a universal implementation of itself, and hence is stable by Theorem 5.1.11). □

It was remarked earlier that in a programming language intended to support data abstraction, all modules should be stable. Theorem 5.1.15 indicates that this requirement is a reasonable one, since criteria very similar to "extension of representation morphisms" have already been proved for some programming notations.

In the research on "representation independence" properties of the typed λ-calculus, "logical relations" play a rôle similar to representation morphisms in this thesis, because observational equivalence of models is characterized by the existence of a logical relation between them [Mitchell 86, Section 6.2]. The "Fundamental Theorem of Logical Relations" [Plotkin 80, p. 365] [Statman 85, p. 92] [MM 85, p. 230] states that the denotations of a term of the typed λ-calculus

(or second-order typed λ-calculus) over models related by a logical relation are again related. This means that program modules consisting of definitions in the (second-order) typed λ-calculus extend logical relations.

Unfortunately, however, logical relations do not fit into the theory of this thesis, because the composition of logical relations is not always a logical relation, and hence the logical relations between models of a certain signature need not form a category. This makes it impossible to decompose the "universal implementation" concept into simple implementation and stability according to the present theory, because the proof of the important Proposition 5.1.9 depends on the composability of representation morphisms. It is thus an interesting problem to find a variant of the "logical relation" concept that is closed under composition and that still allows one to prove the "Fundamental Theorem of Logical Relations" to the effect that all expressions of the typed λ-calculus have related denotations in related models.

I conjecture that for combinatory extensional type structures (for terminology, see [Barendregt 84, Appendix A.1]), a relation $R \subseteq \prod_{i \in I} A^{(i)}$ need only be required to satisfy

$$R_{\sigma\to\tau}\langle f^{(i)}\rangle \implies \forall\langle x^{(i)}\rangle \in R_\sigma\colon \langle f^{(i)}x^{(i)}\rangle \in R_\tau \qquad (*)$$

rather than the usual definition (which has "$\iff$" in this formula), provided it relates the combinators $K_{\sigma\tau}$ and $S_{\sigma\tau\rho}$ of the respective models, *i. e.*,

$$\begin{aligned} \forall\sigma\tau\rho\colon\quad & \langle K^{(i)}_{\sigma\tau}\rangle \in R_{\sigma\to(\tau\to\sigma)} \\ & \langle S^{(i)}_{\sigma\tau\rho}\rangle \in R_{(\sigma\to(\tau\to\rho))\to((\sigma\to\tau)\to(\sigma\to\rho))}. \end{aligned}$$

This should work because in combinatory extensional type structures, all λ-expressions can be translated into expressions with the same semantic value that are built from the combinators using application only [Barendregt 84, Ch. 7]. The implication $(*)$ above ensures that the results of applying related functions to related arguments are again related.

A theory that perfectly fits the framework of this thesis has recently been designed by Nipkow [Nipkow 86]. Nipkow deals with so-called "structures", which

are many-sorted algebras with possibly nondeterministic operations. The signatures of these algebras are a slight variant of tagged algebraic signatures and can be equipped with signature morphisms and inclusions in the same manner. These signatures, together with the (small) structures as models, form an institution. Nipkow does not explicitly introduce a representation relation, but deals with "simulations" between structures, which are the representation morphisms of a representation functor. In particular, a simulation between stuctures with deterministic operations is just a strong correspondence.

Nipkow considers a programming language containing function application, local binding (let), lists, the conditional (if), recursion, and an "angelic choice" operator, and proves that for every program in this language that produces results of "visible" types, the meanings of the program over two structures related by a simulation satisfy a certain "implementation relation". This relation has no direct counterpart in the present theory. However, exploiting the internal details and constructions of Nipkow's proof, it is possible to conclude that every function definable in Nipkow's language is compatible with a given simulation $\sqsubseteq$ between two structures A and C. To this end, one introduces constants $!_{ac}$ for every $a \in A$ and $c \in C$ satisfying $a \sqsubseteq c$, with the interpretations $A_{!_{ac}} = a$ and $C_{!_{ac}} = c$. Then one considers programs consisting of the expression $f(!_{ac})$ in an arbitrary context of declarations (by let and letrec) containing a definition of f. Nipkow's proof then allows one to conclude that when such a program may diverge (*i. e.*, fail to deliver a result value) in C, it also may diverge in A, and every possible result value in C is related by the simulation $\sqsubseteq$ to a possible result value in A. Since this holds for every constant of the form "$!_{ac}$", it follows that the function f is compatible with the simulation $\sqsubseteq$. Thus, the functions definable in Nipkow's language over the models A and C may be added to A and C without destroying the simulation property of $\sqsubseteq$. We may therefore conclude that modules consisting of function definitions in Nipkow's language extend simulations.

Thus, Nipkow's paper exhibits a reasonably rich programming language that by adding a suitable module definition facility can be turned into a language in which all modules are stable for simulation (more precisely, the representation relation associated with simulations), and which therefore supports data abstrac-

tion in the sense of this thesis. During the development of modular programs in this language, it suffices to prove that each cell is a simple implementation of its specification.

However, Nipkow's language does not provide new data type definition facilities. These are an essential ingredient of a useful data abstraction programming language. It is not too difficult to add such facilities to the language while preserving stability of its modules—it suffices to prove that the type constructors (whose instantiations may be regarded as modules) extend simulations. A module can then be regarded as a composition of type definitions and operation definitions, and since each of these extends simulations, the whole module does. It is easily proved that standard type constructors such as product, union or "list" extend simulations.

5.2 Stability for Behavioural Inclusion

This section analyses the stability notion associated with the representation relation "behavioural inclusion" between partial algebras.

First, we present a representation functor whose representation relation is behavioural inclusion.

5.2.1 Definition. Let $\mathbf{Corr}$: $\mathbf{TSig}^{\mathrm{op}} \to \mathbf{LCat}$ be the functor mapping $\langle \Sigma, V \rangle \in |\mathbf{TSig}|$ to the category of V-correspondences between small Σ-algebras (cf. Proposition 4.3.6), and a tagged signature morphism σ: $\langle \Sigma, V \rangle \to \langle \Sigma', V' \rangle$ (with S and S' the sort sets of Σ and Σ') to the functor $\mathbf{Corr}(\sigma^{\mathrm{op}})$: $\mathbf{Corr}\langle \Sigma', V' \rangle \to \mathbf{Corr}\langle \Sigma, V \rangle$ which maps a correspondence $G = \langle G_s \rangle_{s \in S'}$: $A \xrightarrow[V']{} B$ in $\mathbf{Corr}\langle \Sigma', V' \rangle$ to the correspondence $\mathbf{Corr}(\sigma^{\mathrm{op}})(G) := \overline{\sigma} G = \langle G_{\sigma s} \rangle_{s \in S}$: $\overline{\sigma} A \xrightarrow[V]{} \overline{\sigma} B$ in $\mathbf{Corr}\langle \Sigma, V \rangle$ (cf. Proposition 4.3.14). □

5.2.2 Proposition. $\mathbf{Corr}$ *is a representation functor in* $\langle \mathbf{TSig}, \mathbf{TIncl}, \mathbf{TAlg} \rangle$, *and its representation relation is behavioural inclusion.*

Proof. First, we check that for each $\sigma\colon \langle \Sigma, V\rangle \to \langle \Sigma', V'\rangle$ in **TSig**, $\mathbf{Corr}(\sigma^{\mathrm{op}})$ is a functor from $\mathbf{Corr}\langle \Sigma', V'\rangle$ to $\mathbf{Corr}\langle \Sigma, V\rangle$. By Proposition 4.3.14, the functor $\mathbf{Corr}(\sigma^{\mathrm{op}})$ maps a correspondence $G\colon A \xrightarrow[V']{\times} B$ in $\mathbf{Corr}\langle \Sigma', V'\rangle$ to a correspondence $\overline{\sigma}\, G = \mathbf{Corr}(\sigma^{\mathrm{op}})(G)\colon \overline{\sigma}\, A \xrightarrow[V]{\times} \overline{\sigma}\, B$ in $\mathbf{Corr}\langle \Sigma, V\rangle$. For $A \in \mathbf{TAlg}\langle \Sigma', V'\rangle = \mathbf{Alg}(\Sigma')$, we have

$$\begin{aligned}\mathbf{Corr}(\sigma^{\mathrm{op}})(\mathrm{id}(A)) &= \langle (\mathrm{id}(A))_{\sigma s}\rangle_{s\in S} = \langle \mathrm{Id}(A_{\sigma s})\rangle_{s\in S}\\ &= \langle \mathrm{Id}((\overline{\sigma}\, A)_s)\rangle_{s\in S} = \mathrm{id}(\overline{\sigma}\, A),\end{aligned}$$

and if $G\colon A \xrightarrow[V']{\times} B$ and $H\colon B \xrightarrow[V']{\times} C$ in $\mathbf{Corr}\langle \Sigma', V'\rangle$, then

$$\begin{aligned}\mathbf{Corr}(\sigma^{\mathrm{op}})(G\,;H) &= \langle (G\,;H)_{\sigma s}\rangle_{s\in S}\\ &= \langle G_{\sigma s}\,;H_{\sigma s}\rangle_{s\in S}\\ &= \langle (\mathbf{Corr}(\sigma^{\mathrm{op}})G)_s\,;(\mathbf{Corr}(\sigma^{\mathrm{op}})H)_s\rangle_{s\in S}\\ &= (\mathbf{Corr}(\sigma^{\mathrm{op}})G)\,;(\mathbf{Corr}(\sigma^{\mathrm{op}})H).\end{aligned}$$

Hence $\mathbf{Corr}(\sigma^{\mathrm{op}})$ is a functor from $\mathbf{Corr}\langle \Sigma', V'\rangle$ to $\mathbf{Corr}\langle \Sigma, V\rangle$.

Next, we check that **Corr** is a functor. First, **Corr** maps identity signature morphisms to identity functors, because if $\langle \Sigma, V\rangle \in |\mathbf{TSig}|$, ($S$ the sort set of Σ), then $\mathbf{Corr}((\mathrm{id}\langle \Sigma, V\rangle)^{\mathrm{op}})$ maps an arrow $G\colon A \xrightarrow[V]{\times} B$ in $\mathbf{TSig}\langle \Sigma, V\rangle$ to

$$\begin{aligned}\mathbf{Corr}((\mathrm{id}\langle \Sigma, V\rangle)^{\mathrm{op}})G &= \mathbf{Corr}((\mathrm{Id}(S+F))^{\mathrm{op}})G\\ &= \langle G_{\mathrm{Id}(S+F)(s)}\rangle_{s\in S}\\ &= \langle G_s\rangle_{s\in S}\\ &= G.\end{aligned}$$

Second, **Corr** preserves composition, because if $\sigma\colon \langle \Sigma, V\rangle \to \langle \Sigma', V'\rangle$ and $\tau\colon \langle \Sigma', V'\rangle \to \langle \Sigma'', V''\rangle$ in **TSig** (S and S' the sort sets of Σ and Σ'), then $\mathbf{Corr}(\tau^{\mathrm{op}}\,;\sigma^{\mathrm{op}})$ maps a morphism G of $\mathbf{Corr}\langle \Sigma'', V''\rangle$ to

$$\begin{aligned}\mathbf{Corr}(\tau^{\mathrm{op}}\,;\sigma^{\mathrm{op}})G &= \mathbf{Corr}((\sigma\,;\tau)^{\mathrm{op}})G\\ &= \langle G_{(\sigma;\tau)s}\rangle_{s\in S}\\ &= \langle G_{\tau(\sigma s)}\rangle_{s\in S}\end{aligned}$$

$$= \mathbf{Corr}(\sigma^{\mathrm{op}})\langle G_{\tau s}\rangle_{s\in S'}$$
$$= \mathbf{Corr}(\sigma^{\mathrm{op}})(\mathbf{Corr}(\tau^{\mathrm{op}})G)$$
$$= (\mathbf{Corr}(\tau^{\mathrm{op}})\,;\,\mathbf{Corr}(\sigma^{\mathrm{op}}))G.$$

Finally, we check the three axioms of Definition 5.1.1.

Axiom (a): By definition, $\mathbf{Corr}\langle\Sigma,V\rangle$ has object set $\mathbf{TAlg}\langle\Sigma,V\rangle$ for $\langle\Sigma,V\rangle \in |\mathbf{TSig}|$.

Axiom (b): For $\sigma\colon \langle\Sigma,V\rangle \to \langle\Sigma',V'\rangle$ in $\mathbf{TSig}$, the object function of $\mathbf{Corr}(\sigma^{\mathrm{op}})$ maps $A \in \mathbf{TAlg}\langle\Sigma',V'\rangle$ to $\overline{\sigma}\,A = \mathbf{TAlg}(\sigma^{\mathrm{op}})A$ and thus agrees with $\mathbf{TAlg}(\sigma^{\mathrm{op}})$.

Axiom (c): Let $\langle\Sigma_0,V_0\rangle$ and $\langle\Sigma_1,V_1\rangle$ be compatible tagged algebraic signatures, where $\Sigma_i = \langle S_i,\ \alpha_i\colon F_i \to S_i^+\rangle$ for $i \in \{0,1\}$, and let

$$J\colon A \underset{V_0}{\longrightarrow\!\!\!\!\times} B \quad \text{and} \quad K\colon C \underset{V_1}{\longrightarrow\!\!\!\!\times} D$$

be morphisms in $\mathbf{Corr}\langle\Sigma_0,V_0\rangle$ and $\mathbf{Corr}\langle\Sigma_1,V_1\rangle$ such that

$$J/(\langle\Sigma_0,V_0\rangle \sqcap \langle\Sigma_1,V_1\rangle) = K/(\langle\Sigma_0,V_0\rangle \sqcap \langle\Sigma_1,V_1\rangle),$$

i.e., $A/(\langle\Sigma_0,V_0\rangle\sqcap\langle\Sigma_1,V_1\rangle) = C/(\langle\Sigma_0,V_0\rangle\sqcap\langle\Sigma_1,V_1\rangle)$, $B/(\langle\Sigma_0,V_0\rangle\sqcap\langle\Sigma_1,V_1\rangle) = D/(\langle\Sigma_0,V_0\rangle \sqcap \langle\Sigma_1,V_1\rangle)$, and $J_s = K_s$ for $s \in S_0 \cap S_1$ (which is the sort set of $\Sigma_0 \sqcap \Sigma_1$).

Define $L := J \cup K$ (where J and K are regarded as families with domain S_0 and S_1, respectively), so that $L_s = J_s$ for $s \in S_0$ and $L_s = K_s$ for $s \in S_1$. We now show that $L\colon A \sqcup C \underset{(V_0\cup V_1)}{\longrightarrow\!\!\!\!\times} B \sqcup D$ in $\mathbf{Corr}(\langle\Sigma_0,V_0\rangle \sqcup \langle\Sigma_1,V_1\rangle)$.

Recall from Theorem 4.2.6 (c) that $\langle\Sigma_0,V_0\rangle \sqcup \langle\Sigma_1,V_1\rangle = \langle\Sigma_0 \sqcup \Sigma_1,\ V_0 \cup V_1\rangle$. The index set of L is $S_0 \cup S_1$, and for $s \in S_0 \cup S_1$:

- if $s \in S_0$, then $L_s = J_s \subseteq A_s \times B_s = (A \sqcup C)_s \times (B \sqcup D)_s$,
- if $s \in S_1$, then $L_s = K_s \subseteq C_s \times D_s = (A \sqcup C)_s \times (B \sqcup D)_s$.

Hence L is an $(S_0 \cup S_1)$-sorted relation between the carriers of $A \sqcup C$ and $B \sqcup D$.

To check that the operations are compatible with L, consider $f\colon s_1 \ldots s_n \to r$ in $\Sigma_0 \sqcup \Sigma_1$.

- if $f \in F_0$, then $f\colon s_1 \dots s_n \to r$ in Σ_0; in particular, $\{s_1, \dots, s_n, r\} \subseteq S_0$. Hence if

$$(x_i, y_i) \in L_{s_i} \text{ for } i \in \{1, \dots, n\} \text{ and } \langle x_1, \dots, x_n \rangle \in \mathrm{dom}(A \sqcup C)_f,$$

then

$$(x_i, y_i) \in J_{s_i} \text{ for } i \in \{1, \dots, n\} \text{ and } \langle x_1, \dots, x_n \rangle \in \mathrm{dom}\, A_f.$$

Since J is a correspondence, it follows that $\langle y_1, \dots, y_n \rangle \in \mathrm{dom}\, B_f = \mathrm{dom}(B \sqcup D)_f$, and

$$\begin{aligned}\big((A \sqcup C)_f(x_1, \dots, x_n),\ (B \sqcup D)_f(y_1, \dots, y_n)\big)& \\ = \big(A_f(x_1, \dots, x_n),\ B_f(y_1, \dots, y_n)\big) \in J_r = L_r.&\end{aligned}$$

- the case $f \in F_1$ is symmetric.

Hence L is a correspondence.

To see that L is a $(V_0 \cup V_1)$-correspondence, consider $v \in V_0 \cup V_1$.

- if $v \in V_0$, then $(A \sqcup C)_v = A_v \subseteq B_v = (B \sqcup D)_v$ and $L_v = J_v$ is the inclusion function, since J is a V_0-correspondence.

- if $v \in V_1$, then $(A \sqcup C)_v = C_v \subseteq D_v = (B \sqcup D)_v$ and $L_v = K_v$ is the inclusion function, since K is a V_1-correspondence.

Hence L is a $(V_0 \cup V_1)$-correspondence, and so $L\colon A \sqcup C \underset{(V_0 \cup V_1)}{\longrightarrow\!\!\!\!\times} B \sqcup D$ in $\mathbf{Corr}(\langle \Sigma_0, V_0 \rangle \sqcup \langle \Sigma_1, V_1 \rangle)$. Clearly,

$$L/\langle \Sigma_0, V_0 \rangle = J \quad \text{and} \quad L/\langle \Sigma_1, V_1 \rangle = K.$$

Hence **Corr** has the pair-completeness property.

We have proved that **Corr** is a representation functor in $\langle \mathbf{TSig}, \mathbf{TIncl}, \mathbf{TAlg} \rangle$. By Theorem 4.3.7, the representation relation of **Corr** is behavioural inclusion. □

The theorem just proved makes the theory of the preceding section applicable, and we obtain a "simple implementation" concept for correspondences, and a "stability" concept for behavioural inclusion.

It is important to note that the "simple implementation" notion for behavioural inclusion is independent of the visibility of sorts in the environment signatures of the cells involved. The reason is that the correspondence required by the simple implementation criterion must be the identity correspondence when reduced to the environment signatures anyway, and so it automatically satisfies the restrictions that might be imposed on it by visibility of environment sorts. In other words, the simple implementation criterion requires one to treat the environment sorts as if they were all visible, by insisting that the correspondence to be constructed be the identity on the environment.

What is more, modules in programming do not normally define new visible sorts, as explained at the end of Section 4.2. Under this condition, visibility of sorts becomes completely redundant in the simple implementation notion for behavioural inclusion: The correspondence to be constructed must be the identity on the environment sorts, and is unconstrained on the new sorts defined by the cells under consideration.

Since simple implementation is the correctness criterion to be verified in the design of modular programs, this activity is unaffected by visibility of sorts. The visibility of sorts only affects the stability property of modules, which should be the concern of programming language design.

5.2.3 Example. The proof given in Example 4.3.10 that an algebra B is behaviourally included in an algebra A can be re-interpreted as a proof that a program module implementing the *string* data type is a simple implementation of its specification with respect to correspondences.

The abstract import interface Q and the abstract export interface R are given in Figure 5-3. Q describes the sorts to which the *string* data type relates, and R abstractly specifies the desired encapsulated type. The signature Σ of R is the same as in Example 4.3.10, and for a fixed model K of Q, in which $K_{bool} = \{\mathbf{T}, \mathbf{F}\}$ and K_{char} is an arbitrary set C, there is a unique result of $\langle Q, R \rangle$ on K. This result is just the algebra A considered in Example 4.3.10 (identifying C there with C here).

$Q =$

interface

signature

bool, *char*: **sort**

properties

$bool = \{\mathbf{T}, \mathbf{F}\}$

$R =$

interface

signature

bool, *char*, *string*: **sort**

$single$: $char \rightarrow string$

$occurs$: $char\ string \rightarrow bool$

$join$: $string\ string \rightarrow string$

properties

$bool = \{\mathbf{T}, \mathbf{F}\}$

$string = char^*$

$single(x) = \langle x \rangle$

$$occurs(x, s) = \begin{cases} \mathbf{T}, & \text{if } x \in \operatorname{ran} s \\ \mathbf{F}, & \text{if } x \notin \operatorname{ran} s \end{cases}$$

$join(s, t) = s \circ t$

Figure 5-3: The abstract import and export interfaces Q and R

The implementation cell we shall consider has requirement interface $Q' := Q$. The result interface R' depends on a parameter $K \geq 1$ characterizing the "capacity" of the implementation. R' is given in Figure 5-4.

As explained in Example 4.3.10, the cell $\langle Q', R' \rangle$ could be coded in any programming notation that features a "sequence of *char*" data type with the usual access operations *nil*, *cons* etc. We are treating this data type a little differently here than the type *listitem* of the *dictionary* program development in Section 1.4.

$R' =$

interface

signature

Σ (the signature of R)

properties

$bool = \{\mathbf{T}, \mathbf{F}\}$

$string = char^*$

$single(x) = \langle x \rangle$

$$occurs(x,s) = \begin{cases} \mathbf{T}, & \text{if } x \in \operatorname{ran} s \\ \mathbf{F}, & \text{if } x \notin \operatorname{ran} s \end{cases}$$

$join(s,t) =$ **if** $length(s) = 0$ **then** t
else if $occurs(hd\, s, t)$ **then** $join(tl\, s, t)$
else if $length(t) \geq K$ **then** $Error()$
else $join(tl\, s, cons(hd\, s, t))$.

