Constructing and Refining Modules in a Type Theory

Shahad N. Ahmed

A thesis submitted for the degree of Doctor of Philosophy
to the
Department of Computing Science,
University of Glasgow.
June 1994

Thesis
10130
Copy 1

Abstract

The need to apply formal specification and development of programs to large problems has highlighted a need for methods to support modular development. This has two aspects: the modular construction of specifications, and the implementation of modular specifications. This thesis is concerned with both these activities.

The main body of work in the development of modular specifications has been carried out in the context of algebraic specification languages, and model-based languages such as Z. However, these languages fail to provide some important mechanisms for structuring specifications. Furthermore, the complex semantics of these languages lead to complicated definitions of what it means for a program to be an implementation of a specification.

In this thesis, we show that Martin-Löf's Type Theory provides a framework for both the specification and implementation of program modules; and this framework addresses the shortcomings, noted above, in other specification formalisms. The basic theoretical notion underlying our approach is that a specification is a type, and that an implementation of such a specification is any element in the type.

We present a module specification language, and its associated implementation language. The semantics of both the specification and implementation languages are defined in Martin-Löf's Type Theory. We define some specification building operators for our specification language, and show how modular specifications may be incrementally constructed using the specification operators. We give some laws about the specification operators and show how these laws can be used to reason about, and restructure, modular specifications.

We define a notion of refinement that supports the implementation of modular specifications by systematic mathematical transformation. We give some refinement laws for refining modular specifications. We also define some operators on program modules, and show how these operators can be used to systematically implement modular specifications.

Acknowledgements

I am greatly indebted to Joseph Morris, my supervisor, whose encouragement, technical assistance and guidance have been invaluable throughout my studentship. I particularly want to thank Joe for commenting on earlier drafts of this thesis.

The Computing Science Department at Glasgow has provided an excellent environment in which to study and research. I would particularly like to thank Kieran Clenaghan, and Alastair Reid for may interesting discussions during my time in the department.

This work was funded by a Research Studentship from the Science and Engineering Research Council of Great Britain.

Finally, I owe a great deal to Omer, to my parents, and my friends for all their help, encouragement and support.

Declaration

The material presented in this thesis is entirely the result of my own independent research carried out at the Department of Computing Science, University of Glasgow under the supervision of Dr. Joseph M. Morris. Any published or unpublished material used by me has been given full acknowledgement in the text.

Contents

Chapter 1

Introduction

As the size and complexity of software systems has increased, so there has been a growing need for better techniques to construct such systems. Some key concepts put forward to control the complexity of software systems are abstraction by specification, modularisation, and stepwise development of programs from specifications. Specifications describe "what" the system must do without necessarily saying "how" it is to be done. Modularisation helps break up large problems into smaller more manageable pieces. Stepwise development helps to bridge the gap between an abstract specification and a concrete implementation in a number of more manageable steps. This thesis is concerned with the specification and development of modular specifications within the single framework of type theory.

Formal specification languages are mathematical notations that allow software systems to be specified in a precise way, so that questions about the design of a system can be discussed and answered before any program code needs to be written. In particular, we can apply mathematical reasoning to the specification of a system to help identify possible problems in its design so that they can be corrected before the system is implemented.

As well as specifying a software system, a formal specification can also play a direct role in making implementations. If we can give a formal relationship between specifications and implementations then it is possible, at least in principle, to derive programs from specifications by correctness preserving mathematical transformations which are usually called *refinements*. Programs can be developed from specifications by working gradually through a series of refinements until we arrive at a program. This is essentially the discipline of stepwise refinement originally advocated by Wirth [57] and Dijkstra [11].

Specifications of large systems can be as complicated in their structure as programs. We take the view that such specifications should be built in a modular fashion from small and relatively independent specification units or *modules.* Apart from the obvious benefit of breaking large specifications into more manageable pieces, there are other benefits from modularisation. Firstly, it makes the specification more readable and understandable. Secondly, modularisation helps to isolate individual parts of the specification from changes to other parts of the specification; this helps to minimise the amount of re-specification required when we change part of a specification. Thirdly, it may help in making implementations from specifications. Fourthly, it makes it more likely that parts of the specification can be reused in other applications. The last point encourages the building of libraries of reusable specifications and their implementations; this may help reduce the cost—in time and manpower—of specifying and implementing future systems.

The main body of work on modular specifications has been carried out in the context of algebraic specification languages such as CLEAR [5], OBJ [14] and Extended ML [50]. Work has also been done on providing module constructs for model-oriented specification languages; for example, there have been proposals for module facilities for Z [47] and VDM [32]. In our view, the problem with many of these approaches, and in particular the algebraic approach, is that the semantics of such languages are complex and require the complex combination of many different formalisms including logic, set theory and lambda calculus.

Through the work of Martin-Löf [30, 31, 44] and others, it is known that type theories can also be used to provide a framework for specification and program development. In contrast to algebraic semantics, type theory has the advantage that it unifies first order logic, set theory and lambda calculus within one framework. In particular, the notion of *propositions as types* leads to a very simple notion of specification and implementation: specifications are types, and the objects in the types are implementations (programs) satisfying the specification. Note the distinction between specifications and implementations.

However, very little work has been done on the modular development of specifications and implementations in type theory, work by Luo [27], Burstall et al [6] and Nordström et al [44] being the exceptions. It is clear to us that many of the features of type theory provide a framework for the modular development of specifications and programs. In particular, the notion of modules as a collection of inter-dependent components can be readily specified in type theory using the notion of dependent types. The use of type theory provides a simple notion of implementation and also provides a calculus that facilitates the implementation of program modules from specifications.

Our thesis is that type theory provides a framework for both the specification and development of program modules. We show how type theory can be used to give a semantics to a module specification language and an associated implementation language. We show how modular specifications may be incrementally constructed by combining smaller specifications using specification building operators. We also show how implementations of such modular specifications may be calculated by systematic refinements.

1.1 Types as Specifications

Whenever we see an expression like $x : P$ in a programming language we often read it as saying "x has type P". A more unusual way of reading $x : P$ is to think of it as the requirement that says "find a value x that has type P": we think of type P as a specification, and any value in the type P is acceptable as a correct implementation of P. For example, $f : \mathbb{N} \rightarrow \mathbb{N}$ says let f be any function of type $\mathbb{N} \rightarrow \mathbb{N}$, so that both $f = \lambda x.x + 1$ and $f = \lambda x.x * x$, to take two examples, are acceptable implementations for f.

However, simple types such as we know them from programming languages are of little use as specifications since we cannot express the behaviour of objects such as functions, purely using such types. Our approach is to use a very rich type system, such as Martin-Löf's type theory (MLT), which allows us to specify the detailed behaviour of objects purely by their type; objects include functions and—the main interest to us—modules. The task of implementation is then that of finding any value in this type.

Martin-Löf's type theory has the usual standard types such as $\mathbb{N}$ and $\mathbb{B}$, together with the usual operations on them such as $+$, $-$, etc and $\wedge$, $\neg$, etc, respectively. The type theory also includes type constructors such as $\rightarrow$ and $\times$ for forming function and pair types, respectively. Pair values have the usual projection functions *fst* and *snd* to extract the individual elements. One of the interesting features of the type theory is that functions and types can be treated as values. For example, both functions and types may be actual parameters to functions.

Another important feature of the type theory is that types themselves have types. In type theory, types of types are called *kinds*. The kind U_1 (known as the *1st universe*) is the kind of simple types such as those introduced so far (but U_1 is not an element of itself); for example, $\mathbb{N} \in U_1$, $\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{B}) \in U_1$ etc. There is a hierarchy of universes U_1, U_2, ... etc such that U_i $(i > 0)$ and all its elements are elements of U_{i+1}. One unusual type available in the type theory is the empty type Φ that contains no values. Any specification equivalent to Φ is treated as an inconsistent specification since to find a value $x : \Phi$ is impossible.

One of the interesting features of Martin-Löf's type theory is the notion of dependent types. One example of such types are the *dependent function* types. The elements of a dependent function type are functions for which the *type* of their result is determined by the *values* of the arguments applied to them. The type $\prod x \in P.Q(x)$ is an example of a dependent function type, where P is a type and $Q(x)$ is a type expression defined in terms of variable x of type P. The elements of $\prod x \in P.Q(x)$ are functions that take an element—p, say—of type P and return an element of type $Q(p)$. For example, the elements of type

$$\prod x \in \mathbb{N}.(\textbf{if } x \neq 0 \textbf{ then } \mathbb{N} \textbf{ else } \mathbb{S})$$

are functions that take a value—n, say—of type $\mathbb{N}$ and if $n \neq 0$ they return a natural, otherwise if $n = 0$ they return a string.

Another form of dependent type are the *dependent product* types. The type $\sum x \in P.Q(x)$ is a typical dependent product type, where P is a type and $Q(x)$ is a type expression defined in terms of variable x of type P. The elements of this type are pairs $\langle p, q\rangle$ where p has type P and q has type $Q(p)$; observe that the type of q is dependent on the value p. As an example, the elements of type $\sum x \in U_1.x$ are all the pairs where the first component is a type and the second component is an element of that type. The pairs $\langle \mathbb{N}, 6\rangle$, $\langle \mathbb{N}, 0\rangle$, $\langle \mathbb{B}, \mathit{true}\rangle$, $\langle \mathbb{S}, \text{“hello”}\rangle$ etc are all elements of $\sum x \in U_1.x$.

We note in passing that the normal function types $P \rightarrow Q$ are a special case of the dependent function types and are equivalent to $\prod x \in P.Q$ where Q is not dependent on x. Similarly, types of the form $P \times Q$ are equivalent to $\sum x \in P.Q$ where Q is not dependent on x.

1.1.1 Propositions as Types

One of the features of type theory that makes it attractive for program specification is that propositions can be expressed in the type theory we have just been describing. There is no need for additional logical constructs. Martin-Löf's type theory embodies an *intuitionistic* or *constructive* logic that is based on identifying propositions with types. If P is a proposition (i.e. a well-formed formula constructed from the logical connectives $\wedge$, $\vee$ etc), then the “judgement” $p \in P$ means that p is a proof of P. In other words, P is identified with the type (or set) of its proofs. That a propositions is true then means that its corresponding type is non-empty (i.e. “inhabited”). The identity between propositions and types is generally attributed to Curry and Howard [20], and is called the principle of *propositions as types*.

Each of the familiar logical constructs has its counterpart in type theory; for example, $P \wedge Q$ corresponds to the type $P \times Q$, and $\exists x \in P.Q(x)$ corresponds to $\sum x \in P.Q(x)$. Figure 1.1 give the correspondence between the operators of first order logic and their corresponding type constructors in Martin-Löf's type theory. In the first column in Figure 1.1, P and Q denote propositions and in the second column P and Q denote types; p_1 and p_2 denote values of type P.

Proposition	**Type**	**Type Name**
True	T	Top - The unit type
False	ϕ	Empty - the type with no values
$P \wedge Q$	$P \times Q$	Cartesian product
$P \vee Q$	$P + Q$	Disjoint sum
$P \Rightarrow Q$	$P \rightarrow Q$	Function space
$\neg P$	$P \rightarrow \Phi$	
$\forall x \in P.Q(x)$	$\prod x \in P.Q(x)$	Dependent function space
$\exists x \in P.Q(x)$	$\sum x \in P.Q(x)$	Dependent product
$p_1 = p_2$	$[p_1 =_P p_2]$	Equality type on P - inhabited iff $p_1 = p_2$

Figure 1.1: Identity between propositions and their corresponding type

Figure 1.1 introduces some types not seen before. The unit type, T, contains the single value tt; T corresponds to the classical proposition True. The equality type $[p_1 =_P p_2]$ compares the values p_1 and p_2 of type P for equality. If p_1 and p_2 are equal then the type $[p_1 =_P p_2]$ is inhabited by the constant eq. If p_1 and p_2 are not equal then $[p_1 =_P p_2]$ is equivalent to the empty type.

The propositional connectives ($\wedge, \vee$ etc) are used to stand for the corresponding type connectives ($\times, +$ etc) whenever types are used as propositions; for example, if the type $P \times Q$ is used as a proposition we shall write it as $P \wedge Q$. The precedence for each type connective is equivalent to the precedence of the logical connective it identifies, and grouping the logical connectives in decreasing precedence we get: $=$; $\neg$; $\wedge$, $\vee$, $\Rightarrow$.

Of course one has to give an argument that the correspondence between propositions and types is a reasonable one, and that argument has been given, e.g. in [30, 31, 44]. The actual values in a propositional type are not important, its truth is determined only by whether the type is inhabited. Any proposition true intuitionistically is true classically, but one limitation of constructive logic is that propositions that can be proved true classically cannot always be proved true constructively.

1.1.2 Specification and Development

Martin-Löf [30] and others [44] have shown that it is possible to express the properties of programs by combining the notion of dependent types and propositions as types. For example, the following type specifies a function for computing the natural number square root:

$$\sum f \in \mathbb{N} \rightarrow \mathbb{N} \,.\, (\forall n \in \mathbb{N}.\, f(n)^2 \leq n \;\wedge\; (f(n)+1)^2 > n)$$

The "for all" proposition in the above type stands for the corresponding $\prod$-type under the principle of propositions as types. The proposition holds for all f integer square root functions. Since the above type is a $\sum$-type, its elements are pairs whose first component contains a natural number square root function; the second component is a value of the propositional type that constrains f. The actual value of the second component—which is called a *witness*—is not important, it simply witnesses that the first component satisfies the proposition defined by the above type.

Developing a program from a specification is the activity of constructing an element of the specification regarded as a type. The language used to express the values in the types available in MLT constitutes a functional programming language. So programs are expressions without assignment and other side-effects. The development of programs from specifications is formalised in a constructive programming logic which has inference rules for both deducing the correctness of a program, and constructing programs to meet specifications. The constructive implementation of small specifications in type theory is reasonably developed in the literature [3, 44], but little work has been done on the constructive implementation of large specifications.

1.1.3 Different Formulations of Type Theory

Martin-Löf's Type Theory

There are several different formulations of Martin-Löf's type theory [28, 29, 30, 31]. The formulation of MLT in [29] is intensional. Intensional means that judgemental equality is understood as definitional equality; in other words, two types are equal only if there is a type rule stating that they are equal. One consequence of an intensional theory is that equality is decidable.

The type theory we use is an extension of that presented in [30, 31]. This theory is said to be polymorphic and extensional. Extensional equality means the same as in ordinary set theory: two types are equal if they have the same elements. However, judgemental equality is not decidable in the extensional theory. In the extensional theory, any well-typed expression can be computed to a normal form. Furthermore, the extensional theory computes expressions using lazy evaluation.

One of the features of MLT is that it is open to extension, and our thesis makes use of several extensions to the theory. Nordstrom et al [44] and Constable et al [8] have extended the theory to include list and subset types—the work of Constable et al is done in the context of constructing the *Nuprl* proof development system. Backhouse et al [3] show how the theory can be extended in a uniform manner to introduce a rich collection of data types such as sets, bags, guarded expressions and polymorphic functions.

The Calculus of Constructions

The Calculus of Constructions (CC) [10] is another type theory used for program specification and development. CC differs from MLT in several ways. Firstly, it is impredicative: there is no hierarchy of type universes, although there is a kind *Type* of all types (but $Type \notin Type$). Secondly, it does not define a dependent sum type constructor ($\sum$): this omission makes CC less suited to making structured specifications since, as we shall see later, the ability of $\sum$ to combine types and express dependencies within types is important for making module specifications. Thirdly, CC only introduces a primitive collection of types: data-types, such as lists, must be constructed from these primitive types. Fourthly, CC makes a distinction between types used as data types and types used as propositions: there is a kind *Prop* of all types used as propositions, and an axiom that states that all elements of *Prop* are also elements of *Type*. So propositions are types, but types are not necessarily propositions. The advantage of defining an impredicative universe *Prop* is that second-order logic can be interpreted in CC.

More recently, Luo [24, 26] has extended CC with predicative type universes $Type_1$, $Type_2, \ldots$ etc and $\sum$-types to produce the Extended Calculus of Constructions (ECC). ECC can be viewed as an extension of MLT since ECC contains the key features of MLT, and also includes an impredicative universe for propositions.

1.1.4 Why Martin-Löf's Type Theory?

In principle, there is no reason why the methodology presented in this thesis could not be formulated in other type theories such as Nuprl [8]: any type theory with propositions as types and $\sum$-types could be used. Indeed, during our work we have become aware of the work of Luo [27] in specifying and refining structured specifications in ECC.

Although we use an extensional version of MLT, the work presented in this thesis does not rely on the extensional features of the type theory. Indeed, our work can also be formulated in an intensional version of MLT, such as that presented in [44]. Intensional type theories lend themselves to the implementation of machine based proof development systems since type checking and equality are decidable in such theories. Consequently, formulating our work in an intensional version of MLT would make it easier to implement a proof development system for verifying properties about our specifications and implementations. However, we choose to work in the extensional formulation of Martin-Löf's type theory [30, 31, 44], partly since it is amenable to extension, but also because it is well established in the literature.

In summary, Martin-Löf's type theory provides a single framework for the study of both program specification and development: the formal language of type theory is used as a programming language, a specification language, and a programming logic.

1.2 Specifications and Modules

In this thesis, we are concerned with the theory underlying the construction and development of modular specifications. We develop a simple module specification and implementation language whose semantics is given by MLT. The purpose of this section is to introduce the style of module specifications used in this thesis.

By definition, a specification is a type. In particular, a specification consists of the product of a *signature*, and a proposition over the components defined in the signature. A signature is a type and is similar, syntactically, to a signature in the programming language Standard ML [33], and a tuple type as in Quest [7]. The proposition—which is also a type—specifies the properties that an implementation of the specification must satisfy; the proposition is called the *restriction* of the specification.

The members of a specification are implementations that satisfy the specification and are called *modules*. Modules are analogous to ML structures and Quest tuples, where each component has a name, and can be referred to via a Pascal style dot notation. A module that satisfies a specification must meet two conditions: firstly, it must contain all the components defined in the signature of the specification, and secondly, its components must satisfy the restriction of the specification.

1.2.1 Specifications

The simplest form of specifications are called *canonical specifications.* Canonical specifications are the basic building blocks of more complicated structured specifications. The specification *Catalogue*, given in Figure 1.2, is an example of a canonical specification that specifies a catalogue of books for a library. The declarations following the keyword **Elements** denote the signature of the specification, and the proposition following the keyword **Restrictions** is the restriction of the specification. The first component *Book* has type U_1; so *Book* is a type. Note that there are no specific constraints on *Book*: *Catalogue* is only intended to specify a "container" for *Book* values and it says nothing about how to make, or modify, *Book* values. Therefore, *Catalogue* can be viewed as a partial specification that specifies the book catalogue without worrying about details concerning books at this stage. It is our intention to add details about *Book* later: in general, we shall advocate a style of specification

$$
\begin{array}{l}
\mathit{Catalogue} \equiv \\
\quad \textbf{Elements} \\
\quad\quad \mathit{Book} \in U_1, \\
\quad\quad \mathit{Stock} \in U_1, \\
\quad\quad \mathit{empty} \in \mathit{Stock}, \\
\quad\quad \mathit{add} \in \mathit{Book} \rightarrow \mathit{Stock} \rightarrow \mathit{Stock}, \\
\quad\quad \mathit{remove} \in \mathit{Book} \rightarrow \mathit{Stock} \rightarrow \mathit{Stock}, \\
\quad\quad \mathit{instock} \in \mathit{Book} \rightarrow \mathit{Stock} \rightarrow \mathbb{B}, \\
\quad\quad \mathit{isempty} \in \mathit{Stock} \rightarrow \mathbb{B} \\
\quad \textbf{Restrictions} \\
\quad\quad \forall s \in \mathit{Stock}.\forall m, n \in \mathit{Book}. \\
\quad\quad \mathit{isempty}(\mathit{empty}) =_{\mathbb{B}} \mathit{true} \quad \wedge \\
\quad\quad \mathit{isempty}(\mathit{add}(m, s)) =_{\mathbb{B}} \mathit{false} \quad \wedge \\
\quad\quad \mathit{instock}(m, \mathit{empty}) =_{\mathbb{B}} \mathit{false} \quad \wedge \\
\quad\quad \mathit{instock}(m, \mathit{add}(m, s)) =_{\mathbb{B}} \mathit{true} \quad \wedge \\
\quad\quad \neg(m =_{\mathit{Book}} n) \Rightarrow \mathit{instock}(n, \mathit{add}(m, s)) =_{\mathbb{B}} \mathit{instock}(n, s) \quad \wedge \\
\quad\quad \mathit{remove}(m, \mathit{empty}) =_{\mathit{Stock}} \mathit{empty} \quad \wedge \\
\quad\quad \mathit{remove}(m, \mathit{add}(m, s)) =_{\mathit{Stock}} \mathit{remove}(m, s) \quad \wedge \\
\quad\quad \neg(m =_{\mathit{Book}} n) \Rightarrow \mathit{remove}(n, \mathit{add}(m, s)) =_{\mathit{Stock}} \mathit{add}(m, \mathit{remove}(n, s)) \\
\quad \textbf{End}
\end{array}
$$

Figure 1.2: The specification of a Book Catalogue

that often begins by making a partial specification and then adding detail to it until we arrive at a final specification. The second component *Stock* is also specified to be a type. The values of *Stock* denote collections of books. The signature of *Catalogue* also includes operations to add and remove books to and from a library stock. The operation *instock* queries whether a book is in stock; *empty* represents is an empty stock value; and *isempty* queries whether a stock value is empty.

Note that the types of the operations in *Catalogue* are dependent on the type components *Book* and *Stock*. In general, the type of a component may be dependent on any component preceding it in the signature; this includes non-type components such as functions, although in practice, we rarely specify dependencies on non-type components. The restriction may be any proposition allowed in type theory. In practice, the restriction usually takes the from exemplified by the restriction of *Catalogue*; a conjunction of equations that is quantified over the types defined in the signature. Some of the equations in the restriction of *Catalogue* are guarded using logical implication; such equations are called "conditional equations".

The semantics of specifications—and hence modules—are defined in terms of the dependent sum type ($\sum$): the semantics will be given, in Chapters 4 and 5, by translating specifications into terms denoting types in Martin-Löf's type theory. The idea that $\sum$-types can be used to give the type of modules was reported independently

$$
\begin{array}{l}
\mathit{CatalogueModule} \equiv \\
\quad \textbf{module} \\
\qquad \begin{array}{lcl}
\mathit{Book} & = & \mathbb{N}, \\
\mathit{Stock} & = & \mathit{Set}(\mathit{Book}), \\
\mathit{empty} & = & \{\}, \\
\mathit{add} & = & \lambda b \in \mathit{Book}.\lambda s \in \mathit{Stock}.\{b\} \cup s, \\
\mathit{remove} & = & \lambda b \in \mathit{Book}.\lambda s \in \mathit{Stock}.(s - \{b\}), \\
\mathit{isempty} & = & \lambda s \in \mathit{Stock}.(s = \{\}), \\
\mathit{instock} & = & \lambda b \in \mathit{Book}.\lambda s \in \mathit{Stock}.(b \in s)
\end{array} \\
\quad \textbf{proof} \\
\qquad \lambda s.\lambda m.\lambda n.\langle eq, eq, eq, eq, \lambda x.eq, eq, eq, \lambda x.eq\rangle \\
\quad \textbf{end} \in \mathit{Catalogue}
\end{array}
$$

Figure 1.3: A Module Satisfying *Catalogue*

in [43] and [34], but only [43] shows that the behaviour of the components in a module can be specified by adding propositions to its type. A detailed discussion of [43] is given in Chapter 4.

1.2.2 Modules

Figure 1.3 gives an example of a module, named *CatalogueModule*, that satisfies the specification *Catalogue*. Note that we have chosen to implement the *Book* component in *CatalogueModule* by the type $\mathbb{N}$; so book values are implemented as naturals which are intended to denote individual books. The stock of the library is represented by a set of *Book* values. We call that part of a module obtained by omitting the keyword **proof** and the value that follows it, the *computational element* of the module. The value after the keyword **proof** may seem a little unusual; it is called a *witness* and it proves that *CatalogueModule* satisfies *Catalogue*. For the moment, we ignore witnesses—they are discussed in Chapter 4—and we will often omit them when writing modules, replacing them by "...".

The components in a module may be referred to via their names using a Pascal style dot notation. For example, the *Book* component may be referenced by *CatalogueModule.Book*; so *CatalogueModule.Book* $= \mathbb{N}$. The following give some more example of dot notation:

$$\mathit{CatalogueModule.isempty} = \lambda s \in \mathit{Set}(\mathbb{N}).(s = \{\})$$
$$\mathit{CatalogueModule.isempty}(\mathit{CatalogueModule.empty}) = \mathit{true}$$

When we refer to a component by dot notation, all its dependencies on other components are removed by substitution. For example, *CatalogueModule.Stock* $= \mathrm{Set}(\mathbb{N})$; the dependency *Stock* has on *Book* has been removed by substituting the actual implementation of *Book* for the name *Book*.

The computational element of a module can be used like a record value. The types of computational elements will be defined to be signatures. If a module satisfies a specification then its computational element is a member of the signature of the specification; the witness of the module is a member of the restriction of the specification. We write computational elements like modules but omit the keyword **proof** and the witness—computational elements should not be confused with modules in which we replace the witness by "...".

1.2.3 Domains: Name-spaces in Specifications and Modules

We call the collection of component names used in a specification the *domain* of the specification, and we call the collection of component names used in a module the *domain* of the module; and so on for signatures and computational elements. As the component names in specifications and modules are given linearly, we treat domains as lists of names.

The issue of component names within specifications and modules is not addressed in any detail in [44] and [34]; component names are treated as nothing more than bound variables. Consequently, in [44] one specification cannot refer to the components of another specification by name. Furthermore, [44] does not name components in modules: components are referred to by their position in a module.

We take the view that component names play a dual role: on the one hand they are used as bound variables to specify the dependencies between components within specifications and modules, and on the other hand they allow a specification, or module, to refer to components within other specifications and modules. We will give names a formal semantics that caters for both theses roles. A formal treatment of names in modules leads to a formal definition of dot notation on modules. As we shall see in the next section, names are also used by specification operators to allow us to identify components to be renamed, hidden or combined.

1.3 Structured Specifications

The modular development of specifications is supported by operators that allow us to combine and modify specifications. The specification operators are formally defined as transformations that take a number of arguments, which may be specifications, and return specifications as results. The specification operators include *renaming*, which is used to change the names of components in a specification; and *hiding*, which is used to remove components from a specification while still preserving the behaviour of

$BookSpec \equiv$
Elements
$Book \in U_1,$
$mkBook \in (Author \times Title \times Id) \rightarrow Book,$
$author \in Book \rightarrow Author,$
$title \in Book \rightarrow Title,$
$bookId \in Book \rightarrow Id$
Restrictions
$\forall a \in Author. \forall t \in Title. \forall i \in Id$
$author(mkBook(a,t,i)) = a \quad \wedge$
$title(mkBook(a,t,i)) = t \quad \wedge$
$bookId(mkBook(a,t,i)) = i$
End

Figure 1.4: A Specification for Books

the remaining components. Other operations include *sum*, which combines two specifications into one that contains all the components from both; and *enrichment* which is used to add new components and restrictions to specifications. The operations can be used to construct large specifications incrementally in a piecewise manner by constructing smaller relatively independent specifications which are combined, using the operators, to produce large specifications. The operators also have useful algebraic properties which can be used to reason about structured specifications.

We now give some small examples of incrementally constructing specifications. We consider extending our book catalogue example by adding operations on books. We develop a separate specification called *BookSpec*—given in Figure 1.4—which specifies a book type whose values give the author, title and book identifier of a book. In practice, we would parameterise *BookSpec* with respect to the types *Author*, *Title* and *Id*, but for simplicity we shall assume these types as being given in some global context; for example, *Author* and *Title* could be defined as strings, and *Id* by N.

Using the *sum* operator (+), we may construct a new specification for the book catalogue by combining *Catalogue* with *BookSpec* to give the specification $Catalogue_2$:

$$Catalogue_2 = Catalogue + BookSpec$$

The new specification $Catalogue_2$ specifies modules that contain all the components specified by *Catalogue* and *BookSpec*, such that these modules satisfy the original constraints on both *Catalogue* and *BookSpec*. There is a potential problem with name-clashes when summing *Catalogue* and *BookSpec* since both contain a component, *Book*, with the same name. The sum operator handles such name-clashes by treating each occurrence of *Book* as being distinct, so that *Book* appears twice in $Catalogue_2$—we say more about resolving name-clashes in Chapter 7. The restriction of $Catalogue_2$ is the conjunction of the restrictions on *Catalogue* and *BookSpec*.

The *hide* ($\backslash$) operator is used to remove components from the signature of a specification. For example, the specification $Catalogue_3$ illustrates the use of the *hide* operator to remove the components *mkBook*, *author*, *title* and *bookId* from the signature of $Catalogue_2$:

$$Catalogue_3 = Catalogue_2 \setminus \{mkBook, author, title, bookId\}$$

If we were to continue to specify a complete library system, then we might wish to specify a module for a register of library users. Such a specification would be very similar to *Catalogue*, as we would require operations to add and remove library users to and from a library. We would also require operations to query the status of a register and individuals. One possible specification of a register module may be defined simply by renaming some of the components of *Catalogue*. The specification *Register*, below, shows the use of the rename operator to specify a module for a register of library users. The renaming specifies that *Book* is to be renamed as *Person*, *Stock* is to be renamed as *Users* etc.

$$\begin{aligned} Register = Catalogue[&Book\backslash Person, Stock\backslash Users, add\backslash addUsers,\\ &remove\backslash RemoveUsers, instock\backslash registered] \end{aligned}$$

In practice, when we reuse a specification, such as *Catalogue*, it may contain components that we don't require, and these may be hidden using the *hide* (or *derive*—see later) operator. Additionally, we may want to add some new components specified in terms of existing components, and these may be added using the *enrich* operator ($\lhd$). For example, suppose we require an extra operation within a register module so that we can calculate the number of registered users. The new operation, which we call *size*, may be specified as the following enrichment of *Register*:

$$\begin{aligned}
Register_2 = Register \lhd \ & \Lambda\textbf{Elements}\\
& \quad size \in Users \rightarrow \mathbb{N}\\
& \textbf{Restrictions}\\
& \quad \forall s \in Users.\forall m \in Person.\\
& \quad size(empty) = 0 \wedge\\
& \quad size(addUser(m,s)) = \textbf{if } registered(m,s) \textbf{ then}\\
& \qquad\qquad\qquad\qquad\qquad size(s)\\
& \qquad\qquad\qquad\qquad\quad \textbf{else}\\
& \qquad\qquad\qquad\qquad\qquad size(s)+1\\
& \textbf{End}
\end{aligned}$$

1.4 Refinement

We will define a notion of refinement between specifications which facilitates the formal development of program modules from specifications. Refining a specification may be regarded as the task of adding implementation decisions about properties of the specification that were left open by the specifier—the choice of algorithms, data structures and error messages for example. In the context of type theory, we regard refinement as being different from implementation. Specifications are types, and any refinement of a specification is also a specification and hence a type. Implementations are values of a type, and so we regard the implementation task as that of finding values within a type. We don't make implementations in one go, instead we proceed to an implementation by refining the specification. Implementation becomes easier the more we refine a specification, and so we keep refining a specification until we arrive at a specification that strongly suggests an implementation. Then we proceed to find an implementation from the specification by using a constructive proof.

We give a type-theoretic definition of refinement that allows all types used as specifications—not just module specifications—to be refined. However, we concentrate on the refinement of module specifications. In its simplest form, module refinement says that a specification SP_1 refines to a specification SP_2, written $SP_1 \sqsubseteq SP_2$, if every implementation of SP_2 is an implementation of SP_1. In other words, $SP_1 \sqsubseteq SP_2$ if SP_2 is a subtype of SP_1. The actual definition of module refinement is more powerful than that described above as it will allow *data refinements* in which the signature of SP_2 may differ from SP_1. Typically, specifications are data refined by adding new components to their signature, or by changing the type of components within their signature. When data-refining SP_1 to SP_2, the relationship between the signature of SP_1 and SP_2 is given by supplying an *abstraction function* which relates implementations of SP_2 to implementations of SP_1.

The module refinement relation is transitive. Transitivity allows us arrive at a refinement via a sequence of intermediate refinements. Moreover, we may refine a structured specification by refining its individual parts in relative isolation; in other words, the specification operators are monotonic with respect to the refinement relation. The practical consequence of monotonicity is that the "shape" of a specification can be carried over to its refinements. Transitivity and monotonicity give the basic requirements necessary for stepwise development of specifications.

In general, verifying that one specification is a refinement of another is difficult. Therefore, we supply a collection of refinement laws that can help make refinements. Module specifications tend to be large and complex, and in order to refine them we need a systematic approach that simplifies the refinement task. We advocate a piecewise approach to refinement which decomposes the refinement of large specifications into the task of refining their constituent pieces in relative isolation; laws are given that support this methodology. Not all specifications are suited to piecewise refinement, and we consider some refinement laws that use the specification operators to decompose specifications into forms more amenable to piecewise refinement.

1.5 Implementation

Once a specification has been sufficiently refined, its refinement can be implemented to produce a program. A specification is usually refined until it is in a form in which the restrictions of each of its constituent canonical specifications suggests an implementation. Each canonical specification can then be implemented by a constructive proof. This step is usually clerical. Consequently, the implementation of canonical specifications is not very interesting and is not considered in this thesis. Instead, we consider the implementation of structured specifications (i.e. specifications that use the structuring operators). We outline how the shape of a specification may sometimes be used to decompose the implementation task into that of implementing individual pieces of a specification, so that the implementations may be glued together to form an implementation of the specification.

The piecewise approach to refinement, described in the previous section, advocates refining modular specifications by refining their individual parts in relative isolation. Ultimately, the individually refined parts must be implemented and combined. For example, suppose we are given two specifications SP_1 and SP_2 that are combined using the sum operator to make a new specification $SP_1 + SP_2$. If we can implement SP_1 by module m_1, and implement SP_2 by a module m_2, then we might ask whether there is a combinator at the implementation level that can combine m_1 and m_2 to give an implementation of $SP_1 + SP_2$. We will show that for most of our specification operators, analogous combinators do exist at the implementation level. However, we show that some of the implementation combinators, such as *enrich*, only guarantee a correct implementation under certain conditions.