Figure 5-4: The result interface R' of the *string* implementation

There, the type *listitem* and its access functions were listed in the requirement interfaces of modules using them. Here, however, we have omitted the type "sequence of *char*" and its access functions from the interfaces Q and Q' and thus from the algebras we shall consider in order to keep them simple. This is not incorrect, for it means that here we treat the "sequence" type constructor and the functions *nil*, *cons* etc. as a fixed part of the programming notation (this could perhaps be made clear by writing them "nil", "cons" etc.) rather than as entities to be exported and imported by modules. The reader may wish to convince himself that they could be added to Q and R (and the algebras and the correspondence to be discussed) without problems.

The unique result of $\langle Q', R' \rangle$ on K is the algebra B of Example 4.3.10. In that example, a $\{bool, char\}$-correspondence G: $B \underset{\{bool,char\}}{\multimap\!\!\!\!\times} A$ was constructed from the result B of $\langle Q', R' \rangle$ on K to to the result A of $\langle Q, R \rangle$ on K. In particular, we

have $G/\operatorname{Sig}(Q) = \operatorname{id}(K)$, and hence G is the representation morphism required in the proof that $\langle Q', R' \rangle$ is a simple implementation of $\langle Q, R \rangle$.

Since K was an arbitrary element of Q, since the cells $\langle Q, R \rangle$ and $\langle Q', R' \rangle$ have the same signature and since $Q = Q'$, it follows that $\langle Q', R' \rangle$ is a simple implementation of $\langle Q, R \rangle$. □

The remainder of this section analyses the stability notion for behavioural inclusion.

5.2.4 Theorem. *A cell in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$ *is stable for behavioural inclusion, if and only if it extends correspondences.*

The proof rests on the following Lemma.

5.2.5 Lemma. *Let A and B be Σ-algebras, let $G\colon A \multimap B$ be a correspondence, and let $H \sqsupseteq \Sigma$ be an algebraic signature. Then there exists an algebraic signature $\hat{\Sigma}$ with the same sorts as Σ and $\hat{\Sigma}$-algebras $\hat{A}$ and $\hat{B}$ such that $\Sigma \sqsubseteq \hat{\Sigma}$, $\hat{\Sigma} \sim H$, $\hat{\Sigma} \sqcap H = \Sigma$, $\hat{A}/\Sigma = A$ and $\hat{B}/\Sigma = B$, and such that G is the only correspondence from $\hat{A}$ to $\hat{B}$. If Σ, A, and B are small, so are $\hat{\Sigma}$, $\hat{A}$, and $\hat{B}$.*

If, in addition, G is a strong partial homomorphism (an abstraction function) from A to B, then G also is a strong partial homomorphism (an abstraction function) from $\hat{A}$ to $\hat{B}$.

Proof. Write $\Sigma = \langle S, \alpha\colon F \to S^+ \rangle$. If for all $s \in S$, we have $A_s = \emptyset \vee B_s = \emptyset$, then all components of a correspondence from A to B must be empty relations, hence G is unique and we can put $\hat{\Sigma} := \Sigma$, $\hat{A} := A$ and $\hat{B} := B$.

In the following, we may therefore suppose that there exists an $s \in S$ such that $A_s \neq \emptyset$ and $B_s \neq \emptyset$. This means that we can choose $\hat{s} \in S$, $\hat{x} \in A_{\hat{s}}$ and $\hat{y} \in B_{\hat{s}}$ such that

$$\exists s \in S\colon G_s \neq \emptyset \implies (\hat{x}, \hat{y}) \in G_{\hat{s}}.$$

Construct $\hat{\Sigma} = \langle S, \hat{\alpha} : \hat{F} \to S^+ \rangle$ by adding to Σ the following function symbols, distinct from each other and distinct from the symbols of H:

- for each $s \in S$, $(x,y) \in G_s$ a function symbol $!_{sxy}\colon \to s$,
- for each $s \in S$, $z \in B_s$ a function symbol $?_{sz}\colon s \to \hat{s}$.

Clearly, $\hat{\Sigma}$ has the same sorts as Σ and is small if Σ, A and B are. Using formulas 2.3.6 (a) and 2.3.6 (c), we see that $\hat{\Sigma} \sim H$ and $\hat{\Sigma} \sqcap H = \Sigma$.

Define $\hat{A}$ and $\hat{B}$ as follows.

- for $s \in S$: $\quad \hat{A}_s = A_s \qquad \hat{B}_s = B_s$
- for $f \in F$: $\quad \hat{A}_f = A_f \qquad \hat{B}_f = B_f$
- for $s \in S$, $(x,y) \in G_s$: $\quad \hat{A}_{!_{sxy}}() = x \qquad \hat{B}_{!_{sxy}}() = y$
- for $s \in S$, $z \in B_s$: $\quad x \in \operatorname{dom} \hat{A}_{?_{sz}} \Leftrightarrow (x,z) \notin G_s \quad y \in \operatorname{dom} \hat{B}_{?_{sz}} \Leftrightarrow y \neq z$
  $\Rightarrow \hat{A}_{?_{sz}}(x) = \hat{x} \qquad \Rightarrow \hat{B}_{?_{sz}}(y) = \hat{y}$.

Clearly, $\hat{A}/\Sigma = A$ and $\hat{B}/\Sigma = B$. Also, $\hat{A}$ and $\hat{B}$ are small if Σ, A and B are.

The S-sorted relation G is a correspondence from $\hat{A}$ to $\hat{B}$, because every $f \in \hat{F}$ is compatible with G:

- if $f \in F$, then f is compatible with G, because $G\colon A \longrightarrow\!\!\!\!\times\ B$,
- if $f = !_{sxy}\colon \to s$ for $s \in S$, $(x,y) \in G_s$, then $(\hat{A}_{!_{sxy}}(), \hat{B}_{!_{sxy}}()) = (x,y) \in G_s$,
- if $f = ?_{sz}\colon s \to \hat{s}$ for $s \in S$, $z \in B_s$, and $(x,y) \in G_s$ is such that $x \in \operatorname{dom} \hat{A}_{?_{sz}}$, then $(x,z) \notin G_s$, hence $y \neq z$, and thus $y \in \operatorname{dom} \hat{B}_{?_{sz}}$ and $(\hat{A}_{?_{sz}}(x), \hat{B}_{?_{sz}}(y)) = (\hat{x},\hat{y}) \in G_{\hat{s}}$ (as $G_s \neq \emptyset$).

Next, we show that G is unique. Suppose G' is a correspondence from A to B. Then for any $s \in S$, we have

$$(x,y) \in G_s \implies (\hat{A}_{!_{sxy}}(), \hat{B}_{!_{sxy}}()) = (x,y) \in G'_s,$$

and hence $G_s \subseteq G'_s$. Conversely,

$$\begin{aligned}(x,y) \in G'_s &\implies x \notin \operatorname{dom} \hat{A}_{?_{sy}} \qquad \text{(as } y \notin \operatorname{dom} \hat{B}_{?_{sy}} \text{ and } G' \text{ is a corr.)}\\ &\implies (x,y) \in G_s,\end{aligned}$$

and hence $G'_s \subseteq G_s$. It follows that $G' = G$.

Now suppose that in addition, $G\colon A \rightleftharpoons B$. We know already that $G\colon \hat{A} \longrightarrow\!\!\!\!\times\ \hat{B}$, and obviously, this implies that $G\colon \hat{A} \rightarrowtail \hat{B}$. To show that G is a strong partial homomorphism, we show that $G^{\cup}\colon \hat{B} \longrightarrow\!\!\!\!\times\ \hat{A}$ by showing that all $f \in \hat{F}$ are compatible with $G^{\cup}$:

- if $f \in F$, then f is compatible with $G^{\cup}$, because $G\colon A \Longrightarrow\!\!\!\!\times\ B$ and hence $G^{\cup}\colon B \longrightarrow\!\!\!\!\times\ A$.
- if $f = !_{sxy}\colon \to s$ for $s \in S$, $(x,y) \in G_s$, then $(\hat{B}_{!_{sxy}}(), \hat{A}_{!_{sxy}}()) = (y,x) \in G_s^{\cup}$,
- if $f = ?_{sz}\colon s \to \hat{s}$ for $s \in S$, $z \in B_s$, and $(y,x) \in G_s^{\cup}$ is such that $y \in \operatorname{dom} \hat{B}_{?_{sz}}$, then $y \neq z$, and since $(x,y) \in G_s$ and G_s is a partial function, we have $(x,z) \notin G_s$, hence $x \in \operatorname{dom} \hat{A}_{?_{sz}}$ and $(\hat{B}_{?_{sz}}(y), \hat{A}_{?_{sz}}(x)) = (\hat{y},\hat{x}) \in G_{\hat{s}}^{\cup}$ (as $G_s \neq \emptyset$).

Hence G is a strong partial homomorphism from A to B.

If G is an abstraction function from A to B even, then G consists of surjective partial functions only, and so G also is an abstraction function from $\hat{A}$ to $\hat{B}$. □

Proof of Theorem 5.2.4.

Let $\langle Q, R\rangle$ be a cell.

If $\langle Q, R\rangle$ extends correspondences, then $\langle Q, R\rangle$ is stable for behavioural inclusion by Theorem 5.1.15.

Conversely, suppose that $\langle Q, R\rangle$ is stable for behavioural inclusion, *i. e.*, a universal implementation of itself with respect to behavioural inclusion (Theorem 5.1.11). Let the signature of $\langle Q, R\rangle$ be $\langle\langle \Sigma, V\rangle, \langle \Sigma^\bullet, V^\bullet\rangle\rangle$ where $\Sigma = \langle S, \alpha\rangle$ and $\Sigma^\bullet = \langle S^\bullet, \alpha^\bullet\rangle$. Let $A \in Q$ and let $J\colon A' \underset{V}{\longrightarrow\!\!\!\!\times}\ A$ be a representation morphism in $\mathbf{Corr}\langle \Sigma, V\rangle$.

Since A is a base for $\langle Q, R\rangle$, $A' \lesssim_V A$ and $\langle Q, R\rangle$ is a universal implementation of itself, it follows that A' is a base for $\langle Q, R\rangle$, *i. e.*, $A' = A'/E \in Q$, and that there exists a result of $\langle Q, R\rangle$ on A'.

Now let B' be any result of $\langle Q, R\rangle$ on A'. By Lemma 5.2.5, applied to the correspondence $J\colon A' \underset{V}{\longrightarrow\!\!\!\!\times}\ A$ with $H := \Sigma \sqcup \Sigma^\bullet$, we can choose an algebraic

signature $\hat{\Sigma} = \langle \hat{S}, \hat{\alpha} \rangle$ and $\hat{\Sigma}$-algebras $\hat{A}'$ and $\hat{A}$ such that

$$\Sigma \sqsubseteq \hat{\Sigma}, \quad \hat{\Sigma} \sim \Sigma \sqcup \Sigma^\bullet, \quad \hat{\Sigma} \sqcap (\Sigma \sqcup \Sigma^\bullet) = \Sigma, \quad \hat{S} = S, \qquad (1)$$
$$\hat{A}'/\Sigma = A', \quad \hat{A}/\Sigma = A,$$

and such that J is the only correspondence from $\hat{A}'$ to $\hat{A}$.

Now $\langle \hat{\Sigma}, V \rangle$ is a tagged algebraic signature, and it is a site for $\langle Q, R \rangle$, because $\langle \hat{\Sigma}, V \rangle$ is compatible with $\langle \Sigma, V \rangle \sqcup \langle \Sigma^\bullet, V^\bullet \rangle$ by Theorem 4.2.6 (b):

$$\begin{aligned} \hat{S} \cap (V \cup V^\bullet) &= S \cap (V \cup V^\bullet) \\ &= V \cap (S \cup S^\bullet) \qquad (\text{as } \langle \Sigma, V \rangle \sim \langle \Sigma, V \rangle \sqcup \langle \Sigma^\bullet, V^\bullet \rangle), \end{aligned}$$

and because, using Theorem 4.2.6 (c) and (d):

$$\begin{aligned} \langle \hat{\Sigma}, V \rangle \sqcap (\langle \Sigma, V \rangle \sqcup \langle \Sigma^\bullet, V^\bullet \rangle) &= \langle \hat{\Sigma} \sqcap (\Sigma \sqcup \Sigma^\bullet),\ V \cap (V \cup V^\bullet) \rangle \\ &= \langle \Sigma, V \rangle \qquad (\text{using (1)}). \end{aligned}$$

Further, $\hat{A}$ is a base for $\langle Q, R \rangle$, because $\hat{A}/\langle \Sigma, V \rangle = \hat{A}/\Sigma = A \in Q$. Now $J\colon \hat{A}' \longrightarrow\!\!\!\!\times\ \hat{A}$ and, since $J\colon A' \underset{V}{\longrightarrow\!\!\!\!\times}\ A$, it trivially follows that $J\colon \hat{A}' \underset{V}{\longrightarrow\!\!\!\!\times}\ \hat{A}$; in particular (by Theorem 4.3.7), that $\hat{A}' \lesssim_V \hat{A}$.

Define $\hat{B}' := \hat{A}' \sqcup B'$. This is possible, because the meet of the signatures of $\hat{A}'$ and B' is $\langle \hat{\Sigma}, V \rangle \sqcap (\langle \Sigma, V \rangle \sqcup \langle \Sigma^\bullet, V^\bullet \rangle) = \langle \Sigma, V \rangle$, and $\hat{A}'/\langle \Sigma, V \rangle = A' = B'/\langle \Sigma, V \rangle$.

Now $\hat{B}'$ is a result of $\langle Q, R \rangle$ on $\hat{A}'$, as is easily seen. Since $\langle Q, R \rangle$ is a universal implementation of itself with respect to behavioural inclusion, it follows that there exists a result $\hat{B}$ of $\langle Q, R \rangle$ on $\hat{A}$ such that $\hat{B}' \underset{\langle \hat{\Sigma}, V \rangle \sqcup \langle \Sigma^\bullet, V^\bullet \rangle}{\rightsquigarrow} \hat{B}$; so by Theorem 4.3.7, there exists $\hat{K}\colon \hat{B}' \underset{V \cup V^\bullet}{\longrightarrow\!\!\!\!\times}\ \hat{B}$.

Since $\hat{K}/\langle \hat{\Sigma}, V \rangle\colon \hat{A}' \underset{V}{\longrightarrow\!\!\!\!\times}\ \hat{A}$ and J is the unique such correspondence, it follows that $\hat{K}/\langle \hat{\Sigma}, V \rangle = J$.

Defining $K := \hat{K}/(\langle \Sigma, V \rangle \sqcup \langle \Sigma^\bullet, V^\bullet \rangle)$ and $B := \hat{B}/(\langle \Sigma, V \rangle \sqcup \langle \Sigma^\bullet, V^\bullet \rangle)$, we have $K\colon B' \underset{V \cup V^\bullet}{\longrightarrow\!\!\!\!\times}\ B$.

It is easily seen that B is a result of $\langle Q, R \rangle$ on A. Now

$$\begin{aligned} K/\langle \Sigma, V \rangle &= \big(\hat{K}/(\langle \Sigma, V \rangle \sqcup \langle \Sigma^\bullet, V^\bullet \rangle)\big)/\langle \Sigma, V \rangle \\ &= \hat{K}/\langle \Sigma, V \rangle \end{aligned}$$

$$= (\hat{K}/\langle\hat{\Sigma},V\rangle)/\langle\Sigma,V\rangle$$
$$= J/\langle\Sigma,V\rangle$$
$$= J.$$

Hence K is the desired extension of J, and it has been proved that $\langle Q,R\rangle$ extends correspondences. □

5.3 Stability for Behavioural Equivalence

This section analyses the stability notion for the representation relation "behavioural equivalence" between partial algebras and compares it with the stability notion for behavioural inclusion.

First, we present a representation functor for behavioural equivalence. The definition follows the pattern of Definition 4.3.4

5.3.1 Definition. Let **SCorr**: $\mathbf{TSig}^{\mathrm{op}} \to \mathbf{LCat}$ be the functor mapping $\langle\Sigma,V\rangle \in |\mathbf{TSig}|$ to the category of strong V-correspondences between small Σ-algebras (cf. Proposition 4.4.5), and a tagged signature morphism σ: $\langle\Sigma,V\rangle \to \langle\Sigma',V'\rangle$ to the functor $\mathbf{SCorr}(\sigma^{\mathrm{op}})$: $\mathbf{SCorr}\langle\Sigma',V'\rangle \to \mathbf{SCorr}\langle\Sigma,V\rangle$ which is the restriction of $\mathbf{Corr}(\sigma^{\mathrm{op}})$ to strong correspondences. □

5.3.2 Proposition. **SCorr** *is a representation functor in* ⟨**TSig**, **TIncl**, **TAlg**⟩, *and its representation relation is behavioural equivalence.*

In the proof, the following lemma will be used.

5.3.3 Lemma. *Let Σ be an algebraic signature, let V be a subset of its sorts, let A and B be Σ-algebras, and let*

$$G\colon A \xrightarrow[V]{\times} B \quad \text{and} \quad H\colon B \xrightarrow[V]{\times} A$$

be V-correspondences. Then $G \cap H^{\cup}$, the componentwise intersection of G with the converse of H, is a strong V-correspondence from A to B.

Proof. To show that $G \cap H^{\cup}$ is a strong correspondence from A to B, consider $f\colon s_1 \dots s_n \to r$ in Σ and $(x_i, y_i) \in (G \cap H^{\cup})_{s_i}$ for $i \in \{1, \dots, n\}$, *i.e.*, $(x_i, y_i) \in G_{s_i}$ and $(y_i, x_i) \in H_{s_i}$.

If $\langle x_1, \dots, x_n \rangle \in \operatorname{dom} A_f$, then $\langle y_1, \dots, y_n \rangle \in \operatorname{dom} B_f$ because G is a correspondence, and if $\langle y_1, \dots, y_n \rangle \in \operatorname{dom} B_f$, then $\langle x_1, \dots, x_n \rangle \in \operatorname{dom} A_f$ because H is a correspondence. Thus,

$$\langle x_1, \dots, x_n \rangle \in \operatorname{dom} A_f \iff \langle y_1, \dots, y_n \rangle \in \operatorname{dom} B_f.$$

Assume now that both sides of this equivalence are true. Since G is a correspondence, $\big(A_f(x_1, \dots, x_n),\ B_f(y_1, \dots, y_n)\big) \in G_r$, and since H is a correspondence, $\big(B_f(y_1, \dots, y_n),\ A_f(x_1, \dots, x_n)\big) \in H_r$. It follows that

$$\big(A_f(x_1, \dots, x_n),\ B_f(y_1, \dots, y_n)\big) \in (G_r \cap H_r^{\cup}) = (G \cap H^{\cup})_r,$$

and hence that $G \cap H^{\cup}$ is a strong correspondence from A to B.

This strong correspondence is a strong V-correspondence, because for $v \in V$, we have $A_v \subseteq B_v$ as $G\colon A \xrightarrow[V]{\times} B$, and $B_v \subseteq A_v$ as $H\colon B \xrightarrow[V]{\times} A$, and thus $A_v = B_v$. Since G_v is the inclusion from A_v to B_v and H_v is the inclusion function from B_v to A_v, they are both identity functions, and so are $H_v^{\cup}$ and $G_v \cap H_v^{\cup}$, which equals $(G \cap H^{\cup})_v$. Thus, $G \cap H^{\cup}$ is a strong V-correspondence from A to B. □

Proof of Proposition 5.3.2.

For each $\langle \Sigma, V \rangle \in |\mathbf{TSig}|$, $\mathbf{SCorr}\langle \Sigma, V \rangle$ is a subcategory of $\mathbf{Corr}\langle \Sigma, V \rangle$ with the same objects.

For every $\sigma\colon \langle \Sigma, V \rangle \to \langle \Sigma', V' \rangle$ in $\mathbf{TSig}$, $\mathbf{SCorr}(\sigma^{\mathrm{op}})$ is a functor from $\mathbf{SCorr}\langle \Sigma', V' \rangle$ to $\mathbf{SCorr}\langle \Sigma, V \rangle$, because it is a restriction of $\mathbf{Corr}(\sigma^{\mathrm{op}})$, and because that functor maps strong V'-correspondences to strong V-correspondences by Proposition 4.5.7.

It is trivial to check the axioms (a) and (b) of Definition 5.1.1.

Axiom (c), pair-completeness, is obtained from the analogous property of **Corr**.

If $\langle \Sigma_0, V_0 \rangle$ and $\langle \Sigma_1, V_1 \rangle$ are compatible small tagged signatures, and

$$J\colon A \underset{V_0}{\rightrightarrows} B \quad \text{and} \quad K\colon C \underset{V_1}{\rightrightarrows} D$$

are morphisms in $\mathbf{SCorr}\langle \Sigma_0, V_0 \rangle$ and $\mathbf{SCorr}\langle \Sigma_1, V_1 \rangle$ respectively and satisfy

$$J/(\langle \Sigma_0, V_0 \rangle \sqcap \langle \Sigma_1, V_1 \rangle) = K/(\langle \Sigma_0, V_0 \rangle \sqcap \langle \Sigma_1, V_1 \rangle),$$

then

$$J\colon A \underset{V_0}{\rightarrow\!\!\!\!\!\times} B \quad \text{and} \quad K\colon C \underset{V_1}{\rightarrow\!\!\!\!\!\times} D$$

are correspondences satisfying the same law, hence by the pair-completeness of **Corr**, there exists a morphism

$$L\colon A \sqcup C \underset{V_0 \cup V_1}{\rightarrow\!\!\!\!\!\times} B \sqcup D$$

in $\mathbf{Corr}(\langle \Sigma_0, V_0 \rangle \sqcup \langle \Sigma_1, V_1 \rangle)$ such that

$$L/\langle \Sigma_0, V_0 \rangle = J \quad \text{and} \quad L/\langle \Sigma_1, V_1 \rangle = K.$$

Analogously, since

$$J^{\cup}\colon B \underset{V_0}{\rightarrow\!\!\!\!\!\times} A \quad \text{and} \quad K^{\cup}\colon D \underset{V_1}{\rightarrow\!\!\!\!\!\times} C$$

are correspondences satisfying

$$J^{\cup}/(\langle \Sigma_0, V_0 \rangle \sqcap \langle \Sigma_1, V_1 \rangle) = K^{\cup}/(\langle \Sigma_0, V_0 \rangle \sqcap \langle \Sigma_1, V_1 \rangle),$$

there exists a morphism

$$L'\colon B \sqcup D \underset{V_0 \cup V_1}{\rightarrow\!\!\!\!\!\times} A \sqcup C$$

in $\mathbf{Corr}(\langle \Sigma_0, V_0 \rangle \sqcup \langle \Sigma_1, V_1 \rangle)$ such that

$$L'/\langle \Sigma_0, V_0 \rangle = J^{\cup} \quad \text{and} \quad L'/\langle \Sigma_1, V_1 \rangle = K^{\cup}.$$

By Lemma 5.3.3, the componentwise intersection $L \sqcap (L')^{\cup}$ is a morphism

$$L \sqcap (L')^{\cup}\colon A \sqcup C \underset{V_0 \cup V_1}{\rightrightarrows} B \sqcup D$$

in $\mathbf{SCorr}(\langle \Sigma_0, V_0\rangle \sqcup \langle \Sigma_1, V_1\rangle)$, and we have

$$\begin{aligned}(L \cap (L')^{\cup})/\langle \Sigma_0, V_0\rangle &= (L/\langle \Sigma_0, V_0\rangle) \cap ((L')^{\cup}/\langle \Sigma_0, V_0\rangle)\\ &= (L/\langle \Sigma_0, V_0\rangle) \cap (L'/\langle \Sigma_0, V_0\rangle)^{\cup}\\ &= J \cap (J^{\cup})^{\cup}\\ &= J,\end{aligned}$$

and analogously,

$$(L \cap (L')^{\cup})/\langle \Sigma_1, V_1\rangle = K.$$

Hence $(L \cap (L')^{\cup})$ is the desired morphism in $\mathbf{SCorr}(\langle \Sigma_0, V_0\rangle \sqcup \langle \Sigma_1, V_1\rangle)$. □

The theorem just proved makes the concepts of Section 5.1 applicable, and we obtain a "simple implementation" concept for strong correspondences and a "stability" concept for behavioural equivalence.

The same argument as for behavioural inclusion (given just before Example 5.2.3) shows that the simple implementation notion for modules that do not define new visible sorts is not affected by visible sorts at all, so that the visible sorts notion is irrelevant in the development of modular programs.

5.3.4 Example. The proof given in Example 4.4.7, where a "representation" algebra B was proved behaviourally equivalent to an "abstract" algebra A, can be re-interpreted as a proof of the simple implementation relation between cells.

Let $\langle Q, R\rangle$ be the specification cell of Example 5.2.3 (Figure 5-3), and let the implementation cell $\langle Q', R'\rangle$ be obtained from the cell $\langle Q', R'\rangle$ of that example ($Q' = Q$, R' given in Figure 5-4) by removing the capacity constraint from the *join* operation, *i. e.*, by changing the definition of *join* to

$$\begin{aligned}join(s,t) = \textbf{if}\ & length(s) = 0\ \textbf{then}\ t\\ \textbf{else if}\ & occurs(hd\ s, t)\ \textbf{then}\ join(tl\ s, t)\\ \textbf{else}\ & join(tl\ s, cons(hd\ s, t)).\end{aligned}$$

In Example 4.4.7, a strong $\{bool, char\}$-correspondence G: $B \underset{\{bool,char\}}{\Longrightarrow\!\!\!\!\times} A$ was constructed from the result B of $\langle Q', R'\rangle$ to the result A of $\langle Q, R\rangle$ on an

arbitrary algebra $K \in Q$ (recall that Q has a signature consisting of just the two sorts *bool* and *char*, and that Q specifies "*bool* $= \{\mathbf{T}, \mathbf{F}\}$", but puts no constaints on *char*). The components G_{bool} and G_{char} of the strong correspondence G are identities, and thus $G/\operatorname{Sig}(Q) = \operatorname{id}(K)$, and hence G is the desired representation morphism in the proof that $\langle Q', R' \rangle$ is a simple implementation of $\langle Q, R \rangle$. □

In the remainder of this section, the "stability" notion for behavioural equivalence is characterized and compared with the one for behavioural inclusion.

The following criterion will be used in the characterization.

5.3.5 Definition. A cell $\langle Q, R \rangle$ of signature $\langle \langle \Sigma, V \rangle, \langle \Sigma^\bullet, V^\bullet \rangle \rangle$ in the institution $\langle$**TSig, TIncl, TAlg**$\rangle$ *weakly extends abstraction functions* (*weakly extends converse abstraction functions*), if whenever $A \in Q$ and $J\colon A' \overset{\Rightarrow}{\underset{V}{\twoheadrightarrow}} A$ is a V-abstraction function ($J\colon A' \overset{\Rightarrow}{\underset{V}{\rightarrowtail}} A$ is such that $J^\cup\colon A \overset{\Rightarrow}{\underset{V}{\twoheadrightarrow}} A'$ is a V-abstraction function), then

$A' \in Q$,
there exists a result of $\langle Q, R \rangle$ on A', and
whenever B' is a result of $\langle Q, R \rangle$ on A',
then there exists a strong $(V \cup V^\bullet)$-correspondence
$K\colon B' \overset{\Rightarrow}{\underset{V \cup V^\bullet}{\rightarrowtail}} B$
such that B is a result of $\langle Q, R \rangle$ on A and
$K/\langle \Sigma, V \rangle = J$. □

5.3.6 Theorem. *A cell in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$ *is stable for behavioural equivalence, if and only if it weakly extends abstraction functions and converse abstraction functions.*

The proof uses the following lemma.

5.3.7 Lemma. *Let* $\langle \Sigma, V \rangle$ *be a tagged algebraic signature, and let* $G\colon B \overset{}{\underset{V}{\rightarrowtail}} A$ *be a* V*-correspondence between* Σ*-algebras. Then there exists a diagram*

$$B \overset{H}{\underset{V}{\twoheadleftarrow}} C \overset{J}{\hookrightarrow} D \overset{K}{\underset{V}{\twoheadrightarrow}} A,$$

where H and K are V-abstraction functions, C is a weak subalgebra of D with the same hidden carriers as D, J is the inclusion homomorphism from C to D, and $G = H^{\cup}; J; K$.

If, moreover, G is a strong V-correspondence, the diagram can be chosen such that $C = D$ (and thus J is the identity), so that it has the form

$$B \overset{H}{\underset{V}{\twoheadleftarrow}} C \overset{K}{\underset{V}{\twoheadrightarrow}} A,$$

and $G = H^{\cup}; K$.

Proof. Let $\langle \Sigma, V \rangle$ be a tagged algebraic signature, $\Sigma = \langle S, \alpha \rangle$, and let G: $B \underset{V}{\multimap} A$ be a V-correspondence between Σ-algebras.

The carriers of the algebras C and D are defined as follows.

- for $s \in V$: $C_s := B_s$, $D_s := A_s$,
- for $s \in S \setminus V$: $C_s := D_s := \prod\langle\{0\}, G_s\rangle + \prod\langle\{1\}, B_s\rangle + \prod\langle\{2\}, A_s\rangle$

(*i. e.*, each hidden carrier of sort $s \in S \setminus V$ is composed of disjoint "copies" of the relation G_s and the carriers B_s and A_s). Obviously, C and D have the same hidden carriers.

For a set $M \subseteq \{0,1,2\}$ and $s \in S$, let the sets $C_s^M \subseteq C_s$ and $D_s^M \subseteq D_s$ be defined as follows.

- if $s \in V$: $C_s^M := C_s$, $D_s^M := D_s$.
- if $s \in S \setminus V$: $C_s^M := D_s^M := \{\, \langle d, x \rangle \in C_s \mid d \in M \,\}$.

Note that $C_s^M \subseteq D_s^M$ for all s and M. Define J to be the S-sorted inclusion function from C to D.

The abstraction functions H and K will be defined by the following S-sorted functions.

- for $s \in V$: $H_s := \mathrm{Id}(B_s)$, $K_s := \mathrm{Id}(C_s)$,
- for $s \in S \setminus V$:
  $\mathrm{dom}\, H_s := C_s^{\{0,1\}}$, $H_s(\langle 0, (x,y)\rangle) := x$, $H_s(\langle 1, x\rangle) := x$,
  $\mathrm{dom}\, K_s := D_s^{\{0,2\}}$, $K_s(\langle 0, (x,y)\rangle) := y$, $K_s(\langle 2, y\rangle) := y$.

Viewing G, H, J, and K as S-sorted functions, we then have $H^{\cup}\,;J\,;K = H^{\cup}\,;K = G$.

The functions of C are defined as follows. For $f\colon s_1 \dots s_n \to r$ in Σ, let $\operatorname{dom} C_f$ be the set

$$\begin{aligned}
&\{\, \langle p_1,\dots,p_n\rangle \in \textstyle\prod\langle C_{s_1},\dots,C_{s_n}\rangle \mid \\
&\quad \langle p_1,\dots,p_n\rangle \in \textstyle\prod\langle C^{\{0,1\}}_{s_1},\dots,C^{\{0,1\}}_{s_n}\rangle \\
&\qquad \wedge \langle H_{s_1}p_1,\dots,H_{s_n}p_n\rangle \in \operatorname{dom} B_f \\
&\quad \vee \langle p_1,\dots,p_n\rangle \in \textstyle\prod\langle C^{\{0,2\}}_{s_1},\dots,C^{\{0,2\}}_{s_n}\rangle \setminus \prod\langle C^{\{0\}}_{s_1},\dots,C^{\{0\}}_{s_n}\rangle \\
&\qquad \wedge \langle K_{s_1}p_1,\dots,K_{s_n}p_n\rangle \in \operatorname{dom} A_f \\
&\qquad \wedge \big(r \in V \implies A_f(K_{s_1}p_1,\dots,K_{s_n}p_n) \in B_r\big)\,\},
\end{aligned}$$

and if $\langle p_1,\dots,p_n\rangle$ is a member of this set, then

- if $r \in V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle C^{\{0,1\}}_{s_1},\dots,C^{\{0,1\}}_{s_n}\rangle$:

$$C_f(p_1,\dots,p_n) := B_f(H_{s_1}p_1,\dots,H_{s_n}p_n)$$

- if $r \in V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle C^{\{0,2\}}_{s_1},\dots,C^{\{0,2\}}_{s_n}\rangle \setminus \prod\langle C^{\{0\}}_{s_1},\dots,C^{\{0\}}_{s_n}\rangle$:

$$C_f(p_1,\dots,p_n) := A_f(K_{s_1}p_1,\dots,K_{s_n}p_n)$$

- if $r \in S \setminus V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle C^{\{0\}}_{s_1},\dots,C^{\{0\}}_{s_n}\rangle$:

$$C_f(p_1,\dots,p_n) := \big\langle 0,\, \big(B_f(H_{s_1}p_1,\dots,H_{s_n}p_n),\, A_f(K_{s_1}p_1,\dots,K_{s_n}p_n)\big)\big\rangle$$

- if $r \in S \setminus V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle C^{\{0,1\}}_{s_1},\dots,C^{\{0,1\}}_{s_n}\rangle \setminus \prod\langle C^{\{0\}}_{s_1},\dots,C^{\{0\}}_{s_n}\rangle$:

$$C_f(p_1,\dots,p_n) := \big\langle 1,\, B_f(H_{s_1}p_1,\dots,H_{s_n}p_n)\big\rangle$$

- if $r \in S \setminus V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle C^{\{0,2\}}_{s_1},\dots,C^{\{0,2\}}_{s_n}\rangle \setminus \prod\langle C^{\{0\}}_{s_1},\dots,C^{\{0\}}_{s_n}\rangle$:

$$C_f(p_1,\dots,p_n) := \big\langle 2,\, A_f(K_{s_1}p_1,\dots,K_{s_n}p_n)\big\rangle.$$

The functions of D are defined as follows. For $f\colon s_1 \dots s_n \to r$ in Σ, let $\operatorname{dom} D_f$ be the set

$$\begin{aligned}
&\{\, \langle p_1,\dots,p_n\rangle \in \textstyle\prod\langle D_{s_1},\dots,D_{s_n}\rangle \mid \\
&\quad \langle p_1,\dots,p_n\rangle \in \textstyle\prod\langle D^{\{0,2\}}_{s_1},\dots,D^{\{0,2\}}_{s_n}\rangle \\
&\qquad \wedge \langle K_{s_1}p_1,\dots,K_{s_n}p_n\rangle \in \operatorname{dom} A_f
\end{aligned}$$

$$\vee\ \langle p_1,\dots,p_n\rangle \in \prod\langle D_{s_1}^{\{0,1\}},\dots,D_{s_n}^{\{0,1\}}\rangle \setminus \prod\langle D_{s_1}^{\{0\}},\dots,D_{s_n}^{\{0\}}\rangle$$
$$\wedge\ \langle H_{s_1}p_1,\dots,H_{s_n}p_n\rangle \in \operatorname{dom} B_f\,\},$$

and if $\langle p_1,\dots,p_n\rangle$ is a member of this set, then

- if $r \in V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle D_{s_1}^{\{0,2\}},\dots,D_{s_n}^{\{0,2\}}\rangle$:

$$D_f(p_1,\dots,p_n) := A_f(K_{s_1}p_1,\dots,K_{s_n}p_n)$$

- if $r \in V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle D_{s_1}^{\{0,1\}},\dots,D_{s_n}^{\{0,1\}}\rangle \setminus \prod\langle D_{s_1}^{\{0\}},\dots,D_{s_n}^{\{0\}}\rangle$:

$$D_f(p_1,\dots,p_n) := B_f(H_{s_1}p_1,\dots,H_{s_n}p_n)$$

- if $r \in S \setminus V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle D_{s_1}^{\{0\}},\dots,D_{s_n}^{\{0\}}\rangle$, $\langle H_{s_1}p_1,\dots,H_{s_n}p_n\rangle \in \operatorname{dom} B_f$:

$$D_f(p_1,\dots,p_n) := \big\langle 0,\ \big(B_f(H_{s_1}p_1,\dots,H_{s_n}p_n),\ A_f(K_{s_1}p_1,\dots,K_{s_n}p_n)\big)\big\rangle$$

- if $r \in S \setminus V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle D_{s_1}^{\{0,2\}},\dots,D_{s_n}^{\{0,2\}}\rangle$,
  $\neg\big(\langle p_1,\dots,p_n\rangle \in \prod\langle D_{s_1}^{\{0\}},\dots,D_{s_n}^{\{0\}}\rangle \wedge \langle H_{s_1}p_1,\dots,H_{s_n}p_n\rangle \in \operatorname{dom} B_f\big)$:

$$D_f(p_1,\dots,p_n) := \big\langle 2,\ A_f(K_{s_1}p_1,\dots,K_{s_n}p_n)\big\rangle$$

- if $r \in S \setminus V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle D_{s_1}^{\{0,1\}},\dots,D_{s_n}^{\{0,1\}}\rangle \setminus \prod\langle D_{s_1}^{\{0\}},\dots,D_{s_n}^{\{0\}}\rangle$:

$$D_f(p_1,\dots,p_n) := \big\langle 1,\ B_f(H_{s_1}p_1,\dots,H_{s_n}p_n)\big\rangle.$$

It is easily checked that these definitions correctly define algebras—mainly one has to check that for all argument tuples in the explicitly given domains of C_f and D_f the value of C_f and D_f is well-defined; *i. e.*, that exactly one of the five cases is applicable, and that the functions A_f and B_f are applied only to arguments in their domains.

It remains to be checked that H and K are indeed V-abstraction functions, and that C is a weak subalgebra of D.

First, we check that $H\colon C \overset{\twoheadrightarrow}{\underset{V}{\rightrightarrows}} B$, using Proposition 4.5.2. Obviously, H consists of surjective partial functions, the visible sorts of C and B are the same, and

H is the identity on these sorts. If $f\colon s_1 \dots s_n \to r$ in Σ and $p_i \in \operatorname{dom} H_{s_i}$ for $i \in \{1,\dots,n\}$, then $\langle p_1,\dots,p_n\rangle \in \prod\langle C_{s_1}^{\{0,1\}},\dots,C_{s_n}^{\{0,1\}}\rangle$, and hence

$$\langle p_1,\dots,p_n\rangle \in \operatorname{dom} C_f \iff \langle H_{s_1}p_1,\dots,H_{s_n}p_n\rangle \in \operatorname{dom} B_f,$$

and if both sides of this equivalence are true, then

- if $r \in V$:

$$\begin{aligned} H_r(C_f(p_1,\dots,p_n)) &= H_r(B_f(H_{s_1}p_1,\dots,H_{s_n}p_n)) \\ &= B_f(H_{s_1}p_1,\dots,H_{s_n}p_n) \end{aligned}$$

- if $r \in S \setminus V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle C_{s_1}^{\{0\}},\dots,C_{s_n}^{\{0\}}\rangle$:

$$\begin{aligned} &H_r(C_f(p_1,\dots,p_n)) \\ &\quad = H_r\langle 0,\ (B_f(H_{s_1}p_1,\dots,H_{s_n}p_n),\ A_f(K_{s_1}p_1,\dots,K_{s_n}p_n))\rangle \\ &\quad = B_f(H_{s_1}p_1,\dots,H_{s_n}p_n) \end{aligned}$$

- if $r \in S \setminus V$, $\langle p_1,\dots,p_n\rangle \in \prod\langle C_{s_1}^{\{0,1\}},\dots,C_{s_n}^{\{0,1\}}\rangle \setminus \prod\langle C_{s_1}^{\{0\}},\dots,C_{s_n}^{\{0\}}\rangle$:

$$\begin{aligned} H_r(C_f(p_1,\dots,p_n)) &= H_r\langle 1,\ B_f(H_{s_1}p_1,\dots,H_{s_n}p_n)\rangle \\ &= B_f(H_{s_1}p_1,\dots,H_{s_n}p_n). \end{aligned}$$

Thus, H is a V-abstraction function from C to B.

Second, we check that $K\colon D \overset{V}{\twoheadrightarrow} A$, again using Proposition 4.4.4. Obviously, K consists of surjective partial functions, the visible sorts of D and A are the same, and K is the identity on these sorts. If $f\colon s_1 \dots s_n \to r$ in Σ and $p_i \in \operatorname{dom} K_{s_i}$ for $i \in \{1,\dots,n\}$, then $\langle p_1,\dots,p_n\rangle \in \prod\langle D_{s_1}^{\{0,2\}},\dots,D_{s_n}^{\{0,2\}}\rangle$, and hence

$$\langle p_1,\dots,p_n\rangle \in \operatorname{dom} D_f \iff \langle K_{s_1}p_1,\dots,K_{s_n}p_n\rangle \in \operatorname{dom} A_f,$$

and if both sides of this equivalence are true, then

- if $r \in V$:

$$\begin{aligned} K_r(D_f(p_1,\dots,p_n)) &= K_r(A_f(K_{s_1}p_1,\dots,K_{s_n}p_n)) \\ &= A_f(K_{s_1}p_1,\dots,K_{s_n}p_n) \end{aligned}$$

- if $r \in S \setminus V$, $\langle p_1, \ldots, p_n \rangle \in \prod \langle D_{s_1}^{\{0\}}, \ldots, D_{s_n}^{\{0\}} \rangle$, $\langle H_{s_1} p_1, \ldots, H_{s_n} p_n \rangle \in \operatorname{dom} B_f$:

$$\begin{aligned} & K_r(D_f(p_1, \ldots, p_n)) \\ & \qquad = K_r\langle 0, \, (B_f(H_{s_1} p_1, \ldots, H_{s_n} p_n), \, A_f(K_{s_1} p_1, \ldots, K_{s_n} p_n)) \rangle \\ & \qquad = A_f(K_{s_1} p_1, \ldots, K_{s_n} p_n) \end{aligned}$$

- if $r \in S \setminus V$, $\neg\big(\langle p_1, \ldots, p_n \rangle \in \prod \langle D_{s_1}^{\{0\}}, \ldots, D_{s_n}^{\{0\}} \rangle \wedge \langle H_{s_1} p_1, \ldots, H_{s_n} p_n \rangle \in \operatorname{dom} B_f\big)$:

$$\begin{aligned} K_r(D_f(p_1, \ldots, p_n)) &= K_r\langle 2, \, A_f(K_{s_1} p_1, \ldots, K_{s_n} p_n) \rangle \\ &= A_f(K_{s_1} p_1, \ldots, K_{s_n} p_n). \end{aligned}$$

Thus, K is a V-abstraction function from D to A.