1.6 Parameterisation

One important way specifications can be structured is by parameterisation. Parameterisation of specifications allows a specifier to make general purpose specifications that may be instantiated in different ways. Parameterisation therefore provides a useful mechanism for the reuse of specifications. The higher-order features of type theory are particularly suited to describing parameterised specifications. Parameterised specifications can be described directly as functions that take a number of arguments, possibly including specifications, and return a specification as result. We show how this style of parameterisation can be used to structure large specifications, and also to decompose large specifications when developing implementations. We can also introduce parameterised modules. Parameterised modules are functions that return program modules as their result. We show that parametrised modules can be specified by using the dependent function type constructor ($\prod$). In particular, we show that parameterised specifications can be implemented as parameterised programs, and that $\prod$-types play a useful intermediary role in the development of parameterised and non-parameterised specifications.

1.7 Related Work on Specifying Modules

In this section, we discuss related work on specifying and implementing modules in various formal frameworks. We begin by mentioning two approaches to constructing modules in type theory. Then we consider related work in algebraic and model-based specification methods. Roughly speaking, the aim of a model-based specification is to build an abstract model of the program being specified; this is in contrast to algebraic specifications, which describe a system in terms of its desired properties without constructing an explicit model.

1.7.1 Type Theoretic Module Specifications

Nordström and Petersson [43, 44] were the first to show that a type theory such as MLT could be used to specify modules, and their work lays down the foundation on which our thesis is based. Nordstrom et al show that abstract data types—such as stacks—can be specified as $\sum$-types whose elements are tuples containing a type together with associated operations on the type; they consider tuples to be modules. They also show that parameterised modules can be defined as functions returning modules, and may be specified by dependent function types ($\prod$). However, Nordstrom et al do not consider the specification of modules that are not abstract data types; nor do they investigate the use of specification operators, or a definition of refinement for developing implementations. A more detailed review of the semantics of Nordstrom et al is given in Chapter 4 which discusses the semantics of specifications.

Luo [27] shows that modules can be specified, and implemented, in the Extended Calculus of Constructions (ECC). Luo defines specifications not as types, but as pairs; the first component is a type called the *structure type* and specifies the signature of a module; the second component is a predicate—on the elements of the structure type—specifying the behaviour of the components declared by the structure type. Luo argues that, by using pairs, it is easier to separate the computational content of a specification (expressed by the structure type) from the axiomatic requirements; this, it is argued, helps to avoid computationally redundant witnesses appearing in implementations of such specifications. The idea of defining specifications as pairs is also advocated by Burstall et al [6] who use such pairs to make specifications—which they call "deliverables"—in ECC.

Luo also defines a refinement relation on specifications which is similar to that described in section 1.4, and it is also similar to the notion of refinement for deliverables, given in [6]. Luo also identifies two general classes of specification operators called *constructors* and *selectors* which can be used to define operations such as joining two specifications and enriching a specification: constructors and selectors are also monotone with respect to the refinement relation. Name-space issues within specifications and implementations are not considered, so that operations such as renaming and hiding components are not considered. We give a detailed comparison between our work and Luo's in Section 10.3.

1.7.2 Algebraic Specifications

Although we are concerned with the specification and development of modules in type theory, our work is closely related to specifying modules using algebraic specifications. Algebraic specifications provide a means of describing data structures in an abstract property oriented way. Algebraic specifications specify software systems as a collection of abstract data types (ADT's). Superficially, algebraic specifications are similar to the style of module specifications illustrated in section 1.2.1, consisting of a signature and axioms. The signature of an algebraic specification takes the form of a collection of names of types called *sorts*, together with the names and types of the primitive operations on the sorts. The axioms specify the behaviour of the operations in terms of the relationship between the operations declared in the signature.

One of the first and most influential algebraic specification languages was CLEAR [5]. CLEAR is designed to allow algebraic specifications to be constructed in a structured manner. The CLEAR language has operators such as enrichment, amalgamated sum for combining specifications, and an operator called *derive* which is used to remove sorts and functions from a specification. CLEAR also allows specifications to be parameterised by other specifications. Most other algebraic specification languages such as OBJ [14], Larch [17], ASL [51, 55] and Extended ML [49, 48] contain the features of CLEAR. Larch is notable as it consists of a core algebraic specification language—called the Larch shared language—and a collection of "interface languages" each of which is targeted to programming in a specific programming language. Extended ML is a "wide spectrum" language where the programming language Standard ML is a subset of the specification language. ASL was designed as a kernel language for constructing structured specifications; it includes general purpose specification operators which can be combined to produce further specification operators. Much of the work in this thesis is influenced by the work of Sannella and Tarlecki [49, 48] on ASL and Extended ML.

Semantically, an algebraic specification denotes a class of models. The models are algebras whose carriers model the sorts of a specification, and whose operations model the operations specified in the signature of a specification. There are four main approaches to determining which algebras are models of a specification: initial semantics [15], terminal semantics [54], loose semantics [16], and ultra-loose semantics [56]. Roughly speaking, the initial, terminal and loose models of a specification are algebras whose carriers and operations satisfy the axioms, provided the carriers do not contain "junk" terms; junk terms are values that cannot be constructed using operations supplied by the signature of a specification. In the ultra-loose approach, models may contain junk terms. The type-theoretic semantics we advocate is similar to algebraic semantics in the sense that values in a type can be viewed as models of a specification. The type-theoretic semantics is closest to the ultra-loose semantics since there is no restriction on junk terms.

Programs can be developed from an algebraic specification by refining it towards a specification that is so low level that it can be regarded as a program; unlike type theory, the models of an algebraic specification are not regarded as implementations. There are several definitions of refinement for algebraic specifications [12, 18, 51]. Of particular interest to us is the notions of refinement for ASL developed by Sannella and Wirsing [51]: an ASL specification SP_1 is refined by SP_2 iff every loose model of SP_2 is a loose model of SP_1. Sannella and Tarlecki [50] have introduced a generalised definition of ASL refinement, called "constructor" implementation, which includes the notion of a refinement map similar to our abstraction function described in section 1.4. The definition of refinement introduced in Chapter 8 is a type-theoretic equivalent of constructor implementation.

There are some important differences between the algebraic and type theoretic approach to specifying modules. For example, when specifying ADT's in type theory we may use one of the built in types of MLT as a model for a sort; this allows axioms to be specified in terms of a particular model, rather than in an abstract property oriented style. Furthermore, since specifications are types, we can use specifications in the same way we use data-types such as $\mathbb{N}$, $\mathbb{B}$ etc. For example, we can specify a module to be a component within another module; and modules can be the arguments and results of functions. These and other advantages of using types to specify modules are discussed in Chapter 3.

1.7.3 Model-Based Specifications

Another important class of specification languages are the model-based specification languages such as Z [52], The Vienna Development Method (VDM) [21] and the refinement calculus [2, 35, 37]. Model-based methods use mathematical structures—such as sets and relations—to model data, and predicate logic to describe the operations on data. A model-based specification typically consists of the description of a state space followed by predicates describing operations which change the state. Although we do not consider state in our type-theoretic approach to specification, the model-based approach is worth mentioning as it raises general issues about modularity and implementation which are relevant to our work.

Z is notable for encouraging the incremental construction of structured specifications by using *schemas* which allow states and predicates to be grouped together and named. Z allows schemas to import the contents of other schemas, and includes operators to modify and combine schemas. However, there is no mechanism for grouping schemas, and this often leads to problems in structuring large specifications. To solve this problem, [47] proposes an extension to Z that allows modules containing schemas. The main structuring facility in VDM [21] is procedural and functional abstraction, but [13] and [32] propose extensions to VDM allowing modules containing states and operations, together with import and export facilities to combine modules. [39] describes the incremental specification and development of modules using the refinement calculus, but as yet, there is no formal definition of modules.

Both VDM and the refinement calculus have a refinement relation that allows specifications to be refined to programs. In each case, the programming language is an implementable subset of the specification language, and specifications are refined until they contain only programming language constructs. There are two notions of refinement; procedural refinement, which is concerned with replacing specification statements with programming language statements; and data refinement, which is concerned with replacing fancy data-types, such as sets, with programming language data-types such as arrays. The Z notation does not define a formal notion of program development, but [22] shows how Z specifications may be implemented using the refinement calculus. Both [13] and [39] discuss the refinement of modular specifications in VDM and the refinement calculus, respectively.

1.8 This Thesis

In this section, we give a brief summary of the contents of this thesis.

Chapter 2 gives the notations used in this thesis for the types and type rules in Martin-Löf's type theory. We also define some extensions to the theory, and illustrates the proof style used in this thesis.

Chapter 3 gives some example specifications and modules which illustrate important features of our specification language and its associated implementation language. Most of the features will be justified by the fact that specifications are types; but the semantics of specifications are left for later chapters. The features illustrated include the ability to nest specifications, and modules, inside each other; using modules as records; and parameterised specifications and modules.

We give a semantics to specifications and modules by translating them into terms in Martin-Löf's type theory, and Chapter 4 formally defines the terms that may be used as translations for specifications and modules. The definitions are preceded by a detailed review of two other approaches to specifying modules in type theory.

For presentational reasons, Chapter 4 does not give a formal mapping from the syntax of specifications and modules to their translations in Martin-Löf's type theory. That is given in Chapter 5.

Chapter 6 gives the formal definition of a collection of operators on signatures and computational elements, together with some of their properties; the operators are needed in later chapters to define specification and module operators.

Chapter 7 gives formal definitions for a collection of specification operators, and it gives some of the algebraic properties of the specification operators. Chapter 7 also gives an example that shows how the specification operators can be used to construct large specifications in an incremental style.

Chapter 8 gives a formal definition of refinement for specifications in type theory, and it also gives a collection of refinement laws. Particular attention is paid to laws that allow the piecewise refinement of large specifications. An example of piecewise refinement is also given.

Chapter 9 gives laws for implementing structured specifications. The laws make use of a collection of module operators, also defined in Chapter 9; the module operators are analogous to some of the specification operators. Chapter 9 also gives an example of the piecewise implementation of the example refinement given in Chapter 8.

In Chapter 10, we discuss some of the outstanding issues raised in previous chapters. Future work is discussed. And we give a summary of the key points of our thesis.

If the reader wishes to skip details of the formal proofs in this thesis—on a first reading, say—then it is also advisable to skip Chapter 5, as we only use the formal definition of the translation mapping it gives in some of our proofs; Chapter 4 gives an informal description of the translation mapping which should be sufficient for most of our purposes. The purpose of the signature and computational element operators, defined in Chapter 6, is to define specification and module operators. Therefore, Chapter 6 can be skipped until the reader considers the formal definitions of specification and module operators given in Chapters 7 and 9, respectively; although, we recommend at least reading the informal descriptions of the signature and computational element operators, given in Chapter 6, before reading Chapters 7 and 9.

Chapter 2

Types and Specifications

2.1 The Type Theory

Although MLT is reasonably well established in the literature, it is constantly being extended, and notational conventions are still fluid. Therefore, in this chapter we give the basic notations and type rules used in this thesis. We assume the reader is reasonably familiar with type theory and this chapter is only intended to give the particular syntactic conventions we use for what we hope are familiar types, terms and rules. We describe some extensions to the theory which may be unfamiliar, but the extensions are relatively straightforward. We also illustrate the proof style used. Most of the notation for types and type rules is borrowed from [44], but proofs are in the natural deduction style exemplified in [3]. It is not our intention to give a complete description of the type theory and its semantics; for this, we refer the interested reader to [31, 44].

2.1.1 A Note About Notation

We let the capital letters A, B, C, P, Q and R stand for type expressions, and the small letters a, b, c, e, p, q and r (possibly subscripted) stand for type and non-type expressions; both type and non-type expressions may be partial, i.e. they may contain free variables. Variables are x, y, z (possibly subscripted). We let SP (possibly subscripted) stand for types representing specifications, and m (possibly subscripted) stand for modules. The letters f, g, and h will normally stand for functions.

The notation $b(x\backslash e)$ stands for an expression b with each free occurrence of variable x replaced by expression e. If an expression b contains a free variable—x, say—then we will sometimes rewrite b as $[x]b$ to make this fact evident. The notation $[x]b(e)$ stands for b with each free occurrence of variable x replaced by expression e: $[x]b(e) \equiv b(x\backslash e)$

where "$\equiv$" is definitional equality. When it is clear from the context that b is defined in terms of a free variable x, we may abbreviate $[x]b(e)$ to just $b(e)$.

2.1.2 Judgement Notation

We make use of the four "judgemental forms" in Martin-löf's type theory:

$$\begin{array}{l} P\ \textbf{type} \\ p \in P \\ p = q \in P \\ P = Q \end{array}$$

The first is read as P is a well-formed type; the second that p has type P; the third, that p and q are equal elements of type P; and the fourth that P and Q are equal types.

Hypothetical judgements are judgements made under certain assumptions. For example, the judgement $[p =_P q]$ **type** is true under the assumption that $p \in P$ and $q \in P$, and we write such a judgement as

$$[\![p \in P; q \in P \rhd [p =_P q]\ \textbf{type}]\!]$$

The square brackets "$[\![$" and "$]\!]$" denote the scope of the context and the symbol "$\rhd$" separates the context from the conclusion that can be drawn from it; this notation is due to [3]. In general, a hypothetical judgement has the form:

$$[\![x_1 \in P_1; \ldots; x_n \in P_n \rhd J(x_1, \ldots x_n)]\!]$$

where $J(x_1, \ldots, x_n)$ $(n \geq 0)$ stands for a judgement in one of the four judgemental forms given above, and $x_1 \in P_1; \ldots; x_n \in P_n$ represents the context in which the judgement is made.

2.1.3 The Types

The types we use are summarised in Table 2.1. Some of the types, such as the propositional types, have been given previously in Table 1.1, but Table 2.1 also contains data-types, such as naturals and lists, which are used specifically for programming. The string, list, set, subset and singleton types were not in the original theory [31]; subset and list types are extensions given in [44, 8], strings are lists of characters, and sets are defined in [3].

The well-formedness of a type expression is given by formation rules. However, we do not make much use of the formation rules of the theory, and the only formation rule of any significance to us is that for the $\sum$-type:

$$\sum\text{-formation} \quad \frac{P\ \textbf{type} \qquad [\![x \in P \rhd Q(x)\ \textbf{type}]\!]}{\sum x \in P.Q(x)\ \textbf{type}}$$

Type	**Type Name**
$\mathbb{N}$	Naturals
$\mathbb{B}$	Booleans
$\mathbb{S}$	String type - contains lists of characters
T	Unit type - contains the single value tt
ϕ	Empty type - has no elements
$P+Q$	Disjoint sum
$\prod x \in P.Q(x)$	Dependent function space
$\sum x \in P.Q(x)$	Dependent product
$[p_1 =_P p_2]$	Equality type on P - inhabited iff $p_1 = p_2$
$List(P)$	List type - the type of lists of elements of type P
$Set(P)$	Set type - the type of sets of elements of type P
$\{x \in P \mid Q(x)\}$	Subset type - for elements of type P that satisfy $Q(x)$
$\{p\}_P$	Singleton type - contains the single value p of type P
U_i	Type Universes ($i > 0$)

Table 2.1: Summary of types

Type	**Introduction Rule**	**Elimination Rules**
Unit	$tt \in T$	$\dfrac{x \in T}{x = tt \in T}$
ϕ	none	$\dfrac{r \in \phi}{\phi\text{-}elim(r) \in C(r)}$
$\sum$	$\dfrac{p \in P \quad q \in Q(p)}{\langle p, q\rangle \in \sum x \in P.Q(x)}$	$\dfrac{a \in \sum x \in P.Q(x) \quad [\![x \in P, y \in Q(x) \rhd b(x,y) \in C(\langle x,y\rangle)]\!]}{\text{split}(a,b) \in C(a)}$
$\prod$	$\dfrac{[\![x \in P \rhd q(x) \in Q(x)\ ]\!]}{\lambda x.q(x) \in \prod x \in P.Q(x)}$	$\dfrac{f \in \prod(x \in P).Q(x) \quad a \in P}{f(a) \in Q(x)}$
$+$	$\dfrac{p \in P}{inl(p) \in P+Q}$ $\quad$ $\dfrac{q \in Q}{inr(q) \in P+Q}$	$\dfrac{a \in A+B \quad [\![x \in A \rhd c(x) \in C(inl(x))]\!] \quad [\![y \in B \rhd d(y) \in C(inr(y))]\!]}{\text{when}(a,c,d) \in C(a)}$
$=$	$\dfrac{p = q \in P}{eq \in [p =_P q]}$	$\dfrac{c \in [p =_P q]}{p = q \in P} \quad \dfrac{c \in [p =_P q]}{c = eq \in [p =_P q]}$

Table 2.2: Introduction and elimination rules for the propositional types

Type	Introduction Rules	Elimination Rules
$\mathbb{B}$	$true \in \mathbb{B}$ $\quad false \in \mathbb{B}$	$\dfrac{b \in \mathbb{B} \quad d \in C(true) \quad e \in C(false)}{\textbf{if } b \textbf{ then } d \textbf{ else } e \in C(b)}$
$\mathbb{N}$	$0 \in \mathbb{N}$ $\quad \dfrac{n \in \mathbb{N}}{succ(n) \in \mathbb{N}}$	$\dfrac{n \in \mathbb{N} \quad b \in C(0) \quad \|[x \in \mathbb{N}; h \in C(x) \rhd ind(x,h) \in C(succ(x))]\|}{natrec(n,b,ind) \in C(n)}$
List	$nil \in List(P)$ $\dfrac{p \in P \quad l \in List(P)}{p : l \in List(P)}$	$\dfrac{x \in List(A) \quad b \in C(nil) \quad \|[a \in A; l \in List(A); h \in C(l) \rhd ind(a,l,h) \in C(a:l)]\|}{\text{listelim}(x,b,ind) \in C(x)}$
$\{\|\}$	$\dfrac{p \in P \quad q \in Q(p)}{p \in \{x \in P \| Q(x)\}}$	$\dfrac{c \in \{x \in P \| Q(x)\} \quad \|[x \in P, y \in Q(x) \rhd d(x) \in C(x)]\|}{d(c) \in C(c)}$

Table 2.3: Introduction and elimination rules for the data types

Tables 2.2 and 2.3 give the introduction and elimination rules for propositional types and data-types, respectively (the separation of the rules for propositional types and data-types is purely to aid presentations and has no other significance). The rules for type universes have been omitted from Tables 2.2 and 2.3, and are described in Section 2.1.4; the rules for the string type, set types and singleton types are also omitted, and described in Section 2.2. The introduction rules show how to form canonical elements of a type. The elimination rules say how to reason about elements of a type (or equally, since meaning is constructive, how to construct functions over elements of a type). The elimination rules associate with a type a so-called non-canonical form. For example, the $\sum$-type has the non-canonical form *split*; $+$ has the non-canonical form *when* etc. The non-canonical forms of a type are used to construct functions over elements of the type. Table 2.4 gives the operational semantics of the various non-canonical forms.

The non-canonical form *split*, associated with the $\sum$-type, is used to define the usual projections on pairs:

Definition 2.1

$$\begin{aligned} fst(p) &\equiv \text{split}(p,[x,y]x) \\ snd(p) &\equiv \text{split}(p,[x,y]y) \end{aligned}$$

□

The empty type has no introduction rule as it has no elements. However, in the event that we deduce that $r \in \phi$ for some r—an absurdity—then there is an elimination rule which allows us to deduce that the non-canonical expression $\phi\text{-}elim(r)$ inhabits

Type	**Computation Rule**
Σ	$split(\langle p,q\rangle,[x,y]e) = e(x\backslash p, y\backslash q)$
Π	$(\lambda x.e)(p) = e(x\backslash p)$
$+$	$when(inl(p),[x]c,[y]d) = c(p)$ $when(inr(q),[x]c,[y]d) = d(q)$

$\mathbb{B}$	**if** $true$ **then** d **else** $e = d$
	if $false$ **then** d **else** $e = e$

$\mathbb{N}$	$natrec(0,b,[x,h]ind) = b$ $natrec(succ(n),b,[x,h]ind) = ind(n, natrec(n,b,[x,h]ind))$
List	$listelim(nil,b,[x,y,h]ind) = b$ $listelim(a:l,b,[x,y,h]ind) = ind(a,l,listelim(l,b,[x,y,h]ind))$

Table 2.4: Operational semantics for non-canonical forms

all types. The equality type has two elimination rules. The first equality elimination rule states that we are able to construct an object of an equality type whenever we can make the corresponding equality judgement. The second rule rule allows us to deduce that all elements in an equality type are equal to *eq*.

As usual, we shall abbreviate the use of the constructor *succ* such that $succ(0) \equiv 1$, $succ(succ(0)) \equiv 2$ etc. We use *nil* for the empty list and $a : l$ denotes the list formed by appending the element a onto list l. We often abbreviate lists such as $1 : 2 : 3 : nil$ to $[1,2,3]$. The non-canonical forms *natrec* and *listelim* define primitive recursion on naturals and lists, respectively. As the use of *natrec* and *listelim* can often be difficult to parse, we usually define recursive functions on naturals and lists in a clausal form. For example, we would define a recursive list function $f \in List(P) \rightarrow Q$ in a clausal form by:

$$\begin{array}{rcl} f(nil) & = & e_1 \\ f(a:l) & = & e_2(a,l,h) \end{array}$$

where

$$a \in P,\ l \in List(P),\ h = f(l),\ e_1 \in Q,\ e_2(a,l,h) \in Q.$$

Such a definition of f may be written using *listelim* by:

$$f(x) = listelim(x, e_1, e_2(a,l,h))$$

2.1.4 Type Universes

The canonical elements of the type universe U_1 are all the types given in Table 2.1 save the type universes U_i $(i \geq 1)$. We do not give the introduction rules for U_1 as they are too numerous and we shall not need them—the rules are listed in [44]. Each type universe U_i, for $i > 1$, contains all the elements of U_{i-1}, and also includes U_{i-1} itself:

$$\frac{x \in U_i}{x \in U_{i+1}} \qquad U_i \in U_{i+1} \quad (i > 0)$$

There is no widely accepted non-canonical form for type universes, but Nordström et al have introduced *urec* as the non-canonical form for the type U_1. *urec* allows the definition of recursive functions over the structure of types in U_1—in principle, we could introduce a non-canonical form $urec_i$ for each type universe U_i. However, rather than using *urec*, we will write functions on types in a clausal form; the reason for this is that the syntax of the *urec* operator is difficult to use and parse. The clausal forms can be seen as meta-notation which we can always translate into *urec* expressions. For example, we could define a function $F \in U_1 \to U_1$ that converts a $\sum$-type to a $\prod$-type as follows:

$$\begin{array}{lcl} F(\sum x \in P.Q(x)) & = & \prod x \in P.Q(x) \\ F(P) & = & P \quad , \quad P \text{ non-}\sum \end{array}$$

The clause $F(P)$ is included to make F a total function on types in U_1; all functions in the type theory must be total, and so we choose to define F as an identity operation when F is applied to a non $\sum$-type. Many of the functions we define on types are intended to be on $\sum$-types only, so that many of our functions on types will take a similar form to F above.

2.2 Extensions to Type Theory

A feature of Martin-Löf's type theory is that it is amenable to extension by the addition of new types. Backhouse et al [3] describe a systematic approach to adding new types. In this section, we use the approach described by Backhouse et al to define some new types that we will require.

2.2.1 Characters and Strings

The Character type is denoted *Char*. The intended elements are alphabetical characters and mathematical symbols; *Char* is analogous to the *char* type in Pascal. We write characters inside single quotes, and members of *Char* include 'a',…,'z'; 'A',…,'Z'; '0',…,'9'; '+', '⊕'. We could add more characters to *Char*, but those we have listed are sufficient for our purposes. We assume the existence of an equality function, $\_ = \_ \in Char \to Char \to \mathbb{B}$, on characters. The type rules for *Char* are omitted, as we do not make use of them. For the interested reader, *Char* is defined as an enumeration, or finite, type as described in [44] and [31], and its non-canonical form is a Pascal style *Case* statement over its elements; we use the *Case* statement to define the equality function on *Char*, mentioned above.

The character type is used to define the string type, denoted $\mathbb{S}$, whose elements are lists of characters:

$$\mathbb{S} \equiv List(Char)$$

Instead of writing strings as lists, we use the convention of writing strings within single quotes; for example, 'hello' $\equiv$ ['h', 'e', 'l', 'l', 'o']. The empty string is written '', and '' $\equiv nil$.

2.2.2 Set Types

The set type is an extension to the theory introduced by Backhouse et al [3]. We do not give a complete explanation of set types as they are only used in examples where the usual intuitive understanding of sets is sufficient. The type of a set containing elements of type P is written as $Set(P)$. The introduction rules for sets introduce an empty set $\{\}$, and $p :: s$ denotes the set $s \in Set(P)$ with $p \in P$ added to s:

$$\{\} \in Set(P) \qquad \frac{p \in P \quad s \in Set(P)}{p :: l \in Set(P)}$$

The non-canonical from for sets is called *setelim*, and its operational semantics are similar to *listelim*.

$$\begin{array}{lcl} setelim(\{\}, e, [p, s, h]ind) & = & e \in C(\{\}) \\ setelim(p :: s, e, [p, s, h]ind) & = & ind(p, s, setelim(s, e, [p, s, h]ind)) \in C(p :: s) \end{array}$$

where

$$p \in P,\ s \in Set(P),\ [\![x \in Set(P) \rhd C(x)\ \mathbf{type}]\!],\ h \in C(s)$$

In fact, sets are similar to lists, except that *setelim* has restrictions to ensure that sets are independent of the number of occurrences of any element appearing in their construction; and sets are independent of the order in which elements appear in their construction. The restrictions on *setelim* are given by the following rules which are additional computational rules for sets:

$$\begin{array}{lcl} setelim(p :: p :: \{\}, e, [p, s, h]ind) & = & setelim(p :: \{\}, e, [p, s, h]ind) \\ setelim(p :: q :: s, e, [p, s, h]ind) & = & setelim(q :: p :: s, e, [p, s, h]ind) \end{array}$$

2.2.3 Singleton Types

We extend the type theory with a singleton type constructor ($\{\}$) that enables the definition of types containing single elements. For example, $\{true\}_{\mathbb{B}}$ is the type containing the single element *true* from the type $\mathbb{B}$; and $\{[1,2,3]\}_{List(\mathbb{N})}$ is the type containing the single list $[1,2,3]$. The element of a singleton type must be a member of an existing type; the tag on a singleton type allows us to deduce the base type of an element of a singleton type. The formation, introduction and elimination rules for the singleton type are given below.

$$\frac{P\ \textbf{type} \quad p \in P}{\{p\}_P\ \textbf{type}} \qquad \frac{P\ \textbf{type} \quad p \in P}{p \in \{p\}_P} \qquad \frac{x \in \{p\}_P}{x = p \in P}$$

Remark The type $\{p\}_P$ is not equal to the subset type $\{x \in P|[x =_P p]\}$. The elimination rule for the subset type does not allow us to deduce that $e = p \in P$ from $e \in \{x \in P|[x =_P p]\}$. We can approximate a singleton type $\{p\}_P$, using the type $\sum x \in P.[x =_P p]$. The sole element of $\sum x \in P.[x =_P p]$ is the pair $\langle p, eq\rangle$, so it is not equivalent to $\{p\}_P$, but is a reasonable approximation if we ignore the extra component *eq*. The addition of the singleton type can therefore be regarded as just a convenience that avoids the need to deal with pairs; from our approximation of singleton types, we can deduce inference rules similar to the three given above. □

2.3 Proof Style

The inference rules of the type theory admit of a natural deduction style for making proofs of judgements. Natural deduction style proofs can be written in a very systematic manner, and there are several recognised styles for writing them. In this thesis we adopt the proof style given by Backhouse et al in [3], which we illustrate by giving the proof of the proposition:

$$(P \to Q) \wedge (Q \to R) \to (P \to R)$$

This proposition corresponds to showing that $\to$ is transitive. We intend to prove the proposition true constructively by showing the following:

$$\lambda p.\lambda x.(snd(p))((fst(p))(x)) \in (P \to Q) \wedge (Q \to R) \to (P \to R)$$

The derivation is given in Figure 2.1. The line numbers, and comments in quotes are not part of the derivation but are intended to aid the reading of the proof. Each comment gives the assumptions and inference rules used to make the judgement that follows the comment. Proofs make extensive use of hypothetical judgements which are delimited by square brackets: assumptions appear before the "$\triangleright$" symbols, and consequences drawn from the assumptions appear to the right of the "$\triangleright$" symbol.

```
0.0    |[  $p \in (P \to Q) \wedge (Q \to R)$
       ▷     "0.0, ×-elimination"
0.1        $fst(p) \in P \to Q$
             "0.0, ×-elimination"
0.2        $snd(p) \in Q \to R$
0.3.0      |[   $x \in P$
           ▷      "0.3.0, 0.1, →-elimination"
0.3.1           $(fst(p))(x) \in Q$
                  "0.3.1, 0.2, →-elimination"
0.3.2           $(snd(p))((fst(p))(x)) \in R$
           ]|
             "0.3.0, 0.3.2, λ-introduction"
0.4        $\lambda x.(snd(p))((fst(p))(x)) \in (P \to R)$
       ]|
         "0.0, 0.4 , λ-introduction"
1      $\lambda p.\lambda x.(snd(p))((fst(p))(x)) \in (P \to Q) \wedge (Q \to R) \to (P \to R)$
```

Figure 2.1: Derivation of $\to$ transitivity

We now give a brief explanation of the derivation given by Figure 2.1. The proposition we are trying to prove is an implication where the antecedent is $(P \to Q) \wedge (Q \to R)$ and the consequent is $P \to R$. Step 0.0 begins by assuming that p is a proof (member) of the antecedent; using this assumption it is our aim to prove the consequent. Looking ahead to steps 0.4 and 1, having constructed an element of $P \to R$, λ-introduction can be used to complete the derivation. The assumption p in step 0.0 is a pair of type $(P \to Q) \wedge (Q \to R)$. From that assumption we may conclude (steps 0.1 and 0.2) that $fst(p)$ and $snd(p)$ are elements of type $P \to Q$ and $Q \to R$, respectively—we use $\times$-elimination to mean the application of either projection function. Now, in steps 0.3.0, 0.3.1 and 0.3.2 we construct an element of type $P \to R$. First (step 0.3.0) we assume that x is an element of P. Using that assumption we may conclude (step 0.3.1) that applying $fst(p)$ to x ($\to$-elimination) yields a value of type Q, and hence, in step 0.3.2, applying $snd(p)$ to $(fst(p))(x)$ gives a value of type R. Step 0.4 now follows by λ-elimination—note the discharge of assumption 0.3.0. Finally, we obtain the desired conclusion (step 1) by discharging the initial assumption 0.0.

Chapter 3

Typeful Specifications

3.1 Introduction

In this chapter, we outline some features of our specifications and modules. Adopting a type-theoretic view of modules and their specifications allows us to use many of the features of type theory when making specifications, and we illustrate the use of some of those features. We proceed mainly by example, without trying to describe the mathematical underpinnings in any detail. Of course, a formal semantics is important for the purposes of formal reasoning about specifications and modules, but the issues raised here can be understood without a detailed knowledge of the semantics of specifications; we leave the semantics until the next chapter.

We give an example of the use of local components to make specifications and modules. The other features illustrated in this chapter arise as a direct result of interpreting specifications as types. We show that we can use the built-in types and operations of Martin-Löf's type theory to help make specifications and their implementations; we claim such specifications are model-based since they use the built-in types and operations. We also give some examples of hierarchical specifications; these are specifications that have other specifications nested within them. We also show how specifications can be used as data-types, and how modules can be used as first class values within specifications and modules.

In the following, when we give examples of a module satisfying a specification we will omit to prove that the module satisfies the specification; proofs are omitted until we discuss the formal semantics of specifications, in the next chapter. Furthermore, we will omit witnesses from modules, replacing them by "...".

3.2 Local components

When we make a specification, we often specify auxiliary components whose sole purpose is to help specify other components. These auxiliary components are not intended to be available for use by other "client" specifications; they are *local* components intended for use within a specification. Local components are useful. Firstly, we can use them to break up the specifications of complex operations into more manageable pieces. Secondly, we can "hide" components by making them local, and this allows us to modify and reuse old specifications to make new ones; we can hide components not needed in the new specifications. Thirdly, by specifying a component as local, we are saying that it is not intended to be part of the "interface" to the specification. This last point is important, as it tells an implementor which components must appear in an implementation, and which components they are free to change or remove. Our specification language, and its associated implementation language, contain features that allow a specifier to indicate which components are local. Components intended to be part of the interface of a specification are called *visible* components.