To show that C is a weak subalgebra of D, we have to show that J, which was defined as the S-sorted inclusion from C/S to D/S, is a correspondence (and thus a homomorphism) from C to D. If $f\colon s_1 \ldots s_n \to r$ in Σ and $\langle p_1, \ldots, p_n \rangle \in \operatorname{dom} C_f$, then

- if $r \in V$, $\langle p_1, \ldots, p_n \rangle \in \prod \langle C_{s_1}^{\{0,1\}}, \ldots, C_{s_n}^{\{0,1\}} \rangle$, then
  - if $\langle p_1, \ldots, p_n \rangle \in \prod \langle C_{s_1}^{\{0\}}, \ldots, C_{s_n}^{\{0\}} \rangle$, which is a subset of $\prod \langle D_{s_1}^{\{0\}}, \ldots, D_{s_n}^{\{0\}} \rangle$, then $(H_{s_i} p_i, \, K_{s_i} p_i) \in G_{s_i}$ for $i \in \{1, \ldots, n\}$, hence

$$\begin{aligned} & \big(C_f(p_1, \ldots, p_n), \, D_f(p_1, \ldots, p_n)\big) \\ & \qquad = \big(B_f(H_{s_1} p_1, \ldots, H_{s_n} p_n), \, A_f(K_{s_1} p_1, \ldots, K_{s_n} p_n)\big) \\ & \qquad \in G_r, \end{aligned}$$

    and since $r \in V$, G_r is the inclusion from B_r to A_r, and thus $C_f(p_1, \ldots, p_n) = D_f(p_1, \ldots, p_n)$,
  - if $\langle p_1, \ldots, p_n \rangle \in \prod \langle C_{s_1}^{\{0,1\}}, \ldots, C_{s_n}^{\{0,1\}} \rangle \setminus \prod \langle C_{s_1}^{\{0\}}, \ldots, C_{s_n}^{\{0\}} \rangle$, which is a subset of $\prod \langle D_{s_1}^{\{0,1\}}, \ldots, D_{s_n}^{\{0,1\}} \rangle \setminus \prod \langle D_{s_1}^{\{0\}}, \ldots, D_{s_n}^{\{0\}} \rangle$, then

$$\begin{aligned} C_f(p_1, \ldots, p_n) &= B_f(H_{s_1} p_1, \ldots, H_{s_n} p_n) \\ &= D_f(p_1, \ldots, p_n); \end{aligned}$$

- if $r \in V$, $\langle p_1, \ldots, p_n \rangle \in \prod\langle C_{s_1}^{\{0,2\}}, \ldots, C_{s_n}^{\{0,2\}} \rangle \setminus \prod\langle C_{s_1}^{\{0\}}, \ldots, C_{s_n}^{\{0\}} \rangle$, which is a subset of $\prod\langle D_{s_1}^{\{0,2\}}, \ldots, D_{s_n}^{\{0,2\}} \rangle$, then

$$C_f(p_1, \ldots, p_n) = A_f(K_{s_1} p_1, \ldots, K_{s_n} p_n) = D_f(p_1, \ldots, p_n),$$

- if $r \in S \setminus V$, $\langle p_1, \ldots, p_n \rangle \in \prod\langle C_{s_1}^{\{0\}}, \ldots, C_{s_n}^{\{0\}} \rangle$, which is a subset of $\prod\langle D_{s_1}^{\{0\}}, \ldots, D_{s_n}^{\{0\}} \rangle$, then $\langle H_{s_1} p_1, \ldots, H_{s_n} p_n \rangle \in \operatorname{dom} B_f$, and hence

$$\begin{aligned} C_f(p_1, \ldots, p_n) &= \langle 0, \langle B_f(H_{s_1} p_1, \ldots, H_{s_n} p_n), A_f(K_{s_1} p_1, \ldots, K_{s_n} p_n) \rangle \rangle \\ &= D_f(p_1, \ldots, p_n), \end{aligned}$$

- if $r \in S \setminus V$, $\langle p_1, \ldots, p_n \rangle \in \prod\langle C_{s_1}^{\{0,1\}}, \ldots, C_{s_n}^{\{0,1\}} \rangle \setminus \prod\langle C_{s_1}^{\{0\}}, \ldots, C_{s_n}^{\{0\}} \rangle$, which is a subset of $\prod\langle D_{s_1}^{\{0,1\}}, \ldots, D_{s_n}^{\{0,1\}} \rangle \setminus \prod\langle D_{s_1}^{\{0\}}, \ldots, D_{s_n}^{\{0\}} \rangle$, then

$$\begin{aligned} C_f(p_1, \ldots, p_n) &= \langle 1, B_f(H_{s_1} p_1, \ldots, H_{s_n} p_n) \rangle \\ &= D_f(p_1, \ldots, p_n), \end{aligned}$$

- if $r \in S \setminus V$, $\langle p_1, \ldots, p_n \rangle \in \prod\langle C_{s_1}^{\{0,2\}}, \ldots, C_{s_n}^{\{0,2\}} \rangle \setminus \prod\langle C_{s_1}^{\{0\}}, \ldots, C_{s_n}^{\{0\}} \rangle$, which is a subset of $\prod\langle D_{s_1}^{\{0,2\}}, \ldots, D_{s_n}^{\{0,2\}} \rangle \setminus \prod\langle D_{s_1}^{\{0\}}, \ldots, D_{s_n}^{\{0\}} \rangle$, then

$$\begin{aligned} C_f(p_1, \ldots, p_n) &= \langle 2, A_f(K_{s_1} p_1, \ldots, K_{s_n} p_n) \rangle \\ &= D_f(p_1, \ldots, p_n). \end{aligned}$$

Thus, C is a weak subalgebra of D, and so the first part of the lemma has been proved.

Now assume that in addition, G is a strong correspondence. We show that $C = D$. We know already that C is a weak subalgebra of D, and obviously, C and D have the same carriers. It therefore suffices to show that definedness of a function in D for some arguments implies its definedness in C for the same arguments; in other words, that $\operatorname{dom} D_f \subseteq \operatorname{dom} C_f$.

Let $f\colon s_1 \ldots s_n \to r$ in Σ, and let $\langle p_1, \ldots, p_n \rangle \in \operatorname{dom} D_f$.

- if $\langle p_1, \ldots, p_n \rangle \in \prod\langle D_{s_1}^{\{0,2\}}, \ldots, D_{s_n}^{\{0,2\}} \rangle$ and $\langle K_{s_1} p_1, \ldots, K_{s_n} p_n \rangle \in \operatorname{dom} A_f$, then

- if $\langle p_1,\dots,p_n\rangle \in \prod\langle D_{s_1}^{\{0\}},\dots,D_{s_n}^{\{0\}}\rangle$, then $(H_{s_i}p_i,\ K_{s_i}p_i) \in G_{s_i}$ for $i \in \{1,\dots,n\}$, hence $\langle H_{s_1}p_1,\dots,H_{s_n}p_n\rangle \in \operatorname{dom} B_f$, and hence $\langle p_1,\dots,p_n\rangle \in \operatorname{dom} C_f$,
- if $\langle p_1,\dots,p_n\rangle \in \prod\langle D_{s_1}^{\{0,2\}},\dots,D_{s_n}^{\{0,2\}}\rangle \setminus \prod\langle D_{s_1}^{\{0\}},\dots,D_{s_n}^{\{0\}}\rangle$, then $r \in V$ implies $A_f(K_{s_1}p_1,\dots,K_{s_n}p_n) \in A_r = B_r$, and hence $\langle p_1,\dots,p_n\rangle \in \operatorname{dom} C_f$,

- if $\langle p_1,\dots,p_n\rangle \in \prod\langle D_{s_1}^{\{0,1\}},\dots,D_{s_n}^{\{0,1\}}\rangle \setminus \prod\langle D_{s_1}^{\{0\}},\dots,D_{s_n}^{\{0\}}\rangle$ and $\langle H_{s_1}p_1,\dots,H_{s_n}p_n\rangle \in \operatorname{dom} B_f$, then $\langle p_1,\dots,p_n\rangle \in \operatorname{dom} C_f$.

Hence D is equal to C, and the lemma has been proved. □

Proof of Theorem 5.3.6. Let $\langle Q,R\rangle$ be a cell of signature $\langle\langle\Sigma,V\rangle,\langle\Sigma^\bullet,V^\bullet\rangle\rangle$ in the institution $\langle$**TSig, TIncl, TAlg**$\rangle$.

Suppose first that $\langle Q,R\rangle$ is stable for behavioural equivalence, *i. e.*, that $\langle Q,R\rangle$ is a universal implementation of itself with respect to behavioural equivalence.

To see that $\langle Q,R\rangle$ weakly extends abstraction functions and converse abstraction functions, let $A \in Q$ and $J\colon A' \underset{V}{\Longrightarrow\!\!\!\!\times} A$ be such that $J\colon A' \underset{V}{\twoheadrightarrow\!\!\!\!\twoheadrightarrow} A$ or $J^\cup\colon A \underset{V}{\twoheadrightarrow\!\!\!\!\twoheadrightarrow} A'$. Since A is a base for $\langle Q,R\rangle$, since $A' \lesssim_V A$ and since $\langle Q,R\rangle$ is a universal implementation of itself, it follows that A' is a base for $\langle Q,R\rangle$, *i. e.*, $A' = A'/E \in Q$, and that there exists a result of $\langle Q,R\rangle$ on A'.

Now let B' be any result of $\langle Q,R\rangle$ on A'. As in the proof of Theorem 5.2.4, one shows that there exists a strong correspondence $K\colon B' \underset{V\cup V^\bullet}{\Longrightarrow\!\!\!\!\times} B$ satisfying $K/\langle\Sigma,V\rangle = J$ (Lemma 5.2.5 applies if either $J\colon A' \underset{V}{\twoheadrightarrow\!\!\!\!\twoheadrightarrow} A$ or $J^\cup\colon A \underset{V}{\twoheadrightarrow\!\!\!\!\twoheadrightarrow} A'$). It follows that $\langle Q,R\rangle$ weakly extends abstraction functions and converse abstraction functions.

Conversely, suppose that $\langle Q,R\rangle$ weakly extends abstraction functions and converse abstraction functions. We show that $\langle Q,R\rangle$ extends strong correspondences.

Let $A \in Q$, and let $G\colon A' \underset{V}{\Longrightarrow\!\!\!\!\times} A$ be a strong V-correspondence. By Lemma 5.3.7, we can form a diagram

$$A' \underset{V}{\overset{H}{\twoheadleftarrow\!\!\!\!\twoheadrightarrow}} C \underset{V}{\overset{K}{\twoheadrightarrow\!\!\!\!\twoheadrightarrow}} A$$

such that H and K are V-abstraction functions and $G = H^{\cup}\,;K$.

Since $A \in Q$ and $\langle Q, R\rangle$ weakly extends abstraction functions, $C \in Q$ also. Since $\langle Q, R\rangle$ weakly extends converse abstraction functions, it follows that $A' \in Q$ and that there exists a result of $\langle Q, R\rangle$ on A'.

Now let B' be any result of $\langle Q, R\rangle$ on A. Since $\langle Q, R\rangle$ weakly extends converse abstraction functions and $H^{\cup}$ is one, we can pick a strong correspondence $H^{\bullet}$: $B' \underset{V\cup V^{\bullet}}{\Longrightarrow\!\!\!\!\times} D$ such that D is a result of $\langle Q, R\rangle$ on C and $H^{\bullet}/\langle \Sigma, V\rangle = H^{\cup}$.

Since $\langle Q, R\rangle$ weakly extends abstraction functions, we can pick a strong correspondence $K^{\bullet}$: $D \underset{V\cup V^{\bullet}}{\Longrightarrow\!\!\!\!\twoheadrightarrow} B$ such that B is a result of $\langle Q, R\rangle$ on A and $K^{\bullet}/\langle \Sigma, V\rangle = K$.

This gives us a strong correspondence $(H^{\bullet}\,;K^{\bullet})$: $B' \underset{V\cup V^{\bullet}}{\Longrightarrow\!\!\!\!\times} B$ from B' to a result B of $\langle Q, R\rangle$ on A. It satisfies

$$\begin{aligned}(H^{\bullet}\,;K^{\bullet})/\langle \Sigma, V\rangle &= (H^{\bullet}/\langle \Sigma, V\rangle)\,;(K^{\bullet}/\langle \Sigma, V\rangle)\\ &= H^{\cup}\,;K\\ &= G.\end{aligned}$$

Thus, $\langle Q, R\rangle$ extends strong correspondences. Theorem 5.1.15 yields that $\langle Q, R\rangle$ is stable for behavioural equivalence. □

As a simple application of Lemma 5.3.3, we obtain the following theorem.

5.3.8 Theorem. *A single-valued cell that is stable for behavioural inclusion is stable for behavioural equivalence.*

Proof. Let $\langle Q, R\rangle$ be a single-valued cell of signature $\langle\langle \Sigma, V\rangle, \langle \Sigma^{\bullet}, V^{\bullet}\rangle\rangle$ that is stable for behavioural inclusion and hence (Theorem 5.2.4) extends correspondences. We shall show that $\langle Q, R\rangle$ extends strong correspondences, which by Theorem 5.1.15 implies that $\langle Q, R\rangle$ is stable for behavioural equivalence.

Let $A \in Q$, and let J: $A' \underset{V}{\Longrightarrow\!\!\!\!\times} A$ be a strong correspondence. Then J: $A' \underset{V}{\longrightarrow\!\!\!\!\times} A$ and $J^{\cup}$: $A \underset{V}{\longrightarrow\!\!\!\!\times} A'$ are correspondences. Since $\langle Q, R\rangle$ extends correspondences and J is one, it follows that $A' \in Q$ and that there exists a result of $\langle Q, R\rangle$ on A'.

Let B' be a result of $\langle Q, R\rangle$ on A'. Since $\langle Q, R\rangle$ extends correspondences and J is one, we can pick a correspondence K: $B' \underset{V\cup V^{\bullet}}{\longrightarrow\!\!\!\!\times} B$ such that B is a result

of $\langle Q, R\rangle$ on A and $K/\langle\Sigma, V\rangle = J$. Since $\langle Q, R\rangle$ extends correspondences and $J^\cup$ is one, there exists a correspondence $K'\colon B \underset{V\cup V^\bullet}{\longrightarrow\!\!\!\!\times} B''$ such that B'' is a result of $\langle Q, R\rangle$ on A' and $K'/\langle\Sigma, V\rangle = J^\cup$. Because $\langle Q, R\rangle$ is single-valued, $B'' = B'$, and so $K'\colon B \underset{V\cup V^\bullet}{\longrightarrow\!\!\!\!\times} B'$.

By Lemma 5.3.3, the componentwise intersection $K \cap (K')^\cup$ is a strong $(V \cup V^\bullet)$-correspondence from B' to B, and we have

$$\begin{aligned}(K \cap (K')^\cup)/\langle\Sigma, V\rangle &= (K/\langle\Sigma, V\rangle) \cap ((K')^\cup/\langle\Sigma, V\rangle)\\ &= (K/\langle\Sigma, V\rangle) \cap (K'/\langle\Sigma, V\rangle)^\cup\\ &= J \cap (J^\cup)^\cup\\ &= J.\end{aligned}$$

Thus, $\langle Q, R\rangle$ extends strong correspondences. □

We shall now prove a converse theorem: cells that are stable for behavioural equivalence are stable for behavioural inclusion, provided that they are "monotonic".

5.3.9 Definition. A cell $\langle Q, R\rangle$ of signature $\langle\langle\Sigma, V\rangle, \langle\Sigma^\bullet, V^\bullet\rangle\rangle$ in the institution $\langle$**TSig, TIncl, TAlg**$\rangle$ is *monotonic*, if whenever $A \in Q$ and A' is a weak subalgebra of A with the same hidden carriers as A, then

$A' \in Q$,
there exists a result of $\langle Q, R\rangle$ on A', and
whenever B' is a result of $\langle Q, R\rangle$ on A',
then there exists a $(V \cup V^\bullet)$-correspondence
$K\colon B' \underset{V\cup V^\bullet}{\longrightarrow\!\!\!\!\times} B$
such that B is a result of $\langle Q, R\rangle$ on A and
$K/\langle\Sigma, V\rangle$ is the inclusion from A' to A. □

This definition is similar in structure to Definition 5.3.5: this time, the morphisms to be extended are inclusions between algebras with the same hidden carriers, and the morphisms they may be extended into are correspondences.

Monotonicity is a consequence of stability for behavioural inclusion.

5.3.10 Proposition. *A cell in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$ *that is stable for behavioural inclusion is monotonic.*

Proof. A cell that is stable for behavioural inclusion extends correspondences by Theorem 5.2.4. Monotonicity is just this property applied to inclusions between algebras with the same hidden carriers. □

5.3.11 Theorem. *A cell in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$ *that is monotonic and stable for behavioural equivalence is stable for behavioural inclusion.*

Proof. Let $\langle Q, R\rangle$ be a cell of signature $\langle\langle\Sigma, V\rangle, \langle\Sigma^\bullet, V^\bullet\rangle\rangle$ in the institution $\langle$**TSig, TIncl, TAlg**$\rangle$, and assume that $\langle Q, R\rangle$ is monotonic and stable for behavioural equivalence. By Theorem 5.3.6, $\langle Q, R\rangle$ weakly extends abstraction functions and converse abstraction functions.

We show that $\langle Q, R\rangle$ extends correspondences. Let $A \in Q$, and let $G\colon A' \xrightarrow[V]{} A$ be a V-correspondence (*i. e.*, a representation morphism in $\mathbf{Corr}\langle\Sigma, V\rangle$).

By Lemma 5.3.7, we can form a diagram

$$A' \overset{H}{\underset{V}{\Leftarrow\!\!\!\!\!\!\rightarrow}} C \overset{J}{\hookrightarrow} D \overset{K}{\underset{V}{\Rightarrow\!\!\!\!\!\!\rightarrow}} A$$

such that H and K are strong partial V-homomorphisms, C is a weak subalgebra of D with the same hidden carriers, J is the inclusion homomorphism, and $G = H^\cup ; J ; K$. In particular, $H^\cup$ and K are strong correspondences, *i. e.*, representation morphisms in $\mathbf{SCorr}\langle\Sigma, V\rangle$.

Since $A \in Q$ and $\langle Q, R\rangle$ extends strong correspondences, $D \in Q$ also. By monotonicity, $C \in Q$. Since $\langle Q, R\rangle$ extends strong correspondences, it follows that $A' \in Q$ and that there exists a result of $\langle Q, R\rangle$ on A'.

If B' is a result of $\langle Q, R\rangle$ on A', then (since $\langle Q, R\rangle$ extends strong correspondences) there exists a strong $(V \cup V^\bullet)$-correspondence $H^\bullet : B' \underset{V\cup V^\bullet}{\Longrightarrow} E$ such that E is a result of $\langle Q, R\rangle$ on C and $H^\bullet/\langle\Sigma, V\rangle = H^\cup$. By monotonicity, there exists a $(V \cup V^\bullet)$-correspondence $J^\bullet\colon E \underset{V\cup V^\bullet}{\longrightarrow} F$ such that F is a result of $\langle Q, R\rangle$ on D and $J^\bullet/\langle\Sigma, V\rangle = J$. Finally, since $\langle Q, R\rangle$ extends strong correspondences, there exists a strong $(V \cup V^\bullet)$-correspondence $K^\bullet\colon F \underset{V\cup V^\bullet}{\Longrightarrow} B$ such that B is a

result of $\langle Q, R\rangle$ on A and $K^\bullet/\langle\Sigma, V\rangle = K$. These three correspondences form the diagram

$$B' \overset{H^\bullet}{\underset{V\cup V^\bullet}{\Longrightarrow\!\!\!\!\times}} E \overset{J^\bullet}{\underset{V\cup V^\bullet}{\longrightarrow\!\!\!\!\times}} F \overset{K^\bullet}{\underset{V\cup V^\bullet}{\Longrightarrow\!\!\!\!\times}} B.$$

The composition $G^\bullet := H^\bullet; J^\bullet; K^\bullet$ of this diagram is a $(V \cup V^\bullet)$-correspondence from B' to a result B of $\langle Q, R\rangle$ on A, and

$$\begin{aligned} G^\bullet/\langle\Sigma,V\rangle &= (H^\bullet/\langle\Sigma,V\rangle)\,;(J^\bullet/\langle\Sigma,V\rangle)\,;(K^\bullet/\langle\Sigma,V\rangle) \\ &= H^\cup\,;J\,;K \\ &= G. \end{aligned}$$

Hence $\langle Q, R\rangle$ extends correspondences, and, by Theorem 5.1.15, is stable for behavioural inclusion. □

Combining Theorem 5.3.8 and Theorem 5.3.11, we obtain the following corollary.

5.3.12 Theorem. *For cells in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$ *that are single-valued and monotonic, the stability notions for behavioural equivalence and behavioural inclusion are equivalent.* □

Since program modules in concrete programming notations may reasonably be expected to be single-valued and monotonic, the two stability notions are equivalent for most practical purposes.

An example of a programming notation that would allow only modules to be defined that are stable for behavioural equivalence can be derived from the language L investigated by Nipkow in connection with nondeterministic data types [Nipkow 86]. It was remarked earlier that modules consisting of function definitions in this language extend simulations, and that the simulations between partial many-sorted algebras are just the strong correspondences (cf. page 281–283). By removing the nondeterministic "angelic choice" operator from L, we obtain a deterministic language L' featuring application, local binding (**let**), lists, conditional, and recursive function definitions. Function definitions in L' over partial many-sorted algebras define partial functions, and thus a module consisting of

function definitions in L' is a module in $\langle$**TSig, TIncl, TAlg**$\rangle$, because its result on a partial algebra is again a partial algebra. Since modules written in L extend simulations, so do modules written in L', and since the simulations between partial algebras are just the strong correspondences, modules consisting of function definitions written in L' extend strong correspondences and are therefore stable for behavioural equivalence. The language L' does not contain any data type constructors, but we may add to it constructors whose instances (which are modules) extend correspondences. It is an easy exercise to prove that standard type constructors, such as product, union, or "list" extend correspondences.

5.4 Stability for Standard Representation

This section analyses the stability notion of the standard representation relation between partial algebras and compares it with the stability notions of the behavioural representation concepts.

As we shall see, stability for standard representation implies stability for behavioural equivalence and behavioural inclusion under some not unduly strong conditions. However, a cell will be shown that is stable for behavioural inclusion and equivalence, but not stable for standard representation. Thus, not only is the simple implementation concept for abstraction functions more restrictive than those for correspondences and strong correspondences (Prop. 5.1.7 and Example 4.5.5), but standard representation also excludes more cells on the grounds that they are not stable.

First, we define a representation functor for standard representation.

5.4.1 Definition. Let **AFun**: $\mathbf{TSig}^{\mathrm{op}} \to \mathbf{LCat}$ be the functor mapping $\langle \Sigma, V\rangle \in |\mathbf{TSig}|$ to the category of V-abstraction functions between small Σ-algebras (cf. Prop. 4.4.5), and a tagged signature morphism σ: $\langle \Sigma, V\rangle \to \langle \Sigma', V'\rangle$ to the functor $\mathbf{AFun}(\sigma^{\mathrm{op}})$: $\mathbf{AFun}\langle \Sigma', V'\rangle \to \mathbf{AFun}\langle \Sigma, V\rangle$ which is the restriction of $\mathbf{SCorr}(\sigma^{\mathrm{op}})$ (or $\mathbf{Corr}(\sigma^{\mathrm{op}})$) to abstraction functions. □

5.4.2 Proposition. *The functor* **AFun** *is a representation functor in* ⟨**TSig**, **TIncl**, **TAlg**⟩, *and its representation relation is standard representation.*

Proof. For each $\langle \Sigma, V \rangle \in |\mathbf{TSig}|$, the category $\mathbf{AFun}\langle \Sigma, V \rangle$ is a subcategory of $\mathbf{SCorr}\langle \Sigma, V \rangle$ (and of $\mathbf{Corr}\langle \Sigma, V \rangle$) with the same objects.

For every σ: $\langle \Sigma, V \rangle \to \langle \Sigma', V' \rangle$ in **TSig**, $\mathbf{AFun}(\sigma^{\mathrm{op}})$ is a functor from $\mathbf{AFun}\langle \Sigma', V' \rangle$ to $\mathbf{AFun}\langle \Sigma, V \rangle$, because it is a restriction of $\mathbf{SCorr}(\sigma^{\mathrm{op}})$, and that functor maps V'-abstraction functions to V-abstraction functions, as is easily seen.

It is now trivial to verify the axioms (a) and (b) of Definition 5.1.1.

Axiom (c), pair-completeness, will now be derived from the analogous property of **SCorr**.

If $\langle \Sigma_0, V_0 \rangle$ and $\langle \Sigma_1, V_1 \rangle$ are compatible small tagged signatures, and

$$J\colon A \underset{V_0}{\Rightarrow\!\!\!\!\!\rightarrow} B \quad \text{and} \quad K\colon C \underset{V_1}{\Rightarrow\!\!\!\!\!\rightarrow} D$$

are morphisms in $\mathbf{AFun}\langle \Sigma_0, V_0 \rangle$ and $\mathbf{AFun}\langle \Sigma_1, V_1 \rangle$ respectively such that

$$J/(\langle \Sigma_0, V_0 \rangle \sqcap \langle \Sigma_1, V_1 \rangle) = K/(\langle \Sigma_0, V_0 \rangle \sqcap \langle \Sigma_1, V_1 \rangle),$$

then by pair-completeness of **SCorr**, there exists a strong correspondence

$$L\colon A \sqcup C \underset{V_0 \cup V_1}{\Longrightarrow\!\!\!\!\!\times} B \sqcup D$$

such that $L/\langle \Sigma_0, V_0 \rangle = J$ and $L/\langle \Sigma_1, V_1 \rangle = K$.

The strong correspondence L is an abstraction function, because with S_0 and S_1 the sort sets of Σ_0 and Σ_1, we have for every $s \in S_0 \cup S_1$:

- if $s \in S_0$, then

  $L_s = J_s$ is a partial surjective function from A_s to B_s,

  hence from $(A \sqcup C)_s$ to $(B \sqcup D)_s$.

- if $s \in S_1$, then

  $L_s = K_s$ is a partial surjective function from C_s to D_s,

  hence from $(A \sqcup C)_s$ to $(B \sqcup D)_s$.

In either case, L_s is a partial surjective function from $(A \sqcup C)_s$ to $(B \sqcup D)_s$, and thus L: $A \sqcup C \overset{\Rightarrow}{\underset{V_0 \cup V_1}{}} B \sqcup D$, which means that L is the desired morphism in $\mathbf{AFun}(\langle \Sigma_0, V_0 \rangle \sqcup \langle \Sigma_1, V_1 \rangle)$.

We have shown that **AFun** is a representation functor in ⟨**TSig**, **TIncl**, **TAlg**⟩. By Definition 4.5.1, the representation relation of **AFun** is standard representation. □

Due to this theorem, the concepts of Section 5.1 become applicable, and we obtain a "simple implementation" concept for abstraction functions and a "stability" concept for standard representation.

5.4.3 Example. In Example 4.5.3, a correctness proof of the module M_{STORE} from the *dictionary* program development was given using the standard technique for data representation correctness proofs. It will now be shown that this proof can be viewed as a proof that the module M_{STORE} is a simple implementation of its specification cell $\mathcal{M}_{STORE}$ with respect to abstraction functions.

The specification cell $\mathcal{M}_{STORE}$ consists of the interfaces

$$\mathcal{Q}_{STORE} = I_{ITEM} \wedge I_{LISTITEM}$$
$$\mathcal{R}_{STORE} = I_{INSERT} \wedge I_{MIN}.$$

In Example 4.5.3, a $\{bool, item, listitem\}$-abstraction function h was constructed from the result B of M_{STORE} on an arbitrary model C of $\mathcal{Q}_{STORE}$ to a result A of M_{STORE} on the same model C. Since $\{bool, item, listitem\}$ is just the sort set of the environment signature $\mathrm{Sig}(\mathcal{Q}_{STORE})$, the reduct of h to this signature is the identity. Hence h is the representation morphism required by the simple implementation criterion (with respect to abstraction functions). Together with the simple check that each algebra in the abstract import interface $\mathcal{Q}_{STORE}$ is a base for M_{STORE}, this proves that M_{STORE} is a simple implementation of $\mathcal{M}_{STORE}$ with respect to abstraction functions. □

We shall now analyse the stability notion for standard representation and compare it with the ones for behavioural inclusion and behavioural equivalence.

5.4.4 Theorem. *A cell in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$ *is stable for standard representation, if and only if it extends abstraction functions.*

Proof. The proof is precisely analogous to the proof of Theorem 5.2.4; just replace "behavioural inclusion" by "standard representation", and "correspondence" by "abstraction function" (Lemma 5.2.5 still applies). □

It will now be shown that stability for standard representation implies stability for behavioural equivalence, provided that the cell in question is single-valued and that its requirement interface Q is "closed under abstraction functions".

5.4.5 Definition. An interface Q of signature $\langle \Sigma, V \rangle$ is *closed under abstraction functions*, if $B \in Q$ and $h\colon B \overset{}{\twoheadrightarrow}_{V} A$ imply $A \in Q$. □

5.4.6 Theorem. *A cell in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$ *that is single-valued, whose requirement interface is closed under abstraction functions, and that is stable for standard representation is stable for behavioural equivalence.*

The theorem will be obtained immediately from the following lemma.

5.4.7 Lemma. *A cell in* $\langle$**TSig, TIncl, TAlg**$\rangle$ *that is single-valued, whose requirement interface is closed under abstraction functions, and that weakly extends abstraction functions weakly extends converse abstraction functions.*

Proof. Let $\langle Q, R \rangle$ be a cell of signature $\langle \langle \Sigma, V \rangle, \langle \Sigma^\bullet, V^\bullet \rangle \rangle$ that satisfies the assumptions of the lemma.

Let $A \in Q$, and let $J\colon A' \overset{}{\rightarrowtail}_{V} A$ be such that $J^\cup\colon A \overset{}{\twoheadrightarrow}_{V} A'$. Since Q is closed under abstraction functions, it follows that $A' \in Q$. Since $\langle Q, R \rangle$ weakly extends abstraction functions and $J^\cup$ is one, there exists a result B of $\langle Q, R \rangle$ on A and we can pick $K\colon B \overset{}{\rightarrowtail}_{V \cup V^\bullet} B'$ such that B' is a result of $\langle Q, R \rangle$ on A' and $K/\langle \Sigma, V \rangle = J^\cup$.

There exists a result of $\langle Q, R \rangle$ on A' (namely B'). If $B^\bullet$ is a result of $\langle Q, R \rangle$ on A', then $B^\bullet = B'$ by single-valuedness, and $K^\cup\colon B^\bullet \overset{}{\rightarrowtail}_{V \cup V^\bullet} B$ is the desired extension of J, because $K/\langle \Sigma, V \rangle = J^\cup$ and so $K^\cup/\langle \Sigma, V \rangle = J$. □

Proof of Theorem 5.4.6.

Let $\langle Q, R\rangle$ be a cell satisfying the assumptions of the theorem. Since $\langle Q, R\rangle$ extends abstraction functions, it weakly extends abstraction functions. By the previous lemma, $\langle Q, R\rangle$ weakly extends converse abstraction functions, and so by Theorem 5.3.6, $\langle Q, R\rangle$ is stable for behavioural equivalence. □

As a corollary, we obtain

5.4.8 Theorem. *For cells in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$ *that are single-valued, monotonic, and whose requirement interface is closed under abstraction functions, the following implications hold:*

Stability for standard representation

$\Downarrow$

Stability for behavioural equivalence

$\Updownarrow$

Stability for behavioural inclusion.

Proof. Combine Theorem 5.3.12 and Theorem 5.4.6. □

It will now be shown that the top arrow in the previous theorem is a strict implication, because there exist cells that satisfy the three conditions of the theorem, are stable for the behavioural representation concepts, yet are not stable for standard representation. Since the counterexample presented in the proof is conceivable as a module of a programming notation, we may conclude that stability for standard representation is more restrictive than the behavioural stability notions for practical purposes.

5.4.9 Theorem. *The top implication of Theorem 5.4.8 is strict; that is, there exists a cell in the institution* $\langle$**TSig, TIncl, TAlg**$\rangle$ *that is single-valued and monotonic, whose requirement interface is closed under abstraction functions, and that is stable for the behavioural representation notions, yet that is not stable for standard representation.*

Proof. The cell to be presented as an example provides an encryption facility for data types in a machine based on B-ary words of length N ($B \geq 2$, $N \geq 1$). The possible words are identified with the numbers in $W := \{0, \ldots, B^N - 1\}$, and we assume as given an "encryption" and a "decryption" function

$$E, D\colon \prod\langle W, W\rangle \to W$$

with the property that for all "keys" $k \in W$ and "data words" $x \in W$,

$$D(E(x,k),k) = x;$$

that is, a word x that has been encrypted with the key k can be decrypted with the same key.

Let $\langle Q, R\rangle$ be the module shown in Figure 5-5. The signature of this cell will be named $\langle\langle\Sigma,\emptyset\rangle,\langle\Sigma',\emptyset\rangle\rangle$; in particular, there are no visible sorts to be considered.

The type *crypt* generated by the cell $\langle Q, R\rangle$ is an encrypted version of the argument type *item*, provided that the argument types satisfy *word* $= W$ (*i. e.*, *word* is the type of machine words) and *item* $\subseteq W$ (*i. e.*, all *item* values are machine words). This assumption is reasonable in a low-level systems programming situation, where all data are represented by machine words.

However, the cell $\langle Q, R\rangle$ provides a result also for interpretations of *item* and *word* that do not satisfy the requirement above; this is to make them fit into the institution $\langle$**TSig, TIncl, TAlg**$\rangle$, where arbitrary algebras may appear as arguments. (One could prevent this from happening by using another institution, in which all carriers of all algebras have to be subsets of the set W of machine words, and in which all signatures contain a visible sort *word* whose interpretation in all algebras is W.)

The second parameter of *enc* plays a special rôle; it may be called the "salt" (after [MT 79]). In the case that encryption is performed and the number K of B-ary digits needed to represent the values of type *item* is less than the word length N, an *item* value x is "padded" before encryption to the full word length N by prefixing it with $N - K$ digits taken from the "salt" parameter s (so that the value actually encrypted is $x + B^K(s \bmod B^{N-K})$). After decryption, the extra digits are removed again (by the operation $\_ \bmod B^K$), so that

cell

environment signature

item, *word*: **sort**

key: $\rightarrow$ *word*

defined symbols

crypt: **sort**

enc: *item word* $\rightarrow$ *crypt*

dec: *crypt* $\rightarrow$ *item*

requirements

[no requirements]

result

if $word = W$, $item \subseteq W$

then let $K := \lceil \log_B(1 + \max(item \cup \{0\})) \rceil$ be the number of B-ary digits needed for *item* $(0 \leq K \leq N)$,

$crypt = W$,

if $\langle\rangle \notin \mathrm{dom}\, key$ **then** $\mathrm{dom}\, enc = \mathrm{dom}\, dec = \emptyset$

else $enc(x, s) = E(x + B^K(s \bmod B^{N-K}), key())$,

$dec(y) = D(y, key()) \bmod B^K$

else $crypt = item$,

if $\langle\rangle \notin \mathrm{dom}\, key$ **then** $\mathrm{dom}\, enc = \mathrm{dom}\, dec = \emptyset$

else $enc(x, s) = x$,

$dec(y) = y$.

Figure 5-5: The "encryption" module $\langle Q, R \rangle$

the value retrieved is the value that was encrypted (*i. e.*, the first argument of the encryption function). The "salt" value is thus irrelevant to the observable results; however, in case that $K < N$, it allows one to create several possible encryptions for each *item* value without further cost, as the encryption and decryption functions deal with entire words anyway. This provides increased security against key search techniques that might be applied when values and their encryptions are available: While without the salt, each key produces a single encryption of each value, the number of possible encryptions of a value by a key is increased to B^{N-K} with the salt technique. This technique has been used to increase password security in the UNIX operating system [MT 79, p. 597].

Obviously, the cell $\langle Q, R\rangle$ is single-valued and consistent (hence a module), and its requirement interface is closed under abstraction functions, because it comprises all Σ-algebras.

It will now be shown that $\langle Q, R\rangle$ is stable for behavioural inclusion. By Theorem 5.3.8, it then follows that $\langle Q, R\rangle$ is stable for behavioural equivalence also; and by Proposition 5.3.10, it follows that the cell is monotonic. For the proof, Theorem 5.1.15 will be used; that is, we show that $\langle Q, R\rangle$ extends correspondences. Thus, let G: $A' \longrightarrow\!\!\!\!\times\ A$ be a correspondence between Σ-algebras, *i. e.*, an $\{item, word\}$-sorted relation such that if $\langle\rangle \in \operatorname{dom} A'_{key}$, then $\langle\rangle \in \operatorname{dom} A_{key}$ and $\langle A'_{key}(), A_{key}()\rangle \in G_{word}$.

Let B' be the result of $\langle Q, R\rangle$ on A', and let B be the result of $\langle Q, R\rangle$ on A. The desired correspondence H from B' to B is given by

$$
\begin{aligned}
H_{item} &= G_{item},\\
H_{word} &= G_{word},\\
H_{crypt} &= \{\, (B'_{enc}(x', s'),\ B_{enc}(x, s)) \mid\\
&\qquad \langle x', x\rangle \in H_{item},\ \ \langle s', s\rangle \in H_{word},\\
&\qquad \langle x', s'\rangle \in \operatorname{dom} B'_{enc},\ \ \langle x, s\rangle \in \operatorname{dom} B_{enc}\,\}.
\end{aligned}
$$

To verify that H is a correspondence from B' to B, we have to show that the three operations *key*, *enc*, and *dec* are compatible with it.

key: The compatiblity follows trivially from the fact that $H_{word} = G_{word}$ and that G is a correspondence.

enc: Let $\langle x', x\rangle \in H_{item}$ and $\langle s', s\rangle \in H_{word}$ be such that $\langle x', s'\rangle \in \operatorname{dom} B'_{enc}$. Then we must have $\langle\rangle \in \operatorname{dom} A'_{key}$ (for otherwise, $\operatorname{dom} B'_{enc}$ would be empty), hence $\langle\rangle \in \operatorname{dom} A_{key}$ and thus $\langle x, s\rangle \in \operatorname{dom} B_{enc}$ (since $\langle\rangle \in \operatorname{dom} A_{key}$ implies that B_{enc} is total). From the definition of H_{crypt}, it immediately follows that $\langle B'_{enc}(x', s'), B_{enc}(x, s)\rangle \in H_{crypt}$.

dec: We first check that if $\langle\rangle \in \operatorname{dom} A_{key}$, then for all $x \in A_{item}$ and $s \in W$:

$$B_{dec}(B_{enc}(x, s)) = x. \tag{1}$$

For, if $word = W$ and $item \subseteq W$, then (with K the number of B-ary digits needed for *item* as in the definition of $\langle Q, R\rangle$)

$$\begin{aligned} B_{dec}(B_{enc}(x, s)) &= D(E(x + B^K(s \bmod B^{N-K}),\ key()),\ key()) \bmod B^K \\ &= (x + B^K(s \bmod B^{N-K})) \bmod B^K \\ &= x \bmod B^K \\ &= x \qquad (x \in item, \text{ hence } 0 \le x < B^K), \end{aligned}$$

and otherwise,

$$B_{dec}(B_{enc}(x, s)) = B_{dec}(x) = x.$$

Of course, the same argument shows that if $\langle\rangle \in \operatorname{dom} A'_{key}$, then for all $x \in A'_{item}$ and $s \in W$:

$$B'_{dec}(B'_{enc}(x, s)) = x. \tag{1'}$$

Now let $\langle y', y\rangle \in H_{crypt}$ be such that $y' \in \operatorname{dom} B'_{dec}$. Since then $H_{crypt} \neq \emptyset$, we must have $\langle\rangle \in \operatorname{dom} A'_{key}$ (otherwise B'_{enc} would be undefined everywhere), and hence $\langle\rangle \in \operatorname{dom} A_{key}$. Furthermore, by definition of H_{crypt}, we can choose x', s', x, s such that $\langle x', x\rangle \in H_{item}$, $\langle s', s\rangle \in H_{word}$, $y' = B'_{enc}(x', s')$, and $y = B_{enc}(x, s)$. By (1) and (1'), we have

$$\begin{aligned} (B'_{dec}(y'),\ B_{dec}(y)) &= (B'_{dec}(B'_{enc}(x', s')),\ B_{dec}(B_{enc}(x, s))) \\ &= (x', x) \\ &\in H_{item}. \end{aligned}$$

It has thus been shown that H is a correspondence. Clearly, the reduct of H to Σ is the original correspondence G. Thus, $\langle Q, R\rangle$ extends correspondences.

It remains to be shown that $\langle Q, R\rangle$ is not stable for standard representation. Using Theorem 5.4.4, it suffices to show that $\langle Q, R\rangle$ does not extend abstraction functions.

Let A' and A be the Σ-algebras defined by

$$A'_{word} = A_{word} = W,$$
$$A'_{item} = W = \{0, \ldots, B^N - 1\}, \qquad A_{item} = \{0, \ldots, B^{N-1} - 1\},$$
$$A'_{key}() = A_{key}() = k \quad \text{where } k \in W \text{ is chosen arbitrarily,}$$

and let $H\colon A' \twoheadrightarrow A$ be the abstraction function defined by

$$H_{word}(w) = w \qquad \text{for } w \in A'_{word} = W,$$
$$H_{item}(x) = x \bmod B^{N-1} \qquad \text{for } x \in A'_{item} = W.$$

Clearly, H is a surjective strong partial homomorphism from A' to A.

The result algebras B' and B of $\langle Q, R\rangle$ on A' and A have the additional components defined by

$$B'_{crypt} = B_{crypt} = W$$
$$B'_{enc}(x, s) = E(x + B^N(s \bmod 1),\ k) = E(x, k)$$
$$B_{enc}(x, s) = E(x + B^{N-1}(s \bmod B),\ k)$$
$$B'_{dec}(y) = D(y) \bmod B^N = D(y)$$
$$B_{dec}(y) = D(y) \bmod B^{N-1}.$$

Now suppose that G is a correspondence from B' to B that extends H, so that $G_{word} = H_{word}$ and $G_{item} = H_{item}$. Since $\langle 0,0\rangle \in G_{item}$ and $\langle 0,0\rangle \in G_{word}$, it follows that

$$(B'_{enc}(0,0),\ B_{enc}(0,0)) = (E(0,k),\ E(0,k)) \in G_{crypt},$$

and since $\langle 0,0\rangle \in G_{item}$ and $\langle 1,1\rangle \in G_{word}$, that

$$(B'_{enc}(0,1),\ B_{enc}(0,1)) = (E(0,k),\ E(B^{N-1},k)) \in G_{crypt}.$$

We have $E(0,k) \neq E(B^{N-1},k)$, because $D(E(0,k),k) = 0$, which is different from $D(E(B^{N-1},k),k) = B^{N-1}$. Thus, G_{crypt} contains two pairs with the same first components, but different second components. Therefore G_{crypt} cannot be a partial function, and thus no partial homomorphism from B' to B can exist that extends H. It follows that $\langle Q, R \rangle$ does not extend abstraction functions and thus is not stable for standard representation. □

Note that in the counterexample presented in this proof, the fact that encryption is performed is actually irrelevant (we could use the trivial encryption functions defined by $E(x,k) = D(x,k) = x$). The encryption only serves to motivate the introduction of the "salt" parameter that does not contribute anything to the observable results. It is the fact that fewer B-ary digits of the salt are preserved when the *item* parameter requires more digits that causes the cell not to extend abstraction functions.

Figure 5-6 summarizes the relations between the stability notions that have been established in this chapter. The labels next to the arrows indicate the conditions under which the implications hold.

At the end of the previous section, it has been illustrated how to use a stability criterion in a "positive" way, namely to prove that a programming notation supports data abstraction in the sense of this thesis. In the proof of Theorem 5.4.9, it has been shown how to use the stability notion in a "negative" way, namely to prove that a certain type constructor cannot safely be used in connection with standard representation. This kind of proof is practically useful too, because it shows which constructs must not be included in a data abstraction language.

5.4.10 Example. The **array** and **set** data type constructors of PASCAL must not occur in a programming language intended to support data abstraction when the representation relation is behavioural inclusion, behavioural equivalence or standard representation.