3.2.1 An example of local components—The Mean Module

We distinguish local components in a specification by putting a fat dot (•) in front of their declaration in the signature; components without a fat dot are visible. Consider the specification *Mean*, given in Figure 3.1. *Mean* specifies operations for calculating the mean of a collection of naturals. *Data* is intended to be a type for samples of data where the samples consist of a collection of naturals; so *Data* might be implemented by $Bag(\mathbb{N})$, $List(\mathbb{N})$ etc. *clear* is an empty sample. The function *sum* returns the sum of a sample of data; *sum* is a local component whose sole purpose is to help define the component *mean*. The function *size* gives the number of values in a sample; *enter* is used to add a new value to a sample; and *mean* takes the mean of a sample. *mean* is a dependent function: for a non-empty sample the result of *mean* is a natural, but if we try and take the mean of an empty sample then *mean* returns the string "error". We note in passing that *mean* is an example of an operation whose type is dependent on a non-type component, namely, the function *size*. It is unfortunate

$Mean \equiv$
Elements
$Data \in U_1$,
$clear \in Data$,
$\bullet sum \in Data \to \mathbb{N}$,
$size \in Data \to \mathbb{N}$,
$enter \in \mathbb{N} \to Data \to Data$,
$mean \in \prod d \in Data.(\textbf{if } size(d) \neq 0 \textbf{ then } \mathbb{N} \textbf{ else } \mathbb{S})$
Restrictions
$\forall d \in Data. \forall n \in \mathbb{N}.$
$sum(clear) =_{\mathbb{N}} 0 \quad \wedge$
$sum(enter(n, d)) =_{\mathbb{N}} sum(d) + n \quad \wedge$
$size(clear) =_{\mathbb{N}} 0 \quad \wedge$
$size(enter(n, d)) =_{\mathbb{N}} size(d) + 1 \quad \wedge$
$mean(d) = (\textbf{if } size(d) \neq 0 \textbf{ then } sum(d) \text{ div } size(d) \textbf{ else } \text{“error”})$
End

Figure 3.1: The *Mean* specification

that the expression $size(d) \neq 0$ is repeated in both the type of *Mean* and in the axiom that specifies *mean*. We can avoid such repetition by redefining the type of *mean* as follows:

$$mean \in \prod d \in Data.\textbf{if } size(d) \neq 0 \textbf{ then } \{sum(d) \text{ div } size(d)\}_{\mathbb{N}} \textbf{ else } \{\text{“error”}\}_{\mathbb{S}}$$

The new type for *mean* completely specifies the behaviour of *mean* since the only value in the singleton type $\{sum(d) \text{ div } size(d)\}_{\mathbb{N}}$ is $sum(d)$ div $size(d)$; and the only value in type $\{\text{“error”}\}_{\mathbb{S}}$ is “error”. Consequently, there is no need for the final axiom in the restrictions of *Mean*.

When we implement a specification we are obliged to implement both its visible and local components; we will justify this obligation in Section 4.11. The implementations of local components are distinguished by a fat dot before their name. Figure 3.2 gives an implementation of *Mean*, called *MeanModule*, containing implementations for all the components specified by *Mean*, including the local component *sum*. *MeanModule* represents data samples as pairs of naturals: the first component of such a pair is the sum of all the elements in the sample, the second component is the size of the sample.

The main difference between the visible and local components in a module is that clients cannot use dot notation to refer to local components; this shields clients from future changes to local components. An attempt to refer to a local component by dot notation just returns the unit value $tt \in T$; for example, $MeanModule.sum = tt$. Visible components may be referred to by dot notation in the usual way.

$$
\begin{array}{l}
\mathit{MeanModule} \equiv \\
\quad \mathbf{module} \\
\quad\quad \begin{array}{lcl}
\mathit{Data} & = & \mathbb{N} \times \mathbb{N}, \\
\mathit{clear} & = & \langle 0, 0 \rangle, \\
\bullet \mathit{sum} & = & \lambda d \in \mathit{Data}.\mathit{fst}(d), \\
\mathit{size} & = & \lambda d \in \mathit{Data}.\mathit{snd}(d), \\
\mathit{enter} & = & \lambda n \in \mathbb{N}.\lambda d \in \mathit{Data}.\langle \mathit{sum}(d) + n, \mathit{size}(d) + 1 \rangle, \\
\mathit{mean} & = & \lambda d \in \mathit{Data}.\mathbf{if}\ \mathit{size}(d) \neq 0\ \mathbf{then}\ \mathit{sum}(d)\ \mathrm{div}\ \mathit{size}(d)\ \mathbf{else}\ \text{“error”}
\end{array} \\
\quad \mathbf{proof} \\
\quad\quad \ldots \\
\quad \mathbf{end} \in \mathit{Mean}
\end{array}
$$

Figure 3.2: An implementation of *Mean*

3.3 Model-Oriented Specifications

The specifications given in the previous sections specify abstract data types (ADT's); that is, they specify modules containing type components—that are equivalent to sorts in algebraic languages—together with operations on the types. Such specifications are similar to algebraic specifications. However, our specification language is not an algebraic specification language—as we shall see later, its semantics is given in terms of type theory—so that we are not restricted to purely algebraic style specifications.

One difference between our language and algebraic specification languages is that we provide many built-in types such as $\mathbb{N}$, $\mathbb{B}$, *List*, *Set* etc; algebraic specification languages do not have built-in types. The built-in types can be used as "models" to help specify modules. In particular, we can make specifications by using the predefined operations that built-in types come equipped with. In this section, we give two examples of a model-oriented approach to specifying modules.

3.3.1 A model-oriented axiomatic specification

The specifications we have seen so far have been given in a purely property-based style similar to algebraic specifications. However, it is often unclear in a property-based specification whether its restrictions actually specify the properties we desire. The lack of clarity stems from the fact that property-based specifications are overly abstract; they avoid being biased towards any one choice of implementation for data-types. In our opinion, specifying an actual value, or model, for data-types often leads to specifications that are easier to make and understand. Our specification language allows us put constraints on type components declared in the signature of specifications, and so specify a model for type components.

Consider the specification *Catalogue2* given in Figure 3.3. *Catalogue2* is intended to be a model-oriented version of the book catalogue specification, *Catalogue*, which

$Catalogue2 \equiv$
Elements
$Book \in U_1$,
$Stock \in \{Set(Book)\}_{U_1}$,
$empty \in Stock$,
$add \in Book \rightarrow Stock \rightarrow Stock$,
$remove \in Book \rightarrow Stock \rightarrow Stock$,
$instock \in Book \rightarrow Stock \rightarrow \mathbb{B}$,
$isempty \in Stock \rightarrow \mathbb{B}$
Restrictions
$\forall s \in Stock.\forall b \in Book.$
$add(b,s) =_{Stock} \{b\} \cup s \quad \wedge$
$remove(b,s) =_{Stock} s - \{b\} \quad \wedge$
$empty =_{Stock} \{\} \quad \wedge$
$isempty(s) =_{\mathbb{B}} (s = \{\}) \quad \wedge$
$instock(b,s) =_{\mathbb{B}} (b \sqsubseteq s)$
End

Figure 3.3: A model-oriented catalogue specification

$$\{b\} \equiv b :: \{\}$$

$$s \cup t \equiv \mathrm{setelim}(s, t, [x,y,h]\ x :: h)$$

$$b \sqsubseteq s \equiv \mathrm{setelim}(s, \mathrm{false}, [x,y,h]\ \textbf{if}\ (x = b)\ \textbf{then}\ \mathit{true}\ \textbf{else}\ h)$$

$$s - t \equiv \mathrm{setelim}(s, \{\}, [x,y,h]\ \textbf{if}\ (x \in t)\ \textbf{then}\ h\ \textbf{else}\ x :: h)$$

Figure 3.4: Some standard set operations

was defined in a property-based style in Figure 1.2. *Catalogue2* constrains the type component *Stock* to be the type *Set*(*Book*). This is done by specifying *Stock* to be of type $\{Set(Book)\}_{U_1}$ which is a *kind* containing the single type *Set*(*Book*). Hence, we may deduce that $Stock = Set(Book)$, and we exploit this to make the restrictions. In contrast, *Catalogue* declares *Stock* to have the type U_1, which tells us that *Stock* is a type, but tells us nothing more about *Stock* values. Restricting *Stock* to the type *Set*(*Book*) allows us to use set operations on *Stock* values. Figure 3.4 gives the definitions of some set operations such as set union ($\cup$), set membership ($\sqsubseteq$) and set extraction ($-$). The set operations behave as we would expect, and they are used to specify the restrictions on *Catalogue2*.

In our view, the restrictions on *Catalogue2* are easier to understand than those on *Catalogue*. Of course, such a comparison is subjective—in part, our view is due to a familiarity with sets and their properties.

Remark We have observed that when we make a model-oriented specification it often has fewer axioms than a property-based specification of the same module; for

$$
\begin{array}{l}
\mathit{CatalogueModule2} \equiv \\
\quad \textbf{module} \\
\qquad \begin{array}{lcl}
\mathit{Book} & = & \mathbb{N}, \\
\mathit{Stock} & = & \mathit{Set}(\mathit{Book}), \\
\mathit{empty} & = & \{\}, \\
\mathit{add} & = & \lambda b \in \mathit{Book}.\lambda s \in \mathit{Stock}.\{b\} \cup s, \\
\mathit{remove} & = & \lambda b \in \mathit{Book}.\lambda s \in \mathit{Stock}.s - \{b\}, \\
\mathit{instock} & = & \lambda b \in \mathit{Book}.\lambda s \in \mathit{Stock}.(b \sqsubseteq s), \\
\mathit{isempty} & = & \lambda s \in \mathit{Stock}.(s = \{\})
\end{array} \\
\quad \textbf{proof} \\
\quad \ldots \\
\quad \textbf{end} \in \mathit{Catalogue}_2
\end{array}
$$

Figure 3.5: An implementation of *Catalogue2*

example, $\mathit{Catalogue}_2$ has 6 axioms, in comparison to 9 in *Catalogue*. We have also observed that as we add new components to a specification, the number of axioms appears to grows quadratically if it is property-based and linearly if it is model-oriented. We stress that these observations are based on our own experience of making specifications and are not justified by any theoretical, or experimental evidence. □

The usual argument against choosing a model for data-types is that the specification is then biased in favour of implementations that use this model, resulting in fewer choices of implementations for the specification. For example, Figure 3.5 gives the obvious implementation that *Catalogue2* is biased towards. However, although *Catalogue2* specifies *Stock* to be *Set*(*Book*) it is possible to develop implementations of *Catalogue2* which do not implement *Stock* as *Set*(*Book*). Using the technique of data refinement we can change the data-types used by a specification; this allows us to develop modules that use models other than the one given in their specification. For example, we could change the representation of *Stock* from a set to a list, so that in fact *CatalogueModule*, defined earlier in Figure 1.3, is also an implementation of *Catalogue2*. We discuss data refinement in Chapter 8. So apparent bias should not be seen as a problem.

3.3.2 A Non-ADT

Not all specifications we define are ADT's. We often make specifications that specify no type components at all. Such specification usually consist of a collection of operations that are defined in terms of the built-in types of Martin-Löf's type theory. If a specification only specifies a small number of components then it is often easier to make the specification in terms of the built-in types. Some modules naturally lend themselves to specification in terms of the built-in data types; for example, modules containing mathematical operations; in such cases we should feel free to use the built-in types. For example, the specification *Maths*, given in Figure 3.6, specifies

$Maths \equiv$
Elements
$root \in \mathbb{N} \to \mathbb{N}$
$\_div\_ \in \mathbb{N} \to \mathbb{N} \to \mathbb{N},$
$\_mod\_ \in \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
Restrictions
$\forall x \in \mathbb{N}.\forall y \in \mathbb{N}.$
$(root(x))^2 \leq x \quad \wedge$
$x < (root(x)+1)^2 \quad \wedge$
$y > 0 \;\Rightarrow\; x =_{\mathbb{N}} y * (x\ div\ y) + (x\ mod\ y) \quad \wedge$
$y > 0 \;\Rightarrow\; (x\ mod\ y) < y$
End

Figure 3.6: The specification of a maths module

an integer square root function, as well as the functions *div* and *mod* which give the quotient and remainder of integer division, respectively. Both *div* and *mod* are infix operators and we denote this by putting underscores around them (e.g. $\_div\_$). *Maths* contains no type components, only operations defined in terms of the type $\mathbb{N}$.

We do not wish to give the impression that a model-oriented style is always to be preferred over a property-based style. Some problems are more easily expressed in a property-based style, whilst others lend themselves to a model-oriented style; the specifier should be free to choose which style is best for a particular problem. It is interesting to note that although we have called *Maths* a model-oriented specification, its restriction is in a property-based style—the restrictions are not in the style "$root = \ldots$" as used in *Catalogue2*. In fact, in the case of *Maths*, defining an axiom of the form "$root = \ldots$" would require introducing a level of algorithmic detail not appropriate to a specification. Clearly, there are sometimes advantages in combining the model and property-based approaches, using a type-theoretic approach to specification allows us to do this.

3.4 Nested Specifications and Modules

One of the features of our specification language is that we can nest specifications inside other specifications, and modules inside other modules. We nest specifications by using specifications as the types of components within other specifications. We can do this because specifications *are* types. The implementations of nested specifications are modules which have other modules as some of their components. Nesting plays a useful role in structuring specifications and modules. In particular, nesting allows us to make new specifications and modules by using components specified in other specifications and modules.

```
$Means \equiv$
  **Elements**
    $mm \in Mean,$
    $size \in mm.Data \rightarrow mm.Data \rightarrow \mathbb{N},$
    $mean \in mm.Data \rightarrow mm.Data \rightarrow \mathbb{N}$
  **Restrictions**
    $\forall d, e \in mm.Data.$
    $size(d)(e) =_{\mathbb{N}} mm.size(d) + mm.size(e) \quad \wedge$
    $\neg([mm.size(d) =_{\mathbb{N}} 0] \vee [mm.size(e) =_{\mathbb{N}} 0]) \Rightarrow$
      $mean(d)(e) =_{\mathbb{N}} (mm.mean(d) + mm.mean(e))\ div\ 2$
  **End**
```

Figure 3.7: The *Means* specification

3.4.1 A Nested Specification

The specification *Mean*, which we defined earlier in Figure 3.1, specified operations to find the size and mean of a single sample of natural numbers. Suppose we wish to collect two samples—which might be generated by running an experiment twice, say—and calculate the combined mean and size of the two samples. We can use *Mean* to make a new specification that specifies operations to calculate the size and mean of two data samples. Such a specification, called *Means*, is given in Figure 3.7. *Means* contains an example of specification nesting: we use *Mean* as a type inside *Means* to specify component *mm* to be a module that satisfies *Mean*. The operation *mean* takes two data samples and returns the combined mean of both samples. We deliberately say nothing about the result of *mean* if either of the data samples is empty; we leave such a decision for an implementor. We use the components in *mm* to specify both the type and behaviour of *mean*; for example, the domain type of *mean* is given using the type *mm.Data*. The function *size* returns the combined size of two data samples.

Scope and Visibility of Names

Components specified at different levels of nesting in specifications may have the same names. For example, the name *size* occurs at two different levels in *Means*: it occurs in the domain of *Means*, and within the constituent module *mm*. Similarly, *mean* occurs in the domains of *Means* and *mm*. To avoid confusion, we need scope rules that define which declaration of a name is in scope at any particular point. The scope rules for nested specifications will be defined to be similar to the scope rules of a block-structured programming language, like Pascal. Uses of component names are bound to their inner-most (most recent) definition. For example, uses of the name *size* in *Mean*, the type of *mm*, are bound to the declaration of *size* in the signature of *Mean*, rather than the outer declaration of *size* in the signature of *Means*.

Although many specification languages, such as Z and ASL, allow the nesting of specifications, most simply import all the components of one specification into another without introducing any form of hierarchy. In other words, all components within a specification, regardless of nesting, are always in scope. This means that all components in a Z, and ASL, specification occupy the same name-space, so that name-clashes become a problem as specifications increases in size. However, the style of nesting we have described is hierarchical as it introduces different scopes, and so, components with the same name can co-exist.

3.4.2 A Nested Module

Modules may also contain nested modules; by nested module we mean modules which are components within other modules. As an example, consider the module *MeansModule* defined in Figure 3.8. *MeansModule* is an implementation for the specification *Means* with explanation following. The component *mm* is a module that satisfies *Mean*; recall that the type of *mm* is specified to be *Mean*. We can implement *mm* directly using *MeanModule* which we gave earlier—in Figure 3.2—as an example of an implementation for *Mean*. We use the components *Data*, *size* and *mean* from *mm* to implement the functions *size* and *mean*.

```
MeansModule ≡
  module
    mm   = MeanModule,
    size = λd, e ∈ mm.Data. mm.size(d) + mm.size(e),
    mean = λd, e ∈ mm.Data.
             if (mm.size(d) = 0) or (mm.size(e) = 0) then
               0
             else
               (mm.mean(d) + mm.mean(e)) div 2
  proof
    ...
  end ∈ Means
```

Figure 3.8: A module satisfying *Means*

The scope rules for nested modules are similar to those for nested specifications. For example, the function *size* used in the definition of function *mean* in *mm*, refers to *size* defined in *mm*, rather than the *size* defined in *MeansModule*. To refer to the components in a nested module we must use dot notation twice. For example, *MeansModule.mm.enter* is the function *enter* from module *mm* in *MeansModule*.

As an interesting aside, notice that by reusing *Mean* to make *Means*, we are also able to reuse the module *MeanModule* to make *MeansModule*. It is often the case that if we reuse a specification to make a new specification, then we can also reuse its implementations to implement the new specification.

3.5 Specifications as Record Types

Since specifications are types, we can use them just as we would use other types. In this section, we give an example of using specifications as record types. We consider taking our library catalogue example a little further and specify a module of operations on books; for example, the module may contain operations to view the author or title of a book. We model books as records containing a book number, together with author and title fields. This is done by using the specification *BookRec*, given in Figure 3.9, as a type for books. *BookRec* represents the *isbn* number of a book as a natural number, and the *author* and *title* by strings. The intended use of *BookRec* does not require putting any restrictions on its components, so that its restriction is specified to be the unit type T. Of course, we could specify that the *author* and *title* should be limited to a fixed number of characters; for example, we could limit them to 20 characters using the following restriction: $\sharp author =_{\mathbb{N}} 20 \wedge \sharp title =_{\mathbb{N}} 20$.

$BookRec \equiv$
  Elements
    $isbn \in \mathbb{N}$,
    $author \in \mathbb{S}$,
    $title \in \mathbb{S}$
  Restrictions
    T
  End

Figure 3.9: A type for book values

We use *BookRec* to make the specification *Books* which specifies a book module; *Books* is given in Figure 3.10, with explanation following. As stated earlier, we specify the type *Book* to be *BookRec*; note the use of the singleton type constructor to specify that *Book* = *BookRec*. The function *mkBook* takes a book number, author and title, and returns a *Book* value. *mkBook* is an example of a function that returns a module as its result; the module is used like a record. The function *view* takes a module denoting a book value and extracts its title; *view* is an example of a function whose domain is a specification.

The operations specified in *Books* could have been added as extra operations to *Catalogue* or *Catalogue2* in order to make a library specification. However, a more modular solution is to develop a separate specification for books—as we have done above—and then compose this specification with *Catalogue* or *Catalogue2* in some way. To allow

$Books \equiv$
Elements
$Book \in \{\text{BookRec}\}_{U_1}$,
$mkBook \in \mathbb{N} \rightarrow \mathbb{S} \rightarrow \mathbb{S} \rightarrow Book$,
$view \in Book \rightarrow \mathbb{S}$
Restrictions
$\forall n \in N. \forall a \in \mathbb{S}. \forall t \in \mathbb{S}.$
$mkBook(n)(a)(t) =_{Book}$ **module**
$isbn = \text{n},\ author = \text{a},\ title = \text{t}$
proof
tt
end $\wedge$
$view =_{Book \rightarrow \mathbb{S}} \lambda b \in Book.(b.title)$
End

Figure 3.10: The specification of a book module

such modular development of specifications requires specification building operations. We discuss such operations later, in Chapter 7.

However, even with our, at present, limited repertoire of structuring features, we can still package *Catalogue* and *Books* into a single specification by using nesting. Figure 3.11 gives the specification of a module that contains a book module and a catalogue module. Recall that both *Books* and *Catalogue* contain a type named *Book*. The restriction on *Catalogue3* ensures that the *Book* components in *bookmodule* and *catalogue* are both implemented using the same type; this ensures that *Book* values used in *bookmodule* are compatible with those used in *catalogue*. Without the restriction on *Catalogue3* we would be free to implement the *Book* component in *bookmodule* independently from the *Book* component in *catalogue*. This would be undesirable, since *Book* values generated by operations in *bookmodule* might no longer be compatible with *Book* values used in *catalogue*. The restriction on *Catalogue3* can be seen as a 'gluing' invariant between *Catalogue* and *Books*.

$Catalogue3 \equiv$
Elements
$bookmodule \in Books$,
$catalogue \in Catalogue$
Restrictions
$bookmodule.Book =_{U_1} catalogue.Book$
End

Figure 3.11: The specification of a book catalogue

3.6 Parameterisation

One of the advantages of using type theory as a framework for specification is that types, and specifications, can be treated as first class values. This allows us to define functions returning specifications; such functions are parameterised specifications. Since types are first class values, we can parameterise specifications by types to produce generic specifications that may be instantiated in different ways. By using specifications as types, we may also parameterise specifications with respect to modules, and this provides another useful means of structuring specifications.

The formal parameters of a parameterised specification may be any typed value allowed in the type theory; this includes values such as integers and boolean, as well as data types, modules and specifications themselves. The body of a parameterised specification is simply a specification defined in terms of some formal parameter names. In this section we only consider parameterised specifications whose formal parameters are data types and values; a discussion of specifications as formal parameters is left until Chapter 7 since the meaningful use of specifications as parameters requires the use of specification operators.

3.6.1 Generic Specifications

The specification *Sequence*, given in Figure 3.12, is a parameterised specification, and is a generic specification for sequences. The formal parameter $elem \in U_1$ indicates that *Sequence* takes a type as argument: *elem* is the type of the elements in the sequences that *Sequence* specifies. *Sequence* defines some standard sequence operations: *empty* is the empty sequence; the infix operator $\langle \_ \rangle$ makes a unit sequence; the binary infix operator $\_ \bullet \_$ concatenates two sequences; *head* returns the first elements in a sequence; and *tail* yields all but the first element in a sequence. Instantiated instances of *Sequence* are specification obtained by substituting the formal parameter *elem* by an actual parameter, in the body of *Sequence*. For example, *Sequence*($\mathbb{N}$) specifies sequences of naturals; *Sequence*($\mathbb{B}$) specifies sequences of boolean etc.

Once instantiated, *Sequence* can be implemented like any other specification. As an example, we consider an implementation of *Sequence*($\mathbb{N}$). List types are available as a standard type in Martin-Löf's type theory. Consequently, we can define a module satisfying *Sequence*($\mathbb{N}$) using the type List($\mathbb{N}$) to implement *Seq*. We can use the standard list operators ($+\!\!+$), *nil*, *tl* and *hd* to implement the operations defined by *Sequence*. Such an implementation—named *SeqImp*—is given in Figure 3.13. Since *Sequence* does not specify a value for *head*(*empty*) we have the freedom to choose, and in *SeqImp* we define it to be 0. We could have implemented *Sequence* as a

$Sequence \equiv$
$\quad \lambda elem \in U_1.$
$\quad\quad$ **Elements**
$\quad\quad\quad Seq \in U_1,$
$\quad\quad\quad empty \in Seq,$
$\quad\quad\quad \langle \_ \rangle \in elem \rightarrow Seq,$
$\quad\quad\quad \_ \bullet \_ \in Seq \rightarrow Seq \rightarrow Seq,$
$\quad\quad\quad head \in Seq \rightarrow elem,$
$\quad\quad\quad tail \in Seq \rightarrow Seq$
$\quad\quad$ **Restrictions**
$\quad\quad\quad \forall x \in elem. \forall s_1, s_2, s_3 \in Seq.$
$\quad\quad\quad empty \bullet s_1 = s_1 \quad \wedge$
$\quad\quad\quad s_1 \bullet empty = s_1 \quad \wedge$
$\quad\quad\quad s_1 \bullet (s_2 \bullet s_3) = (s_1 \bullet s_2) \bullet s_3 \quad \wedge$
$\quad\quad\quad head(\langle x \rangle \bullet s_1) = x \quad \wedge$
$\quad\quad\quad tail(\langle x \rangle \bullet s_1) = s_1$
$\quad\quad$ **End**

Figure 3.12: A Parameterised Specification of Sequences

$SeqImp \equiv$
$\quad$ **module**

Seq	$=$	$\mathrm{List}(\mathbb{N}),$
$empty$	$=$	nil,
$\langle \_ \rangle$	$=$	$\lambda x.(x : \mathrm{nil}),$
$\_ \bullet \_$	$=$	$\lambda s_1.\lambda s_2.s_1 +\!\!+ s_2,$
$head$	$=$	$\lambda s.$**if** $s = empty$ **then** 0 **else** $hd(s),$
$tail$	$=$	$\lambda s.tl(s)$

$\quad$ **proof**
$\quad \ldots$
$\quad$ **end** $\in Sequence(\mathbb{N})$

Figure 3.13: An Implementation of *Sequence*($\mathbb{N}$)

parameterised module, but we leave the discussion of parameterised modules for the next section.

3.6.2 Parameterisation for Structuring

The specification *ParMeans*, given in Figure 3.14, is another example of a parameterised specification. *ParMeans* is similar to the specification *Means*, given previously in Figure 3.7; both specifications contain operations allowing the calculations of the means of collections of data. The formal parameter *mm* records the fact that *ParMeans* is dependent on a module satisfying the specification *Mean* where *Mean* was given previously in Figure 3.1.

```
ParMeans ≡
  λmm ∈ Mean.
    Elements
      $size \in mm.Data \rightarrow mm.Data \rightarrow \mathbb{N}$,
      $mean \in mm.Data \rightarrow mm.Data \rightarrow \mathbb{N}$
    Restrictions
      $\forall d, e \in mm.Data.$
      $size(d)(e) =_{\mathbb{N}} mm.size(d) + mm.size(e) \quad \wedge$
      $\neg([mm.size(d) =_{\mathbb{N}} 0] \vee [mm.size(d) =_{\mathbb{N}} 0]) \Rightarrow$
        $mean(d)(e) =_{\mathbb{N}} (mm.mean(d) + mm.mean(e))\ div\ 2$
    End
```

Figure 3.14: A parameterised version of *Means*

ParMeans takes a module *mm* ∈ *Mean* as its argument and uses it to make the body of *ParMeans*. In contrast to *Means*, we do not introduce *mm* into the signature of *ParMeans*; this leads to *ParMeans* having a cleaner signature than *Means*. *ParMeans* is an example of where parameterisation is used to structure a specification. *ParMeans* is not intended to be instantiated with many different parameters, but will instead be instantiated with whatever module we finally choose as an implementation for *Mean*, the type of the formal parameter of *ParMeans*.

We choose to implement *ParMeans* as a parameterised module; a parameterised module is a function that returns a module as its result. Figure 3.15 gives an example of one possible implementation of *ParMeans*. *ParMeansModule* is a function that takes any module *mm* ∈ *Mean* and returns a module satisfying *ParMeans*(mm) (i.e. $[\![mm \in Mean \rhd ParMeansModule(mm) \in ParMeans(mm)]\!]$). So, whatever module we finally choose to instantiate *ParMeans* with, *ParMeansModule* guarantees to produce an implementation for the instantiation of *ParMeans*.

```
ParMeansModule ≡
  λmm ∈ Mean.
    module
      size   =  $\lambda d, e \in mm.Data.mm.size(d) + mm.size(e)$,
      mean   =  $\lambda d, e \in mm.Data.$
                  if $(mm.size(d) = 0)$ or $(mm.size(e) = 0)$ then
                    0
                  else
                    $(mm.mean(d) + mm.mean(e))\ div\ 2$
    proof
      ...
    end $\in \prod mm \in Mean.ParMeans(mm)$
```

Figure 3.15: A parameterised version of *MeansModule*

As a separate point, note that *ParMeans* takes a module satisfying *Mean* as its parameter, not the specification *Mean* itself. The only reason for passing the specification

Mean as a parameter is if we want to incorporate *Mean* into the body of *ParMeans* by using some specification operator. But that is not our intention here; we are only interested in using the components supplied by *Mean*. The example shows how the type membership relation between specifications and modules can be used to mix specifications and modules to make specifications.

3.7 Overloading

One feature of specifications and modules is that we allow different components within a signature to have the same name. We call this feature "overloading" of names; note that this use of the word "overloading" is different from its normal use where it means giving a symbol, or name, different meanings depending on the context in which the symbol is used. In practice, we make little deliberate use of overloading, but it can arise during program development, so we allow it. Potentially, overloading can cause ambiguities when it is not clear to which declaration the use of an overloaded name is bound. Additional scope rules are needed to resolve such ambiguities.

In a signature such as $y_1 \in A_1, \ldots, y_n \in A_n$, each name y_i $(0 \leq i \leq n)$ is usually in scope in all types, $A_{i+1} \ldots A_n$, occurring to the right of y_i. But if some y_i is overloaded, then each declaration of y_i is only in scope from its point of declaration up to, but not beyond, the next declaration of the name y_i. Consequently, an overloaded name always binds to its most recent declaration. Similarly, occurrences of an overloaded component name in the restriction of a specification always bind to the last declaration of that name in the signature of the specification.

The following specification illustrates the scope rules for overloaded names:

$$S = \textbf{Elements } x \in \mathbb{N}, y \in \{x\}_{\mathbb{N}}, x \in \mathbb{N}, z \in \{x\}_{\mathbb{N}} \textbf{ Restrictions } x =_{\mathbb{N}} y \textbf{ End}$$

The name x is overloaded in S. The type of y is a singleton type containing the value of the first component x; in other words, y is specified as being equal to x. The type of z is also a singleton type $\{x\}_{\mathbb{N}}$, but in this case x is bound to the x declared as the third component of S. The x referred to by the restriction is also bound to the x declared as the third component of the signature.

We also permit component names to be overloaded in the computational element of a module. In a computational element, any occurrence of an overloaded component name p, for example, always binds to the closest definition of p that is in scope. For example, consider the following computational element:

$$mn = \textbf{module } x = 1, y = x, x = 2, z = x \textbf{ end} \in S$$

Note that component name x is overloaded in mn. The occurrence of x in $y = x$ binds to $x = 1$. But the occurrence of x in $z = x$ binds to $x = 2$ since $x = 2$ is the closest definition of x to $z = x$.

If a component name p, for example, is overloaded in a computational element m, then $m.p$ is the value bound to the last definition of p in m; in other words, $m.p$ is the value bound to the definition of p that is closest to the end of m. For example, if we consider computational element mn above, then $mn.x = 2$ since $x = 2$ is the last definition of x in mn. We can refer to component $x = 1$ by temporarily renaming x in $x = 1$ to some unused component name y, for example, and then refer to $x = 1$ as $mn.y$; renaming component names in computational elements is discussed in Chapter 6.

3.8 An Observational Equality

When we make a specification, we should try to ensure that it does not over-specify the system it is intended to specify. Over-specification may result in a specification that excludes acceptable implementations of a system. In other words, some modules may behave like acceptable implementations of a system, without actually satisfying its specification; we say such modules are ***observationally equivalent*** to implementations of the specification. One cause of over-specification in our specifications is the use of computational equality to equate expressions in the restrictions of our specifications. In this section, we show how loosening the equality relations used in specifications leads to looser specifications that admit observationally equivalent implementations previously excluded.

We proceed by example, considering the specification for sequences of naturals and showing why it fails to admit a particular module as an implementation. We then show how the equalities in the sequence specification may be loosened to admit the previously excluded implementation. Finally, we indicate how the task of loosening equalities can be generalised.

3.8.1 The Problem

Consider the specification *SeqNat*, given in Figure 3.16, which specifies sequences of naturals—*SeqNat* is produced by expanding the application of the generic specification *Sequences* to $\mathbb{N}$. *SeqNat* is a typical example of the specification of an abstract data type: it specifies a type, together with operations over the type. When we specify abstract data types we are usually interested in the so-called *ultra-loose* [56] interpretation of the semantics of the data type. However, *SeqNat* does not specify all the modules that an ultra-loose interpretation of *SeqNat* would allow. Consider an implementation of *SeqNat* where *Seq* is implemented by the type List($\mathbb{N} \times \mathbb{B}$). Each integer element in a list is tagged (paired) with a boolean value; if the tag is *true* then

$SeqNat \equiv$
Elements
$Seq \in U_1,$
$empty \in Seq,$
$\langle \_ \rangle \in \mathbb{N} \to Seq,$
$\_ \bullet \_ \in Seq \to Seq \to Seq,$
$head \in Seq \to \mathbb{N},$
$tail \in Seq \to Seq$
Restrictions
$\forall x \in \mathbb{N}.\forall s_1, s_2, s_3 \in Seq.$
$empty \bullet s_1 = s_1 \quad \wedge$
$s_1 \bullet empty = s_1 \quad \wedge$
$s_1 \bullet (s_2 \bullet s_3) = (s_1 \bullet s_2) \bullet s_3 \quad \wedge$
$head(\langle x \rangle \bullet s_1) = x \quad \wedge$
$tail(\langle x \rangle \bullet s_1) = s_1$
End

Figure 3.16: A Specification of Sequences of Naturals

the tagged integer is part of the sequence, and if the tag is *false* then the element is not part of the sequence. Consider the two sequences s_1 and s_1:

$$\begin{aligned} s_1 &= [(2,\text{false}),(4,\text{true}),(6,\text{true}),(8,\text{false})] \\ s_2 &= [(4,\text{true}),(6,\text{true})] \end{aligned}$$

If we treat sequence as "black boxes" and only look inside them using *head* then s_1 and s_2 can be viewed as equivalent; for example, applying any sequence operation to s_1 and s_2, respectively, and then applying *head* will give equal results. Functions such as *head* which can look at the constituent parts of the values in a type are called "observers" of the type. Sequences s_1 and s_2 are said to be *observationally equivalent* with respect to the observers of *Seq* in *SeqNat*, and we write this as $s_1 \cong_{Seq} s_2$. Removing elements from such sequences is simply a matter of tagging the elements with *false*: such an implementation is often used where explicitly removing elements from sequences is an expensive operation. By using $\text{List}(\mathbb{N} \times \mathbb{B})$ to represent sequences, *tail* may be implemented as a function that marks the first occurrence of a *true* tag to *false*; and the *head* function returns the first element with a *true* tag. A complete implementation for sequences using $Seq = \text{List}(\mathbb{N} \times \mathbb{B})$ is given by *SeqImp2* in Figure 3.17. Note that in defining *SeqImp2* we assume the existence of the functions *hd* and *tl* which are the usual head and tail functions, respectively, on list values.