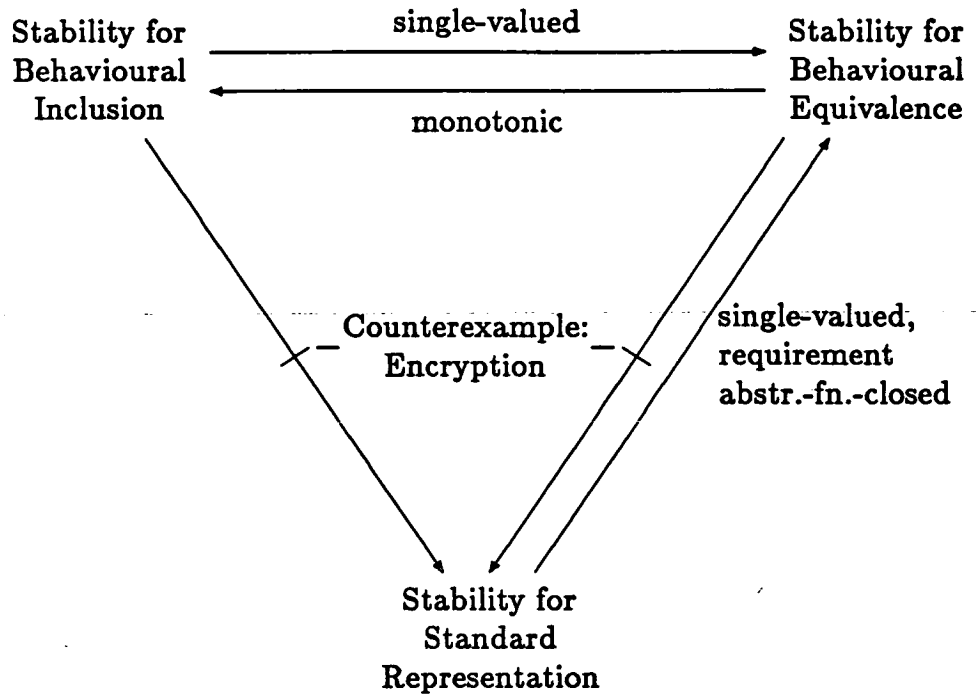


Figure 5-6: Implications between stability notions in ⟨**TSig, TIncl, TAlg**⟩

The reason is that the two constructs allow one to write the function definitions *equal1* and *equal2* that were discussed in Section 1.2 and that translate into the modules M_1 and M_2 shown in Figures 5-7 and 5-8. These modules do not extend correspondences nor weakly extend abstraction functions nor extend abstraction functions, and hence (by Theorems 5.2.4 5.3.6 and 5.4.4) are not stable for behavioural inclusion nor behavioural equivalence nor standard representation.

Consider M_1 first. The following two algebras A and A' are members of the requirement interface of M_1:

$$\begin{aligned} A_{boolean} &= A'_{boolean} = \{\mathbf{T}, \mathbf{F}\} \\ A_{true} &= A'_{true} = \mathbf{T} \\ A_{false} &= A'_{false} = \mathbf{F} \\ A_T &= \{0\} \quad A'_T = \{0, 1\}. \end{aligned}$$

M_1 = **module**

environment signature

boolean, *T*: **sort**

true, *false*: $\rightarrow$ *boolean*

defined symbols

equal1: *T T* $\rightarrow$ *boolean*

requirements

"*T* is an ordinal type"

result

function *equal1*(*x*, *y*: *T*): *boolean*;

var *a*: **array**[*T*] **of** *boolean*;

begin *a*[*x*] := *false*;

a[*y*] := *true*;

equal1 := *a*[*x*] **end**;

Figure 5-7: The PASCAL module M_1

The following relation G is a $\{boolean\}$-abstraction function G: $A' \underset{\{boolean\}}{\Longrightarrow} A$ (and hence a $\{boolean\}$-correspondence as well):

$$G_{boolean} = \mathrm{Id}(\{\mathbf{T}, \mathbf{F}\})$$
$$G_T = \{(0,0), (1,0)\}.$$

The results B' and B of M_1 on the bases A' and A have the same sorts as A' and A, hence if a correspondence from B' to B extending G did exist, it would have to be equal to G. However, G is not a correspondence from B' to B, because *equal1* is not compatible with it: We have $(0,0) \in G_T$ and $(1,0) \in G_T$, but

$$(B'_{equal1}(0,1),\ B_{equal1}(0,0)) = (\mathbf{F}, \mathbf{T}) \notin G_{boolean}.$$

Thus, there exists no correspondence (let alone a strong correspondence or abstraction function) from B' to B that extends G, hence M_1 does not extend

M_2 = **module**

environment signature

boolean, T: **sort**

defined symbols

equal2: $T\ T \rightarrow$ *boolean*

requirements

boolean = {**T**, **F**}

"T is an ordinal type"

result

function *equal2*$(x, y: T)$: *boolean*;

begin *equal2* := x **in** $[y]$ **end**;

{$[y]$ is the set with single member y,

in is the membership test}.

Figure 5-8: The PASCAL module M_2

correspondences nor weakly extend abstraction functions nor extend abstraction functions.

For M_2 the argument is precisely the same, except that from the algebras A and A' the operations *true* and *false* should be removed so that they become elements of the requirement interface of M_2. The correspondence G and the remainder of the argument are as for M_1. □

Chapter 6

Conclusions

Modular Programming and Data Abstraction

A theory of modular programming and data abstraction has been presented that explains why these design methods lead to correct programs.

Modular programming is viewed as the construction of a "structured correctness argument" (sketched in Figure 3-5), in which the correctness of a program composed of a number of modules is established by separate correctness proofs of the individual modules.

The basic entities of the theory are "cells" (Def. 3.1.10), which consists of an import (or "requirement") and an export (or "result") interface, and which represent program modules as well as program or module specifications.

The external export and import interfaces of a program under design form a cell called the "global cell" of the program development. In the structured correctness argument for a modular program, each program module is specified by a "module specification", and both modules and module specifications are cells in the theory. The module specifications must form a "decomposition" (Def. 3.2.10) of the global cell. They may be derived from a design graph recording the import and export interfaces of each module, and this automatically establishes the semantic part of the "decomposition" notion (cf. Example 3.2.11 and the discussion following it).

Every module of the program must be "correct" with respect to its specification. The correctness notion appropriate for modular programming is called "refinement" (Def. 3.1.18), and the "composability theorem of refinements" (Theorem 4.1.12) asserts that if each module of a program is a refinement of its

specification, then the result of composing the modules is a refinement of the global cell, which means that the resulting program is correct with respect to the global import and export interfaces. This theorem generalizes previous theorems by myself [Schoett 81] and by Back and Mannila [BM 84].

Data abstraction is viewed as an extension to modular programming. The requirement that the entities defined in a program satisfy the interfaces is relaxed, and it is only required that these program entities "represent" entities satisfying the interfaces. Nevertheless, the users of an interface may depend on the interface as it is and need not be concerned with the fact that they will be supplied with program entities that need not satisfy the interface.

The correctness notion appropriate for modular programming with data abstraction is called "universal implementation" (Def. 4.1.5), and it is shown that this notion can be used as the correctness notion in a structured correctness argument: the "composability theorem for universal implementations" (Thm. 4.1.7) asserts that if each module of a program is a universal implementation of its specification, then the result of composing the modules is a universal implementation of the global cell.

The universal implementation property of a cell is cumbersome to prove directly; hence universal implementation can not be proposed as a correctness criterion for practical programming. Appropriate for practice is the "simple implementation" criterion (Def. 5.1.5), which corresponds to established methods of proving the correctness of data type representations (Example 5.4.3). It requires, however, that the representation relation can be characterized by "representation morphisms" (Def. 5.1.1). The simple implementation relation is transitive and thus allows the "vertical composition" of implementation steps.

"Stable" cells are those for which the simple implementation property implies the universal implementation property (Def. 5.1.10). Hence proving the simple implementation property of stable cells is sufficient to guarantee that they are universal implementations of their specifications, which is what is needed for a valid structured correctness argument.

I suggest that stability should be regarded an essential design criterion for a programming language that is to support data abstraction: the language should

allow only stable modules to be defined. With such a language as the target language of a modular program development, programmers need only verify the simple implementation property of their modules, which means that they may use established practical proof methods.

Generality through Institutions

The theory of this thesis is based on a variant of the "institution" notion of Goguen and Burstall [GB 84]. The "signatures" and "models" of an institution can represent the "type environments" (sets of identifiers with associated type information) and the "environments" (identifiers with associated semantic values) of a programming language. By choosing appropriate institutions, the theory can be applied to modular programming in various frameworks. In particular, there is no limit to the kinds of program entities that may occur in a programming language and may be imported and exported by its modules: data types, functions, procedures, data values, data objects, labels, processes, or even modules themselves.

The "institution" notion of this thesis (Def. 2.3.5 and 2.3.9) differs from the one introduced by Goguen and Burstall in two respects. There is no notion of "sentences" in this thesis, because there is no need to consider specifications on the syntactical level—only the semantics of a specification, *i. e.*, the set of models it describes, enters the theory (as a "specification" or "interface" in Def. 3.1.6), and the language in which specifications are written is irrelevant to the theory. Second, the institutions of this thesis are equipped with a partial ordering between signatures, which allows us to combine signatures and models without linking them by explicit signature morphisms. If a signature S is included in a signature T according to the partial ordering, it means that all the program entities described by S occur in T also. There is thus an automatic identification of program entities between different signatures, which reflects the idea that the signatures reside in a common "name space", so that a name occurring in different signatures must refer to the same program entity.

The particular institution considered on the "concrete level" of this thesis, the institution of partial many-sorted algebras (Thm. 2.3.6 and 2.3.11), deals with programs in strongly typed functional programmming languages such as the typed λ-calculus [Barendregt 84, App. A], the language ML [HMM 86] without assignment, polymorphism and exceptions, or the deterministic applicative algorithmic sublanguage of CIP-L [CIP 85]. The example program development in Section 1.4 is carried out in such a language.

Generality is further achieved by considering abstract "representation relations" (Def. 4.1.1) and "representation categories" (Def. 5.1.1) in this thesis. The purpose of a representation relation is to express when one model is to be considered a correct "representation" of another one. The representation model will usually consist of program entities defined in some programming notation, while the model it represents may be an "abstract model" that satisfies the specifications of these program entities. A natural way of defining a representation relation is to require that two models in the relation must have the same "observable behaviour", which means that every program produces the same output when based on the one model as when based on the other model.

The "simple implementation" notion, which is proposed as the correctness criterion to be verified by module designers, requires that a representation relation has an associated "representation category", in which there is a "representation morphism" from each representation to the model it represents. In the institution of partial algebras (with distinguished "visible sorts"), three representation concepts were considered:

representation relation	representation morphisms
behavioural inclusion	correspondences
behavioural equivalence	strong correspondences
standard representation	abstraction functions

The association of representation relations with representation categories appears to be natural and useful. In particular, it becomes possible to formulate

the "simple implementation" concept, which reflects established practical methods for proving the correctness of data representations (Example 5.4.3).

The theory of data abstraction for nondeterministic data types recently published by Nipkow [Nipkow 86] has not influenced the design of the present theory, yet it fits perfectly into the framework of "representation relation" and "representation category" (page 281–283).

Representation Relations between Partial Algebras

It has been argued in Section 4.2 that representation relations between (total and hence between) partial algebras must be based on a set of "visible sorts" (or analogous concepts of "visible values") that must be represented identically, since otherwise they would allow useless representations such as, for example, the term algebra as a representation of every total algebra of that signature with nonempty and finite or countable carriers.

Three representation relations between partial algebras have been introduced and compared.

"Standard representation" (Section 4.5) is the representation relation associated with the "abstraction function" proof method introduced by Hoare [Hoare 72], which has formed the basis of most approaches to data representation.

"Behavioural equivalence" (Section 4.4) is based on the notion of the "observable behaviour" of an algebra, where values of the visible sorts may be input to computations, and result values of visible sort can be discriminated. The behavioural equivalence relation is characterized by the existence of a "strong correspondence" between two algebras (Def. 4.4.3 and Thm. 4.4.6), and this yields a useful practical proof method in which the usual "abstraction function" is generalized to a relation (Example 4.4.7).

An important advantage of behavioural equivalence and the proof method based on strong correspondences over the standard representation concept is that they are strictly more general (Thm. 4.5.4 and Example 4.5.5), thus allowing more representations, and that "representation bias" [Jones 80, Ch. 15] in a

specification no longer restricts the range of correct representations. This eliminates one of the main criticisms of the "abstract model" specification technique for encapsulated data types.

A second advantage of behavioural equivalence over the standard representation relation is that the stability notion for behavioural equivalence is strictly more general (for cells that are single-valued and whose requirement interface is closed under abstraction functions) than the stability notion for standard representation (Thm. 5.4.9). This means that in a programming language designed to support data abstraction (*i. e.*, a language with stable modules), more constructs are allowed when the representation relation is behavioural equivalence than when the representation relation is standard representation. Thus, behavioural equivalence is superior to standard representation in every respect except familiarity, and should be used in programming practice instead of standard representation.

"Behavioural inclusion" (Section 4.3) is a new representation relation that combines the "partial implementation" idea of Kamin and Archer [KA 84] with the behaviour idea. It is strictly more general than the "implementation" notion of Kamin and Archer (Thm. 4.3.11 and Example 4.3.12), and admits correctness proofs by means of "correspondences".

For cells that are single-valued and monotonic, the stability notions for behavioural inclusion and behavioural equivalence are equivalent to each other (Thm. 5.3.12). It will often be most useful to prove that the modules of a programming language are stable for behavioural inclusion, because this implies that they are stable for behavioural equivalence also under the natural condition that they are single-valued (Thm. 5.3.8).

Design of Programming Languages for Data Abstraction

The theory of this thesis shows that "simple implementation" can be used as the correctness notion in a modular program development, provided that the notation in which the modules are finally coded allows to define only stable modules.

I suggest that stability of modules should be regarded as an essential design criterion for a programming language intended to support data abstraction.

To verify that a programming language has stable modules and hence supports data abstraction, the following steps are necessary.

- Designing an institution whose signatures correspond to the type environments and whose models correspond to the semantic environments of the language.
- Defining a representation category. This defines an associated representation relation (which may or may not be chain-closed and hence allow to compose infinite systems). The representation relation should depend only on the "observable behaviour" of the models.
- Proving that the modules of the language are stable, *e.g.*, by showing that they extend representation morphisms.

After these steps have been performed, this thesis provides a sound and practical methodology for modular program development with data abstraction in this language.

The claim that stability is the appropriate design criterion for a data abstraction language is supported by two further aguments.

First, the stability notion is defined as the condition under which the simple implementation property implies the universal implementation property. Thus, there is no way of introducing another, perhaps more general, notion instead of stability without changing the methodology or one of the two implementation notions. The simple implementation notion, however, corresponds closely to practical data representation correctness proofs, while the universal implementation notion is both necessary and sufficient for the composability of implementations (except perhaps for a slight generalization using a notion of "concrete model", cf. page 181).

Second, examples show that stability is a reasonable requirement of programming languages. Nipkow's nondeterministic language L [Nipkow 86], which

features application, local binding (let), lists, conditional, recursion, and angelic choice, is stable for the representation relation associated with his "simulation" concept (cf. page 281–283), and the deterministic sublanguage obtained by removing the angelic choice operator is stable for behavioural equivalence, as discussed at the end of Section 5.3. Also the sufficient condition for stability that modules "extend representation morphisms" (Def. 5.1.14 and Thm. 5.1.15), is very similar to the "representation independence" theorems that have been proved for variants of the typed λ-calculus (cf. page 280 f.). Finally, the type constructors **array** and **set** of PASCAL, which are not compatible with data abstraction (cf. page 16 f.), are ruled out by the stability criterion for any of the three representation relations between partial algebras that have been considered (Example 5.4.10).

Practical Program Design

The development of the *dictionary* program in Section 1.4 is an example of modular programming with data abstraction as supported by the theory of this thesis. The modules and interfaces of that development fit the corresponding formal notions (in the institution of tagged algebraic signatures and partial many-sorted algebras), the final design graph yields a decomposition of the global cell $\langle I_{ITEM} \wedge I_{LISTITEM}, I_{DICT} \rangle$, the correctness notion employed for the modules I_{DICT}, I_{INOUT}, I_{INPUT} and I_{OUTPUT} is the refinement notion of the theory, and the conventional data representation correctness proof of M_{STORE} sketched on page 54–57 and performed formally in Example 4.5.3 proves that M_{STORE} is a simple implementation (with respect to any of the three representation categories introduced) of its specification. What is missing, however, is a proof that the modules of this development are stable. The proper way to prove this would be to formalize the notation used to define these modules, and to show that this language allows to define only stable modules. To include such a proof in this thesis, however, would require a lot of additional formalism, yet not be interesting, because the notation is so primitive.

In the *dictionary* example, modular programming was viewed as the construction of a design graph. From a design graph, module specifications as required by the structured correctness argument and by the composability theorems are derived immediately (Example 3.2.11). If (the signatures of) these module specifications form a syntactical decomposition of (the signature of) the global cell, then the semantic decomposition property follows from the fact that the specifications were derived from a design graph. Hence design graphs make it unnecessary to prove the semantic component of the "decomposition" notion.

Design graphs do not help, however, to establish the syntactical decomposition property of the (signatures of the) specifications derived from them. I have translated the requirements of the syntactic decomposition notion back to design graphs, but the resulting criteria showed clearly that they were derived in this way and did not seem to be of any practical use. It appears that to ensure the syntactical decomposition property, there must be a global coordinator for a design graph who ensures that all signatures remain compatible, and that each symbol is "defined" in at most one place. This seems to be a consequence of the idea that one single "name space" (*i. e.*, signature) should underlie a decomposition.

The *dictionary* program development of Section 1.4 has illustrated the strategy of "access function refinement", which allows one to determine the elementary access functions to an encapsulated type in the course of refining more problem-oriented operations referring to the type. The basis for the refinement process is an abstract model (*i. e.*, a set of values) for the encapsulated type, which allows the operations to be specified and refined independently of each other. There is a danger of devising an abstract model that is "too big", *i. e.*, a value set with more values than necessary. Such a "biased" specification may guide the refinement process of the access operations in an inefficient direction.

Thus, different specification techniques are appropriate in different situations: Abstract model specifications work best when an "unbiased" value set for an encapsulated data type can be determined, while the implicit definition technique is appropriate when it is easier to determine a set of useful elementary access functions to the type.

It is advantageous, therefore, to use a specification notation that allows one to write both abstract model and implicit specifications of encapsulated data types, so that the most appropriate technique can be chosen in every situation. The mathematical notation used for specifications in this thesis has this property, since one may, but need not, specify value sets for data types. This language is "rigorous" in the sense of [Jones 80, p. 13 f.], but not fully formal (which would be necessary for machine processing).

In this context it is noteworthy that the theory of this thesis is completely independent of a specification notation. Specifications enter the picture only via their semantics, namely the set of models satisfying them, and it is irrelevant how this set is described. Since any set of models is admissible as a specification in the theory, there are no restrictions on the expressive power of the notation that may be used. It is not even necessary to fix a specification language at all; new language elements may be introduced whenever desired as long as it is clear what they mean, *i. e.*, it is clear which models satisfy a specification and which do not.

Contributions of the Thesis

The theory of this thesis consists of two levels. The "abstract" level of the theory, consisting of Section 2.3, Chapter 3, and Sections 4.1 and 5.1, deals with modular programming and data abstraction in the general setting of institutions, representation relations and representation categories. The "concrete" level of the theory, consisting of Sections 2.2, 4.2 to 4.5, and 5.2 to 5.4, deals with specific representation relations in the institution of (tagged) algebraic signatures and partial many-sorted algebras.

The abstract level of the theory is more general than most previous approaches to data abstraction, because it applies to an arbitrary institution. In parallel with this thesis, Sannella and Tarlecki developed an approach to data abstraction that is also based on institutions [ST 84a] [ST 85]. In this approach, data abstraction is based on a notion of "observational equivalence" between mod-

els, which depends on whether certain "sentences" are "satisfied" by the models. The approach of this thesis is more abstract and more general than the approach of Sannella and Tarlecki, because it is based on an abstract representation relation that need not even be symmetric. For example, behavioural inclusion and standard representation are not symmetric and can be treated in the theory of this thesis, but not (yet) in the theory of Sannella and Tarlecki.

The specific "institution" notion used in this thesis differs from the one introduced by Goguen and Burstall [GB 84] in that "sentences" and the "satisfaction relation" are omitted and an additional "inclusion" relation between signatures is introduced. This relation occurs naturally in practice, and it allows one to relate and combine signatures, models, and specifications without providing explicit signature morphisms (as is necessary, *e. g.*, in the specification-building operations of ASL [ST 85]).

Specifications in the abstract theory are sets of models. This means that specification languages and their semantics need not be considered explicitly. Furthermore, there is no need to use a fixed specification language at all when applying the theory—the only requirement on specifications is that it must be clear which models satsfy them and which do not. The idea of treating specifications as model sets is due to Lipeck [Lipeck 83, p. 15 f.]. However, Lipeck is a little more restrictive in that he considers only sets of algebras that are closed under isomorphism. This restricts the semantics of the specification notations that can be used in connection with Lipeck's approach.

Cells with requirements can be composed with each other and are correctly dealt with in structured correctness arguments: the correctness notion "universal implementation" (of which "refinement" is a special case) can be verified independently for each cell in a system and guarantees that the composition of the cells is correct. In particular, when the system is installed on a base that represents a model satisfying the global import specification, then all cells are guaranteed to be supplied with bases satisfying their requirements. This is in contrast to Lipeck's theory [Lipeck 83], which deals with cells with requirements, but in which the composability of independently designed cells is not always guaranteed, so that "re-implementation" of modules may become neces-

sary [Lipeck 83, p. 78–89]. In other theories [SW 82] [GM 82], the problem has led to the introduction of "fitting conditions" on the cells to be composed, so that they cannot be designed independently of each other (cf. page 22 f.).

The theory allows for infinite systems in case the representation relation is chain-closed. This is not directly relevant for practice, but it would allow, for example, to model parameterized modules (*e. g.*, the parameterized type constructors of a programming language) by introducing all their possible instantiations as cells into a system (of which only a finite subset would be used in a program). That the theory of [Schoett 81] might be extendable to infinite systems was suggested to me by Gordon Plotkin [personal communication, 1983] after he had read an early version of [Schoett 83].

The "stability" criterion for cells is *derived* from the "universal" and "simple implementation" notions, so that it is the most general condition under which the "simple implementation" criterion implies "universal implementation". This supports the claim that stability should be a design criterion for "data abstraction" programming languages, because the "simple implementation" notion reflects practical data representation correctness proofs (Example 5.4.3) and the "universal implementation" notion expresses the requirements for composability. Thus, the only way the stability notion could be generalized would be to generalize "universal implementation" by distinguishing "concrete models" as suggested on page 181. In contrast, previous approaches to the correctness problem of data abstraction have introduced *ad hoc* criteria for the modules of a system and proved that they are sufficient to guarantee composability (the "homomorphism expansion property" in [Schoett 81], "conservativity" in [Lipeck 83], the "soundness conditions" of [Nipkow 86]), and there was no search for the most general such criterion.