Although *SeqImp2* behaves as we would expect an implementation of sequences to behave, it does not satisfy *SeqNat*. In particular the equation,

$$\forall x \in \mathbb{N}.\forall s \in Seq.tail(\langle x \rangle \bullet s) =_{Seq} s$$

```
SeqImp2 ≡
  module
    Seq     = List(ℕ × 𝔹),
    empty   = nil,
    ⟨_⟩     = λx.(⟨x, true⟩ : nil),
    _ • _   = λs₁.λs₂.s₁++s₂,
    head    = λs.if s = nil then
                   0
                 else
                   if snd(hd(s)) then
                     fst(hd(s))
                   else
                     head(tl(s)),
    tail    = λs.if s = nil then
                   nil
                 else
                   if snd(hd(s)) then
                     ⟨fst(hd(s)), false⟩ : tl(s)
                   else
                     hd(s) : tail(tl(s))
  proof
    ...
  end ∈ SeqNat
```

Figure 3.17: A loose implementation of *SeqNat*

does not hold. For example, given $x = 1$ and $s = [(2, \text{true}), (3, \text{true})]$ then:

$$
\begin{aligned}
&\quad \mathit{tail}(\langle x\rangle \bullet s)\\
&= \text{“defn } (\_ \bullet \_)\text{”}\\
&\quad \mathit{tail}([(1,\text{true}),(2,\text{true}),(3,\text{true})])\\
&= \text{“defn } \mathit{tail}\text{”}\\
&\quad [(1,\text{false}),(2,\text{true}),(3,\text{true})]\\
&\neq \text{“list equality”}\\
&\quad [(2,\text{true}),(3,\text{true})]\\
&= \text{“given”}\\
&\quad s
\end{aligned}
$$

The reason for the inequality, above, is that we are comparing lists using the equality $=_{Seq}$ which is the built in computational equality $=_{List(\mathbb{N}\times\mathbb{B})}$ on $\text{List}(\mathbb{N} \times \mathbb{B})$. Under this equality, two lists are equal iff they contain the same number of elements, and the corresponding elements are equal. However, from our intuitive understanding we recognise that $\mathit{tail}(\langle x\rangle \bullet s)$ is observationally equivalent to s. (i.e. $\mathit{tail}(\langle x\rangle \bullet s) \cong_{Seq} s$). In order to allow implementations such as *SeqImp2* to satisfy *SeqNat* we must weaken the computational equality, $=_{Seq}$, to an observational equality $\cong_{Seq}$. We do this by adding $\cong_{Seq}$ to *SeqNat*.

3.8.2 A Solution

The specification *SeqNat2*, in Figure 3.18, gives a loose specification of sequences, with an explanation following. We add an observational equality relation, $\cong_{Seq}$, as a new component to *SeqNat* and use it in place of the computational equality $=_{Seq}$. The type of $\cong_{Seq}$ is defined as $Seq \rightarrow Seq \rightarrow U_1$: for any $s_1, s_2 \in Seq$, $s_1 \cong_{Seq} s_2$ is a type, and hence, a proposition. The equality $\cong_{Seq}$ is specified as a congruence w.r.t the operations defined by *SeqNat*. A formal definition of $\cong_{Seq}$ is given by $Congruence(\cong_{Seq})$. Equations (1), (2) and (3) specify $\cong_{Seq}$ to be an equivalence. Axiom (4) states that if two sequences are observationally equivalent then the sequences obtained by applying *tail* to each of them are also observationally equivalent. Axiom (5) states that for two observationally equivalent sequences their *head* values are identical. Finally, axiom (6) states that if s_1 and s_2 are observationally equivalent, and if s_3 and s_4 are observationally equivalent, then $s_1 \bullet s_3$ is observationally equivalent to $s_2 \bullet s_4$. Note that the axiom $head(\langle x \rangle \bullet s_1) =_{\mathbb{N}} x$ uses computational equality on naturals; computational equalities are always sufficient for comparing the values of primitive types such as $\mathbb{N}$, $\mathbb{B}$ etc.

Since *SeqNat2* specifies the observational equality $\cong_{Seq}$ as part of its signature, modules satisfying *SeqNat2* will contain an implementation of $\cong_{Seq}$ as a local component. For example, the loose implementation *SeqImp2* can be made into an implementation of *SeqNat2* by adding a suitable implementation of $\cong_{Seq}$ such as:

$$\_ \cong_{Seq} \_ = \lambda s_1.\lambda s_2.[\mathrm{norm}(s_1) =_{List(\mathbb{N})} \mathrm{norm}(s_2)]$$

here

```
norm ∈ $\mathrm{List}(\mathbb{N} \times \mathbb{B}) \rightarrow \mathrm{List}(\mathbb{N})$
norm = $\lambda s.$if $s = nil$ then
           $nil$
         else
           if $snd(hd(s))$ then
             $fst(hd(s)) : norm(tl(s))$
           else
             $norm(tl(s))$
```

The function *norm* takes any sequence $s \in Seq$ ($Seq = List(\mathbb{N} \times \mathbb{B})$) and transforms it into a sequence of natural numbers by removing the tags on the elements of s; elements with *false* tags are removed entirely. Function *norm* essentially normalises *Seq* values to a canonical form. Hence, $\cong_{Seq}$ defines any two sequences $s_1, s_2 \in Seq$ as observationally equivalent if their normalised forms are computationally equivalent under $List(\mathbb{N})$ equality.

$SeqNat2 \equiv$

Elements

$Seq \in U_1,$

$\bullet\_ \cong_{Seq} \_ \in Seq \to Seq \to U_1,$

$empty \in Seq,$

$\langle \_ \rangle \in \mathbb{N} \to Seq,$

$\_ \bullet \_ \in Seq \to Seq \to Seq,$

$head \in Seq \to \mathbb{N},$

$tail \in Seq \to Seq$

Restrictions

$\forall x \in \mathbb{N}. \forall s_1, s_2, s_3 \in \mathrm{Seq}.$

$Congruence(\cong_{Seq}) \quad \wedge$

$empty \bullet s_1 \cong_{Seq} s_1 \quad \wedge$

$s_1 \bullet empty \cong_{Seq} s_1 \quad \wedge$

$s_1 \bullet (s_2 \bullet s_3) \cong_{Seq} (s_1 \bullet s_2) \bullet s_3 \quad \wedge$

$head(\langle x \rangle \bullet s_1) =_{\mathbb{N}} x \quad \wedge$

$tail(\langle x \rangle \bullet s_1) \cong_{Seq} s_1$

End

where

$Congruence(\cong_{Seq}) \equiv$

$\forall s_1, s_2, s_3, s_4 \in Seq.$

$$s_1 \cong_{Seq} s_1 \quad \wedge \tag{1}$$

$$s_1 \cong_{Seq} s_2 \Rightarrow s_2 \cong_{Seq} s_1 \quad \wedge \tag{2}$$

$$(s_1 \cong_{Seq} s_2 \wedge s_2 \cong_{Seq} s_3) \Rightarrow s_1 \cong_{Seq} s_3 \quad \wedge \tag{3}$$

$$s_1 \cong_{Seq} s_2 \Rightarrow \mathrm{tail}(s_1) \cong_{Seq} \mathrm{tail}(s_2) \quad \wedge \tag{4}$$

$$s_1 \cong_{Seq} s_2 \Rightarrow \mathrm{head}(s_1) =_{\mathbb{N}} \mathrm{head}(s_2) \quad \wedge \tag{5}$$

$$(s_1 \cong_{Seq} s_2) \wedge (s_3 \cong_{Seq} s_4) \Rightarrow s_1 \bullet s_3 \cong_{Seq} s_2 \bullet s_4 \tag{6}$$

Figure 3.18: A looser version of *SeqNat*

3.8.3 A Generalisation of Observational Equivalence

The choice of whether to define a congruence between the values of a type is a specification design decision. Not every abstract data type we specify requires a congruence. If we specify ADT's whose values are to be treated as "black boxes"—such as stacks, queues, containers etc—then we should specify a congruences between their values. On the other hand, if we intend to specify operations that compare the values of an ADT for equality then we should not use congruences. The penalty for using congruences is that the specification becomes more complicated. The penalty for not using congruences is that we may disallow implementations that might well have been suitable for our needs. It could be left up to an implementor to decide whether a congruence is used or not, but adding a congruence "weakens" a specification, and will allow implementations that may not have been considered by the specifier. This may lead to implementations that do not satisfy the original specification, although they will be observationally equivalent to implementations of the original specification.

If we specify an abstract data type—T, say—and decide to use congruence for equality between T-valued expressions, then we can generate the axioms that specify the congruence in a systematic manner. To define a congruence $\_ \cong_T \_ \in T \to T \to U_1$ over type T, $\cong_T$ must first be an equivalence:

$$\begin{array}{ll}
\forall t_1, t_2, t_3 \in T. & \\
t_1 \cong_T t_1 \quad \wedge & \text{(reflexive)} \\
t_1 \cong_T t_2 \Rightarrow t_2 \cong_T t_1 \quad \wedge & \text{(symmetric)} \\
(t_1 \cong_T t_2 \wedge t_2 \cong_T t_3) \Rightarrow t_1 \cong_T t_3 & \text{(transitive)}
\end{array}$$

When we defined the congruence on sequence values in Figure 3.18, it was shown that the operations in *SeqNat* also generated axioms about the congruence. In general, if the specification of T specifies m operations f_j $(0 \leq j \leq m)$ then each operations generates an axiom. For each operation $f_j \in T_1 \to T_2 \to \ldots \to T_n$ the following axiom is generated:

$$\begin{array}{l}
\forall t_1, t_1' \in T_1 \ldots \forall t_{n-1}, t_{n-1}' \in T_{n-1}. \\
\quad (t_1 \equiv_{T_1} t_1') \wedge \ldots \wedge (t_{n-1} \equiv_{T_{n-1}} t_{n-1}') \Rightarrow f_j(t_1, \ldots, t_{n-1}) \equiv_{T_n} f_j(t_1', \ldots, t_{n-1}')
\end{array}$$

where for each type T_i $(1 \leq i \leq n)$, if T_i is a primitive built-in type (such as $\mathbb{N}$, $\mathbb{B}$ etc) then $\equiv_{T_i}$ is the usual computational equality $=_{T_i}$; and if $T_i = T$ then $\equiv_{T_i}$ is the equivalence $\cong_T$.

All the axioms described by the above "prescription" are sufficient to specify a congruence on type T, assuming only operations f_j $(0 \leq j \leq m)$ are used on T-values. If we decide to add new operations on T-values to the specification, then we must also add new axioms about the congruence: the new operations generate axioms in the same way each operation f_j did above.

3.9 Conclusion and Summary

The examples of specifications and modules given above do not describe all the features of our specification language, but only serve to illustrate the style of specifications and modules we use. Wirsing and Broy [56] have used congruence relations to specify observational equivalences within algebraic specifications; they define a simple algebraic specification language that equips every sort in a specification with a congruence relation instead of the usual computational equality. Although our type-theoretic approach to specifications is similar in many ways to algebraic specifications, it is important to stress that many of the specifications presented in this chapter are different from algebraic specifications; the examples of model-oriented specifications, and specifications being used as data-types, are intended to convince the reader of this fact.

We have not discussed the semantics of specifications in this chapter. However, if we are do any formal reasoning with specifications then we need a formal semantics, and this is discussed in Chapter 4.

In summary, we have seen that specifications and modules may have local components that can be used to factor the specification and implementation of complicated components. We also show how the built-in types and operations of the type theory may be used to make specifications and modules. Specifications can also be used as normal data types; this allows us to nest specifications and modules, and also to use modules as records.

Chapter 4

The Semantics of Specifications

4.1 Introduction

It is important that specification and implementation languages should have a formal semantics. Firstly, a formal semantics can resolve possible ambiguities about the meaning of specifications. Secondly, the semantics can be used to deduce the properties of a program from its specification. Thirdly, the semantics can be used to justify laws for refining and implementing specifications. In this chapter, we define the semantics of our canonical specifications and modules in Martin-Löf's type theory.

The are several reasons why we choose to define the semantics of specifications and modules in type theory. Firstly, a type-theoretic semantics for specifications and modules is simple because the formal language of type theory can be used as a programming language, a specification language and a programming logic. Consequently, the semantics of specifications and modules can be given in the single framework of type theory; no other theory is required. Secondly, features of the type theory such as dependent sum types, and propositions, are particularly suited to specifying modules: for example, the dependent sum type readily captures the inter-dependencies between components in a module. Thirdly, type theory provides a simple definition of a module satisfying a specification: recall that a specification is a type, and that the values of the type are modules satisfying the specification.

Our semantics for specifications and modules is based on the approach of Nordström, Petersson and Smith [43, 44] (henceforth known as the NPS approach) to specifying modules in Martin-Löf's Type Theory. The NPS approach has influenced other work on specifying modules in type theory, most notably the *deliverables* approach advocated by Burstall et al [6] and Luo [27]. Both approaches use dependent sum types, in conjunction with the principle of propositions as types, to specify modules in type theory.

We will give a semantics to our specifications and modules by translating them into terms in MLT. This chapter defines the terms of MLT that we use as translations for specifications and modules. A formal definition of a mapping from the syntax of specifications and modules to their translations in MLT will be given in Chapter 5. This split in the presentation of the semantics is purely for presentation. The terms used as translations for specifications and modules can be understood without a formal definition of the translation mapping; its definition in this chapter would simply obscure the definitions of those terms.

In the NPS and deliverables approaches, specifications and modules are written directly as terms in MLT, but we chose not to do this as the translations of our specifications and modules are complicated terms; it is easier for a programmer to write them in their un-translated form. The translations are complicated by the need to give a semantics to component names and local components within specifications and modules. Neither the NPS nor deliverables approaches address those issues. We need to give a semantics to component names and local components to define specification operations such as renaming and hiding, and module operations such as dot notation. However, giving a semantics to names is problematic: the translations of specifications and modules must distinguish between the use of names as bound variables to describe dependencies between components, and the use of names to refer to components in specifications and modules.

One feature of our semantics is that we do not add any new types to MLT. The translations of specifications and modules are all existing terms in MLT, and can be seen as constituting a meta-theory sitting on top of MLT.

4.2 A review of semantics

We begin by reviewing the NPS and deliverables approaches. In those approaches, a programmer writes specifications and modules as terms in type theory: specifications denote dependent sum types and modules denote nested pairs. Therefore, it is unnecessary to translate specifications and modules into terms in the type theory. Neither approach includes component names or local components in specifications and modules. The NPS and deliverables approaches differ in their treatment of restrictions in specifications, and witnesses in modules; we will illustrate these differences below.

We review the NPS and deliverables approaches by example. First, we give an example specification and module written in our specification and implementation language, respectively. Then we show how our example specification and module would be written in the NPS and deliverables approaches and say what the new specifications and modules denote. Finally, we give a critique of both approaches.

$MS \equiv$
  Elements
    $G \in U_1$,
    $\_ \oplus \_ \in G \to G \to G$,
    $id \in G$
  Restrictions
    $\forall a, b, c \in G.$
    $[(a \oplus b) \oplus c =_G a \oplus (b \oplus c)] \quad \wedge$
    $[a \oplus id =_G a] \quad \wedge$
    $[id \oplus a =_G a]$
  End

Figure 4.1: The specification of monoids

$nm \equiv$
  module
    $G \;=\; \mathbb{N}$,
    $\oplus \;=\; +$,
    $id \;=\; 0$
  proof
    $\lambda a \in \mathbb{N}.\lambda b \in \mathbb{N}.\lambda c \in \mathbb{N}.\langle eq, \langle eq, eq\rangle\rangle$
  end $\in MS$

Figure 4.2: A module describing a monoid

4.2.1 A running example—Monoids

We consider the specification of monoids which is given in Figure 4.1 with explanation following. In mathematics a monoid is an algebra consisting of a carrier set, a binary operation and an identity value; all of which satisfy certain relationships. Component G is a type representing the carrier of the monoid; $\oplus$ is an associative infix binary operator with type $G \to G \to G$; and id is a two-sided identity for $\oplus$. A familiar monoid is the naturals ($\mathbb{N}$) with the integer addition operator ($+$) and identity 0; in mathematics such a monoid is usually packaged as the tuple $\langle \mathbb{N}, +, 0\rangle$. Other examples of monoids include $\langle List(\mathbb{N}), +\!\!+, nil\rangle$ where $+\!\!+$ is list concatenation and nil is the empty list; and $\langle Bool, \wedge, true\rangle$ where $\wedge$ is the logical-and operator. We can describe monoids by modules in our implementation language; for example, the module nm, given in Figure 4.2, describes the monoid $\langle \mathbb{N}, +, 0\rangle$. Note that nm includes a witness; the purpose of the witness will become clear later.

4.2.2 Specifications in the NPS approach

MS_1, in Figure 4.3, is a specification of monoids in the NPS approach, with explanation following. There is a strong syntactic resemblance between MS_1 and our specifi-

$$
\begin{array}{l}
MS_1 \equiv \\
\quad \sum G \in U_1. \\
\quad \sum \_ \oplus \_ \in G \to G \to G. \\
\quad \sum id \in G. \\
\qquad (\prod a \in G. \prod b \in G. \prod c \in G. \\
\qquad\quad ([(a \oplus b) \oplus c =_G a \oplus (b \oplus c)] \times \\
\qquad\quad [a \oplus id =_G a] \times \\
\qquad\quad [id \oplus a =_G a]))
\end{array}
$$

Figure 4.3: A Specification of monoids in the NPS approach

cation MS. However, specifications in the NPS approach are terms in type theory that denote dependent types; MS_1 is what a programmer would write as a specification for monoids. Dependent product types can express the type dependencies that may exist between components in a module; for example, MS_1 expresses the fact that the types of the second ($\oplus$) and third (id) components in a monoid are dependent on the value of the first component (G). The names G, $\oplus$ and id are only bound variables. Dependent product types alone cannot specify the behaviour of components in specifications. That is done by adding a final field which is a proposition specifying the behaviour of the preceding components. The proposition specifying the properties of G, $\oplus$ and id in MS_1 is $Ax(MS_1)$, where

$$
\begin{array}{rl}
Ax(MS_1) = & (\prod a \in G. \prod b \in G. \prod c \in G. \\
& \quad ([(a \oplus b) \oplus c =_G a \oplus (b \oplus c)] \times \\
& \quad [a \oplus id =_G a] \times \\
& \quad [id \oplus a =_G a]))
\end{array}
$$

Using the isomorphism between types and propositions, it can easily be shown that $Ax(MS_1)$ corresponds to the classical proposition:

$$\forall a, b, c \in G.((a \oplus b) \oplus c = a \oplus (b \oplus c) \ \wedge \ a \oplus id = a \ \wedge \ id \oplus a = a)$$

which is identical to the restriction in our specification MS.

4.2.3 Modules in the NPS approach

In the NPS approach, modules are terms in type theory, and these terms denote pairs. The pairs are members of the types that specifications denote. A module m is said to "satisfy" a specification SP if $m \in SP$. For example, nm_1, below, is a module that describes a monoid, and nm_1 satisfies MS_1:

$$nm_1 = \langle \mathbb{N}, \langle +, \langle 0, \lambda a \in \mathbb{N}.\lambda b \in \mathbb{N}.\lambda c \in \mathbb{N}.\langle eq, \langle eq, eq \rangle \rangle \rangle \rangle \rangle \in MS_1$$

Note that nm_1 is constructed from nested pairs since MS_1 is a nested dependent sum type. The module nm_1 describes the same monoid as the module nm which was given in Figure 4.2. In the NPS approach, the components in modules are not named, and modules do not allow local components. We can refer to the components in nm_I by compositions of *fst* and *snd*. The lack of names means that such modules cannot express dependencies between their components. The final value in a module is a witness. For example, the final value in nm_I,

$$\lambda a \in \mathbb{N}.\lambda b \in \mathbb{N}.\lambda c \in \mathbb{N}.\langle eq, \langle eq, eq\rangle\rangle,$$

is a witness that is a member of the proposition $Ax(MS_1)$, and is equal to the witness of the module nm.

To see how the components in modules that satisfy MS_1 obey the proposition $Ax(MS_1)$, consider any module $\langle G, \langle \oplus, \langle id, r\rangle\rangle\rangle \in MS_1$. The extra component r is a witness and belongs to the type $Ax(MS_1)$. Since we know that $r \in Ax(MS_1)$ we know that for the particular values G, $\oplus$ and id in $\langle G, \langle \oplus, \langle id, r\rangle\rangle\rangle$ the type $Ax(MS_1)$ is inhabited, i.e it is true. Therefore, we can conclude that the corresponding classical proposition is true for G, $\oplus$ and Id; the value r is a witness that proves that the proposition is true. The actual value of r is not important.

4.2.4 Specifications in the deliverables approach

In the deliverables approach, specifications are terms in type theory that denote dependent product types. MS_2, in Figure 4.4, is a specification of monoids, in the deliverables approach. MS_2 is a straight translation of MS_1. In the deliverables approach, the left field of a specification is called the *structure type* of the specification. The structure type of MS_2 is a dependent product type S, where

$$S = \sum G \in U_1. \sum \_ \oplus \_ \in G \to G \to G.G$$

The fields in S are the data-types of the components included in MS_2. The right field of MS_2 is a proposition that specifies axioms about the components included in S; we name this proposition $Ax(MS_2)$. The main difference between specifications in the NPS and deliverables approaches is that in the deliverables approach axioms are defined independently of the bound component names given in the structure type. For example, note that the bound names G and $\oplus$, in S, are not in scope in $Ax(MS_2)$. $Ax(MS_2)$ is defined in terms of the single bound variable $m \in S$, and refers to the components of m by compositions of *fst* and *snd*. We abbreviate the use of projection functions by m_1, m_2 and m_3; m_1 is the carrier of a monoid; m_2 is the associative binary operator; and m_3 is the two sided identity value.

The main advantage of the deliverables approach—over the NPS approach—is that specifications can easily be factored into their "computational contents" (given by the structure type) and their axiomatic requirements. In the NPS approach, the axioms are nested deep within a $\sum$-type and this makes factorisation harder. Factoring a specification has several advantages. Firstly, it simplifies the definition of operations

$$MS_2 \equiv$$

$$\sum m \in \begin{pmatrix} \sum G \in U_1. \\ \sum \text{-}\oplus\text{-} \in G \to G \to G. \\ G \end{pmatrix} . \begin{pmatrix} \forall a, b, c \in m_1. \\ (a\ (m_2)\ b)\ (m_2)\ c = a\ (m_2)\ (b\ (m_2)\ c) \ \wedge \\ a\ (m_2)\ (m_3) = a \ \wedge \\ (m_3)\ (m_2)\ a = a \end{pmatrix}$$

here m_1, m_2 and m_3 are abbreviations defined by:

$$\begin{aligned} m_1 &= \mathit{fst}(m)\ ; \\ m_2 &= \mathit{fst}(\mathit{snd}(m))\ ; \text{ and} \\ m_3 &= \mathit{snd}(\mathit{snd}(m)) \end{aligned}$$

Figure 4.4: A specification for monoids in the deliverables approach

on specifications as it allows definitions to be factored into two parts: one part defines the operation over the structure type, and the other part defines the operation over the axioms. Secondly, it makes it easier to separate redundant witnesses from modules satisfying such specifications; this is described in the next section.

4.2.5 Modules in the deliverables approach

Modules in the deliverables approach are similar to modules in the NPS approach as they are also terms in type theory that denote nested pairs. For example, nm_2, below, is a module in the deliverables approach, and it satisfies MS_2:

$$nm_2 = \langle\ \langle \mathbb{N}, \langle +, 0 \rangle \rangle\ ,\ \lambda a \in \mathbb{N}.\lambda b \in \mathbb{N}.\lambda c \in \mathbb{N}.\langle eq, \langle eq, eq \rangle \rangle\ \rangle \in MS_2$$

The only difference between modules in the NPS and deliverables approaches is the way in which components are nested. In the deliverables approach, the witness in a module—m, say—is nested so that it is $\mathit{snd}(m)$, and the other "computational" components are, collectively, $\mathit{fst}(m)$.

As with specifications, there are some advantages to the ease with which a deliverables module can be factored into its witness and computational components. Firstly, it simplifies proofs concerning modules: most constructive proofs concerning modules need to refer to the witness within a module, and the easier this is, the clearer the proof. Secondly, it simplifies the definition of many operations on modules by allowing the definition to be factored into an operation over the computational components of a module, and an operation over the witness.

4.2.6 A Critique

The NPS and deliverables approaches illustrate that, in principle, type theory can be used as the basis for the semantics of specifications and modules. Our main criticism of both the NPS and deliverables approaches is the omission of a name-space for components within specifications and modules. The failure to address the name-space issue impedes the use of both approaches as the basis for a full-blown specification and implementation language for modules. Both approaches fail to recognise that component names play a dual role: on the one hand they specify dependencies between components, and on the other hand they name components within specifications and modules, so that client specifications and modules can refer to components by name. Both the NPS and deliverables approach only recognise the role of component names in specifications as that of specifying dependencies.

At first sight, MS_1 does appear to introduce the names G, $\oplus$ and id for the components it specifies. However, G, $\oplus$ and id are only bound variables within MS_1, and so can be changed without changing the meaning of the specification. The lack of component names in the deliverables style means that a programmer must use *fst* and *snd* to write the proposition that specifies the behaviour of the components in a specification. Clearly, this is undesirable as it complicates the specification, and makes references to components position dependent. The types that MS_1 and MS_2 denote are—type theoretically—isomorphic, but MS_2 is more complicated to write than MS_1. Nevertheless, the ease with which specifications and modules can be factored in the deliverables approach does make formal reasoning about specifications and modules easier than in the NPS approach.

Ideally, we would like a fusion of the NPS and deliverables approach in which the axioms can be defined using component names, but where specifications and modules can still be factored easily. In addition, we would like to add component names and local components to specifications and modules. The fusion of the two approaches, together with the additions described here, form the basis for the semantics of our specifications and modules.

4.3 An Introduction to Our Semantics

In this section, we give an introduction to our semantics of specifications and modules. Our semantics are given by translating specifications and modules into terms in Martin-Löf's Type Theory (MLT). The translation of a specification is a term in MLT that denotes a type. This type is a dependent product type whose left component is a translation of the signature of the specification, and whose right component is a translation of the restriction of the specification. It follows that the translations of signatures and restrictions are also terms in MLT that denote types. The translation of a module is a term in MLT that denotes a pair. The left and right components of this pair are the translations of the computational element and witness of the module, respectively.

To distinguish between specifications as they are written, and their translation, we write the translation of a specification—SP, say—as $[\![SP]\!]$. We also write "specification in MLT" to mean a term in MLT obtained by translating a specification. Similarly, we write the translation of a signature—S, say—as $[\![S]\!]$, and write "signature in MLT" to mean a term in MLT obtained by translating a signature; and so on for modules, computational elements, restrictions and witnesses. In addition, we sometimes write "specification as a type" and "signature as a type" to mean "specification in MLT" and "signature in MLT", respectively.

We will require that members of specifications in MLT be modules in MLT. So the choice of translations for specifications also determines the translation of modules. A module m is said to "satisfy" a specification SP if $[\![m]\!] \in [\![SP]\!]$. Moreover, if $[\![m]\!] \in [\![SP]\!]$ then the computational element of $[\![m]\!]$ will be a member of the signature of $[\![SP]\!]$, and the witness of $[\![m]\!]$ will be a member of the restriction of $[\![SP]\!]$.

The main technical problem in defining translations for specifications concerns translations for signatures. The translation of a signature must describe the type of each component in the signature, and the domain of the signature, and it must distinguish between local and visible names. As we require the members of signatures in MLT to be computational elements in MLT, the choice of translations for signatures also determines the translations of computational elements. Computational elements in MLT will be treated like records, and we will again refer to their components using a Pascal style dot notation. Further discussion about the translations of signatures and computational elements is left until Sections 4.4 and 4.6, respectively.

4.3.1 An Example Specification in MLT

$$
\begin{array}{l}
[\![MS]\!] \equiv \\
\quad \sum m \in [\![G \in U_i, \oplus \in G \to G \to G, id \in G]\!]. \\
\qquad\quad \forall a, b, c \in m.G. \\
\qquad\quad (a\ m.\oplus\ b)\ m.\oplus\ c = a\ m.\oplus\ (b\ m.\oplus\ c)\ \wedge \\
\qquad\quad a\ m.\oplus\ m.id = a\ \wedge \\
\qquad\quad m.id\ m.\oplus\ a = a
\end{array}
$$

Figure 4.5: The translation of specification MS

Figure 4.5 gives the translation of specification MS (as given in Figure 4.1). The expression $[\![G \in U_i, \oplus \in G \to G \to G, id \in G]\!]$ denotes the translation of the signature of MS. The structure of $[\![MS]\!]$ is similar to that of a deliverables' specification. But, unlike the deliverables approach, the restriction of $[\![MS]\!]$ refers to the components in the computational element, m, by dot notation instead of *fst* and *snd*. For example, the restriction of $[\![MS]\!]$ refers to component G as $m.G$. Unlike dot notation on modules, dot notation on computational elements can refer to local components.

4.3.2 An Example Module in MLT

The pair $[\![nm]\!]$, below, is the translation of the module *nm*, given in Figure 4.2, as an implementation of *MS*:

$$[\![nm]\!] = \langle [\![\textbf{module } G = \mathbb{N}, \oplus = +, Id = 0 \textbf{ end}]\!], \lambda a, b, c \in \mathbb{N}.\langle eq, \langle eq, eq \rangle\rangle\rangle$$

The left component of $[\![nm]\!]$ denotes the translation of the computational element of *nm*. The right component of $[\![nm]\!]$ is the translation of the witness of *nm*. Note that the witness of *nm* and its translation are syntactically identical; this is because the witness of *nm* is already a term in MLT. $[\![nm]\!]$ is a member of type $[\![MS]\!]$, so *nm* satisfies *MS*. Furthermore, the computational element of $[\![nm]\!]$ is a member of the signature of $[\![MS]\!]$, and the witness of $[\![nm]\!]$ is a member of the restriction of $[\![MS]\!]$.

Remark Section 4.8 contains a detailed example of the translation of a specification and module, and it may help the reader to refer to Section 4.8 when reading the formal semantics of specifications and modules which we give in the following sections. □

4.4 Signatures in MLT

In this section we consider the translation of signatures. The translation of a signature is a term in MLT that denotes a product type. The left component of this type is just the domain of the signature packaged as a singleton type. The right component is called the "loose signature" of the signature. A loose signature is similar to the $\sum$-types used as specifications by the NPS and deliverables approaches, but where the restriction is omitted and where names are just treated as bound variables. We will give the domain of a signature a formal meaning as a term in MLT that denotes a list of tagged names for components in a signature; the tags indicate whether the name is local or visible. The word "domain" is overloaded as we will also use it to mean the domain packaged as a singleton type. In general, when we refer to the domain of a signature we will mean a list of tagged names, and if we use domain to mean a type then this will be clear from the context; if there is potentially any doubt about the use of domain to mean a type, we will write "domain as a type".

4.4.1 An Illustration

We illustrate the translation of signatures by considering the translation of:

$$S = G \in U_1, \_\oplus\_ \in G \rightarrow G \rightarrow G, id \in G.$$

The translation of S, written $[\![S]\!]$, is given below, with an explanation following:

$$\{[inl(`G'), inl(`\oplus'), inl(`id')]\}_{List(\mathbb{S}+\mathbb{S})} \times (\textstyle\sum x \in U_1.\sum y \in x \rightarrow x \rightarrow x.\sum z \in x.T)$$

The singleton type to the left of the product ($\times$) is the domain of $[\![S]\!]$, and the $\sum$-type on the right is the loose signature of $[\![S]\!]$. The names x, y, z used in the loose signature are just bound variables that express the dependencies between components. Note that the loose signature is terminated by the unit type T; T is needed for technical reasons which we will explain later. The domain, as a type, is a singleton type containing the domain as a list. We define the domain as a singleton type since all computational elements meeting a signature have the same domain (list of names). The terms 'G', '$\oplus$' and 'id' are the translations of the names of the components in S and have type $\mathbb{S}$. The names in the domain are tagged so that we can distinguish between local and visible components: visible component names are tagged with *inl*, and local names are tagged with *inr*. Tagged names have type $(\mathbb{S} + \mathbb{S})$. The names in the domain occur in the same order as they appear in signatures. The special case of the empty signature (Φ) is translated to the type $\{nil\}_{List(\mathbb{S}+\mathbb{S})} \times T$: the loose signature is the unit type T, and the domain is the empty list.

4.4.2 The Signature Judgement

In the following, we give the formal definition of a judgement **sig**, such that S **sig** holds whenever S is a term in MLT that is the translation of some signature. The judgement **sig** is not intended as a new judgement for the type theory, but is a meta-judgement defined in terms of the existing judgements. Consequently, signatures do not have a formal standing within the type theory. For example, there are no new formation, elimination or introduction rules for signatures or their values. We use $\{\}$- and $\sum$-formation, -introduction, and -elimination rules to construct and reason about signatures as types. The advantage of this approach is that we avoid extending the type theory, with the consequent obligation to prove that the extension is consistent with the existing theory.

To help define the judgement **sig**, we define a judgement **lsg**, such that P **lsg** holds if P is a well-formed loose signature; in other words, P **lsg** holds if P is a nested $\sum$-type terminated by T, or P is just T.

Definition 4.1 (Well-Formed Loose Signature)
The judgement **lsg** is defined inductively as follows:

T **lsg**
$\sum x \in A.B(x)$ **lsg** iff A **type** and $|[x \in A \rhd B(x)$ **lsg**$]|$
$|[x \in A \rhd T$ **lsg**$]|$ iff A **type**

□

Fact 4.1 Given a type term P, if P **lsg** then P **type**. □

The type of a domain—as a list—is $List(\mathbb{S} + \mathbb{S})$. We introduce the symbol $\mathbb{D}$ to denote the type $List(\mathbb{S} + \mathbb{S})$.

Notation $\mathbb{D} = List(\mathbb{S} + \mathbb{S})$ □

Using the judgement **lsg**, we give the formal definition of a well-formed signature in MLT as the product of a domain (as type) and a loose signature:

Definition 4.2 (Well-Formed Signature in MLT)
$\{l\}_{\mathbb{D}} \times S$ **sig** iff S **lsg** and $l \in \mathbb{D}$
□

Fact 4.2 Given a type term S, if S **sig** then S **type**. □

Note, the definition of judgement **sig** does not require the length of the domain, l, to be equal to the number of components included in the loose signature S. In practice, the number of components in l and S are the same, but as we shall see later, we can define all the operations we will need without making use of this fact. The definition of **sig** does not prevent duplicate names appearing in a domain, and indeed duplicate names occur in the domain of a signature whenever we translate a signature that overloads the names of some of its components.