The concrete level of the theory of this thesis deals with the representation relations "behavioural inclusion", "behavioural equivalence", and "standard representation" between partial many-sorted algebras, and with their associated representation categories and stability notions.

It is shown that even the very restrictive representation relation "there exists a surjective homomorphism" between total algebras admits useless representations in which the computations produce just formal terms (Thm. 4.2.2). This is an argument in favour of representation relations that are based on a set of "visible sorts". Values of visible sorts must be represented by themselves, and this means that result values of visible sorts that are produced by a correct representation are the same as in the algebra represented, so that it is no problem to interpret these results.

A new representation relation "behavioural inclusion" has been introduced, which combines the "partial implementation" idea of Kamin and Archer [KA 84] with the behaviour idea. It is strictly more general than the concept of Kamin and Archer (Thm. 4.3.11 and Example 4.3.12).

The "behavioural equivalence" representation relation is shown to be strictly more general than the "standard representation" relation that is based on abstraction functions (Thm. 4.5.4 and Example 4.3.12), and it is shown that behavioural equivalence eliminates the "representation bias" problem that occurs in connection with standard representation (page 258 f.).

Two new types of many-sorted relations between partial many-sorted algebras are introduced, "correspondences" and "strong correspondences". These concepts lead to characterizations of "behavioural inclusion" and "behavioural equivalence" (Theorem 4.3.7 and 4.4.6), thus yielding "simple implementation" and "stability" notions for these representation relations (Sections 5.2 and 5.3), and they provide practical methods for proving data representations correct (Examples 4.3.10 and 4.4.7). The idea that data representations can be proved correct using relations instead of functions is not new—it occurs, for example, in [Ginzburg 68], [Milner 71], [Jones 80, p. 264], and [Reynolds 81, p. 311]. However, these ideas seem not to have been transferred to algebraic data type theory and linked with its "behaviour" notions before. The "strong correspondence" concept has influenced recent work by Nipkow [Nipkow 86], who generalized it to algebras with nondeterministic operations.

Characterizations of the stability notions for behavioural inclusion, behavioural equivalence, and standard representation are given: A cell is stable for be-

havioural inclusion iff it "extends" correspondences (Def. 5.1.14 and Thm. 5.2.4); it is stable for behavioural equivalence iff it "weakly extends" both abstraction functions and converse abstraction functions (Def. 5.3.5 and Thm. 5.3.6); and it is stable for standard representation iff it extends abstraction functions (Thm. 5.4.4). For a large class of cells, the stability notions for behavioural inclusion and behavioural equivalence are equivalent, while the stability notion for standard representation is more restrictive (Theorems 5.4.8 and 5.4.9).

The program development example of Section 1.4 introduces the "access function refinement" strategy, which allows one to determine the elementary access functions to an encapsulated data type in the process of designing code for more complex, problem-oriented operations. This strategy works best on the basis of an abstract model specification of the encapsulated type, which leads to the conclusion that implicit specification techniques for encapsulated data types are not always the best ones, and that practical specification notations should allow one to write both "abstract model" and "implicit" specifications of encapsulated data types.

Improvements to the Theory

There are various ways in which the theory of this thesis could be further developed and improved.

From a practical point of view, an important problem is to ensure that the specification cell system derived from a design graph (cf. Examples 3.1.5, 3.1.14, 3.2.9 and 3.2.11) is syntactically correct (*i. e.*, that the cell signatures form a syntactic decomposition of the global cell signature). At present, it appears to be necessary that the choice of names for the program entities occurring in a design graph is managed centrally. To some extent, this interferes with the freedom to design modules independently of each other, because module design may involve the design of new import interfaces, and the names occurring in these interfaces may only be chosen in consultation with the central "name manager".

A way to solve this problem might be to introduce composition operations for cells that do not use the idea of a uniform name space, but in which the cells are connected by explicit signature morphisms. Such operations are familiar from theories of parameterized specifications and parameterized data types such as [BG 77], [BG 80], [TWW 82], [Ehrig *et al.* 80], [SW 82] and [Lipeck 83], and from programming notations involving parameterized or "generic" modules, such as CLU [Liskov *et al.* 81], ADA [ANSI 83] and CIP-L [CIP 85].

In connection with the introduction of such operations, it would be worth while to check whether the renaming axiom of an institution syntax (axiom (c) of Def. 2.3.5) can be replaced by a more elegant axiom. At present, the renaming axiom is by far the most complex of the axioms characterizing an institution, and it takes a considerable amount of formal labour to verify (see the proofs of Thm. 2.3.6 and 4.2.6); yet it is only used in a single place, namely the proof of the "renaming" lemma (Lemma 4.1.9).

On the semantic level, the new operations do not seem capable of creating any new problems, because it appears that each composition of modules using the new operations can be represented in the present theory by a system in which each module instantiation occurs as a separate cell, and in which the symbols are renamed so that precisely those symbols are equal that are identified with each other via signature morphisms in the more complex composition operations.

It would also be important for practice to have programming languages available that have been proved sound for data abstraction in the sense of this thesis, that is, whose modules are guaranteed to be stable. So far, stability has been established only for "toy languages" consisting mainly of recursive function definitions (informally in [Hoare 72] and [Schoett 81], formally in [Nipkow 86], where functions may even be nondeterministic), and it would be interesting to see how such proofs "scale up" to realistic programming notations. A formal semantics for the programming language under consideration is, of course, a prerequisite for a stability proof.

The presentation of the theory of modular programming in Chapter 3 could be improved slightly by changing some of the basic definitions so that they more closely match their informal motivation.

The definition of a "system site" (Def. 3.2.4) could be improved by sharpening the condition that for $i \in I$

$$\bar{E} \sqcup \bigsqcup_{k \ll i} D_k \text{ is a site for } \langle E_i, D_i \rangle.$$

to

whenever $K \subseteq I$ is $<$-downward closed and i is $<$-minimal in $I \setminus K$
then $\bar{E} \sqcup \bigsqcup_{k \in K} D_k$ is a site for $\langle E_i, D_i \rangle$,

since that definition makes it clear that the cells of a system can be composed in any order compatible with the dependence relation $<$. Fortunately, this does not change the mathematical content of the theory, sice the two definitions are equivalent according to Lemmas 3.4.3 (c) and 3.5.2.

In a similar way, the definitions of "decomposition" (Def. 3.2.10) and "composition" (Def. 3.3.6) could be changed by quantifying over downward closed index sets K in the statement that each cell must be supplied with a proper base (Clause 3.2.10 (b) and the definition of $\bar{Q}$ in Def. 3.3.6). This would not change anything either, due to the monotonicity of interface conjunction (Prop. 3.1.8).

From a theoretical point of view, it seems interesting to explore the link between institutions and sheaves that was observed by John Gray. One presentation of sheaf theory in which this link is clearly visible is [FS 79].

It would also be interesting to study further the interplay betwen representation relations and representation categories. The concept of a "representation relation" between models arises naturally in connection with data abstraction, and it suffices for the "universal implementation" concept and the composability theorem. The "representation categories" are needed only for the more practical "simple implementation" concept. It is an interesting open problem to characterize those representation relations for which representation categories exist.

The proof of the composability theorem for universal implementations does not make use of the preordering property of a representation relation, so that

this condition could be removed from the definition of a "representation relation" (Def. 4.1.1). It was mentioned on page 181 that apparently the only way to generalize the "universal implementation" concept would be to exploit the fact that not all the models of an institution might actually be definable in a programming notation. With representation relations that need not be preorderings, this idea could be expressed elegantly by making $A' \rightsquigarrow A$ hold only if A' is definable (a "concrete model").

While removing the condition that representation relations be preorderings would not affect the composability theorem, it creates a problem in connection with representation categories. At present, the fact that the representation morphisms form a category and characterize the representation relation via $A' \rightsquigarrow A \iff \exists J\colon A' \rightsquigarrow A$ implies that the representation relation must be a preordering anyway. Thus, a way would have to be found to weaken these axioms while preserving the essential theorems of Section 5.1, particularly the decomposition of "universal implementation" into "simple implementation" and "stability", and the convenient "representation morphism extension" criterion for stability.

... for it is only theory that makes men completely incautious.

Bertrand Russell [Russell 50, p. 178]

Bibliography

THE ENTRIES in this list are sorted by author, therefore the citation keys may be slightly out of sequence. Uncited entries are marked "—".

Bibliographies on data types and abstract data types have been published by Dungan [Dungan 79] and by Kutzler and Lichtenberger [KL 83]. Klaeren's book [Klaeren 83] also contains a good bibliography.

Abbreviations of journal, institution, and series titles:

ACM	= Association for Computing Machinery
ACM SIGPLAN	= ACM Special Interest Group on Programming Languages
EATCS	= European Association for Theoretical Computer Science
IEEE	= Institute of Electrical and Electronic Engineers
LNCS	= Lecture Notes in Computer Science
MIT	= Massachusetts Institute of Technology
R.A.I.R.O.	= Revue Française d'Automatique, d'Informatique, et de Recherche Opérationnelle
SIAM	= Society for Industrial and Applied Mathematics

[ANSI 83] American National Standards Institute: *The Programming Language Ada Reference Manual.* ANSI/MIL-STD-1815A-1983. Berlin *et al.*: Springer 1983 (= LNCS 155).

— Arbib, M. A., Manes, E. G.: Machines in a Category: An Expository Introduction. *SIAM Review* 16 (1974), p. 163–192.

— Arbib, M. A., Manes, E. G.: Adjoint Machines, State-Behaviour Machines, and Duality. *Journal of Pure and Applied Algebra* 6 (1975), p. 313–344.

— Back, R. J. R.: On Correct Refinement of Programs. *Journal of Computer and System Sciences* 23 (1981), p. 49–68.

[BM 84] Back, R. J. R., Mannila, H.: A Semantic Approach to Program Modularity. *Information and Control* 60 (1984), p. 138–167.

[Barendregt 84] Barendregt, H. P.: *The Lambda Calculus. Its Syntax and Semantics.* Revised ed., Amsterdam, New York, Oxford: North-Holland 1984 (= Studies in Logic and the Foundations of Mathematics 103).

[Barwise 74] Barwise, K. J.: Axioms for Abstract Model Theory. *Annals of Mathematical Logic* 7 (1974), p. 221–265.

[Barwise 77] Barwise, J. (ed.): *Handbook of Mathematical Logic.* Amsterdam, New York, Oxford: North-Holland 1977 (= Studies in Logic and the Foundations of Mathematics 90).

[BW 84] Bauer, F. L., Wössner, H.: *Algorithmische Sprache und Programmentwicklung.* Second, improved ed., Berlin *et al.*: Springer 1984.

[BR 83] Benecke, K., Reichel, H.: Equational Partiality. *Algebra Universalis* 16 (1983), p. 219–232.

— Bergstra, J. A., Broy, M., Tucker, J. V., Wirsing, M.: On the Power of Algebraic Specifications. In: *Mathematical Foundations of Computer Science 1981. Proceedings .... (...)* Ed.: J. Gruska, M. Chytil. Berlin, Heidelberg, New York: Springer 1981 (= LNCS 118). P. 193–204.

[Birkhoff 67] Birkhoff, G.: *Lattice Theory.* Third ed., Providence, Rh. I.: American Mathematical Society 1967 (= Amer. Math. Soc. Colloquium Publications 25).

— Birkhoff, G., Lipson, J. D.: Heterogeneous Algebras. *Journal of Combinatorial Theory* 8 (1970), p. 115–133.

[Bothe 81] Bothe, K.: A Comparative Study of Abstract Data Type Concepts. *Elektronische Informationsverarbeitung und Kybernetik* 17 (1981), p. 237–257.

[Bourbaki 66] Bourbaki, N.: *Éléments de mathématique. Fascicule XVII. Théorie des ensembles. Chapitres 1 et 2. (...)* Troisième édition revue et corrigée, Paris: Hermann 1966 (= Actualités scientifiques et industrielles 1212). Translated into English as: *Elements of Mathematics. Theory of Sets.* Paris: Hermann / Reading *et al.*: Addison-Wesley 1968 (Actualités scientifiques et industrielles / ADIWES International Series in Mathematics).

[BS 82] British Standard Specification for Computer Programming Language Pascal. 1982 (= BS 6192).

— Broy, M., Dosch, W., Partsch, H., Pepper, P., Wirsing, M.: Existential Quantifiers in Abstract Data Types. In: *Automata, Languages and Programming. Sixth Colloquium ... 1979.* Ed.: H. A. Maurer. Berlin, Heidelberg, New York: Springer 1979 (= LNCS 71). P. 73–87.

— Broy, M., Wirsing, M.: Partial recursive functions and abstract data types. *Bulletin of the EATCS* 11 (June 1980), p. 34–41.

[BW 82] Broy, M., Wirsing, M.: Partial Abstract Types. *Acta Informatica* 18 (1982), p. 47–64.

[BW 83] Broy, M., Wirsing, M.: Algebraic Definition of a Functional Programming Language and its Semantic Models. *R.A.I.R.O. Informatique théorique / Theoretical Informatics* 17 (1983), p. 137–161.

— Broy, M., Wirsing, M., Pair, C.: A Systematic Study of Models of Abstract Data Types. *Theoretical Computer Science* 33 (1984), p. 139–174.

— Burmeister, P.: Quasi-Equational Logic for Partial Algebras. In: *Fundamentals of Computation Theory. Proceedings .... (...)* Ed.: F. Gecseg. Berlin, Heidelberg, New York: Springer 1981 (= LNCS 117). P. 71–80.

[Burmeister 82] Burmeister, P.: Partial algebras—survey of a unifying approach towards a two-valued model theory for partial algebras. *Algebra Universalis* 15 (1982), p. 306–358.

— Burmeister, P., Reichel, H.: On the Current State and Trends in the Theory of Partial Algebras. Darmstadt, Technische Hochschule Darmstadt, Fachbereich Mathematik, Preprint No. 719, December 1982.

— Burstall, R. M., Darlington, J.: A Transformation System for Developing Recursive Programs. *Journal of the ACM* 24 (1977), p. 44–67.

[BG 77] Burstall, R. M., Goguen, J. A.: Putting Theories Together to Make Specifications. In: *5th International Joint Conference on Artificial Intelligence - 1977. IJCAI 77. (...). Volume Two.* P. 1045–1058.

[BG 80] Burstall, R. M., Goguen, J. A.: The Semantics of Clear, a Specification Language. In: *Abstract Software Specification.* Ed.: D. Bjørner. Berlin, Heidelberg, New York: Springer 1980 (= LNCS 86). P. 292–332.

[BG 81] Burstall, R. M., Goguen, J. A.: An informal introduction to specifications using Clear. In: *The correctness problem in computer science.* Ed.: R. S. Boyer, J S. Moore. London *et al.*: Academic Press 1981 (International Lecture Series in Computer Science). P. 185–213.

— Burstall, R. M., Goguen, J. A.: Algebras, Theories and Freeness: an Introduction for Computer Scientists. Edinburgh, University of Edinburgh, Department of Computer Science 1982 (= Internal Report CSR-101-82).

[BMS 81] Burstall, R. M., MacQueen, D. B., Sannella, D. T.: HOPE: An Experimental Applicative Language. Edinburgh, University of Edinburgh, Department of Computer Science 1980 (Update 1981) (= Internal Report CSR-62-80).

[Cartwright 84] Cartwright, R.: Recursive Programs as Definitions in First-Order Logic. *SIAM Journal on Computing* 13 (1984), p. 374–408.

— Chang, C. C., Keisler, H. J.: *Model Theory.* Second ed., Amsterdam, New York, Oxford: North-Holland 1977 (= Studies in Logic and the Foundations of Mathematics 73).

[CIP 85] The CIP Language Group ...: *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L.* Berlin *et al.*: Springer 1985 (= LNCS 183).

[Cohn 81] Cohn, P. M.: *Universal Algebra.* Revised ed., Dordrecht, Boston, London: D. Reidel 1981 (= Mathematics and its Applications 6).

[DS 85] Däßler, K., Sommer, M.: *PASCAL. Einführung in die Sprache; DIN-Norm 66256; Erläuterungen.* Second ed., Berlin *et al.*: Springer 1985.

[DN 66] Dahl, O.-J., Nygaard, K.: SIMULA—an ALGOL-Based Simulation Language. *Communications of the ACM* 9 (1966), p. 671–678.

— Delong. H.: *A Profile of Mathematical Logic.* Reading, Mass. *et al.*: Addison-Wesley 1970.

[Denert 79] Denert, E.: Software-Modularisierung. *Informatik-Spektrum* 2 (1979), p. 204–218.

[Dijkstra 68] Dijkstra, E. W.: The Structure of the "THE"-Multiprogramming System. *Communications of the ACM* 11 (1968), p. 341–346.

[Dijkstra 72] Dijkstra. E. W.: Notes on Structured Programming. In: O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare: *Structured Programming.* London, New York: Academic Press 1972 (= A.P.I.C. Studies in Data Processing 8).

— Dijkstra, E. W.: *A Discipline of Programming.* Englewood Cliffs, N. J.: Prentice-Hall 1976 (Prentice-Hall Series in Automatic Computation).

— Dijkstra, E. W.: The Effective Arrangement of Logical Systems. In: *Mathematical Foundations of Computer Science 1976. (...)* Ed.: A. Mazurkiewicz. Berlin, Heidelberg, New York: Springer 1976 (= LNCS 45). P. 39–51.

— Dijkstra, E. W.: My Hopes of Computing Science. In: *4th International Conference on Software Engineering. ... 1979. (...).* IEEE 1979. P. 442–448.

— Dijkstra, E. W.: On the Role of Scientific Thought. In: E. W. Dijkstra: *Selected Writings on Computing: A Personal Perspective.* New York, Heidelberg, Berlin: Springer 1982 (Texts and Monographs in Computer Science). P. 60–66.

— DIN 44300. Informationsverarbeitung. Begriffe. In: *Informationsverarbeitung 1. Normen ....* Ed.: DIN Deutsches Institut für Normung e. V. Fourth, modified ed., Berlin, Köln: Beuth Verlag 1978. P. 109–128.

[Donahue 79] Donahue, J.: On the Semantics of "Data Type". *SIAM Journal on Computing* 8 (1979), p. 546–560.

[Dungan 79] Dungan, D. M.: Bibliography on Data Types. *ACM SIGPLAN Notices* 14 (1979), p. 11, 31–59.

— Ehrich, H. D.: On Realization and Implementation. In: *Mathematical Foundations of Computer Science 1981. Proceedings .... (...)* Ed.: J. Gruska, M. Chytil. Berlin, Heidelberg, New York: Springer 1981 (= LNCS 118). P. 271–280.

[Ehrich 82] Ehrich, H.-D.: On the Theory of Specification, Implementation and Parameterization of Abstract Data Types. *Journal of the ACM* 29 (1982), p. 206–227.

— Ehrich, H. D., Lipeck, U.: Proving Implementations Correct – Two Alternative Approaches. In: *Information Processing 80.* Ed.: S. H. Lavington. Amsterdam, New York, Oxford: North-Holland 1980. P. 83–88.

[EK 83] Ehrig, H., Kreowski, H.-J.: Compatibility of Parameter Passing and Implementation of Parameterized Data Types. *Theoretical Computer Science* 27 (1983), p. 255–286.

[EKMP 82] Ehrig, H., Kreowski, H.-J., Mahr, B., Padawitz, P.: Algebraic Implementation of Abstract Data Types. *Theoretical Computer Science* 20 (1982), p. 209–263.

[Ehrig *et al.* 80] Ehrig, H., Kreowski, H.-J., Thatcher, J., Wagner, E., Wright, J.: Parameterized Data Types in Algebraic Specification Languages. In: *Automata, Languages and Programming. 7th Colloquium ... 1980.* Ed.: J. de Bakker, J. van Leeuwen. Berlin, Heidelberg, New York: Springer 1980 (= LNCS 85). P. 157–168.

[EM 85] Ehrig, H., Mahr, B.: *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics.* Berlin *et al.*: Springer 1985 (= EATCS Monographs on Theoretical Computer Science 6).

[FS 79] Fourman, M. P., Scott, D. S.: Sheaves and Logic. In: *Applications of Sheaves. Proceedings ... 1977.* Ed.: M. P. Fourman, C. J. Mulvey, D. S. Scott. Berlin, Heidelberg, New York: Springer 1979 (= Lecture Notes in Mathematics 753). P. 302–401.

— Fraenkel, A. A., Bar-Hillel, Y., Levy, A.: *Foundations of Set Theory.* Second revised ed., Amsterdam, New York, Oxford: North-Holland 1973 (= Studies in Logic and the Foundations of Mathematics 67).

[Ganzinger 83] Ganzinger, H.: Parameterized Specifications: Parameter Passing and Implementation with Respect to Observability. *ACM Transactions on Programming Languages and Systems* 5 (1983), p. 318–354.

[GGM 76] Giarratana, V., Gimona, F., Montanari, U.: Observability Concepts in Abstract Data Type Specification. In: *Mathematical Foundations of Computer Science 1976. (...)* Ed.: A. Mazurkiewicz. Berlin, Heidelberg, New York: Springer 1976. (= LNCS 45). P. 576–587.

[Ginzburg 68] Ginzburg, A.: *Algebraic Theory of Automata.* New York, London: Academic Press 1968.

— Goguen, J. A.: Parameterized Programming. *IEEE Transactions on Software Engineering* SE-10 (1984), p. 528–543.

[GB 80] Goguen, J. A., Burstall, R. M.: CAT, a System for the Structured Elaboration of Correct Programs from Structured Specifications. SRI International, Technical Report CSL-118, 1980.

[GB 84] Goguen, J. A., Burstall, R. M.: Introducing Institutions. In: *Logics of Programs. Workshop ... 1983.* Ed.: E. Clarke, D. Kozen. Berlin *et al.*: Springer 1984 (= LNCS 164). P. 221–256.

[GM 82] Goguen, J., Meseguer, J.: Universal Realization, Persistent Interconnection and Implementation of Abstract Modules. In: *Automata, Languages and Programming. Ninth Colloquium ... 1982.* Ed.: M. Nielsen, E. M. Schmidt. Berlin, Heidelberg, New York: Springer 1982 (= LNCS 140). P. 265–281.

[GTW 78] Goguen, J. A., Thatcher, J. W., Wagner, E. G.: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In: *Current Trends in Programming Methodology. Volume IV. Data Structuring.* Ed.: R. T. Yeh. Englewood Cliffs, N. J.: Prentice-Hall 1978. P. 80–149.

— Goguen, J. A., Thatcher, J. W., Wagner, E. G., Wright, J. B.: Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM* 24 (1977), p. 68–95.

[Grätzer 79] Grätzer, G.: *Universal Algebra.* Second ed., New York, Heidelberg, Berlin: Springer 1979.

— Guttag, J. V.: The Specification and Application to Programming of Abstract Data Types. Toronto, University of Toronto, Technical Report CSRG-59, 1975.

[Guttag 77] Guttag, J.: Abstract Data Types and the Development of Data Structures. *Communications of the ACM* 20 (1977), p. 396–404.

[GH 78] Guttag, J. V., Horning, J. J.: The Algebraic Specification of Abstract Data Types. *Acta Informatica* 10 (1978), p. 27–52.

— Guttag, J., Horning, J., Wing, J.: Some Notes on Putting Formal Specifications to Productive Use. *Science of Computer Programming* 2 (1982), p. 53–68.

[GHM 78] Guttag, J. V., Horowitz, E., Musser, D. R.: Abstract Data Types and Software Validation. *Communications of the ACM* 21 (1978), p. 1048–1064.

[Harel 80] Harel, D.: On Folk Theorems. *Communications of the ACM* 23 (1980), p. 379–389.

[HMM 86] Harper, R., MacQueen, D., Milner, R.: Standard ML. Edinburgh, University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science 1986 (= LFCS Report Series ECS-LFCS-86-2).

[Haynes 84] Haynes, C. T.: A Theory of Data Type Representation Independence. In: *Semantics of Data Types. International Symposium. ... 1984. (...)* Ed.: G. Kahn, D. B. MacQueen, G. Plotkin. Berlin *et al.*: Springer 1984. (= LNCS 173). P. 157–175.

— Higgins, P. J.: Algebras with a Scheme of Operators. *Mathematische Nachrichten* 27 (1963), p. 115–132.

— Hilfinger, P. N.: *Abstraction Mechanisms and Language Design.* Cambridge, Mass., London: The MIT Press 1983 (ACM Distinguished Dissertations).

[Hoare 69] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. *Communications of the ACM* 12 (1969), p. 576–580, 583.

[Hoare 72] Hoare, C. A. R.: Proof of Correctness of Data Representations. *Acta Informatica* 1 (1972), p. 271–281.

— Hofstadter, D. R.: *Gödel, Escher, Bach: An Eternal Golden Braid.* Harmondsworth, Middlesex et al.: Penguin Books 1980. Also: Stanford Terrace, Hassocks, Sussex: The Harvester Press 1979 (= Harvester Studies in Cognitive Science 12).

— Hupbach, U. L.: Abstract Implementation and Parameter Substitution. In: *Proceedings of the Third Hungarian Computer Science Conference.* Ed.: M. Arató, L. Varga. Budapest: Publishing House of the Hungarian Academy of Sciences 1981. P. 11–25.

[HR 83] Hupbach, U. L., Reichel, H.: On Behavioural Equivalence of Data Types. *Elektronische Informationsverarbeitung und Kybernetik* 19 (1983), p. 297–305.

— Ichbiah, J. D.: On the Design of Ada. In: *Information Processing 83. Proceedings of the IFIP 9th World Computer Congress. (...)* Ed.: R. E. A. Mason. Amsterdam, New York, Oxford: North-Holland 1983. Also: Amsterdam: Elsevier 1983. P. 1–10.

— Ichbiah, J. D., Barnes, J. G. P., Heliard, J. C., Krieg-Brückner, B., Roubine, O., Wichmann, B. A.: Rationale for the Design of the ADA Programming Language. *ACM Sigplan Notices* 14 (1979), No. 6, Part B.

— Jones, C. B.: Constructing a Theory of a Data Structure as an Aid to Program Development. *Acta Informatica* 11 (1979), p. 119–137.

[Jones 80] Jones, C. B.: *Software Development: A Rigorous Approach.* London *et al.*: Prentice-Hall 1980 (Prentice-Hall International Series in Computer Science).

[Kamin 83] Kamin, S.: Final Data Types and Their Specification. *ACM Transactions on Programming Languages and Systems* 5 (1983), p. 97–121.

[KA 84] Kamin, S., Archer, M.: Partial Implementations of Abstract Data Types: A Dissenting View on Errors. In: *Semantics of Data Types. International Symposium. ... 1984. (...)* Ed.: G. Kahn, D. B. MacQueen, G. Plotkin. Berlin *et al.*: Springer 1984. (= LNCS 173). P. 317–336.

[Klaeren 83] Klaeren, H. A.: *Algebraische Spezifikation. Eine Einführung.* Berlin, Heidelberg, New York: Springer 1983.

— Klaeren, H. A.: A Constructive Method for Abstract Algebraic Software Specification. *Theoretical Computer Science* 30 (1984), p. 139–204.

— Kneebone, G. T.: *Mathematical Logic and the Foundations of Mathematics.* London *et al.*: D. van Nostrand 1963.

[Knuth 81] Knuth, D. E.: *The Art of Computer Programming. Volume 2 / Seminumerical Algorithms.* Second ed., Reading, Mass., *et al.*: Addison-Wesley 1981.

[Kunen 80] Kunen, K.: *Set Theory. An Introduction to Independence Proofs.* Amsterdam, New York, Oxford: North-Holland 1980 (= Studies in Logic and the Foundations of Mathematics 102).

— Kupka, I.: Van Wijngaarden Grammars as a Special Information Processing Model. In: *Mathematical Foundations of Computer Science 1980. Proceedings (...)* Ed.: P. Dembinski. Berlin, Heidelberg, New York: Springer 1980 (= LNCS 88). P. 387–401.

[KL 83] Kutzler, B., Lichtenberger, F.: *Bibliography on Abstract Data Types.* Berlin *et al.*: Springer 1983 (= Informatik-Fachberichte 68).

[Lakatos 76] Lakatos, I.: *Proofs and Refutations. The Logic of Mathematical Discovery.* Cambridge *et al.*: Cambridge University Press 1967.

— Lamport, L.: Letter to P. M. Taylor, May 12, 1983.

— Lehmann, D. J., Smyth, M. B.: Algebraic Specification of Data Types: A Synthetic Approach. *Mathematical Systems Theory* 14 (1981), p. 97–139.

[Levy 79] Levy, A.: *Basic Set Theory.* Berlin, Heidelberg, New York: Springer 1979 (Perspectives in Mathematical Logic).

[Lipeck 83] Lipeck, U.: Ein algebraischer Kalkül für einen strukturierten Entwurf von Datenabstraktionen. Dortmund, Universität Dortmund, Abteilung Informatik, Dissertation 1982 (= Forschungsbericht Nr. 148, 1983).

— Liskov, B.: Modular Program Construction Using Abstractions. In: *Abstract Software Specification.* Ed.: D. Bjørner. Berlin, Heidelberg, New York: Springer 1980 (= LNCS 86). P. 354–389.

[Liskov *et al.* 81] Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J. C., Scheifler, R., Snyder, A.: *CLU Reference Manual.* Berlin, Heidelberg, New York: Springer 1981 (= LNCS 114).

[LB 79] Liskov, B. H., Berzins, V.: An Appraisal of Program Specifications. In: *Research Directions in Software Technology.* Ed.: P. Wegner. Cambridge, Mass.: MIT Press 1979 (= The MIT Press Series in Computer Science 2). P. 276–301.

[LZ 75] Liskov, B., Zilles, S.: Specification Techniques for Data Abstractions. *IEEE Transactions on Software Engineering* SE-1 (1975), p. 7–19.

— Loeckx, J.: Algorithmic Specifications of Abstract Data Types. In: *Automata, Languages and Programming. ... 1981.* Ed.: S. Even, O. Kariv. Berlin, Heidelberg, New York: Springer 1981 (= LNCS 115). P. 129–147.

[Mac Lane 71] Mac Lane, S.: *Categories for the Working Mathematician.* New York, Heidelberg, Berlin: Springer 1971 (= Graduate Texts in Mathematics 5).

— MacQueen, D. B., Sannella, D. T.: Completeness of Proof Systems for Equational Specifications. *IEEE Transactions on Software Engineering* SE-11 (1985), p. 454–461.

[MM 84] Mahr, B., Makowsky, J. A.: Characterizing Specification Languages which admit Initial Semantics. *Theoretical Computer Science* 31 (1984), p. 49–59.

[MG 85] Meseguer, J., Goguen, J. A.: Initiality, Induction, and Computability. In: *Algebraic Methods in Semantics.* Ed.: M. Nivat, J. C. Reynolds. Cambridge *et al.*: Cambridge University Press 1985. P. 459–541.

[Milner 71] Milner, R.: An Algebraic Definition of Simulation between Programs. In: *Second International Joint Conference on Artificial Intelligence. ... 1971.* London: The British Computer Society 1971. P. 481–489.

— Milner, R.: A Proposal for Standard ML. Edinburgh, University of Edinburgh, Department of Computer Science 1983 (= Internal Report CSR-157-83).

— Milner, R.: The Standard ML Core Language. Edinburgh, University of Edinburgh, Department of Computer Science 1984 (= Internal Report CSR-168-84).

[Mitchell 86] Mitchell, J. C.: Representation independence and data abstraction (Summary). Manuscript 1986. To appear in the Proceedings of the Thirteenth ACM Conference on Principles of Programming Languages 1986.

[MM 85] Mitchell, J. C., Meyer, A. R.: Second-order logical relations (Extended Abstract). In: *Logics of Programs. ... 1985. Proceedings.* Ed.: R. Parikh. Berlin *et al.*: Springer 1985 (= LNCS 193). P. 225–236.

[MP 84] Mitchell, J. C., Plotkin, G. D.: Abstract types have existential type. In: ... *Twelfth Annual ACM Symposium on Priciples of Programming Languages.* New York: ACM 1984. P. 37–51.

— Morgan, C., Sufrin, B.: Specification of the UNIX Filing System. *IEEE Transactions on Software Engineering* SE-10 (1984), p. 128–142.

[MT 79] Morris, R., Thompson, K.: Password Security: A Case History. *Communications of the ACM* 22 (1979), p. 594–597.

— Naur, P.: Formalization in Program Development. *BIT* 22 (1982), p. 437–453.

[Nipkow 86] Nipkow, T.: Non-deterministic Data Types: Models and Implementations. *Acta Informatica* 22 (1986), p. 629–661.

— Orejas, F.: Characterizing Composability of abstract Implementations. Barcelona, Universitat Politècnica de Barcelona, Facultat d'Informàtica, Report de Recerca RR 82/08, 1982. Short version in: *Foundations of Computation Theory. Proceedings ... 1983.* Ed.: M. Karpinski. Berlin *et al.*: Springer 1983 (= LNCS 158). P. 335–346.

[Parnas 72a] Parnas, D. L.: Information Distribution Aspects of Design Methodology. In: *Information Processing 71. Proceedings of IFIP Congress 71.* Amsterdam, New York, Oxford: North-Holland 1972. P. 339–344.

[Parnas 72b] Parnas, D. L.: On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM* 15 (1972), p. 1053–1058.

[Parnas 79] Parnas, D. L.: The role of program specification. In: *Research Directions in Software Technology.* Ed.: P. Wegner. Cambridge, Mass.: MIT Press 1979 (= The MIT Press Series in Computer Science 2). P. 364–370.

— Parnas, D. L., Clements, P. C.: A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering* SE-12 (1986), p. 251–257.

— Parnas, D. L., Clements, P. C., Weiss, D. M.: The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering* SE-11 (1985), p. 259–266.

— Pepper, P., Broy, M., Bauer, F. L., Partsch, H., Dosch, W., Wirsing, M.: Abstrakte Datentypen: Die algebraische Spezifikation von Rechenstrukturen. *Informatik-Spektrum* (1982) 5, p. 107–119.

[Plotkin 80] Plotkin, G. D.: Lambda-Definability in the Full Type Hierarchy. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Ed.: J. P. Seldin, J. R. Hindley. London *et al.*: Academic Press 1980. P. 363–373.

[Reichel 81] Reichel, H.: Behavioural Equivalence—A Unifying Concept for Initial and Final Specification Methods. In: *Proceedings of the Third Hungarian Computer Science Conference.* Ed.: M. Arató, L. Varga. Budapest: Publishing House of the Hungarian Academy of Sciences 1981. P. 27–39.

[Reichel 84] Reichel, H.: *Structural Induction on Partial Algebras. Introduction to Theory and Application of Partial Algebras – Part II.* Berlin (DDR): Akademie-Verlag 1984 (= Mathematical Research · Mathematische Forschung 18).

[Reynolds 81] Reynolds, J. C.: *The Craft of Programming.* London *et al.*: Prentice-Hall 1981 (Prentice-Hall International Series in Computer Science).

[Reynolds 83] Reynolds, J. C.: Types, Abstraction and Parametric Polymorphism. In: *Information Processing 83. Proceedings of the IFIP 9th World Computer Congress. (...)* Ed.: R. E. A. Mason. Amsterdam, New York, Oxford: North-Holland 1983. Also: Amsterdam: Elsevier 1983. P. 513–523.

[Russell 50] Russell, B.: Ideas that have Harmed Mankind. In: Russell, B.: *Unpopular Essays.* London: George Allen & Unwin 1976 (Unwin Paperbacks). P. 160–180. First published 1950.

— Sakabe, T., Inagaki, Y., Honda, N.: Specification of Abstract Data Types with Partially Defined Operations. In: 6^{th} *International Conference on Software Engineering. ... 1982. (...)* IEEE Computer Society Press 1982. P. 218–224.

— Sannella, D. T.: Semantics, Implementation and Pragmatics of Clear, A Program Specification Language. Edinburgh, University of Edinburgh, Department of Computer Science, PhD Thesis 1982 (= CST-17-82).

— Sannella, D. T.: A Set-Theoretic Semantics for Clear. *Acta Informatica* 21 (1984), p. 443–472.

[ST 84a] Sannella, D., Tarlecki, A.: On Observational Equivalence and Algebraic Specification. Edinburgh, University of Edinburgh, Department of Computer Science 1984 (= Internal Report CSR-172-84). Extended Abstract in: *Mathematical Foundations of Software Development. Proceedings ... (TAPSOFT) ... 1985. Volume 1: Colloquium on Trees in Algebra and Programming (CAAP '85).* Ed.: H. Ehrig *et al.* Berlin *et al.*: Springer 1985 (= LNCS 185) P. 308–322.

[ST 84b] Sannella, D., Tarlecki, A.: Program Specification and Development in Standard ML. In: ... *Twelfth Annual ACM Symposium on Priciples of Programming Languages.* New York: ACM 1984. P. 67–77.

[ST 85] Sannella, D., Tarlecki, A.: Specifications in an Arbitrary Institution. Edinburgh, University of Edinburgh, Department of Computer Science 1985 (= Internal Report CSR-184-85). Partial Extended Abstract in: *Semantics of Data Types. International Symposium. ... 1984. (...)* Ed.: G. Kahn, D. B. MacQueen, G. Plotkin. Berlin *et al.*: Springer 1984 (= LNCS 173). P. 337–356.

[SW 82] Sannella, D., Wirsing, M.: Implementation of Parameterised Specifications. Edinburgh, University of Edinburgh, Department of Computer Science 1982 (= Internal Report CSR-103-82). Extended Abstract in: *Automata, Languages and Programming. Ninth Colloquium ... 1982.* Ed.: M. Nielsen, E. M. Schmidt. Berlin, Heidelberg, New York: Springer 1982 (= LNCS 140). P. 473–488.

[SW 83] Sannella, D., Wirsing, M.: A Kernel Language for Algebraic Specification and Implementation. Edinburgh, University of Edinburgh, Department of Computer Science 1983 (= Internal Report CSR-131-83). Extended Abstract in: *Foundations of Computation Theory. Proceedings ... 1983.* Ed.: M. Karpinski. Berlin *et al.*: Springer 1983 (= LNCS 158). P. 413–427.

— Scherlis, W. L., Scott, D. S.: First Steps Towards Inferential Programming. In: *Information Processing 83. Proceedings of the IFIP 9th World Computer Congress. (...)* Ed.: R. E. A. Mason. Amsterdam, New York, Oxford: North-Holland 1983. Also: Amsterdam: Elsevier 1983. P. 199–212.

[Schoett 81] Schoett, O.: Ein Modulkonzept in der Theorie Abstrakter Datentypen. Hamburg, Universität Hamburg, Fachbereich Informatik 1981 (= Bericht Nr. 81, IfI-HH-B-81/81).

[Schoett 83] Schoett, O.: A Theory of Program Modules, their Specification and Implementation (Extended Abstract). Edinburgh, University of Edinburgh, Department of Computer Science 1983 (= Internal Report CSR-155-83).

[Schoett 85] Schoett, O.: Behavioural Correctness of Data Representations. Edinburgh, University of Edinburgh, Department of Computer Science 1985 (= Internal Report CSR-185-85).

[Statman 85] Statman, R.: Logical Relations and the Typed λ-Calculus. *Information and Control* 65 (1985), p. 85–97.

— Strecker, G. E.: An introduction to categorical methods. Amsterdam, Vrije Universiteit Amsterdam, Rapport nr. 133, 1980.

— Subrahmanyam, P. A.: Nondeterminism in Abstract Data Types. In: *Automata, Languages and Programming. ... 1981.* Ed.: S. Even, O. Kariv. Berlin, Heidelberg, New York: Springer 1981 (= LNCS 115). P. 148–164.

— Sufrin, B.: Formal Specification of a Display-Oriented Text Editor. *Science of Computer Programming* 1 (1982), p. 157–202.

— Swartout, W., Balzer, R.: On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM* 25 (1982), p. 438–440.

[Tarlecki 83] Tarlecki, A.: Free Constructions in Algebraic Institutions. Edinburgh, University of Edinburgh, Department of Computer Science 1983 (= Internal Report CSR-149-83).

— Tarlecki, A.: Quasi-Varieties in Abstract Algebraic Institutions. Edinburgh, University of Edinburgh, Department of Computer Science 1984 (= Internal Report CSR-173-84).

[TWW 82] Thatcher, J. W., Wagner, E. G., Wright, J. B.: Data Type Specification: Parameterization and the Power of Specification Techniques. *ACM Transactions on Programming Languages and Systems* 4 (1982), p. 711–732.

[van Wijngaarden *et al.* 76] van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T., Fisker, R. G., ed.: *Revised Report on the Algorithmic Language Algol 68.* Berlin, Heidelberg, New York: Springer 1976.

[Wand 79] Wand, M.: Final Algebra Semantics and Data Type Extensions. *Journal of Computer and System Sciences* 19 (1979), p. 27–44.

— Wand, M.: First-Order Identities as a Defining Language. *Acta Informatica* 14 (1980), p. 337–357.

— Wand, M.: Specifications, Models, and Implementations of Data Abstractions. *Theoretical Computer Science* 20 (1982), p. 3–32.

— Whitehead, A. N., Russell, B.: *Principia Mathematica. To* *56. Second ed., Cambridge: Cambridge University Press 1927.

— Wirsing, M., Broy, M.: An Analysis of Semantic Models for Algebraic Specifications. In: *Theoretical Foundations of Programming Methodology. Lecture Notes* .... Ed.: M. Broy, G. Schmidt. Dordrecht, Boston, London: D. Reidel 1982 (= NATO Advanced Study Institutes Series C Vol. 91). P. 351–412.

[Wirsing *et al.* 83] Wirsing, M., Pepper, P., Partsch, H., Dosch, W., Broy, M.: On Hierarchies of Abstract Data Types. *Acta Informatica* 20 (1983), p. 1–33.

— Wirth, N.: Program Development by Stepwise Refinement. *Communications of the ACM* 14 (1971), p. 221–227.

— Wulf, W. A.: Abstract Data Types: A Retrospective and Prospective View. In: *Mathematical Foundations of Computer Science 1980. Proceedings of the 9th Symposium. (...).* Ed.: P. Dembinski. Berlin, Heidelberg, New York: Springer 1980 (= LNCS 88). P. 94–112.

List of Figures

List of Definitions

Author's address:

Oliver Schoett
Institut für Informatik
Technische Universität München
Postfach 20 24 20

8000 München 2

West Germany

infma501@dm0tui1s.bitnet

This thesis was typeset using Donald E. Knuth's TEX typesetting program with an early version of Leslie Lamport's LATEX macro package and fonts from the 'Almost Computer Modern' font family. The design was done by the author and conforms to Edinburgh University thesis regulations. The thesis was printed on a CANON CX print engine at 300 dots-per-inch resolution. This engine was coupled to an 'Advanced Personal Machine' developed at the University of Edinburgh, and was driven by software written by Robert G. Struthers.

Declaration

This thesis has been written by myself, and the work reported is my own.

Some basic ideas of this work have been presented in [Schoett 81], which, apart from minor corrections and editorial changes, is identical to a thesis submitted and accepted for the *Diplom-Informatiker* degree at the University of Hamburg, West Germany. In particular, [Schoett 81] anticipates the module concept, a behavioural implementation notion, the identification of the correctness problem of data abstraction in modular programming, and the idea of solving it by a constraint on the implementation modules.

The present thesis goes beyond this by providing an institutional framework suitable for program entities other than data types and partial functions, for arbitrary specification languages and for different representation concepts, by dealing with requirements of implementation cells and with infinite systems, by providing the general concepts of "universal implementation", "simple implementation" and "stability", by analysing and comparing three different concrete representation notions, and by introducing correspondences as a tool useful in the theory as well as in practical data representation correctness proofs.

Two internal reports of the Department of Computer Science, [Schoett 83] and [Schoett 85], contain material from this thesis. The latter has been accepted for publication in *Science of Computer Programming* on condition that it be revised according to the referees' suggestions.

Edinburgh, 27 September 1986

Oliver Schoett