Loose signatures are terminated by the unit type T in order to disambiguate between potentially ambiguous loose signatures. We illustrate the problem by considering the following signatures:

$$\begin{aligned} S_1 &= y \in (\textstyle\sum x \in P.Q(x)) \\ S_2 &= y \in P, z \in Q \end{aligned}$$

S_1 includes a single component y that is a pair, and S_2 includes two components y and z of type P and $Q(y)$, respectively. The loose signatures of S_1 and S_2 are LS_1 and LS_2, respectively, where:

$$\begin{aligned} LS_1 &= \textstyle\sum y \in (\sum x \in P.Q(x)).T \\ LS_2 &= \textstyle\sum y \in P.\sum z \in Q.T \end{aligned}$$

If we do not terminate loose signatures with the unit type then the loose signature of S_1 and S_2 is:

$$LS = \sum x \in P.Q(x)$$

LS is ambiguous: LS could be interpreted as a loose signature including two components of type P and Q, respectively, or a loose signature including a single component that is a pair. Without a terminating unit type, both interpretations of LS seem reasonable. This ambiguity is undesirable since S_1 and S_2 are different signatures. Such an ambiguity arises in all signatures whose final component is a pair, and can always be solved by using the unit type to mark the end of a loose signature.

4.4.3 Some Operations on Signatures

In order to reason about signatures in MLT, we define two functions, *Dom* and *Lsg*, which both take a signature in MLT, and return its domain and loose signature, respectively. The definitions of *Dom* and *Lsg* are given by definitions 4.3 and 4.4, respectively, and an explanatory remark follows the definitions.

Definition 4.3 (*Dom*)
Given $\{l\}_{\mathbb{D}} \times S$ **sig**, and any P **type** (P not a signature) then $Dom \in U_1 \to \mathbb{D}$ is defined as follows:

$$\begin{array}{lcl} Dom(\{l\}_{\mathbb{D}} \times S) & = & l \\ Dom(P) & = & nil \end{array}$$

□

Definition 4.4 (*Lsg*)
Given $\{l\}_{\mathbb{D}} \times S$ **sig**, and any P **type** (P not a signature) then $Lsg \in U_1 \to U_1$ is defined as follows:

$$\begin{array}{lcl} Lsg(\{l\}_{\mathbb{D}} \times S) & = & S \\ Lsg(P) & = & P \end{array}$$

□

Remark For the sake of presentation, *Dom* and *Lsg* are given in a clausal form, although, strictly speaking, they should be defined using the *urec* elimination operator on types. Note that the functions *Dom* and *Lsg* are defined over all types, not just signatures, since operations on types must be total in MLT. Both definitions, have two clauses, one for signatures and one for non-signatures, and these cover all possible arguments to *Dom* and *Lsg*. In practice, we are only interested in applying *Dom* and *Lsg* to signatures in MLT, and the definition of *Dom* and *Lsg* over non-signatures is not important. □

Fact 4.3 If S **sig** then $Lsg(S)$ **lsg**. □

4.4.4 Scope within Signatures

This section describes how signatures in MLT incorporate scope. The scope of a name in a signature is the scope of the bound variable it is translated to in the loose signature. Hence, the scope of names is enforced by the scope rules for bound variables in quantified terms of the type theory—we do not need to impose any extra rules, which is one of the advantages of our choice of translations for signatures. The scope rules for quantified expressions in MLT are summarised by saying that the scope of a bound variable extends from its declaration to the next unmatched parenthesis. But, if a variable is declared twice, such as x in $\sum x \in P. \sum x \in Q.R(x)$, then free occurrences of the variable bind to the closest declaration of the variable; so, x in $R(x)$ binds to $x \in Q$. Full details about the scope of bound variables in MLT are given in [44].

We illustrate the incorporation of scope within signatures with an example. Consider the signature:

$$NS = p \in P_0,\ q \in (p \in P_1(p), r \in R(p)),\ p \in P_2(p,q),\ s \in S(p,q)$$

NS overloads the name p, and moreover, p occurs a further time in the type of q. The loose signature of NS is given below:

$$\sum p \in P_0. \sum q \in (\sum p \in P_1(p). \sum r \in R(p).T). \sum p \in P_2(p,q). \sum s \in S(p,q).T$$

Let us consider the scope of the bound variables in the loose signature of NS. The scope rules for bound variables tell us that the name p in $R(p)$ is bound to $p \in P_1(p)$, rather than $p \in P_0$, since $p \in P_1(p)$ is the closest declaration of p. The name p in $S(p,q)$ is bound to $p \in P_2(p,q)$ since $p \in P_2(p,q)$ is closer than $p \in P_0$. The name p in $P_1(p)$ and $P_2(p,q)$ is bound to $p \in P_0$. So, the scope of each bound variable in the loose signature of NS is exactly the desired scope of the corresponding component name in NS.

4.5 Specifications in MLT

This section considers the translation of specifications. In Section 4.3, we outlined the translation of a specification as being a term in MLT that denotes a dependent product type whose left and right fields are the translation of the signature and restriction, respectively, of the specification. Figure 4.5 also gave an example, $[\![MS]\!]$, of the translation of the specification, MS, for monoids. We now generalise the outline given in Section 4.3, by considering the following specification:

$$SP = \textbf{Elements } S \textbf{ Restrictions } R(y_1, \ldots, y_n) \textbf{ End},$$

We assume that signature S includes components named $y_1, \ldots, y_n$. The translation of SP is given below with an explanation following:

$$[\![SP]\!] = \sum x \in [\![S]\!].R(x.y_1, \ldots, x.y_n)$$

$[\![S]\!]$ denotes the translation of signature S, and $R(x.y_1, \ldots, x.y_n)$ is the translation of the restriction $R(y_1, \ldots, y_n)$. The translation of a restriction is a term in MLT that denotes a proposition—and, hence, a type—dependent on $x \in [\![S]\!]$. Although a restriction is already a term in MLT that denotes a proposition, its translation is obtained by substituting each free occurrence of component names y_i $(1 \leq i \leq n)$ by $x.y_i$; recall that the substitution is necessary so that the translation of the restriction can refer to the components $y_1, \ldots, y_n$ in $x \in [\![S]\!]$.

4.5.1 Some Specification Notation

In the following, we give the formal definition of the judgement **spec** which is used to determine whether a type is a well-typed specification. Just like the judgement **sig**, the judgement **spec** is not intended to be a new judgement for the type theory, but rather, it is a meta-judgement that holds whenever a term in MLT is the translation of some specification.

Definition 4.5 (Judgement spec)
$\sum m \in S.R(m)$ **spec** iff S **sig** and $|[m \in S \rhd R(m)$ **type**$]|$
□

Notation If $\sum m \in S.R(m)$ **spec** then we often write $\langle S \mid (m)R \rangle$ as a shorthand for $\sum m \in S.R(m)$. □

The functions *Sig* and *Ax*, defined below, are needed when defining operations and properties over specifications. The function *Sig* is used to return the signature (in MLT) of a specification. The function *Ax* returns the restriction (in MLT) of a specification. Note that *Ax* actually returns a function: the translations of restrictions are dependent on computational elements satisfying the signature of a specification, so they are functions that take computational elements and return propositions.

Definition 4.6 (*Sig* and *Ax*)
Given $\sum m \in S.R(m)$ **spec** then functions $Sig \in U_1 \to U_1$ and $Ax \in \prod X \in U_1.(X \to U_1)$ are defined as follows:

$$\begin{array}{rclcl} Sig(\sum m \in S.R(m)) & = & S & \in & U_1 \\ Ax(\sum m \in S.R(m)) & = & \lambda m \in S.R(m) & \in & S \to U_1 \end{array}$$

□

Fact 4.4 If SP **spec** then $Sig(SP)$ **sig**. □

Remark Functions *Sig* and *Ax* are actually defined using the *urec* operator, and so are total functions over all types, not just specifications. For any non-specification P, we define $Sig(P) = P$ and $Ax(P) = \lambda m \in Sig(P).T$. However, since we never need to consider the application of *Sig* and *Ax* to types other than module specifications, we have omitted the full definition of *Ax* and *Sig* from Definition 4.6. □

4.6 Computational Elements in MLT

In this section we consider the translation of computational elements. The translations of computational elements are members of the translation of signatures, so they are terms in MLT that denote pairs. A computational element m is said to "meet" (or "satisfy") a signature S if $[\![m]\!] \in [\![S]\!]$. The left component of a computational element $[\![m]\!] \in [\![S]\!]$ is always the domain of $[\![S]\!]$, and so—further overloading the word domain—we call it the domain of $[\![m]\!]$. The right component of $[\![m]\!]$ is called the *value tuple* of $[\![m]\!]$, and is a member of the loose signature of $[\![S]\!]$.

4.6.1 An Example Translation

We illustrate the translation of computational elements by considering the following:

$$m = \textbf{module } G = \mathit{List}(\mathbb{N}), \oplus = +\!\!+, \mathit{id} = \mathit{nil} \textbf{ end} \in S$$

The computational element m meets the signature, S, for monoids, given in Section 4.4. The translation of m is:

$$[\![m]\!] = \langle [\mathit{inl}(`G'), \mathit{inl}(`\oplus'), \mathit{inl}(`\mathit{id}')], \langle \mathit{List}(\mathbb{N}), \langle +\!\!+, \langle \mathit{nil}, \mathit{tt}\rangle\rangle\rangle\rangle \in [\![S]\!]$$

The left component of $[\![m]\!]$ is the domain of $[\![m]\!]$, and is equal to the domain of $[\![S]\!]$:

$$[\mathit{inl}(`G'), \mathit{inl}(`\oplus'), \mathit{inl}(`\mathit{id}')] = \mathit{Dom}([\![S]\!])$$

The right component of $[\![m]\!]$ is the value tuple of $[\![m]\!]$ and is a member of the loose signature of $[\![S]\!]$:

$$\langle \mathit{List}(\mathbb{N}), \langle +\!\!+, \langle \mathit{nil}, \mathit{tt}\rangle\rangle\rangle \in \mathit{Lsg}([\![S]\!])$$

Note that the final component in a value tuple is always the unit value, tt, since loose signatures are always terminated by the unit type T. The translation of the empty module, **module end** , is $\langle \mathit{nil}, \mathit{tt}\rangle$ where nil is the empty domain, and $tt \in T$ is an empty value tuple.

The names in the domain of a computational element are the names of corresponding components in its value tuple. For example, the component $\mathit{List}(\mathbb{N})$ has the name 'G'; $+\!\!+$ has name '$\oplus$'; and nil has the name 'id'. The domain of a computational element will be used to define a dot notation so that we can refer to the components in the value tuple by name; for example, $[\![m]\!].`G'$ refers to $\mathit{List}(\mathbb{N})$. In this thesis, we will not require to make use of a formal definition of dot notation but, for completeness, one is given in Appendix A.

4.6.2 Notations for Computational Elements in MLT

This section give some formal notations on computational elements in MLT.

Definition 4.7 (Signature Satisfaction)
Let m be a computational element, and S **sig**. m meets S iff $m \in S$.
□

Figure 4.6 gives two rules to judge whether a computational element meets a signature. The first rule is for the empty signature, and the second rule is for non-empty signatures. The rules are not new inference rules for the type theory, but are deduced as a consequence of the definition of signatures and computational elements. The rules are justified using the introduction and elimination rules for singleton types and $\sum$-types. Informally, *Comp1* and *Comp2* can be seen as introduction rules for signature in MLT.

$$\text{Comp1} \quad \langle nil, tt\rangle \in \{nil\}_{\mathbb{D}} \times T$$

$$\text{Comp2} \quad \frac{a \in \mathbb{S} + \mathbb{S} \quad e \in A \quad \{l\}_{\mathbb{D}} \times B(e)\ \mathbf{sig} \quad \langle l, b\rangle \in \{l\}_{\mathbb{D}} \times B(e)}{\langle a : l, \langle e, b\rangle\rangle \in \{a : l\}_{\mathbb{D}} \times (\sum x \in A.B(x))}$$

Figure 4.6: Inference rules for Computational elements

We define two projection functions, *dom* and *val*, which each take a computational element, and return its domain and value tuple, respectively.

Definition 4.8 (*dom* **and** *val*)
Given S **sig** and $m \in S$ then functions $dom \in S \to \{Dom(S)\}_{\mathbb{D}}$ and $val \in S \to Lsg(S)$ are defined as follows:

$$\begin{array}{lclcl} dom(m) & = & fst(m) & \in & \{Dom(S)\}_{\mathbb{D}} \\ val(m) & = & snd(m) & \in & Lsg(S) \end{array}$$

□

Fact 4.5 Given S **sig** and $m \in S$ then $m = \langle dom(m), val(m)\rangle \in S$ □

4.6.3 Dependencies within Computational Elements

Value tuples do not encode dependencies between components in a computational element, as value tuples are ordinary pair values and cannot express dependencies between their components. We obtain the value tuple of a computational element with dependencies by removing the dependencies using substitution. For example, consider the following computational element, m_1, in which the component y_2 is dependent on the component y_1:

$$m_1 = \textbf{module } y_1 = e_1, y_2 = e_2(y_1) \textbf{ end},$$

The value tuple of m_1 is $\langle e_1, \langle e_2(y_1 \backslash e_1), tt\rangle\rangle$; here the dependency that e_2 has on y_1 is removed by substituting e_1, the value bound to y_1, for occurrences of y_2 in e_2. The rules governing the order in which dependencies are substituted enforce the scope of each component name within a computational component, and are discussed in section 5.6.1.

4.7 Modules in MLT

In this section, we consider modules in MLT. In Section 4.3, we outlined modules in MLT as terms that denote pairs whose left and right components are the translation of the computational element and witness, respectively. Section 4.3 also gave an example translation of the module *nm* satisfying specification *MS* of monoids. Let us generalise the outline given in Section 4.3. Modules in MLT are members of specifications in MLT, so they are terms in MLT that denote pairs. Consider the following module:

$$m = \textbf{module}\ y_1 = e_1, \ldots, y_n = e_n\ \textbf{proof}\ p\ \textbf{end} \in SP$$

We assume that m satisfies the specification *SP*:

$$SP = \textbf{Elements}\ S\ \textbf{Restrictions}\ R(y_1, \ldots, y_n),$$

The translation of m is given below with an explanation following.

$$[\![m]\!] = \langle [\![\textbf{module}\ y_1 = e_1, \ldots, y_n = e_n\ \textbf{end}]\!], p\rangle \in [\![SP]\!]$$

where

$$\begin{array}{lcl} \mathit{fst}([\![m]\!]) & \in & \mathit{Sig}([\![SP]\!]) \\ \mathit{snd}([\![m]\!]) & \in & \mathit{Ax}([\![SP]\!])(\mathit{fst}([\![m]\!]) \end{array}$$

The left component of $[\![m]\!]$ is the translation of the computational element containing the components in m, and the right component is the translation of the witness of m. The witness of a module is already a term in MLT, and is syntactically identical to its translation.

4.7.1 Some Module Notation

We now define some notation for modules in MLT.

Definition 4.9 (Specification Satisfaction)
Let m be a module, *SP* **spec**. m is said to satisfy (or implement) *SP* iff $m \in SP$.
□

We define two projection functions, *ce* and *pf*, which return the computational element and witness, respectively, of a module. The definitions of *ce* and *pf* are given by Definition 4.10. Note the use of the $\prod$-type constructor to give the type of *pf*: *pf* is a dependent function since the type of a witness is dependent on the computational element of a module. Definition 4.10 is worth remembering as it is often used in proofs concerning specifications.

Definition 4.10 (*ce* and *pf*)
Given *SP* **spec** and $m \in SP$ then functions *ce* and *pf* are defined as follows:

$$\begin{array}{lcl} ce(m) & = & \mathit{fst}(m) \\ pf(m) & = & \mathit{snd}(m) \end{array}$$

where

$$\begin{array}{lcl} ce & \in & SP \to \mathit{Sig}(SP) \\ pf & \in & \prod m \in SP.(\mathit{Ax}(SP)(ce(m))) \end{array}$$

□

Fact 4.6 Given *SP* **spec** and $m \in SP$ then $m = \langle ce(m), pf(m)\rangle \in SP$ □

$PointSpec \equiv$
Elements
$Point \in U_1$,
$mkPoint \in \mathbb{N} \to \mathbb{N} \to Point$,
$X \in Point \to \mathbb{N}$,
$Y \in Point \to \mathbb{N}$
Restrictions
$\forall x, y \in \mathbb{N}.$
$X(mkPoint(x)(y)) =_{\mathbb{N}} x \quad \wedge$
$Y(mkPoint(x)(y)) =_{\mathbb{N}} y$
End

Figure 4.7: The Specification of Cartesian Points

$PointModule \equiv$
module
$Point = \mathbb{N} \times \mathbb{N}$,
$mkPoint = \lambda x, y \in \mathbb{N}.\langle x, y + 100\rangle$,
$X = \lambda p \in Point.\mathrm{fst}(p)$,
$Y = \lambda p \in Point.\mathrm{snd}(p) - 100$
proof
$\lambda x \in \mathbb{N}.\lambda y \in \mathbb{N}.\langle eq, eq\rangle$
end $\in PointSpec$

Figure 4.8: An Implementation for PointSpec

4.8 An Example Translation: PointSpec

In this section, we exemplify the use of the notations defined in this chapter by giving an example specification and module, and their corresponding translations in MLT. The specification *PointSpec*, given in Figure 4.7, specifies a module that contains a type to represent points on a plane. *PointSpec* also specifies an operation to make a *Point* value from a Cartesian representation, as well as operations to return the x- and y-coordinates for Cartesian equivalents of a *Point* value. The module *PointModule*, defined in Figure 4.8, gives one possible implementation for *PointSpec* (i.e $[\![PointModule]\!] \in [\![PointSpec]\!]$). *PointModule* is made by choosing to represent *Point* values as pairs of natural numbers.

Let us consider the translations of *PointSpec* and *PointModule*. The translation of *PointSpec* (written as $[\![PointSpec]\!]$) is the dependent type:

$$\sum m \in Sig([\![PointSpec]\!]).Ax([\![PointSpec]\!])(m)$$

Here, the translations of the signature and restriction of *PointSpec* are elaborated by the following equations:

$$\begin{aligned}
Sig(\llbracket PointSpec \rrbracket) &= \left\{ \begin{array}{l} [inl(\text{`}Point\text{'}), \\ inl(\text{`}mkPoint\text{'}), \\ inl(\text{`}X\text{'}), \\ inl(\text{`}Y\text{'})] \end{array} \right\}_{\mathbb{D}} \times \left(\begin{array}{l} \sum x_1 \in U_1. \\ \sum x_2 \in \mathbb{N} \to \mathbb{N} \to x_1. \\ \sum x_3 \in x_1 \to \mathbb{N}. \\ \sum x_4 \in x_1 \to \mathbb{N}.T \end{array} \right) \\
Ax(\llbracket PointSpec \rrbracket)(m) &= \begin{array}[t]{l} \forall x, y \in \mathbb{N}. \\ m.X(m.mkPoint(x)(y)) =_{\mathbb{N}} x \quad \wedge \\ m.Y(m.mkPoint(x)(y)) =_{\mathbb{N}} y \end{array}
\end{aligned}$$

The translation of *PointModule* (written as $\llbracket PointModule \rrbracket$) and is the pair:

$$\langle ce(\llbracket PointModule \rrbracket), pf(\llbracket PointModule \rrbracket)\rangle \in \llbracket PointSpec \rrbracket$$

here

$$\begin{aligned}
ce(\llbracket PointModule \rrbracket) &= \left(\begin{array}{l} [inl(\text{`}Point\text{'}), \\ inl(\text{`}mkPoint\text{'}), \\ inl(\text{`}X\text{'}), \\ inl(\text{`}Y\text{'})] \end{array}, \begin{array}{l} \langle \mathbb{N} \times \mathbb{N}, \\ \langle \lambda x, y \in \mathbb{N}.\langle x, y + 100\rangle, \\ \langle \lambda p \in \mathbb{N} \times \mathbb{N}.\text{fst}(p), \\ \langle \lambda p \in \mathbb{N} \times \mathbb{N}.\text{snd}(p) - 100, \\ tt\rangle\rangle\rangle\rangle \end{array} \right) \\
pf(\llbracket PointModule \rrbracket) &= \lambda x \in \mathbb{N}.\lambda y \in \mathbb{N}.\langle eq, eq\rangle
\end{aligned}$$

4.9 Type and Specification Equality

One of the problems with using types as propositions is that intuitionistic logic does not always obey the properties of classical logic; for example, intuitionistic logic does not obey the law of the excluded middle (i.e. $P \vee \neg P$ is not always true). More importantly for us, properties such as $\wedge$-associativity and $\wedge$-commutativity do not hold with respect to type equality. Consequently, a specification of the form $\langle S \mid (m) P \wedge Q\rangle$ may not be equal to $\langle S \mid (m) Q \wedge P\rangle$ since $P \wedge Q = Q \wedge P$ may not be true; note that here we are using the shorthand notation for specifications, which we defined immediately after Definition 4.5. However, we would like the specifications $\langle S \mid (m) P \wedge Q\rangle$ and $\langle S \mid (m) Q \wedge P\rangle$ to be defined as equivalent since they admit the same implementations (disregarding witnesses). In this section, we define a specification equality, based on the notion of weak type equality, which admits many equivalences—such as commutativity—not allowed by type equality.

4.9.1 Weak Type Equality

In Martin-Löf's type theory—and other type theories—the type equality given by the theory is not always suitable for comparing propositions. It is more usual to use weak type equality as an equality test between propositions. Informally, two types P and Q are said to be weakly equivalent, written $P \Leftrightarrow Q$, if we can always generate a member of Q from a member of P, and vice versa. We use the following definition of weak type equality:

Definition 4.11 (Weak Type Equality $\Leftrightarrow$)
Given two types P and Q, then $P \Leftrightarrow Q$ iff there exists some functions $f \in P \rightarrow Q$ and $g \in Q \rightarrow P$. $\Box$

Weak type equality is an equivalence: it is reflexive, symmetric and transitive. We do not give all the properties of weak type equality as they are reasonably well described in the literature; it admits many of the equalities of classical logic, but not the law of the excluded middle.

Fact 4.7 $P = Q$ implies $P \Leftrightarrow Q$ $\Box$

Fact 4.8 $P \wedge Q \Leftrightarrow Q \wedge P$ $\Box$

Fact 4.9 $P \wedge (Q \wedge R) \Leftrightarrow (P \wedge Q) \wedge R$ $\Box$

Fact 4.10 $P \wedge P \Leftrightarrow P$ $\Box$

In the following, when we say two restrictions, or propositions, are equivalent, we mean equivalent with respect to weak type equality; for example, P is equivalent to Q means $P \Leftrightarrow Q$.

4.9.2 Specification Equality

Weak type equality to used to define a practical definition of equality for specifications. We say that two specifications SP_1 and SP_2 are equivalent, written $SP_1 \Leftrightarrow SP_2$, whenever the signatures of SP_1 and SP_2 are equal and their restrictions are equivalent. Note that we overload the symbol $\Leftrightarrow$ by using it to denote specification equality and weak type equality. Whenever the arguments of $\Leftrightarrow$ are specifications, it denotes specification equality; and if any of its arguments are not specifications then it denotes weak type equality.

Definition 4.12 (Specification Equality)
Given SP_1 **spec** and SP_2 **spec**, then $SP_1 \Leftrightarrow SP_2$ iff:

$$Sig(SP_1) = Sig(SP_2) \ \mathbf{type} \ ; \text{ and}$$

$$\forall m \in Sig(SP_1).\ Ax(SP_1)(m) \Leftrightarrow Ax(SP_2)(m)$$

$\Box$

Specification equality is an equivalence (Facts 4.11–4.13) and gives us equalities that do not hold under type equality, such as Facts 4.14, 4.15 and 4.16 (justified by Facts 4.8, 4.9, and 4.10, respectively).

Fact 4.11 $SP \Leftrightarrow SP$ □

Fact 4.12 $SP_1 \Leftrightarrow SP_2$ implies $SP_2 \Leftrightarrow SP_1$ □

Fact 4.13 $SP_1 \Leftrightarrow SP_2$ and $SP_2 \Leftrightarrow SP_3$ implies $SP_1 \Leftrightarrow SP_3$ □

Fact 4.14 $\langle S \mid P \wedge Q\rangle \Leftrightarrow \langle S \mid Q \wedge P\rangle$ □

Fact 4.15 $\langle S \mid P \wedge (Q \wedge R)\rangle \Leftrightarrow \langle S \mid (P \wedge Q) \wedge R\rangle$ □

Fact 4.16 $\langle S \mid P \wedge P\rangle \Leftrightarrow \langle S \mid P\rangle$ □

In the remainder of this thesis, when we say two specifications are equivalent, we mean equivalent with respect to specification equality ($\Leftrightarrow$); for example, SP_1 is equivalent to SP_2 means $SP_1 \Leftrightarrow SP_2$.

4.10 Some Alternative Semantics

The types and notations presented in this chapter are not the only way of defining the semantics of specifications and modules in type theory. Our work on the semantics of specifications has led us to contemplate several alternative definitions for specifications in MLT. In this section, we consider some of these alternatives.

4.10.1 An Alternative Naming Strategy

We could have chosen alternative ways of defining the semantics of component names within signatures. One option considered was to pair each type in a loose signature with a singleton type containing the name of a component. For example, the translation of a signature of a monoid specification would then be:

$$\begin{array}{l}\sum x \in (\{\mathit{inl}(\text{`}G\text{'})\}_{\mathbb{D}} \times U_1).\\ \sum y \in (\{\mathit{inl}(\text{`}\oplus\text{'})\}_{\mathbb{D}} \times x \to x \to x).\\ \sum z \in (\{\mathit{inl}(\text{`}\mathit{id}\text{'})\}_{\mathbb{D}} \times x).T\end{array}$$

The values in such types are tuples in which each component is paired with a component name. For example, the following is a value of the above type:

$$\langle\langle \mathit{inl}(\text{`}G\text{'}), \mathit{List}(\mathbb{N})\rangle, \langle \mathit{inl}(\text{`}\oplus\text{'}), +\!\!+\rangle, \langle \mathit{inl}(\text{`}\mathit{id}\text{'}), \mathit{nil}\rangle, \mathit{tt}\rangle$$

The alternative semantics, above, are isomorphic to the original semantics of signatures and computational elements. However, the alternative semantics are harder to manipulate since component values are interleaved with their names. In particular, the alternative semantics require the use of projection functions to define operations on the name-space of modules. In contrast, using the original semantics we can define operations on the name-space using list operations. As we shall see later, reasoning about projection functions requires extra work to ensure type correctness. However, the advantages of using the original semantics are limited to operations on the name-space, as both the original and alternative semantics require the use of projection functions to define operations over the component values in a module.

4.10.2 An Alternative Semantics for Specifications

It is possible to define translations for specifications so that modules satisfying specifications do not contain witnesses; in other words, so that modules are just computational elements. Such translations can be defined using the subset type constructor. For example, given a specification $SP_1 = \sum m \in S.R(m)$, we can define the type containing just the computational elements satisfying SP_1 as $SP_2 = \{m \in S|R(m)\}$. The member of the subset type SP_2 are all the computational elements $m \in S$ such that the proposition $R(m)$ is true.

The problem with using the subset type is that the subset elimination rule does not allow us to assume that a member of SP_2 satisfies $R(m)$. For example, the only property we can deduce about the members of SP_2 is that they are members of S. The problem generalises to not being allowed to assume that $Q(x)$ is true under the assumption that $x \in \{y \in P|Q(y)\}$. The problem is well-known in type theory and stems from the fact that the members of a subset type do not contain witnesses that prove that they satisfy the restrictions of the subset type. Several solutions have been proposed, though none are totally satisfactory for our purposes. We do not discuss the issue further and refer the interested reader to [46, 44].

4.11 Conclusion and Summary

Although Nordström et al [43, 44] have shown that modules can be specified in MLT, they do not give a formal treatment to component names within specifications and modules. The semantics of our specifications and modules include a definition of domains in MLT, so component names have a formal semantics. These semantics are certainly complicated—in comparison to the NPS and deliverables approaches—by the inclusion of component names. But this disadvantage is out-weighed by the importance of component names to the development of a usable specification and implementation language. After all, names provide a means by which one specification, or module, can use the operations of another. As we will see in Chapter 6, component names also play an important role in the design of useful specification and module operators.

One consequence of our semantics for specifications is that if a specification contains local components, then all modules satisfying it contain implementations of the local components. We justify the implementation of local components on the grounds that they are needed to calculate the types of other components in a module; these types are needed to prove a module satisfies its specification. We illustrate this point by considering the specification SP, and the module $m \in SP$, both given in Figure 4.9; for simplicity, we assume there are no dependencies between the components in m. Note that the type of y_3, in SP, is dependent on the local component y_2. Therefore, we need e_2, the implementation of y_2 in m, to calculate the type of e_3 ($e_3 \in A_3(e_2)$); we need to prove $e_3 \in A_3(e_2)$ to prove that $m \in SP$. The restriction of SP is also dependent on y_2, so we also need e_2 to prove p is a witness for m, i.e. to prove $p \in R(e_1, e_2, e_3)$. We need to prove $p \in R(e_1, e_2, e_3)$ to prove $m \in SP$.

$$
\begin{array}{ll@{\qquad\qquad}ll}
SP = & \textbf{Elements} & m = & \textbf{module} \\
 & \quad y_1 \in A_1, & & \quad y_1 = e_1, \\
 & \quad \bullet y_2 \in A_2, & & \quad \bullet y_2 = e_2, \\
 & \quad y_3 \in A_3(y_2) & & \quad y_3 = e_3 \\
 & \textbf{Restrictions} & & \textbf{proof} \\
 & \quad R(y_1, y_2, y_3) & & \quad p \\
 & \textbf{End} & & \textbf{end} \in SP
\end{array}
$$

Figure 4.9: A Specification SP and its implementation $m \in SP$

There are some disadvantages to defining specifications and modules in a constructive type theory. Firstly, modules contain witnesses that have no use as programs: witnesses are only needed to give constructive proofs about the properties of modules. Secondly, as noted in section 4.9, constructive logics do not obey all the properties of classical logics. This latter point occasionally makes it harder to reason about specifications presented in type theory, compared to those presented in specification languages, such as Z and CLEAR, which use classical logic.

There are compensating advantages to defining specifications and modules in type theory. Firstly, type theory provides a single framework for constructing specifications and modules. Secondly, the formal language of type theory can be used as a programming logic for reasoning about specifications and modules. Thirdly, the membership relation ($\in$) provides a simple definition of implementation: a module m satisfies a specification SP if $m \in SP$.

In summary, we have proposed a definition of specifications and modules in MLT, and defined some basic operations to manipulate them. A specification is a type, and the elements of this type are modules satisfying the specification.

Chapter 5

The Translation Mapping

5.1 Introduction

Although we discussed the translation of specifications and modules in Chapter 4, and defined the terms used as their translations, we did not define a mapping from specifications and modules to their translations in MLT. We need such a mapping so that we have a formal method of translating specifications and modules correctly when defining and proving laws about them. In this chapter, we define a formal mapping, called the *translation* mapping, from specifications and modules to their translations in MLT.

Most of the technical details concerning the definition of the translation mapping are easily inferred from the definitions of specifications and modules in MLT, given in Chapter 4. The definition of the translation mapping is slightly complicated over signatures and computational elements, as it must factor signatures into their domains and loose signatures; and computational elements into their domains and value tuples.

We will proceed by giving grammars for the syntax of specifications, modules, signatures and computational elements. Then we define the translation mapping by structural induction on the terms of each grammar. Not all terms defined by the grammars are well-typed, as grammars are only syntax rules, so we will give some laws that use the translation mapping to determine if a term is well-typed.

5.2 The Syntax of Specifications and Modules

Figure 5.1 gives the formal syntax for specifications and modules, with an explanation following. We let the tokens S, Sl, b and e stand for terms satisfying the grammar *sig*, *siglist*, *bind* and *Exp* respectively; R and A are tokens for terms satisfying *Type*;

Specifications

$spec ::= \textbf{Elements}\ S\ \textbf{Restrictions}\ R$

Signatures

$sig \quad ::= \quad \Phi \mid Sl$

$siglist \quad ::= \quad y \in A \mid \bullet y \in A \mid y \in A, Sl \mid \bullet y \in A, Sl$

Modules

$mod ::= \textbf{module}\ \ \textbf{proof}\ e\ \textbf{end} \mid \textbf{module}\ b\ \textbf{proof}\ e\ \textbf{end}$

Computational Elements

$comp \quad ::= \quad \textbf{module}\ \ \textbf{end} \mid \textbf{module}\ \mathrm{b}\ \textbf{end}$

$bind \quad ::= \quad y = e \mid \bullet y = e \mid y = e, b \mid \bullet y = e, b$

Types

$Type ::= \mathbb{N} \mid \mathbb{B} \mid \mathbb{S} \mid \sum x \in P.Q(x) \mid \prod x \in P.Q(x) \mid P + Q \mid List(P) \mid \ldots$
$\mid spec \mid sig$

Expressions

$Exp ::= y \mid `y\text{'} \mid \lambda e.e \mid e_1(e_2) \mid e_1.e_2 \mid \langle e_1, e_2 \rangle \mid fst(e) \mid snd(e) \mid \ldots$
$\mid Type \mid mod \mid comp$

Primitive classes

Exp_0 - Terms of MLT.

Ident - Identifiers (i.e. Sequences of characters)

Figure 5.1: A Summary of Syntax

x and y are token for terms satisfying *Ident*. The grammar *Type* admits terms that denote types in specifications and modules, and includes types used as restrictions in specifications. Note that *Type* admits specifications and signatures as they are also types; for example, specifications are used as types when defining nested specifications. The grammar *Exp* admits all terms denoting values that can appear in specifications and modules; these include all terms in MLT, including types, as well as specifications, modules , signatures and computational elements. For brevity, we omit to define most of the terms that *Type* and *Exp* inherit from MLT, but the omitted terms can be inferred from the formation, introduction and elimination rules of MLT. Note that we assume the existence of a grammar Exp_0 defining only terms in MLT.

Let F and G be any grammars. We write $e : E$ to mean that e is a term satisfying grammar E, and write $f : E \dashrightarrow F$ to mean that f is a mapping from terms satisfying E to terms satisfying F. Mappings, such as f, are rewrite rules on syntactic objects, and should not be confused with functions in MLT.

$$\begin{array}{lcl}
[\![y]\!] & = & y \\
[\![\text{`}y\text{'}]\!] & = & \text{`}[\![y]\!]\text{'} \\
[\![\lambda x \in (P).e]\!] & = & \lambda[\![x]\!] \in [\![P]\!].[\![e]\!] \\
[\![e_1(e_2)]\!] & = & [\![e_1]\!]([\![e_2]\!]) \\
[\![e_1.e_2]\!] & = & [\![e_1]\!].[\![e_2]\!] \\
[\![\langle e_1, e_2\rangle]\!] & = & \langle[\![e_1]\!], [\![e_2]\!]\rangle \\
 & \vdots & \\
[\![\mathbb{N}]\!] & = & \mathbb{N} \\
[\![\Pi x \in P.Q(x)]\!] & = & \Pi[\![x]\!] \in [\![P]\!].[\![Q(x)]\!] \\
[\![\Sigma x \in P.Q(x)]\!] & = & \Sigma[\![x]\!] \in [\![P]\!].[\![Q(x)]\!] \\
[\![P+Q]\!] & = & [\![P]\!] + [\![Q]\!] \\
[\![\text{List}(P)]\!] & = & \text{List}([\![P]\!])
\end{array}$$

Figure 5.2: The Translation of Standard terms of MLT

5.3 The Translation Mapping

In the following, we will define a mapping $[\![\_]\!] : \mathit{Exp} \rightarrow \mathit{Exp}_0$, called the translation mapping. $[\![\_]\!]$ maps specifications, signatures, modules and computational elements—which are all terms in *Exp*—to their translations in MLT. But $[\![\_]\!]$ also map the other terms in *Exp* to MLT as they may use specifications and modules as subterms; for example, *Exp* admits terms denoting functions that use modules in their body. In other words, $[\![\_]\!]$ maps all $e : \mathit{Exp}$ to terms in MLT by translating any specifications, signatures, modules and computational elements that appear in e.

The definition of $[\![\_]\!]$ is by structural induction on terms satisfying *Exp*. Figure 5.2 gives the definition of $[\![\_]\!]$ on some of the terms of *Exp* inherited from MLT; $[\![\_]\!]$ is defined in a similar manner on the remaining terms inherited from MLT, and the definitions are omitted for brevity. Note that $[\![\_]\!]$ satisfies $\forall e : \mathit{Exp}_0.[\![e]\!] = e$, i.e., $[\![\_]\!]$ is an identity mapping when restricted to terms in MLT. The definition of $[\![\_]\!]$ on terms satisfying *spec*, *sig*, *mod* and *comp* is slightly more complicated, and is the subject of the remainder of this chapter.

5.4 Translating Signatures

In this section, we extend the translation mapping, $[\![\_]\!]$, over signatures. The extension is by structural induction on the terms of the grammar *sig*. Recall that the translation of a signature is the product of its domain and loose signature. Therefore, we will define two mappings, $[\![\_]\!]_{Lsg}$ and $[\![\_]\!]_{Dom}$, which translate signatures into their loose signature and domain, respectively; these are used to extend $[\![\_]\!]$ over signatures. The main complication in the definitions of $[\![\_]\!]_{Lsg}$ and $[\![\_]\!]_{Dom}$ is ensuring they map component identifiers in signatures to bound variables and names, respectively. Hence, we begin by defining the translation of component identifiers.

5.4.1 Translating Component Identifiers

In Section 4.4, we noted that the component identifiers in signatures play a dual role as both component names and bound variables. To distinguish between the two roles of a component identifier, we define two mappings $[\![\_]\!]_{id}$ and $[\![\_]\!]_{var}$ which translate a component identifier—y, say—into a component name 'y' $\in \mathbb{S}$ and a variable y, respectively.

Definition 5.1 (**$[\![\_]\!]_{id}$ and $[\![\_]\!]_{var}$**)
Given y : *ident*, the mappings $[\![\_]\!]_{id}$: *Ident* $\rightarrow$ *Exp* and $[\![\_]\!]_{var}$: *Ident* $\rightarrow$ *Exp* are defined as follows:

$$\begin{array}{lll} [\![y]\!]_{id} & = & \text{`}y\text{'} \text{ ; and} \\ [\![y]\!]_{var} & = & y_{var} \end{array}$$

□

Strictly speaking, we do not need $[\![\_]\!]_{var}$ as it is identical to the clause defining the translation mapping over identifiers (i.e. $[\![y]\!] = y$). However, using $[\![\_]\!]_{var}$ emphasizes the translation of an identifier to a variable.

5.4.2 Translating Loose Signatures

Definition 5.2 gives a mapping, $[\![\_]\!]_{Lsg}$, that translates signatures to their loose signature in MLT. For example, given the signature $S = y_1 \in A_1, \ldots, y_n \in A_n$, then:

$$[\![S]\!]_{Lsg} = \sum [\![y_1]\!]_{var} \in [\![A_1]\!]. \sum [\![y_2]\!]_{var} \in [\![A_2]\!] \ldots \sum [\![y_n]\!]_{var} \in [\![A_n]\!].T$$

Note that $[\![\_]\!]_{var}$ translates each component identifiers y_i $(1 \leq i \leq n)$ to a bound variable in $[\![S]\!]_{Lsg}$. The translation mapping maps each type A_j $(1 \leq j \leq n)$ onto MLT; in the process, each occurrence of y_i in A_j becomes $[\![y_i]\!]$, and is bound to the corresponding variable declaration $[\![y_i]\!]_{var} \in [\![A_i]\!]$.

Definition 5.2 (**$[\![\_]\!]_{Lsg}$**)
The mapping $[\![\_]\!]_{Lsg}$: *sig* $\rightarrow$ *Type* is defined inductively, as follows:

$$\begin{array}{lll} [\![\Phi]\!]_{Lsg} & = & T \text{ ;} \\ [\![y \in A]\!]_{Lsg} & = & \sum [\![y]\!]_{var} \in [\![A]\!].T \text{ ;} \\ [\![\bullet y \in A]\!]_{Lsg} & = & \sum [\![y]\!]_{var} \in [\![A]\!].T \text{ ;} \\ [\![y \in A, S]\!]_{Lsg} & = & \sum [\![y]\!]_{var} \in [\![A]\!].[\![S]\!]_{Lsg} \text{ ; and} \\ [\![\bullet y \in A, S]\!]_{Lsg} & = & \sum [\![y]\!]_{var} \in [\![A]\!].[\![S]\!]_{Lsg} \end{array}$$

□

5.4.3 Translating Domains

We define a mapping, $[\![\_]\!]_{Dom}$, that translates a signature to its domain, as a type, in MLT. For example, given a signature $S = y_1 \in A_1, \ldots, y_n \in A_n$ then:

$$[\![S]\!]_{Dom} = \{[inl([\![y_1]\!]_{id}), \ldots, inl([\![y_n]\!]_{id})]\}_{\mathbb{D}}$$

Note that $[\![\_]\!]_{id}$ translates component identifiers into component names with type $\mathbb{S}$—contrast this with the use of $[\![\_]\!]_{var}$ in the definition of $[\![\_]\!]_{Lsg}$, in which component identifiers are translated into variables. $[\![\_]\!]_{Dom}$ is defined in terms of an auxiliary mapping, $[\![\_]\!]_{dom}$, which translates a signature to a domain, as a list: $[\![\_]\!]_{Dom}$ takes a signature—S, say—and "lifts" $[\![S]\!]_{dom}$ to the domain of S, as a type.

Definition 5.3 ($[\![\_]\!]_{dom}$)
The mapping $[\![\_]\!]_{dom} : sig \to Exp$ is defined as follows:

$$\begin{array}{lcl}
[\![\Phi]\!]_{dom} & = & nil\ ; \\
[\![y \in A]\!]_{dom} & = & [inl([\![y]\!]_{id})]\ ; \\
[\![\bullet y \in A]\!]_{dom} & = & [inr([\![y]\!]_{id})]\ ; \\
[\![y \in A, S]\!]_{dom} & = & [inl([\![y]\!]_{id})] +\!\!+ [\![S]\!]_{dom}\ ; \\
[\![\bullet y \in A, S]\!]_{dom} & = & [inr([\![y]\!]_{id})] +\!\!+ [\![S]\!]_{dom}\ .
\end{array}$$

□

Definition 5.4 ($[\![\_]\!]_{Dom}$)
Given any $S : sig$, the mapping $[\![\_]\!]_{Dom} : sig \to Type$ is defined as follows:

$$[\![S]\!]_{Dom} = \{[\![S]\!]_{dom}\}_{\mathbb{D}}$$

□

5.4.4 The Translation of Signatures

We use the mappings $[\![\_]\!]_{Lsg}$ and $[\![\_]\!]_{Dom}$ to extend the translation mapping so that it maps signatures to their translations in MLT; recall that the translation of a signature is the product of its domain and loose signature. The extension of the translation mapping over signatures is given by Definition 5.5.

Definition 5.5 (Signature types)
The translation mapping is extended over all $S : sig$, as follows:

$$[\![S]\!] = [\![S]\!]_{Dom} \times [\![S]\!]_{Lsg}$$

□

We will often write $[\![S]\!]_{sig}$ in place of $[\![S]\!]$ to denote the application of the translation mapping to any $S : sig$ ($[\![S]\!]_{sig} \equiv [\![S]\!]$).

5.4.5 Type-Checking Signatures

The grammar *sig* gives the syntax rules for signatures, but it does not guarantee that terms satisfying *sig* are well-typed. For example, the term $x \in \mathbb{N}, y \in \{x\}_{\mathbb{B}}$ is syntactically correct with respect to *sig*, but is not well-typed: if $x \in \mathbb{N}$ then $\{x\}_{\mathbb{B}}$ is ill-typed since x cannot be a member of both $\mathbb{B}$ and $\mathbb{N}$. If we use a term $S : sig$ as a type then it must be well-typed; this is checked by using the judgement **sig** to ensure that $[\![S]\!]$ **sig** holds.

In general, determining if $[\![S]\!]$ **sig** holds can be broken down into a collection of convenient rules given by the following theorems:

Theorem 5.1 $[\![\Phi]\!]$ **sig** $\square$

Theorem 5.2 if $[\![A]\!]$ **type** then $[\![y \in A]\!]$ **sig** $\square$

Theorem 5.3 if $[\![A]\!]$ **type** then $[\![\bullet y \in A]\!]$ **sig** $\square$

Theorem 5.4 if $[\![A]\!]$ **type** and $|[[\![y]\!]_{var} \in [\![A]\!] \rhd [\![S]\!]\ \mathbf{sig}]|$ then $[\![y \in A, S]\!]$ **sig** $\square$

Theorem 5.5 if $[\![A]\!]$ **type** and $|[[\![y]\!]_{var} \in [\![A]\!] \rhd [\![S]\!]\ \mathbf{sig}]|$ then $[\![\bullet y \in A, S]\!]$ **sig** $\square$

Proof (of Theorems 5.1—5.5)
Each proof follows from the definition of the mapping $[\![\_]\!]$ over signatures, and the definition of the judgement **sig**. $\square$

5.5 Translating Specifications

In this section, we extend the translation mapping so that it maps specifications into their translations in MLT. Recall, from Section 4.5, that the translation of a specification SP,

$$SP = \textbf{Elements } y_1 \in A_1, \ldots, y_n \in A_n \textbf{ Restrictions } R(y_1, \ldots, y_n),$$

is denoted by $[\![SP]\!]$:

$$[\![SP]\!] = \textstyle\sum x \in [\![y_1 \in A_1, \ldots, y_n \in A_n]\!]_{sig}.[\![R(x.y_1, \ldots, x.y_n)]\!]$$

The definition of $[\![SP]\!]$, given above, virtually defines the translation mapping over specifications. All that is missing is a definition of a mapping that translates restrictions of the form $R(y_1, \ldots, y_n)$ to $R(m.y_1, \ldots, m.y_n)$. In the following, we define such a mapping, which we call the *open* mapping, and use it to give the formal definition of the translation mapping over terms satisfying the grammar *spec*.

5.5.1 The *Open* Mapping

Definition 5.6 defines the **open** mapping with explanation following. Given a term satisfying *sig*, such as $S = y_1 \in A_1, \ldots, y_n \in A_n$, and a term satisfying *Exp*, such as $R(y_1, \ldots, y_n)$, then:

$$\textbf{open } S \textbf{ in } R(y_1, \ldots, y_n) \equiv [m]R(m.y_1, \ldots, m.y_n).$$

Intuitively, **open** S **in** $R(y_1, \ldots, y_n)$ returns the term obtained by substituting each component identifier y_i $(1 \leq i \leq n)$ in $R(y_1, \ldots, y_n)$ by $m.y_i$. The prefix $[m]$ indicates that $R(m.y_1, \ldots, m.y_n)$ is dependent on $m \in S$. The **open** transformation is analogous to the Pascal **with**-statement, as it allows the components of a structure $m \in S$ to be used in terms, such as $R(y_1, \ldots, y_n)$, without the explicit use of dot notation.

Definition 5.6 (Open)
The **open** mapping, **open_in_** : $sig \rightarrow Exp \rightarrow Exp$, is defined recursively:

$$\begin{array}{lcl}
\textbf{open } \Phi \textbf{ in } e & \equiv & [m]e \\
\textbf{open } y \in A \textbf{ in } e & \equiv & [m]e(y\backslash m.[y]) \\
\textbf{open } \bullet y \in A \textbf{ in } e & \equiv & [m]e(y\backslash m.[y]) \\
\textbf{open } y \in A, S \textbf{ in } e & \equiv & \textbf{open } S \textbf{ in } e(y\backslash m.[y]) \\
\textbf{open } \bullet y \in A, S \textbf{ in } e & \equiv & \textbf{open } S \textbf{ in } e(y\backslash m.[y])
\end{array}$$

□

Unlike other mappings defined in this chapter, **open** does not always return terms in MLT; in particular, if e is not a term in MLT then neither is **open** S **in** e. However, we can translate **open** S **in** e to MLT by applying the translation mapping ($[\![\_]\!]$).

5.5.2 Translating Specifications

We use the **open** mapping to extend the translation mapping so that it maps specifications into their translations in MLT.

Definition 5.7 (Extending $[\![\_]\!]$ over specifications)
The translation mapping is extended over terms with syntax *spec*, as follows:

$$[\![\textbf{Elements } S \textbf{ Restrictions } R]\!] = \textstyle\sum x \in [\![S]\!]_{sig}.[\![\textbf{open } S \textbf{ in } R]\!](x)$$

□

Note that we use the translation mapping to translate **open** S **in** R into MLT. From the definition of **open**, we know that **open** S **in** R contains a free variable $m \in [\![S]\!]_{sig}$. We bind m to $x \in [\![S]\!]_{sig}$ by applying $[\![\textbf{open } S \textbf{ in } R]\!]$ to x; hence, the restriction $[\![\textbf{open } S \textbf{ in } R]\!](x)$ is dependent on $x \in [\![S]\!]_{sig}$.

5.5.3 Type-Checking Specifications

The grammar *spec* only gives the syntax rules for specifications, and does not guarantee type correctness. We check that a term—SP, say—satisfying the grammar *spec*, is well-typed by using the judgement **spec** defined previously in section 4.5: SP is well typed iff $[\![SP]\!]$ **spec**. Determining if $[\![SP]\!]$ **spec** holds can be broken down into two requirements. Firstly, the signature of SP must be well-typed. Secondly, the restriction of SP must be a type under the assumption that SP's signature is well-typed. Theses requirements are stated formally by the following theorem:

Theorem 5.6 (A law about judgement spec)

$[\![$**Elements** S **Restrictions** $R]\!]$ **spec**

iff

$[\![S]\!]_{sig}$ **sig**; and
$|[m \in [\![S]\!]_{sig} \triangleright [\![\textbf{open } S \textbf{ in } R]\!](m) \textbf{ type}]|$.

□

Proof We omit the proof, but it follows immediately from the translation mapping and the definition of **spec**. □

5.6 Translating Computational Elements

In this section, we extend the translation mapping so that it maps computational elements to their translations in MLT. Recall, from Section 4.6, that the translation of a computational element is a pair containing its domain and value tuple. Therefore, we proceed by defining two mappings $[\![\_]\!]_{val}$ and $[\![\_]\!]_{dom}$ which translate computational elements into their value tuple and domain, respectively; these mapping are then used to extend the translation mapping by structural induction on the terms of the grammar *comp* for computational elements. We will also use the extension to $[\![\_]\!]$ to give some typing rules for computational elements.

5.6.1 Translating Value Tuples

Definition 5.8 gives a mapping, $[\![\_]\!]_{val}$, that translates computational elements to their value tuple in MLT. For example, given the computational element:

$m = \textbf{module } y_1 = e_1, \ldots, y_n = e_n \textbf{ end}$

then:

$[\![m]\!]_{val} = \langle [\![e_1]\!], \langle [\![e_2(y_1 \backslash e_1)]\!], \langle \ldots, \langle [\![e_n(y_1 \backslash e_1, \ldots, y_{n-1} \backslash e_{n-1})]\!], tt \rangle \ldots \rangle \rangle \rangle$

Note that the translation mapping ($[\![\_]\!]$) is used to translate each term e_i ($1 \leq i \leq n$) into MLT. The substitutions on each e_i remove any dependencies between components in a computational element; this is necessary as the pair constructors used to construct value tuples cannot express dependencies between components. We assume that substitutions of the form $b(y\backslash e)$ only substitute free occurrences of y, so that overloaded definitions of y in b, and occurrences of y bound to such definitions, are not substituted; this ensures that a component identifier is only substituted by the value it is bound to by the scope rules for computational elements.

Definition 5.8 ($[\![\_]\!]_{val}$)
The mapping $[\![\_]\!]_{val} : comp \to Exp$ is defined as follows:

$$\begin{array}{lcl} [\![\textbf{module end}]\!]_{val} & = & tt\ ; \\ [\![\textbf{module } y = e \textbf{ end}]\!]_{val} & = & \langle [\![e]\!], tt \rangle\ ; \\ [\![\textbf{module } \bullet y = e \textbf{ end}]\!]_{val} & = & \langle [\![e]\!], tt \rangle\ ; \\ [\![\textbf{module } y = e, b \textbf{ end}]\!]_{val} & = & \langle [\![e]\!], [\![\textbf{module } b(y\backslash e) \textbf{ end}]\!]_{val} \rangle\ ; \\ [\![\textbf{module } \bullet y = e, b \textbf{ end}]\!]_{val} & = & \langle [\![e]\!], [\![\textbf{module } b(y\backslash e) \textbf{ end}]\!]_{val} \rangle. \end{array}$$

□

5.6.2 Translating Domains

Definition 5.9 gives the mapping $[\![\_]\!]_{dom}$ that translates computational elements to their domains in MLT. For example, given the computational element m—defined in the previous section—then:

$$[\![m]\!]_{dom} = [inl([\![y_1]\!]_{id}), \ldots, inl([\![y_n]\!]_{id})]$$

Note the use of the mapping $[\![\_]\!]_{id}$ to translate each component identifier y_i ($1 \leq in \leq$) into a component name of type $\mathbb{S}$.

Definition 5.9 ($[\![\_]\!]_{dom}$)
The mapping $[\![\_]\!]_{dom} : comp \to Exp$ is defined as follows:

$$\begin{array}{lcl} [\![\textbf{module end}]\!]_{dom} & = & nil\ ; \\ [\![\textbf{module } y = e \textbf{ end}]\!]_{dom} & = & [inl([\![y]\!]_{id})]\ ; \\ [\![\textbf{module } \bullet y = e \textbf{ end}]\!]_{dom} & = & [inr([\![y]\!]_{id})]\ ; \\ [\![\textbf{module } y = e, b \textbf{ end}]\!]_{dom} & = & [inl([\![y]\!]_{id})] +\!\!+ [\![\textbf{module } b \textbf{ end}]\!]_{dom}\ ; \\ [\![\textbf{module } \bullet y = e, b \textbf{ end}]\!]_{dom} & = & [inr([\![y]\!]_{id})] +\!\!+ [\![\textbf{module } b \textbf{ end}]\!]_{dom}. \end{array}$$

□

5.6.3 Translating Computational Elements

We use the mappings $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{val}$ to extend the translation mapping so that it maps computational elements to their translations in MLT.

Definition 5.10 (Translating Computational Elements)
The translation mapping is defined over all m : *comp* as follows:

$$[\![m]\!] = \langle [\![m]\!]_{dom}, [\![m]\!]_{val} \rangle$$

□

Definition 5.10 is justified by recalling, from Section 4.6, that a computational element in MLT is a pair containing its domain and value tuple.

5.6.4 Type-Checking Computational Elements

The grammar *comp* only gives the syntax rules for computational elements, and does not guarantee type correctness. From the definition of well-typed computational elements (Definition 4.7), it follows that a term m : *comp* is well-typed with respect to a signature, $[\![S]\!]$ **sig**, if $[\![m]\!] \in [\![S]\!]$. A judgement of the form $[\![m]\!] \in [\![S]\!]$ can be made using theorem 5.7 which recognises the factoring of names and values in the translations of computational elements.

Theorem 5.7
Let $[\![S]\!]$ **sig** hold. $[\![m]\!] \in [\![S]\!]$ iff $[\![m]\!]_{dom} \in [\![S]\!]_{Dom}$ and $[\![m]\!]_{val} \in [\![S]\!]_{Lsg}$.
□

Proof Theorem 5.7 is justified by the $\times$-introduction rules of the type theory; the proof is trivial, and is omitted. □

We also define a collection of laws that allow a judgement $[\![m]\!] \in [\![S]\!]$ to be made in a piecewise manner. The laws are given by Theorems 5.8–5.12; informally, the laws can be viewed as introduction rules for signatures.

Theorem 5.8 $[\![$**module end**$]\!] \in [\![\Phi]\!]$ □

Theorem 5.9 if $[\![e]\!] \in [\![A]\!]$ then $[\![$**module** $y = e$ **end**$]\!] \in [\![y \in A]\!]$ □

Theorem 5.10 if $[\![e]\!] \in [\![A]\!]$ then $[\![$**module** $\bullet y = e$ **end**$]\!] \in [\![\bullet y \in A]\!]$ □

Theorem 5.11 if $[\![e]\!] \in [\![A]\!]$ and $[\![$**module** $b(y\backslash e)$ **end**$]\!] \in [\![S(y\backslash e)]\!]$ then $[\![$**module** $y = e, b$ **end**$]\!] \in [\![y \in A, S]\!]$ □

Theorem 5.12 if $[\![e]\!] \in [\![A]\!]$ and $[\![$**module** $b(y\backslash e)$ **end**$]\!] \in [\![S(y\backslash e)]\!]$ then $[\![$**module** $\bullet y = e, b$ **end**$]\!] \in [\![\bullet y \in A, S]\!]$ □

Proof (of Theorems 5.8–5.12)
In each case, the proof follows from the definition of the translation mapping and the inference rules *Struct1* and *Struct2* given previously in Figure 4.6. The proofs are trivial and are omitted. □

5.7 Translating Modules

In this section, we extend the translation mapping so that it maps modules to their translations in MLT. Recall, from Section 4.7, that the translation of a module,

$$m = \textbf{module } y_1 = e_1, \ldots, y_n = e_n \textbf{ proof } p \textbf{ end},$$

is denoted by $[\![m]\!]$:

$$[\![m]\!] = \langle [\![\textbf{module } y_1 = e_1, \ldots, y_n = e_n \textbf{ end}]\!], [\![p]\!] \rangle$$

The definition of $[\![m]\!]$ is essentially an informal definition of the translation mapping over modules. In the following, we give the formal definition of the translation mapping over modules by structural induction on the terms of the grammar *mod* for modules.

5.7.1 Translating Modules

Definition 5.11 gives the formal definition of the translation mapping over modules.

Definition 5.11 (Extending $[\![\_]\!]$ over modules)
The translation mapping is extended over terms satisfying *mod*, as follows:

$$\begin{array}{lcl} [\![\textbf{module proof } p \textbf{ end}]\!] & = & \langle [\![\textbf{module end}]\!], [\![p]\!] \rangle \, ; \\ [\![\textbf{module } b \textbf{ proof } p \textbf{ end}]\!] & = & \langle [\![\textbf{module } b \textbf{ end}]\!], [\![p]\!] \rangle \end{array}$$

□

Note that the translation mapping is used to translate both the computational element and witness of a module.

5.7.2 Type-Checking Modules

From the definition of well-typed modules (Definition 4.9), it follows that a term $m : mod$ is well-typed with respect to a specification $[\![SP]\!]$ **spec** if $[\![m]\!] \in [\![SP]\!]$. Theorems 5.13 and 5.14 are used to make judgements of the form $[\![m]\!] \in [\![SP]\!]$ by factoring them into two separate judgements.

Theorem 5.13
If $Sig([\![SP]\!]) = \Phi$ and $[\![p]\!] \in Ax([\![SP]\!])([\![\textbf{module end}]\!])$ then
$[\![\textbf{module proof } p \textbf{ end}]\!] \in [\![SP]\!]$
□

Theorem 5.14
If $[\![\textbf{module } \mathrm{b} \textbf{ end}]\!] \in Sig([\![SP]\!])$ and $[\![p]\!] \in Ax([\![SP]\!])([\![\textbf{module } \mathrm{b} \textbf{ end}]\!])$ then
$[\![\textbf{module } b \textbf{ proof } p \textbf{ end}]\!] \in [\![SP]\!]$
□

Proof (of Theorems 5.13 and 5.14)
The proof follows immediately from Definition 4.9, and is omitted.
□

5.8 Conclusion and Summary

Although the semantics of our specification language is presented in a denotational style, it is not a conventional denotational semantics. In conventional denotational semantics, such as [53], a language is mapped onto the Lambda calculus. However, the translation mapping maps specifications and modules onto Martin-Löf's type theory. Denotational semantics use the Lambda calculus because its properties are well-known and can be used to reason about languages expressed in terms of it. However, the Lambda calculus has its limitations, as many modern programming concepts, such as parallelism and specifications, are difficult to express in the Lambda calculus; for more details see [40].

In summary, we have presented a formal mapping from the syntax of specifications and modules to their translations in Martin-Löf's Type Theory. We have used the mapping to give some laws for type-checking specifications and modules in a piecewise manner.

Chapter 6

Signature and Computational Element Operators

6.1 Introduction

Canonical specifications and canonical modules are unsuited to making very large specifications and modules. For example, understanding a canonical specification is difficult if there are too many components in its signature, or too many axioms in its restriction. A more convenient way of making large specifications and modules is to construct them by combining smaller specifications and modules, respectively. Such a style encourages us to make specifications and modules incrementally, as well as helping us to decompose specifications and modules into more manageable pieces. We call specifications constructed from smaller specifications structured specifications; and we call modules constructed from smaller modules structured modules. In Chapter 7, we will define specification operators that combine and modify specifications to make structured specifications. In order to define such specification operators we will require operators that combine and modify signatures and computational elements. That is the subject of this chapter.

The signature operations we define are signature renaming, signature hiding and signature concatenation. Signature renaming renames components within signatures. Signature hiding hides visible components in a signature by making them local components. Signature concatenation combines two signatures to make a signature containing the components from both signatures. We also define renaming, hiding and concatenation operators on computational elements.

In order to define the signature concatenation operator—which we write as an infix binary operator ($\_ \oplus \_$)—we will introduce dependent signatures in MLT. Dependent signatures are signatures whose components depend on the components of other signatures. The signature concatenation operator requires dependent signatures because for any signature concatenation, $S_1 \oplus S_2$ for example, we allow S_2 to be dependent on the components supplied by S_1. We allow such dependencies since we often use $\oplus$ to enrich signatures with new components defined in terms of components from other signatures.

The resolution of name clashes is another important issue in the design of the signature concatenation operator. Name clashes can arise when we concatenate two signatures that have some component names in common. Name clashes must be resolved as otherwise they can cause ambiguities in signatures. The computational element concatenation operator must also resolve potential name clashes.

In order to avoid the proliferation of operator symbols, we will overload the symbol for each signature operator and also use it for the corresponding computational element operator; for example, we will use $\oplus$ to denote both the signature concatenation and computational element concatenation operators. It will always be clear from the context whether we are using such operator symbols to denote signature operators or computational element operators.

We will also give some laws about each operator defined in this chapter, and these will be used, later, to prove laws about specification and module operators. However, for reasons of space, we do not supply a proof for all the laws given in this chapter.

6.2 Dependent Signatures

We begin by defining dependent signatures. A dependent signature is a signature whose components are dependent on those of another signature. For example, S_2 (below) is dependent on S_1 as it is defined using Sqr in S_1.

$$\begin{aligned} S_1 &= Sqr \in U_1, size \in Sqr \to \mathbb{N} \\ S_2 &= area \in Sqr \to \mathbb{N}, size \in Sqr \to \mathbb{N} \end{aligned}$$

Strictly speaking, a dependent signature is not a signature, but an abstraction (i.e. an expression containing free component names) whose free component names can be instantiated to produce a signature.

6.2.1 The Dependent Signature Judgement

A dependent signature in MLT—dependent on some signature S, for example—is any expression of the form $\{l\}_{\mathbb{D}} \times Q$ where $\{l\}_{\mathbb{D}}$ is a domain, and Q is an abstraction that can be instantiated by the components in the loose-signature of S to produce a loose signature. Given any loose signature

$$Lsg(S) = \sum y_1 \in A_1 \ldots \sum y_n \in A_n.T,$$

Q can be instantiated as a loose signature if, and only if, the following holds:

$$\langle\!|[y_1 \in A_1, \ldots, y_n \in A_n \rhd Q(y_1)\ldots(y_n)\ \mathbf{lsg}]|\!\rangle \quad (1)$$

To formally define dependent signatures, we introduce the dependent loose-signature judgement. This has the form $P \vdash Q$ **lsg** for any P **lsg** and Q an abstraction. $P \vdash Q$ **lsg** holds if, and only if, Q is a loose-signature when instantiated by the components supplied by P. For example, condition (1) (above) is equivalent to $Lsg(S) \vdash Q$ **lsg**.

Definition 6.1 (Dependent Loose-signature)
Given $\sum x \in A.B(x)$ **lsg**, then

$$\begin{array}{lcl} T \vdash Q\ \mathbf{lsg} & = & Q\ \mathbf{lsg} \\ (\sum x \in A.B(x)) \vdash Q\ \mathbf{lsg} & = & \langle\!|[x \in A \rhd (B(x) \vdash Q(x)\ \mathbf{lsg})]|\!\rangle \end{array}$$

□

We give a formal definition of dependent signatures by defining the dependent signature judgement. This has the form $S_1 \vdash S_2$ **sig**; it holds if, and only if, S_2 is a dependent signature that is dependent on S_1.

Definition 6.2 (Dependent Signatures)
$S \vdash \{l\}_{\mathbb{D}} \times Q$ **sig** iff S **sig**, $l \in \mathbb{D}$ and $(Lsg(S) \vdash Q\ \mathbf{lsg})$.
□

Here are a few useful facts about the dependent signature judgement:

Fact 6.1 $[\![\Phi]\!] \vdash [\![S]\!]\ \mathbf{sig} = [\![S]\!]\ \mathbf{sig}$ □

Fact 6.2 $[\![y \in A]\!] \vdash [\![S]\!]\ \mathbf{sig} = \langle\!|[[\![y]\!] \in [\![A]\!] \rhd [\![S]\!]\ \mathbf{sig}]|\!\rangle$ □

Fact 6.3 $[\![\bullet y \in A]\!] \vdash [\![S]\!]\ \mathbf{sig} = \langle\!|[[\![y]\!] \in [\![A]\!] \rhd [\![S]\!]\ \mathbf{sig}]|\!\rangle$ □

Fact 6.4 $[\![y \in A, S_1]\!] \vdash [\![S]\!]\ \mathbf{sig} = \langle\!|[[\![y]\!] \in [\![A]\!] \rhd ([\![S_1]\!] \vdash [\![S]\!]\ \mathbf{sig})]|\!\rangle$ □

Fact 6.5 $[\![\bullet y \in A, S_1]\!] \vdash [\![S]\!]\ \mathbf{sig} = \langle\!|[[\![y]\!] \in [\![A]\!] \rhd ([\![S_1]\!] \vdash [\![S]\!]\ \mathbf{sig})]|\!\rangle$ □

6.2.2 Signature Instantiation

Let S_2 be a dependent signature that is dependent on S_1. Given any computational element $m \in S_1$, we can instantiate each free occurrence of a component name—p, for example—in S_2 by the corresponding component $m.p$ in m, so that S_2 becomes a non-dependent signature. We write such instantiations as $S_2(\!|m|\!)_{S_1}$ and call them signature instantiations. In this section, we give a formal definition of signature instantiation.

We give some examples of signature instantiation in Facts 6.6–6.10 below.

Fact 6.6 $S(\!|\mathbf{module\ \ end}|\!)_\Phi = S$ □

Fact 6.7 $S(\!|\mathbf{module}\ y = e\ \mathbf{end}|\!)_{y \in A} = S(e)$ □

Fact 6.8 $S(\!|\textbf{module} \bullet y = e \textbf{ end}|\!)_{\bullet y \in A} = S(e)$ □

Fact 6.9 $S(\!|\textbf{module } y = e, b \textbf{ end}|\!)_{y \in A, S_1} = S(e)(\!|\textbf{module } b(y \backslash e) \textbf{ end}|\!)_{S_1(y\backslash e)}$ □

Fact 6.10 $S(\!|\textbf{module} \bullet y = e, b \textbf{ end}|\!)_{\bullet y \in A, S_1} = S(e)(\!|\textbf{module } b(y \backslash e) \textbf{ end}|\!)_{S_1(y\backslash e)}$ □

We define signature instantiation using loose-signature instantiation. An application of loose-signature instantiation has the form $Q\langle\!|p|\!\rangle_P$ where P **lsg**, and $p \in P$ is a value tuple, and $P \vdash Q$ **lsg** holds. $Q\langle\!|p|\!\rangle_P$ is Q applied to each component value in p, so that Q becomes instantiated as a non-dependent loose-signature. For example, if $p = \langle e_1, \langle \ldots, \langle e_n, tt\rangle\rangle\rangle \in P$ then $Q\langle\!|p|\!\rangle_P = Q(e_1)\ldots(e_n)$.

Definition 6.3 (Loose-signature Instantiation)
Given P **lsg**, $\sum x \in A.B(x) \vdash Q$ **lsg** and any $\langle a, b\rangle \in \sum x \in A.B(x)$ then:

$$\begin{array}{lcl} P\langle\!|tt|\!\rangle_T & = & P \\ Q\langle\!|\langle a,b\rangle|\!\rangle_{\sum x \in A.B(x)} & = & Q(a)\langle\!|b|\!\rangle_{B(a)} \end{array}$$

□

Definition 6.4 (Signature Instantiation)
Given S **sig**, $S \vdash \{l\}_{\mathbb{D}} \times Q$ **sig** and any $m \in S$ then:

$$(\{l\}_{\mathbb{D}} \times Q)(\!|m|\!)_S = \{l\}_{\mathbb{D}} \times Q\langle\!|val(m)|\!\rangle_{Lsg(S)} \textbf{ sig}$$

□

6.3 Signature Renaming

In this section, we define the *signature renaming* operator. An application of this operator has the form $S[p\backslash q]$ for p and q any component names and S a signature. $S[p\backslash q]$ is S with component p renamed to q. An attempt to rename a non-existent component leaves a signature unchanged. Consider the following signature:

$$S_1 = Book \in U_1, Stock \in U_1, add \in Book \to Stock \to Stock$$

Renaming *Book* to *Video*, in S_1, is written as $S_1[Book\backslash Video]$. $S_1[Book\backslash Video]$ can be expressed in canonical form as follows:

$$S_1[Book\backslash Video] = Video \in U_1, Stock \in U_1, add \in Video \to Stock \to Stock$$

An attempt to rename an overloaded component name in any signature, S for example, only renames the first component that has that overloaded name in S. For example, given a signature $S = (y \in A_1, p \in A_2, p \in A_3)$, in which p is overloaded, then $S[p\backslash q] = (y \in A_1, q \in A_2, p \in A_3)$. Note that $p \in A_3$ remains unchanged in $S[p\backslash q]$. This feature of renaming allows each component with an overloaded name to be given a distinct name. For example, $(S[p\backslash q])[p\backslash r]$ renames $p \in A_2$ to q and $p \in A_3$ to r.

6.3.1 A Formal Definition of Signature Renaming

We define signature renaming in Definition 6.5, with explanation following. Signature renaming only modifies the domain of a signature; the loose signature remains unchanged. Signature renaming is defined using the *domain substitution* function, given in Definition 6.7, an application of which has the form $l[p\backslash\backslash q]$ for any $p, q \in \mathbb{S}$ and $l \in \mathbb{D}$ a domain. $l[p\backslash\backslash q]$ is l with the first tagged occurrence of p (i.e. the first occurrence of $inl(p)$ or $inr(p)$) replaced by q, where q is tagged with the same tag that p has. Note that we use the notation given in Definition 6.6 to define the right-hand side of the equality in Definition 6.5; see the remark below. The notation overloads the signature renaming notation in our specification language. To disambiguate this overloading, if S **sig** (i.e. S is a type) then $S[p\backslash q]$ is a use of the notation in Definition 6.6, but if S is a signature term of our specification language then so is $S[p\backslash q]$.

Definition 6.5 (Signature Renaming)
Let S be a signature in our specification language (i.e. $[\![S]\!]$ **sig**) and $[\![p]\!], [\![q]\!] \in \mathbb{S}$.

$$[\![S[p\backslash q]]\!] = [\![S]\!][[\![p]\!]\backslash[\![q]\!]]$$

□

Definition 6.6 (Notation for Signature Renaming in MLT)
Given S **sig** and $p, q \in \mathbb{S}$ then $S[p\backslash q] = \{Dom(S)[p\backslash\backslash q]\}_{\mathbb{D}} \times Lsg(S)$
□

Definition 6.7 (Domain Substitution)
Let $a, p, q \in \mathbb{S}$ and $l \in \mathbb{D}$. The domain substitution function $(\_[\_\backslash\backslash\_]) \in \mathbb{D} \to \mathbb{S} \to \mathbb{S} \to \mathbb{D}$ is defined as follows:

$$\begin{aligned}
nil[p\backslash\backslash q] &= nil \\
(inl(a) : l)[p\backslash\backslash q] &= \begin{cases} inl(q) : l & \text{if } a = p \\ inl(a) : (l[p\backslash\backslash q]) & \text{if } a \neq p \end{cases} \\
(inr(a) : l)[p\backslash\backslash q] &= \begin{cases} inr(q) : l & \text{if } a = p \\ inr(a) : (l[p\backslash\backslash q]) & \text{if } a \neq p \end{cases}
\end{aligned}$$

□

Remark It is possible to define signature renaming without introducing Definition 6.6. However, its introduction gives us a concise way of denoting expression of the form $\{Dom(S)[p\backslash\backslash q]\}_{\mathbb{D}} \times Lsg(S)$ which will appear frequently in theorems, proofs and definitions. We will introduce similar notations for each of the other signature, and computational element, operations and use them to define the respective signature, or computational element, operation. In each case, the new notation overloads the notation for the corresponding signature, or computational element, operator, but it should always be clear from the context how to disambiguate the overloading. □

6.3.2 Laws of Signature Renaming

The application of signature renaming to any canonical signature can be unfolded to regain a canonical signature. This statement is justified by Theorems 6.1–6.5. For example, Theorem 6.4 gives a rule, defined in terms of textual substitution, for applying signature renaming to a non-empty canonical signature; in Theorem 6.4, the expression $S(p\backslash q)$ stands for S with each free occurrence of p replaced by q. Theorems 6.1–6.5 constitute a complete set of rules for calculating the result of applying signature renaming to any canonical signature.

Theorem 6.1 $\Phi[p\backslash q] = \Phi$ □

Proof Omitted □

Theorem 6.2 $(y \in A)[p\backslash q] = \begin{cases} y \in A & \text{if } p \neq y \\ q \in A & \text{if } p = y \end{cases}$ □

Proof Similar to that of Theorem 6.4 below. □

Theorem 6.3 $(\bullet y \in A)[p\backslash q] = \begin{cases} \bullet y \in A & \text{if } p \neq y \\ \bullet q \in A & \text{if } p = y \end{cases}$ □

Proof Similar to that of Theorem 6.4 below. □

Theorem 6.4 $(y \in A, S)[p\backslash q] = \begin{cases} y \in A, (S[p\backslash q]) & \text{if } p \neq y \\ q \in A, (S(p\backslash q)) & \text{if } p = y \end{cases}$ □

Proof The proof is given in Figure 6.1. After the second step of the proof, the proof proceeds in two parts which consider the cases $p = y$ and $p \neq y$, respectively □

Theorem 6.5 $(\bullet y \in A, S)[p\backslash q] = \begin{cases} \bullet y \in A, (S[p\backslash q]) & \text{if } p \neq y \\ \bullet q \in A, (S(p\backslash q)) & \text{if } p = y \end{cases}$ □

Proof Similar to that of Theorem 6.4 above. □

6.4 Signature Hiding

In this section, we define the *signature hiding* operator. An application of this operator has the form $S\backslash i$ for S any signature and i a set of component names. $S\backslash i$ is S with each visible component named in i redefined as a local component. For example, consider the following signature:

$$S_1 = sqr \in \mathbb{N} \rightarrow \mathbb{N}, \bullet pi \in \mathbb{N}, area \in \mathbb{N} \rightarrow \mathbb{N}$$

$S_1\backslash\{sqr, pi\}$ is the signature S_1 with the components sqr and pi made local. Note that pi is already local to S_1 and it remains local in $S_1\backslash\{sqr, pi\}$. The canonical form of $S_1\backslash\{sqr, pi\}$ is defined below:

$$S_1\backslash\{sqr, pi\} = \bullet sqr \in \mathbb{N} \rightarrow \mathbb{N}, \bullet pi \in \mathbb{N}, area \in \mathbb{N} \rightarrow \mathbb{N}$$

If a component name, p for example, is overloaded in a signature S then $S\backslash\{p\}$ hides all the components named p in S. If we wish to keep some of the components named p visible, then these should be temporarily renamed prior to applying signature hiding.

Proof **(of Theorem 6.4)**

$[\![(y \in A, S)[p\backslash q]]\!]$

$=$ "Fact 6.19"

$\{[\![(y \in A, S)]\!]_{dom}[p\backslash\!\backslash q]\}_{\mathbb{D}} \times [\![(y \in A, S)]\!]_{Lsg}$

$=$ "definition of $[\![\_]\!]_{dom}$"

$\{(inl(y) : [\![S]\!]_{dom})[p\backslash\!\backslash q]\}_{\mathbb{D}} \times [\![(y \in A, S)]\!]_{Lsg}$

Case $p = y$:

$\{(inl(y) : [\![S]\!]_{dom})[p\backslash\!\backslash q]\}_{\mathbb{D}} \times [\![(y \in A, S)]\!]_{Lsg}$

$=$ "assumption $p = y$, definition of domain substitution, definition of $[\![\_]\!]_{Lsg}$"

$\{inl(q) : [\![S]\!]_{dom}\}_{\mathbb{D}} \times \sum y \in A.[\![S]\!]_{Lsg}$

$=$ " $[\![S]\!]_{dom} = [\![S(p\backslash q)]\!]_{dom}$, renaming bound variable y to q"

$\{inl(q) : [\![S(p\backslash q)]\!]_{dom}\}_{\mathbb{D}} \times \sum q \in A.[\![S(p\backslash q)]\!]_{Lsg}$

$=$ "definitions of $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{Lsg}$"

$\{[\![q \in A, (S(p\backslash q))]\!]_{dom}\}_{\mathbb{D}} \times [\![q \in A, (S(p\backslash q))]\!]_{Lsg}$

$=$ "definition of $[\![\_]\!]$ on signature terms"

$[\![q \in A, (S(p\backslash q))]\!]$

Case $p \neq y$:

$\{(inl(y) : [\![S]\!]_{dom})[p\backslash\!\backslash q]\}_{\mathbb{D}} \times [\![(y \in A, S)]\!]_{Lsg}$

$=$ "assumption $p \neq y$, definition of domain substitution, definition of $[\![\_]\!]_{Lsg}$"

$\{inl(y) : ([\![S]\!]_{dom}[p\backslash\!\backslash q])\}_{\mathbb{D}} \times \sum y \in A.[\![S]\!]_{Lsg}$

$=$ "Fact 6.19 gives $[\![S[p\backslash q]]\!]_{dom} = [\![S]\!]_{dom}[p\backslash\!\backslash q]$ and $[\![S[p\backslash q]]\!]_{Lsg} = [\![S]\!]_{Lsg}$"

$\{inl(y) : [\![S[p\backslash q]]\!]_{dom}\}_{\mathbb{D}} \times \sum y \in A.[\![S[p\backslash q]]\!]_{Lsg}$

$=$ "definitions of $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{Lsg}$"

$\{[\![y \in A, (S[p\backslash q])]\!]_{dom}\}_{\mathbb{D}} \times [\![y \in A, (S[p\backslash q])]\!]_{Lsg}$

$=$ "definition of $[\![\_]\!]$ on signature terms"

$[\![y \in A, (S[p\backslash q])]\!]$

□

Figure 6.1: A proof of Theorem 6.4

6.4.1 A Formal Definition of Signature Hiding

Like signature renaming, signature hiding only changes the domain of a signature. Signature hiding is defined using the *domain hiding* function given in Definition 6.10, and the notation given in Definition 6.9 (recall the remark in Section 6.3.1). An application of domain hiding has the form $l \backslash\backslash i$ for $l \in \mathbb{D}$ a domain, and $i \in Set(\mathbb{S})$ a set of component names. For all names $y \in i$, $l \backslash\backslash i$ is l with each visible name $inl(y)$, in l, replaced by a local name $inr(y)$. The formal definition of signature hiding is given in Definition 6.8 below.

Definition 6.8 (Signature Hiding)
Let S be a signature in our specification language (i.e. $[\![S]\!]$ **sig**), and $[\![i]\!] \in Set(\mathbb{S})$.

$$[\![S\backslash i]\!] = [\![S]\!] \backslash [\![i]\!]$$

□

Definition 6.9 (Notation for Signature Hiding in MLT)
Given S **sig** and $i \in Set(\mathbb{S})$ then $S \backslash i = \{Dom(S) \backslash\backslash i\}_{\mathbb{D}} \times Lsg(S)$
□

Definition 6.10 (Domain Hiding)
Let $y \in \mathbb{S}$, $l \in \mathbb{D}$ and $i \in Set(\mathbb{S})$. The domain hiding function $(\_\backslash\backslash\_) \in \mathbb{D} \to Set(\mathbb{S}) \to \mathbb{D}$ is defined as follows :

$$\begin{array}{lcl} nil \backslash\backslash i & = & nil \\ (inl(y) : l) \backslash\backslash i & = & \begin{cases} inr(y) : (l \backslash\backslash i) & \text{if } y \in i \\ inl(y) : (l \backslash\backslash i) & \text{if } y \notin i \end{cases} \\ (inr(y) : l) \backslash\backslash i & = & inr(y) : (l \backslash\backslash i) \end{array}$$

□

6.4.2 Laws of Signature Hiding

If S is a canonical signature, then the result of a signature hiding $S \backslash i$ ($i \in Set(\mathbb{S})$) can always be expressed by putting bullets (•) in front of all the visible components of S which are named in i. This property is justified by Theorems 6.6–6.10 below; these theorems are concerned with unfolding applications of signature hiding to produce canonical signatures.

Theorem 6.6 $\Phi \backslash i = \Phi$ □

Proof

$$
\begin{aligned}
& [\![\Phi\backslash i]\!] \\
= \ & \text{“definition of signature hiding”} \\
& \{Dom([\![\Phi]\!])\backslash\!\!\backslash i\}_{\mathbb{D}} \times Lsg([\![\Phi]\!]) \\
= \ & \text{“}Dom([\![\Phi]\!]) = nil \text{ and } nil\backslash\!\!\backslash i = nil,\ Lsg([\![\Phi]\!]) = T\text{”} \\
& \{nil\}_{\mathbb{D}} \times T \\
= \ & \text{“definition of } [\![\Phi]\!]\text{”} \\
& [\![\Phi]\!]
\end{aligned}
$$

□

Theorem 6.7 $(y \in A)\backslash i = \begin{cases} \bullet y \in A & \text{if } y \in i \\ y \in A & \text{if } y \notin i \end{cases}$ □

Proof Similar to that of Theorem 6.9 below □

Theorem 6.8 $(\bullet y \in A)\backslash i = \bullet y \in A$ □

Proof Similar to that of Theorem 6.10 below □

Theorem 6.9 $(y \in A, S)\backslash i = \begin{cases} \bullet y \in A, (S\backslash i) & \text{if } y \in i \\ y \in A, (S\backslash i) & \text{if } y \notin i \end{cases}$ □

Proof Given in Figure 6.2. After the third step, the proof proceeds by a two part case analysis for the case $y \in i$ and $y \notin i$. □

Theorem 6.10 $(\bullet y \in A, S)\backslash i = \bullet y \in A, (S\backslash i)$. □

Proof

$$
\begin{aligned}
& [\![(\bullet y \in A, S)\backslash i]\!] \\
= \ & \text{“Fact 6.20”} \\
& \{[\![\bullet y \in A, S]\!]_{dom}\backslash\!\!\backslash i\}_{\mathbb{D}} \times [\![(\bullet y \in A, S)\backslash i]\!]_{Lsg} \\
= \ & \text{“definitions of } [\![\_]\!]_{dom} \text{ and } [\![\_]\!]_{Lsg}\text{”} \\
& \{(inr(y) : [\![S]\!]_{dom})\backslash\!\!\backslash i\}_{\mathbb{D}} \times \textstyle\sum y \in A.[\![S]\!]_{Lsg} \\
= \ & \text{“definition of domain hiding”} \\
& \{inr(y) : ([\![S]\!]_{dom}\backslash\!\!\backslash i)\}_{\mathbb{D}} \times \textstyle\sum y \in A.[\![S]\!]_{Lsg} \\
= \ & \text{“definition of signature hiding, twice”} \\
& \{inr(y) : [\![S\backslash i]\!]_{dom}\}_{\mathbb{D}} \times \textstyle\sum y \in A.[\![S\backslash i]\!]_{Lsg} \\
= \ & \text{“definitions of } [\![\_]\!]_{dom} \text{ and } [\![\_]\!]_{Lsg}\text{”} \\
& \{[\![\bullet y \in A, (S\backslash i)]\!]_{dom}\}_{\mathbb{D}} \times [\![\bullet y \in A, (S\backslash i)]\!]_{Lsg} \\
= \ & \text{“definition of } [\![\_]\!] \text{ on signature terms”} \\
& [\![\bullet y \in A, (S\backslash i)]\!]
\end{aligned}
$$

□

Proof (of Theorem 6.9)

$[\![(y \in A, S)\backslash i]\!]$

= "Fact 6.20"

$\{[\![(y \in A, S)]\!]_{dom}\backslash\!\backslash i\}_{\mathbb{D}} \times [\![(y \in A, S)\backslash i]\!]_{Lsg}$

= "definition of $[\![\_]\!]_{dom}$"

$\{(inl(y) : [\![S]\!]_{dom})\backslash\!\backslash i\}_{\mathbb{D}} \times [\![y \in A, S]\!]_{Lsg}$

Case $y \in i$:

$\{(inl(y) : [\![S]\!]_{dom})\backslash\!\backslash i\}_{\mathbb{D}} \times [\![y \in A, S]\!]_{Lsg}$

= "assumption $y \in i$, definition of domain hiding, definition of $[\![\_]\!]_{Lsg}$"

$\{inr(y) : ([\![S]\!]_{dom}\backslash\!\backslash i)\}_{\mathbb{D}} \times \sum y \in A.[\![S]\!]_{Lsg}$

= "definitions of domain hiding and signature hiding"

$\{inr(y) : [\![S\backslash i]\!]_{dom}\}_{\mathbb{D}} \times \sum y \in A.[\![S\backslash i]\!]_{Lsg}$

= "definitions of $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{Lsg}$"

$\{[\![\bullet y \in A, (S\backslash i)]\!]_{dom}\}_{\mathbb{D}} \times [\![\bullet y \in A, (S\backslash i)]\!]_{Lsg}$

= "definition of $[\![\_]\!]$"

$[\![\bullet y \in A, (S\backslash i)]\!]$

Case $y \notin i$:

$\{(inl(y) : [\![S]\!]_{dom})\backslash\!\backslash i\}_{\mathbb{D}} \times [\![y \in A, S]\!]_{Lsg}$

= "assumption $y \notin i$, definition of domain hiding, definition of $[\![\_]\!]_{Lsg}$"

$\{inl(y) : ([\![S]\!]_{dom}\backslash\!\backslash i)\}_{\mathbb{D}} \times \sum y \in A.[\![S]\!]_{Lsg}$

= "definitions of domain hiding and signature hiding"

$\{inl(y) : [\![S\backslash i]\!]_{dom}\}_{\mathbb{D}} \times \sum y \in A.[\![S\backslash i]\!]_{Lsg}$

= "definitions of $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{Lsg}$"

$\{[\![y \in A, (S\backslash i)]\!]_{dom}\}_{\mathbb{D}} \times [\![y \in A, (S\backslash i)]\!]_{Lsg}$

= "definition of $[\![\_]\!]$"

$[\![y \in A, (S\backslash i)]\!]$

□

Figure 6.2: A proof of Theorem 6.9

The properties of signature hiding are determined by those of the domain hiding function $(\_\backslash\!\backslash\_)$. Lemmas 6.1– 6.3 give some properties of $(\_\backslash\!\backslash\_)$ which we will use to justify some laws of signature hiding; we state these lemmas without proof. Lemma 6.1 says that hiding an empty set of names leaves a domain unchanged; Lemma 6.2 says that two applications of $(\_\backslash\!\backslash\_)$ can be composed into a single application of $(\_\backslash\!\backslash\_)$. Lemma 6.3 says that the order in which names are hidden is not important.

Lemma 6.1 $l\backslash\!\backslash\{\} = l$ where $l \in \mathbb{D}$. □

Lemma 6.2 $(l\backslash\!\backslash i)\backslash\!\backslash j = l\backslash\!\backslash(i \cup j)$ where $l \in \mathbb{D}$ and $i, j \in Set(\mathbb{S})$. □

Lemma 6.3 $(l\backslash\!\backslash i)\backslash\!\backslash j = (l\backslash\!\backslash j)\backslash\!\backslash i$ where $l \in \mathbb{D}$ and $i, j \in Set(\mathbb{S})$. □

Theorem 6.11, below, says that hiding an empty set of components in a signature does not change the signature.

Theorem 6.11 Given S **sig**, $S\backslash\{\} = S$. □

Proof

$$\begin{aligned}
& S\backslash\{\} \\
= & \text{ “definition of signature hiding in MLT”} \\
& \{Dom(S)\backslash\!\backslash\{\}\}_{\mathbb{D}} \times Lsg(S) \\
= & \text{ “Lemma 6.1”} \\
& \{Dom(S)\}_{\mathbb{D}} \times Lsg(S) \\
= & \text{ “semantics of signatures”} \\
& S
\end{aligned}$$

□

Two, or more, consecutive applications of signature hiding can be composed into a single application of signature hiding:

Theorem 6.12 $(S\backslash i)\backslash j = S\backslash(i \cup j)$ where S **sig** and $i, j \in Set(\mathbb{S})$. □

Proof

$$\begin{aligned}
& (S\backslash i)\backslash j \\
= & \text{ “definition of signature hiding in MLT, twice”} \\
& \{(Dom(S)\backslash\!\backslash i)\backslash\!\backslash j\}_{\mathbb{D}} \times Lsg(S) \\
= & \text{ “Lemma 6.2”} \\
& \{Dom(S)\backslash\!\backslash(i \cup j)\}_{\mathbb{D}} \times Lsg(S) \\
= & \text{ “definition of signature hiding in MLT”} \\
& S\backslash(i \cup j)
\end{aligned}$$

□

The order in which we compose consecutive applications of signature hiding does not affect the final result of the applications:

Corollary 6.1 $(S\backslash i)\backslash j = (S\backslash j)\backslash i$. □

Proof Apply Theorem 6.12 to show that $(S\backslash i)\backslash j = S\backslash(i \cup j)$. Next, use set union commutativity to show that $S\backslash(i \cup j) = S\backslash(j \cup i)$. Then apply Theorem 6.12, to show that $S\backslash(j \cup i) = (S\backslash j)\backslash i$. □

6.5 Signature Concatenation

In this section, we define the signature concatenation operator. An application of this operator has the form $S_1 \oplus S_2$ for S_1 and S_2 signatures. $S_1 \oplus S_2$ is a signature containing all the components defined in S_1 followed by all the components in S_2. The right-hand argument, S_2, may be dependent on S_1 (i.e. $S_1 \vdash S_2$ **sig**). Consider the following signatures:

$$\begin{aligned} S_1 &= \mathit{Sqr} \in U_1, \mathit{size} \in \mathit{Sqr} \to \mathbb{N} \\ S_2 &= \mathit{area} \in \mathit{Sqr} \to \mathbb{N}, \mathit{size} \in \mathit{Sqr} \to \mathbb{N} \end{aligned}$$

Note that S_2 is dependent on the component *Sqr* in S_1. The result of the signature concatenation $S_1 \oplus S_2$ is given below:

$$S_1 \oplus S_2 = \mathit{Sqr} \in U_1, \mathit{size} \in \mathit{Sqr} \to \mathbb{N}, \mathit{area} \in \mathit{Sqr} \to \mathbb{N}, \mathit{size} \in \mathit{Sqr} \to \mathbb{N}$$

Note that name *size*, which is used in both S_1 and S_2, is overloaded in $S_1 \oplus S_2$. The signature concatenation operator overloads any name that appears in both its arguments. Overloading resolves any potential name-clashes arising in the result of a concatenation. We discuss the issue of name-clashes in more detail in Section 6.12.

6.5.1 A Formal Definition of Signature Concatenation

Two signatures are concatenated by concatenating their domains, using list concatenation ($+\!\!+$), and concatenating their loose-signatures, using the *loose-signature concatenation* function. An application of loose signature concatenation has the form $P \otimes Q$ for P and Q loose signatures. $P \otimes Q$ is a loose signature containing all the components in P followed by all the components in Q (i.e. $P \otimes Q$ is P with its final unit type (T) replaced by Q). Q may be dependent on the components given in P (i.e. $P \vdash Q$ **lsg**). Loose signature concatenation is defined in Definition 6.11.

We illustrate the use of loose-signature concatenation by considering the loose-signatures of S_1 and S_2 above:

$$\begin{aligned} [\![S_1]\!]_{Lsg} &= \textstyle\sum \mathit{Sqr} \in U_1. \sum \mathit{size} \in \mathit{Sqr} \to \mathbb{N}. T \\ [\![S_2]\!]_{Lsg} &= [\mathit{Sqr}][\mathit{size}] \textstyle\sum \mathit{area} \in \mathit{Sqr} \to \mathbb{N}. \sum \mathit{size} \in \mathit{Sqr} \to \mathbb{N}. T \end{aligned}$$

The prefix $[\mathit{Sqr}][\mathit{size}]$ on $[\![S_2]\!]_{Lsg}$ indicates that $[\![S_2]\!]_{Lsg}$ is dependent on $[\![S_1]\!]_{Lsg}$. The result of the loose-signature concatenation $[\![S_1]\!]_{Lsg} \otimes [\![S_2]\!]_{Lsg}$ is:

$$\sum \mathit{Sqr} \in U_1. \sum \mathit{size} \in \mathit{Sqr} \to \mathbb{N}. \sum \mathit{area} \in \mathit{Sqr} \to \mathbb{N}. \sum \mathit{size} \in \mathit{Sqr} \to \mathbb{N}. T$$

The free variable Sqr in $[\![S_2]\!]_{Lsg}$ becomes bound to $Sqr \in U_1$. The overloading of *size* is resolved by the usual scope rules for quantified expressions in MLT.

Definition 6.11 (Loose Signature Concatenation ($\otimes$))
Given any P **lsg**, $\sum x \in A.B(x)$ **lsg**, and $\sum x \in A.B(x) \vdash Q$ **lsg**:

$$\begin{array}{lcl} T \otimes P & = & P \\ (\sum x \in A.B(x)) \otimes Q & = & \sum x \in A.(B(x) \otimes Q(x)) \end{array}$$

□

The formal definition of signature concatenation is given in Definition 6.12, and uses the notation defined in Definition 6.13 (recall the remark in Section 6.3.1). Note that in Definition 6.13, we write the right-hand argument of $\oplus$ as $(\{l\}_{\mathbb{D}} \times Q)$ so that we can refer to its domain l and loose signature Q: if the right-hand argument of $\oplus$ is a dependent signature then we cannot apply *Dom* and *Lsg* to it, as *Dom* and *Lsg* are not defined on dependent signatures.

Definition 6.12 (Signature Concatenation)
Given $[\![S_1]\!]$ **sig** and $[\![S_1]\!] \vdash [\![S_2]\!]$ **sig** then

$$[\![S_1 \oplus S_2]\!] = [\![S_1]\!] \oplus [\![S_2]\!]$$

□

Definition 6.13 (Notation for Signature Concatenation in MLT($\oplus$))
Given any S_1 **sig**, and $S_1 \vdash (\{l\} \times Q)$ **sig** where $l \in \mathbb{D}$ and $Lsg(S_1) \vdash Q$ **lsg**:

$$\begin{array}{lcl} Dom(S_1 \oplus (\{l\} \times Q)) & = & Dom(S_1) +\!\!+ l \\ Lsg(S_1 \oplus (\{l\} \times Q)) & = & Lsg(S_1) \otimes Q \end{array}$$

□

Given any signature concatenation $S_1 \oplus S_2$ such that S_2 is not a dependent signature, then we can apply *Lsg* and *Dom* to S_2 to get the following facts:

Fact 6.11 If S_1 **sig** and S_2 **sig** then $Dom(S_1 \oplus S_2) = Dom(S_1) +\!\!+ Dom(S_2)$ □

Fact 6.12 If S_1 **sig** and S_2 **sig** then $Lsg(S_1 \oplus S_2) = Lsg(S_1) \otimes Lsg(S_2)$ □

6.5.2 Laws of Signature Concatenation

The properties of signature concatenation are determined by those of list concatenation and loose-signature concatenations. Lemmas 6.4 and 6.5 give some properties of loose-signature concatenation which we will use to justify some laws of signature concatenation. Lemma 6.4 says that the unit type is a two sided identity for ($\otimes$), and Lemma 6.5 says that ($\otimes$) is associative.

Lemma 6.4 $T \otimes P = P$ and $P \otimes T = P$ where P **lsg** and T is the unit type □

Proof Trivial, and is omitted □

Lemma 6.5 $P \otimes (Q \otimes R) = (P \otimes Q) \otimes R$ where P **lsg**, Q **lsg** and R **lsg** □

Proof Omitted □

The empty signature Φ is a right- and left-identity for signature concatenation:

Theorem 6.13 $S \oplus \Phi = S$ □

Proof

$$\begin{array}{ll} & [\![S \oplus \Phi]\!] \\ = & \text{“definition of } \oplus \text{, definition of } \Phi\text{”} \\ & [\![S]\!] \oplus (\{nil\}_{\mathbb{D}} \times T) \\ = & \text{“definition of signature concatenation”} \\ & \{Dom([\![S]\!]) +\!\!+ nil\}_{\mathbb{D}} \times (Lsg([\![S]\!]) \otimes T) \\ = & \text{“list calculus, Lemma 6.4”} \\ & \{Dom([\![S]\!])\}_{\mathbb{D}} \times Lsg([\![S]\!]) \\ = & \text{“definition of signature”} \\ & [\![S]\!] \end{array}$$

□

Theorem 6.14 $\Phi \oplus S = S$ □

Proof Similar to that of Theorem 6.13 above □

Theorems 6.15–6.18 (below) are a collection of laws that can be used to unfold applications of signature concatenation to canonical signatures.

Theorem 6.15 $(y \in A) \oplus S_2 = \begin{cases} y \in A & \text{if } S_2 = \Phi \\ y \in A, S_2 & \text{if } S_2 \neq \Phi \end{cases}$ □

Proof Similar to that of Theorem 6.17 below □

Theorem 6.16 $(\bullet y \in A) \oplus S_2 = \begin{cases} \bullet y \in A & \text{if } S_2 = \Phi \\ \bullet y \in A, S_2 & \text{if } S_2 \neq \Phi \end{cases}$ □

Proof Similar to that of Theorem 6.17 below □

Theorem 6.17 $(y \in A, S_1) \oplus S_2 = y \in A, (S_1 \oplus S_2)$ □

Proof

$[\![(y \in A, S_1) \oplus S_2]\!]$

$=$ "Fact 6.21"

$\{[\![y \in A, S_1]\!]_{dom} +\!\!+ [\![S_2]\!]_{dom}\}_{\mathbb{D}} \times ([\![y \in A, S_1]\!]_{Lsg} \otimes [\![S_2]\!]_{Lsg})$

$=$ "definitions of $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{Lsg}$"

$\{inl(y) : ([\![S_1]\!]_{dom} +\!\!+ [\![S_2]\!]_{dom})\}_{\mathbb{D}} \times ((\sum y \in A.[\![S_1]\!]_{Lsg}) \otimes [\![S_2]\!]_{Lsg})$

$=$ "definition of $\oplus$, definition of $\otimes$"

$\{inl(y) : [\![S_1 \oplus S_2]\!]_{dom}\}_{\mathbb{D}} \times (\sum y \in A.([\![S_1]\!]_{Lsg} \otimes [\![S_2]\!]_{Lsg}))$

$=$ "definitions of $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{Lsg}$"

$\{[\![y \in A, (S_1 \oplus S_2)]\!]_{dom}\}_{\mathbb{D}} \times [\![y \in A, (S_1 \oplus S_2)]\!]_{Lsg}$

$=$ "definition of $[\![\_]\!]$ on signature terms"

$[\![y \in A, (S_1 \oplus S_2)]\!]$

□

Theorem 6.18 $(\bullet y \in A, S_1) \oplus S_2 = \bullet y \in A, (S_1 \oplus S_2)$ □

Proof Similar to that of Theorem 6.17 above □

Signature concatenation is not commutative; this is because list concatenation and loose-signature concatenation are not commutative. However, signature concatenation is associative:

Theorem 6.19 ($\oplus$ associativity)
If S_1 **sig**, S_2 **sig** and S_3 **sig** then $S_1 \oplus (S_2 \oplus S_3) = (S_1 \oplus S_2) \oplus S_3$.
□

Proof

$S_1 \oplus (S_2 \oplus S_3)$

$=$ "Fact 6.11, Fact 6.12"

$S_1 \oplus (\{Dom(S_2) +\!\!+ Dom(S_3)\}_{\mathbb{D}} \times (Lsg(S_2) \otimes Lsg(S_3)))$

$=$ "definition of signature concatenation"

$\{Dom(S_1) +\!\!+ (Dom(S_2) +\!\!+ Dom(S_3))\}_{\mathbb{D}} \times (Lsg(S_1) \otimes (Lsg(S_2) \otimes Lsg(S_3)))$

$=$ "associativity of $+\!\!+$, associativity of $\otimes$ (Lemma 6.5)"

$\{(Dom(S_1) +\!\!+ Dom(S_2)) +\!\!+ Dom(S_3)\}_{\mathbb{D}} \times ((Lsg(S_1) \otimes Lsg(S_2)) \otimes Lsg(S_3))$

$=$ "definition of signature concatenation"

$(\{Dom(S_1) +\!\!+ Dom(S_2)\}_{\mathbb{D}} \times (Lsg(S_1) \otimes Lsg(S_2))) \oplus S_3$

$=$ "Fact 6.11, Fact 6.12"

$(S_1 \oplus S_2) \oplus S_3$

□

6.6 Translating Signature Operators

In this section, we summarise the syntax of the signature operations defined in this chapter, and give some useful facts about the operations. The grammar *sigops*, in Definition 6.14, gives the syntax of signature operators; token S (possibly subscripted) stands for *sigops* terms, and token Sl (possibly subscripted) stands for *siglist* terms. *sigops* is an extension of the grammar *sig* for canonical signatures. We assume that *sigops* is added to the syntax of types (i.e. *Type* ::= *sigops*).

Definition 6.14 (Syntax of Signature Operators)

$$\begin{array}{lll} \mathit{sigops} & ::= & \Phi \mid Sl \mid S[p\backslash q] \mid S\backslash i \mid S_1 \oplus S_2 \\ \mathit{siglist} & ::= & y \in A \mid \bullet\, y \in A \mid y \in A, Sl \mid \bullet\, y \in A, Sl \mid Sl[p\backslash q] \mid Sl\backslash i \mid Sl_1 \oplus Sl_2 \end{array}$$

□

We obtain the following facts from the definition of the translation mapping on canonical signatures, which is $[\![S]\!] = \{[\![S]\!]_{dom}\}_{\mathbb{D}} \times [\![S]\!]_{Lsg}$ for any $S : sig$; recall that $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{Lsg}$ translate canonical signatures into their domains and loose-signatures, respectively. Recall, from Chapter 5, that the semantic function $[\![\_]\!]_{Id}$ translates an identifier token into a component name of type $\mathbb{S}$. The term $[\![i]\!]$ is the set of component names obtained by translating the term i : *Exp* into MLT.

Fact 6.13 $[\![S[p\backslash q]]\!]_{dom} = [\![S]\!]_{dom}[[\![p]\!]_{Id}\backslash\!\backslash[\![q]\!]_{Id}]$ □

Fact 6.14 $[\![S\backslash i]\!]_{dom} = [\![S]\!]_{dom}\backslash\!\backslash[\![i]\!]$ □

Fact 6.15 $[\![S_1 \oplus S_2]\!]_{dom} = [\![S_1]\!]_{dom} +\!\!+ [\![S_2]\!]_{dom}$ □

Fact 6.16 $[\![S[p\backslash q]]\!]_{Lsg} = [\![S]\!]_{Lsg}$ □

Fact 6.17 $[\![S\backslash i]\!]_{Lsg} = [\![S]\!]_{Lsg}$ □

Fact 6.18 $[\![S_1 \oplus S_2]\!]_{Lsg} = [\![S_1]\!]_{Lsg} \otimes [\![S_2]\!]_{Lsg}$ □

Fact 6.19 $[\![S[p\backslash q]]\!] = \{[\![S]\!]_{dom}[[\![p]\!]_{Id}\backslash\!\backslash[\![q]\!]_{Id}]\}_{\mathbb{D}} \times [\![S]\!]_{Lsg}$ □

Fact 6.20 $[\![S\backslash i]\!] = \{[\![S]\!]_{dom}\backslash\!\backslash[\![i]\!]\}_{\mathbb{D}} \times [\![S]\!]_{Lsg}$ □

Fact 6.21 $[\![S_1 \oplus S_2]\!] = \{[\![S_1]\!]_{dom} +\!\!+ [\![S_2]\!]_{dom}\}_{\mathbb{D}} \times ([\![S_1]\!]_{Lsg} \otimes [\![S_2]\!]_{Lsg})$ □

6.7 Computational Element Renaming

In this section, we define the *computational element renaming* operator. An application of this operator has the form $m[p\backslash q]$ for p and q names and m a computational element. $m[p\backslash q]$ is m with component p renamed to q. There is a close relationship between computational element renaming and signature renaming: given any signature S and any $m \in S$ then $m[p\backslash q] \in S[p\backslash q]$.

We give an example of computational element renaming by considering the following signature S_1 and computational element $m_1 \in S$:

$$\begin{array}{rcl} S_1 & = & \mathit{Book} \in U_1, \mathit{Stock} \in U_1, \mathit{add} \in \mathit{Book} \to \mathit{Stock} \to \mathit{Stock} \\ m_1 & = & \textbf{module} \\ & & \quad \mathit{Book} = \mathbb{N}, \\ & & \quad \mathit{Stock} = \mathit{Set}(\mathit{Book}), \\ & & \quad \mathit{add} = \lambda b \in \mathit{Book}.\lambda s \in \mathit{Stock}.\ s \cup \{b\} \\ & & \textbf{end} \in S_1 \end{array}$$

$m_1[\mathit{Book}\backslash \mathit{Video}]$ is m_1 with the component *Book* renamed to *Video*. $m_1[\mathit{Book}\backslash \mathit{Video}]$ has the signature $S_1[\mathit{Book}\backslash \mathit{Video}]$, and its canonical form is:

$$\begin{array}{l} \textbf{module} \\ \quad \mathit{Video} = \mathbb{N}, \\ \quad \mathit{Stock} = \mathit{Set}(\mathit{Video}), \\ \quad \mathit{add} = \lambda b \in \mathit{Video}.\lambda s \in \mathit{Stock}.\ s \cup \{b\} \\ \textbf{end} \in S_1[\mathit{Book}\backslash \mathit{Video}] \end{array}$$

The properties of computational element renaming are similar to those of signature renaming. For example, an attempt to rename a non-existent component leaves a computational element unchanged; and an attempt to rename any overloaded component name, p for example, in any computational element, m for example, only renames the first version of p in m.

6.7.1 A Definition of Computational Element Renaming

We give a formal definition of computational element renaming in Definition 6.15 with explanation following. Note that Definition 6.15 uses the notation given in Definition 6.16 (recall the remark in Section 6.3.1). We use the domain substitution function $\_[\_\backslash\backslash\_]$ (given previously in Definition 6.7) to substitute the name p by q in the domain of m. Computational element renaming does not alter the value tuple of a computational element. Note that in Definition 6.16, we also state that given $m \in S$, then $m[p\backslash q] \in S[p\backslash q]$; a proof of this fact is given in Figure 6.3.

Definition 6.15 (Computational Element Renaming)
Given $[\![S]\!]$ **sig**, a computational element $[\![m]\!] \in [\![S]\!]$ and $[\![p]\!], [\![q]\!] \in \mathbb{S}$, then:

$$[\![m[p\backslash q]]\!] = [\![m]\!][[\![p]\!]\backslash[\![q]\!]] \in [\![S[p\backslash q]]\!]$$

□

Definition 6.16 (Computational Element Renaming in MLT)
Given S **sig**, a computational element $m \in S$ and $p, q \in \mathbb{S}$, then:

$$m[p\backslash q] = \langle \mathit{dom}(m)[p\backslash\backslash q], \mathit{val}(m)\rangle \in S[p\backslash q]$$

□

0.0 $|[$ S **sig** ; $m \in S$; $p, q \in \mathbb{S}$

$\rhd$ "0.0, S **sig**, $\times$-elimination"

0.1 $dom(m) \in \{Dom(S)\}_{\mathbb{D}}$

"0.1, $\{\}_{\mathbb{D}}$-elimination"

0.2 $dom(m) = Dom(S) \in \mathbb{D}$

"0.2, $=$-substitution"

0.3 $dom(m)[p\backslash\backslash q] = Dom(S)[p\backslash\backslash q] \in \mathbb{D}$

"0.3, $\{\}_{\mathbb{D}}$-introduction"

0.4 $dom(m)[p\backslash\backslash q] \in \{Dom(S)[p\backslash\backslash q]\}_{\mathbb{D}}$

"0.0, $\times$-elimination"

0.5 $val(m) \in Lsg(S)$

"definition of signature renaming"

0.6 $Lsg(S) = Lsg(S[p\backslash q])$

"0.5, 0.6, $=$-substitution"

0.7 $val(m) \in Lsg(S[p\backslash q])$

"0.4, 0.7, signature satisfaction"

0.8 $\langle dom(m)[p\backslash\backslash q], val(m)\rangle \in S[p\backslash q]$

$]|$

Figure 6.3: Proof of the Type Correctness of Computational Element Renaming

6.7.2 Laws of Computational Element Renaming

Theorems 6.20–6.24 (below) give laws for renaming canonical computational elements. Note the similarity, in function, of these laws to the laws for renaming canonical signatures, given in Section 6.3.2. Note that Theorems 6.23 and 6.24 use computational element concatenation ($\oplus$) which will be defined in Section 6.9: the concatenation of two computational elements m_1 and m_2 (written as $m_1 \oplus m_2$) is a computational element containing the components of m_1 followed by the components of m_2.

Theorem 6.20 **module end**$[p\backslash q] =$ **module end** $\square$

Proof Omitted $\square$

Theorem 6.21 $\textbf{module } y = e \textbf{ end}[p\backslash q] = \begin{cases} \textbf{module } y = e \textbf{ end} & \text{if } p \neq y \\ \textbf{module } q = e \textbf{ end} & \text{if } p = y \end{cases}$ $\square$

Proof Given in Figure 6.4 $\square$

Theorem 6.22 $\textbf{module } \bullet y = e \textbf{ end}[p\backslash q] = \begin{cases} \textbf{module } \bullet y = e \textbf{ end} & \text{if } p \neq y \\ \textbf{module } \bullet q = e \textbf{ end} & \text{if } p = y \end{cases}$ $\square$

Proof Similar to that of Theorem 6.21 above $\square$

Proof (of Theorem 6.21)

$[\![\textbf{module } y = e \textbf{ end}[p\backslash q]]\!]$

= "Fact 6.32"

$[\![\textbf{module } y = e \textbf{ end}]\!][[\![p]\!]\backslash[\![q]\!]]$

= "definition of computational element renaming"

$\langle[\![\textbf{module } y = e \textbf{ end}]\!]_{dom}[[\![p]\!]\backslash\backslash[\![q]\!]], [\![\textbf{module } y = e \textbf{ end}]\!]_{val}\rangle$

= "definitions of $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{val}$"

$\langle[inl([\![y]\!])][[\![p]\!]\backslash\backslash[\![q]\!]], \langle[\![e]\!], tt\rangle\rangle$

Case $p = y$:

$\langle[inl([\![y]\!])][[\![p]\!]\backslash\backslash[\![q]\!]], \langle[\![e]\!], tt\rangle\rangle$

= "assumption $p = y$, definition of domain substitution"

$\langle[inl([\![q]\!])], \langle[\![e]\!], tt\rangle\rangle$

= "definitions of $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{val}$"

$\langle[\![\textbf{module } q = e \textbf{ end}]\!]_{dom}, [\![\textbf{module } q = e \textbf{ end}]\!]_{val}\rangle$

= "definition of the translation mapping on computational elements"

$[\![\textbf{module } q = e \textbf{ end}]\!]$

Case $p \neq y$:

$\langle[inl([\![y]\!])][[\![p]\!]\backslash\backslash[\![q]\!]], \langle[\![e]\!], tt\rangle\rangle$

= "assumption $p = y$, definition of domain substitution"

$\langle[inl([\![y]\!])], \langle[\![e]\!], tt\rangle\rangle$

= "definitions of $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{val}$"

$\langle[\![\textbf{module } y = e \textbf{ end}]\!]_{dom}, [\![\textbf{module } y = e \textbf{ end}]\!]_{val}\rangle$

= "definition of the translation mapping on computational elements"

$[\![\textbf{module } y = e \textbf{ end}]\!]$

□

Figure 6.4: A proof of Theorem 6.21

Theorem 6.23 $\textbf{module } y = e, b \textbf{ end}[p\backslash q]$
$$= \begin{cases} \textbf{module } y = e \textbf{ end} \oplus \textbf{ module } b(y\backslash e) \textbf{ end}[p\backslash q] & \text{if } p \neq y \\ \textbf{module } q = e, b(p\backslash q) \textbf{ end} & \text{if } p = y \end{cases}$$
□

Proof Similar to that of Theorem 6.21 above □

Theorem 6.24 $\textbf{module } \bullet y = e, b \textbf{ end}[p\backslash q]$
$$= \begin{cases} \textbf{module } \bullet y = e \textbf{ end} \oplus \textbf{ module } b(y\backslash e) \textbf{ end}[p\backslash q] & \text{if } p \neq y \\ \textbf{module } \bullet q = e, b(p\backslash q) \textbf{ end} & \text{if } p = y \end{cases}$$
□

Proof Similar to that of Theorem 6.21 above □

6.8 Computational Element Hiding

In this section, we define the computational element hiding operator. An application of this operator has the form $m\backslash i$ for m any computational element and i a set of component names. $m\backslash i$ is m with all visible components named in i redefined as local components. Given any signature S, and any $m \in S$, then $m\backslash i \in S\backslash i$.

We give an example application of computational element hiding by considering the following signature S_1 and computational element $m_1 \in S_1$:

$$\begin{aligned} S_1 &= \mathit{sqr} \in \mathbb{N} \to \mathbb{N}, \bullet \mathit{pi} \in \mathbb{N}, \mathit{area} \in \mathbb{N} \to \mathbb{N} \\ m_1 &= \textbf{module} \\ &\qquad \mathit{sqr} = \lambda x \in \mathbb{N}.x * x, \\ &\qquad \bullet \mathit{pi} = 314, \\ &\qquad \mathit{area} = \lambda r \in \mathbb{N}.(\mathit{pi} * \mathit{sqr}(r))\ \mathit{div}\ 100 \\ &\quad \textbf{end} \in S_1 \end{aligned}$$

$m_1\backslash\{\mathit{sqr}, \mathit{pi}\}$ is m_1 with the components *sqr* and *pi* made local. The signature of $m_1\backslash\{\mathit{sqr}, \mathit{pi}\}$ is $S_1\backslash\{\mathit{sqr}, \mathit{pi}\}$. The canonical form of $m_1\backslash\{\mathit{sqr}, \mathit{pi}\}$ is:

$$\begin{aligned} &\textbf{module} \\ &\quad \bullet \mathit{sqr} = \lambda x \in \mathbb{N}.x * x, \\ &\quad \bullet \mathit{pi} = 314, \\ &\quad \mathit{area} = \lambda r \in \mathbb{N}.(\mathit{pi} * \mathit{sqr}(r))\ \mathit{div}\ 100 \\ &\textbf{end} \in S_1\backslash\{\mathit{sqr}, \mathit{pi}\} \end{aligned}$$

The properties of computational element hiding are similar to those of signature hiding. For example, an attempt to hide a non-existent component leaves a computational element unchanged; and if a component name, p for example, is overloaded in a computational element m then $m\backslash\{p\}$ hides all the components named p in m.

6.8.1 A Definition of Computational Element Hiding

We give a formal definition of computational element hiding in Definition 6.17 with explanation following. Note that Definition 6.17 uses the notation given in Definition 6.18 (recall the remark in Section 6.3.1). We use the domain hiding function $\_\backslash\!\backslash\_$ (given previously in Definition 6.10) to tag, with *inr*, all names in the domain of m that also appear in i. Computational element hiding does not alter the value tuple of a computational element; so $\mathit{val}(m)$ remains unchanged. Note that in Definition 6.18, we state that $m\backslash i \in S\backslash i$; we omit the proof of this property as it is similar, in style, to the proof of the type correctness of computational element renaming.

Definition 6.17 (Computational Element Hiding)
Given $[\![S]\!]$ **sig**, any computational element $[\![m]\!] \in [\![S]\!]$ and $[\![i]\!] \in \mathit{Set}(\mathbb{S})$, then:

$$[\![m\backslash i]\!] = [\![m]\!]\backslash[\![i]\!] \in [\![S\backslash i]\!]$$

□

Definition 6.18 (Computational Element Hiding in MLT)
Given S **sig**, any computational element $m \in S$ and $i \in Set(\mathbb{S})$, then:

$$m\backslash i = \langle dom(m)\backslash\!\backslash i, val(m)\rangle \in S\backslash i$$

□

6.8.2 Laws of Computational Element Hiding

Given any canonical computational element m, $m\backslash i$ ($i \in Set(\mathbb{S})$) can be rewritten by putting bullets in front of all visible components in m which are also named in i. This property is justified by Theorems 6.25–6.29 below. Note that Theorems 6.28 and 6.29 use the computational element concatenation operator ($\oplus$) which will be defined in Section 6.9.

Theorem 6.25 $\textbf{module end}\backslash i = \textbf{module end}$ □

Proof Omitted □

Theorem 6.26 $\textbf{module } y = e \textbf{ end}\backslash i = \begin{cases}\textbf{module } y = e \textbf{ end} & \text{if } y \notin i\\ \textbf{module } \bullet y = e \textbf{ end} & \text{if } y \in i\end{cases}$
□

Proof Omitted □

Theorem 6.27 $\textbf{module } \bullet y = e \textbf{ end}\backslash i = \textbf{module } \bullet y = e \textbf{ end}$ □

Proof Omitted □

Theorem 6.28 $\textbf{module } y = e, b \textbf{ end}\backslash i$
$$= \begin{cases}\textbf{module } y = e \textbf{ end} \oplus \textbf{module } b(y\backslash e) \textbf{ end}\backslash i & \text{if } y \notin i\\ \textbf{module } \bullet y = e \textbf{ end} \oplus \textbf{module } b(y\backslash e) \textbf{ end}\backslash i & \text{if } y \in i\end{cases}$$
□

Proof Omitted □

Theorem 6.29 $\textbf{module } \bullet y = e, b \textbf{ end}\backslash i$
$$= \textbf{module } \bullet y = e \textbf{ end} \oplus \textbf{module } b(y\backslash e) \textbf{ end}\backslash i$$
□

Proof Omitted □

Hiding an empty set of components in a computational element does not change the computational element:

Theorem 6.30 If S **sig** and $m \in S$ then $m\backslash\{\} = m \in S$ □

Proof Omitted □

Two consecutive applications of computational element hiding can be composed into a single application:

Theorem 6.31 If S **sig**, $m \in S$ and $i, j \in \mathit{Set}(\mathbb{S})$ then $m \backslash i \backslash j = m \backslash (i \cup j) \in S \backslash (i \cup j)$ $\square$

Proof Omitted $\square$

The order in which consecutive applications of computational element hiding are made does not affect the final result of the applications:

Theorem 6.32 If S **sig**, $m \in S$ and $i, j \in \mathit{Set}(\mathbb{S})$ then $m \backslash i \backslash j = m \backslash j \backslash i \in S \backslash j \backslash i$ $\square$

Proof Omitted $\square$

6.9 Computational Element Concatenation

In this section, we define the computational element concatenation operator. An application of this operator has the form $m_1 \oplus m_2$ for m_1 and m_2 any computational elements. $m_1 \oplus m_2$ is a computational element containing all the components in m_1 followed by all the components in m_2. Given any signatures S_1 and S_2, and computational elements $m_1 \in S_1$ and $m_2 \in S_2$, then $m_1 \oplus m_2 \in S_1 \oplus S_2$.

We give an example of computational element concatenation by considering the following signatures and computational elements:

$$\begin{aligned}
S_1 &= \mathit{Sqr} \in U_1, \mathit{size} \in \mathit{Rect} \to \mathbb{N} \\
S_2 &= \mathit{area} \in \mathbb{N} \to \mathbb{N}, \mathit{size} \in \mathit{Rect} \to \mathbb{N} \\
m_1 &= \textbf{module } \mathit{Sqr} = \mathbb{N}, \mathit{size} = \lambda s \in \mathbb{N}.s \textbf{ end} \in S_1 \\
m_2 &= \textbf{module } \mathit{area} = \lambda s \in \mathbb{N}.s * s, \mathit{size} = \lambda s \in \mathbb{N}.s * 2 \textbf{ end} \in S_2
\end{aligned}$$

The result of the concatenating m_1 and m_2 is given below:

$$\begin{array}{l}
m_1 \oplus m_2 = \textbf{module} \\
\qquad \begin{array}{lcl}
\mathit{Sqr} &=& \mathbb{N}, \\
\mathit{size} &=& \lambda s \in \mathbb{N}.s, \\
\mathit{area} &=& \lambda s \in \mathbb{N}.s * s, \\
\mathit{size} &=& \lambda s \in \mathbb{N}.s * 2
\end{array} \\
\qquad \textbf{end} \in S_1 \oplus S_2
\end{array}$$

Like signature concatenation, the computational element concatenation operator resolves potential name-clashes by overloading any component name used by both its arguments. For example, the name *size* which appears in m_1 and m_2 (above) is overloaded in $m_1 \oplus m_2$.

6.9.1 Definition of Computational Element Concatenation

We concatenate two computational elements by concatenating their domains, and by concatenating their value tuples. Domains are concatenated using list concatenation. Value tuples are concatenated using *tuple concatenation* which is described below.

An application of tuple concatenation has the form $p \odot q$ for p and q any value tuples. The result of $p \odot q$ is a value tuple containing all the components in p followed by all the components in q. In other words, $p \odot q$ is p with its final component $tt \in T$ replaced by q. For example, given two loose signatures P and Q, and two value tuples,

$$\begin{array}{rcl} p & = & \langle e_1, \langle \ldots, \langle e_i, tt \rangle \ldots \rangle \rangle \in P \text{ and} \\ q & = & \langle e_j, \langle \ldots, \langle e_n, tt \rangle \ldots \rangle \rangle \in Q, \end{array}$$

then

$$p \odot q = \langle e_1, \ldots, \langle e_i, \langle e_j, \langle \ldots, \langle e_n, tt \rangle \ldots \rangle \rangle \rangle \ldots \rangle \in P \otimes Q.$$

We give a formal definition of tuple concatenation in Definition 6.19. Note that we use the loose signature concatenation operator ($\otimes$) to give the type of a tuple concatenation.

Definition 6.19 (Tuple concatenation ($\odot$))
Given $\sum x \in A.B(x)$ **lsg**, Q **lsg**, $\langle a, b \rangle \in \sum x \in A.B(x)$ and $q \in Q$, then:

$$\begin{array}{lcll} tt \odot q & = & q & \in \; T \otimes Q = Q \\ \langle a, b \rangle \odot q & = & \langle a, b \odot q \rangle & \in \; (\sum x \in A.B(x)) \otimes Q = \sum x \in A.(B(x) \otimes Q) \end{array}$$

□

Fact 6.22 Given P **lsg**, Q **lsg**, $p \in P$ and $q \in Q$, then $p \odot q \in P \otimes Q$ □

We give a formal definition of computational element concatenation ($\oplus$) in Definition 6.20, with an explanation following. Note that Definition 6.20 uses the notation given in Definition 6.21 (recall the remark in Section 6.3.1). We concatenate the domains of m_1 and m_2 using list concatenation ($+\!\!+$). We use tuple concatenation ($\odot$) to concatenate the value tuples of m_1 and m_2. Note that $m_1 \oplus m_2 \in S_1 \oplus S_2$; we give a proof of this property in Figure 6.5.

Definition 6.20 (Computational Element Concatenation)
Given any $[\![S_1]\!]$ **sig**, $[\![S_2]\!]$ **sig**, $[\![m_1]\!] \in [\![S_1]\!]$ and $[\![m_2]\!] \in [\![S_2]\!]$, then

$$[\![m_1 \oplus m_2]\!] = [\![m_1]\!] \oplus [\![m_1]\!] \in [\![S_1 \oplus S_2]\!]$$

□

Definition 6.21 (Computational Element Concatenation in MLT($\oplus$))
Given any S_1 **sig**, S_2 **sig**, $m_1 \in S_1$ and $m_2 \in S_2$, then

$$m_1 \oplus m_2 = \langle dom(m_1) +\!\!+ dom(m_2), val(m_1) \odot val(m_2) \rangle \in S_1 \oplus S_2$$

□

Proof

0.0 |[S_1 **sig** ; S_2 **sig**

0.1 $m_1 \in S_1$; $m_2 \in S_2$

$\triangleright$ "0.1, definition of *dom*, twice"

0.2 $dom(m_1) \in \{Dom(S_1)\}_{\mathbb{D}}$; $dom(m_2) \in \{Dom(S_2)\}_{\mathbb{D}}$

"0.2, $\{\}_{\mathbb{D}}$-elimination"

0.3 $dom(m_1) = Dom(S_1) \in \mathbb{D}$

"0.2, $\{\}_{\mathbb{D}}$-elimination"

0.4 $dom(m_2) = Dom(S_2) \in \mathbb{D}$

"0.3, 0.4, =-substitution"

0.5 $dom(m_1) \mathbin{+\!\!+} dom(m_2) = Dom(S_1) \mathbin{+\!\!+} Dom(S_2) \in \mathbb{D}$

"0.5, $\{\}_{\mathbb{D}}$-introduction"

0.6 $dom(m_1) \mathbin{+\!\!+} dom(m_2) \in \{Dom(S_1) \mathbin{+\!\!+} Dom(S_2)\}_{\mathbb{D}}$

"0.6, definition of signature concatenation"

0.7 $dom(m_1) \mathbin{+\!\!+} dom(m_2) \in Dom(S_1 \oplus S_2)$

"0.1, definition of *val*, twice"

0.8 $val(m_1) \in Lsg(S_1)$; $val(m_2) \in Lsg(S_2)$

"0.8, Fact 6.22"

0.9 $val(m_1) \odot val(m_2) \in Lsg(S_1) \otimes Lsg(S_2)$

"0.9, definition of signature concatenation"

0.10 $val(m_1) \odot val(m_2) \in Lsg(S_1 \oplus S_2)$

"0.7, 0.10, signature satisfaction"

0.11 $\langle dom(m_1) \mathbin{+\!\!+} dom(m_2), val(m_1) \odot val(m_2) \rangle \in S_1 \oplus S_2$

"0.11, notation for computational element concatenation"

0.12 $m_1 \oplus m_2 \in S_1 \oplus S_2$

]|

□

Figure 6.5: Proof of the Type Correctness of Computational Element Concatenation

6.9.2 Laws of Computational Element Concatenation

Theorem 6.33 is a law of $\oplus$ for the special case where the signature of the right-hand argument of $\oplus$ depends on the left-hand argument of $\oplus$. Note that in Theorem 6.33, we express the signature $S_2(\!|m_1|\!)_{S_1}$ using the signature instantiation notation defined in Section 6.2.2: $S_2(\!|m_1|\!)_{S_1}$ is a non-dependent signature obtained by replacing each free occurrence of a component name in S_2 with the corresponding component in m_1.

Theorem 6.33
Given S_1 **sig**, $S_1 \vdash S_2$ **sig**, $m_1 \in S_1$ and $m_2 \in S_2(\!|m_1|\!)_{S_1}$ then $m_1 \oplus m_2 \in S_1 \oplus S_2$
□

Proof Similar to the proof of the type correctness of $\oplus$ given in Figure 6.5 □

The empty computational element is a right and left identity for $\oplus$:

Theorem 6.34 $m \oplus \textbf{module end} = m$ □

Proof Omitted □

Theorem 6.35 $\textbf{module end} \oplus m = m$ □

Proof Omitted □

Theorems 6.36–6.39 are a collection of laws used to unfold applications of computational element concatenations to canonical computational elements. These laws can be used to express a canonical computational element as the concatenation of some smaller computational elements.

Theorem 6.36 $\textbf{module } y = e \textbf{ end} \oplus \textbf{module } b \textbf{ end} = \textbf{module } y = e, b \textbf{ end}$ □

Proof Given in Figure 6.6 □

Theorem 6.37 $\textbf{module } \bullet y = e \textbf{ end} \oplus \textbf{module } b \textbf{ end} = \textbf{module } \bullet y = e, b \textbf{ end}$ □

Proof Similar to that of Theorem 6.36 above □

Theorem 6.38 $\textbf{module } y = e, b_1 \textbf{ end} \oplus \textbf{module } b_2 \textbf{ end}$
$= \textbf{module } y = e \textbf{ end} \oplus \textbf{module } b_1(y \backslash e) \textbf{ end} \oplus \textbf{module } b_2 \textbf{ end}$
□

Proof Similar to that of Theorem 6.36 above □

Theorem 6.39 $\textbf{module } \bullet y = e, b_1 \textbf{ end} \oplus \textbf{module } b_2 \textbf{ end}$
$= \textbf{module } \bullet y = e \textbf{ end} \oplus \textbf{module } b_1(y \backslash e) \textbf{ end} \oplus \textbf{module } b_2 \textbf{ end}$
□

Proof Similar to that of Theorem 6.36 above □

Theorem 6.40 (below) says that computational element concatenation is associative. The proof of Theorem 6.40 relies on the associativity of value tuple concatenation:

Proof (of Theorem 6.36)

$\quad [\![\textbf{module } y = e \textbf{ end} \oplus \textbf{module } b \textbf{ end}]\!]$

$=$ "definition of translation function over $\oplus$ (Fact 6.34)"

$\quad [\![\textbf{module } y = e \textbf{ end}]\!] \oplus [\![\textbf{module } b \textbf{ end}]\!]$

$=$ "definition of translation mapping on computational elements, twice"

$\quad \langle [\mathit{inl}([\![y]\!])], \langle [\![e]\!], \mathit{tt}\rangle\rangle \oplus \langle [\![\textbf{module } b \textbf{ end}]\!]_{dom}, [\![\textbf{module } b \textbf{ end}]\!]_{val}\rangle$

$=$ "definition of $\oplus$"

$\quad \langle [\mathit{inl}([\![y]\!])] +\!\!\!+ [\![\textbf{module } b \textbf{ end}]\!]_{dom}, \langle [\![e]\!], \mathit{tt}\rangle \odot [\![\textbf{module } b \textbf{ end}]\!]_{val}\rangle$

$=$ "definition of $\odot$"

$\quad \langle [\mathit{inl}([\![y]\!])] +\!\!\!+ [\![\textbf{module } b \textbf{ end}]\!]_{dom}, \langle [\![e]\!], [\![\textbf{module } b \textbf{ end}]\!]_{val}\rangle\rangle$

$=$ "definitions of $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{val}$"

$\quad \langle [\![\textbf{module } y = e, b \textbf{ end}]\!]_{dom}, [\![\textbf{module } y = e, b \textbf{ end}]\!]_{val}\rangle$

$=$ "definition of translation mapping on computational elements"

$\square \quad [\![\textbf{module } y = e, b \textbf{ end}]\!]$

Figure 6.6: A proof of Theorem 6.36

Lemma 6.6 Given P **lsg**, Q **lsg** and R **lsg**; and $p \in P$, $q \in Q$ and $r \in R$, then $p \odot (q \odot r) = (p \odot q) \odot r \in (P \otimes Q \otimes R)$ □

Proof Omitted □

Theorem 6.40 (Associativity of $\oplus$)
Given S_1 **sig**, S_2 **sig**, S_3 **sig**, and $m_1 \in S_1$, $m_2 \in S_2$, $m_3 \in S_3$ then:

$$(m_1 \oplus m_2) \oplus m_3 = m_1 \oplus (m_2 \oplus m_3) \in (S_1 \oplus S_2) \oplus S_3 = S_1 \oplus (S_2 \oplus S_3)$$

□

Proof

$\quad (m_1 \oplus m_2) \oplus m_3$

$=$ "definition of $\oplus$"

$\quad \langle dom(m_1 \oplus m_2) +\!\!\!+ dom(m_3), val(m_1 \oplus m_2) \odot val(m_3)\rangle$

$=$ "definition of $\oplus$"

$\quad \langle (dom(m_1) +\!\!\!+ dom(m_2)) +\!\!\!+ dom(m_3), (val(m_1) \odot val(m_2)) \odot val(m_3)\rangle$

$=$ "associativity of $+\!\!\!+$, associativity of $\odot$ (Lemma 6.6)"

$\quad \langle dom(m_1) +\!\!\!+ (dom(m_2) +\!\!\!+ dom(m_3)), val(m_1) \odot (val(m_2) \odot val(m_3))\rangle$

$=$ "definition of $\oplus$"

$\quad \langle dom(m_1) +\!\!\!+ dom(m_2 \oplus m_3), val(m_1) \odot val(m_2 \oplus m_3)\rangle$

$=$ "definition of $\oplus$"

$\quad m_1 \oplus (m_2 \oplus m_3)$

□

6.9.3 Projection Functions on Computational Elements

In this section, we describe two operators that split a computational element into two separate computational elements. We use these operators to factor large computational elements into smaller pieces. This is useful when we want to reason about some, but not all, of the components in a large computational element.

Given any two signatures S_1 and S_2, and any two computational elements $m_1 \in S_1$ and $m_2 \in S_2$, the projection function ${}^{S_1S_2}\Pi_1$ takes $m_1 \oplus m_2 \in S_1 \oplus S_2$ and returns $m_1 \in S_1$. The projection function ${}^{S_1S_2}\Pi_2$ takes $m_1 \oplus m_2 \in S_1 \oplus S_2$ and returns $m_2 \in S_2$. The signature S_2 may be dependent on S_1. In that case, ${}^{S_1S_2}\Pi_2(m_1 \oplus m_2) = m_2 \in S_2(\!|m_1|\!)_{S_1}$.

Remark We omit the formal definitions of ${}^{S_1S_2}\Pi_1$ and ${}^{S_1S_2}\Pi_2$, but it may help the reader to know that we define ${}^{S_1S_2}\Pi_1$ and ${}^{S_1S_2}\Pi_2$ using dot notation on computational elements □

We will only require the properties stated in Facts 6.23–6.25, below; in each of these facts we assume that S_1 **sig**, $S_1 \vdash S_2$ **sig**, $m_1 \in S_1$, $m_2 \in S_2(\!|m_1|\!)_{S_1}$ and $m \in S_1 \oplus S_2$.

Fact 6.23 ${}^{S_1S_2}\Pi_1(m_1 \oplus m_2) = m_1 \in S_1$ □

Fact 6.24 ${}^{S_1S_2}\Pi_2(m_1 \oplus m_2) = m_2 \in S_2(\!|m_1|\!)_{S_1}$ □

Fact 6.25 ${}^{S_1S_2}\Pi_1(m) \oplus {}^{S_1S_2}\Pi_2(m) = m \in (S_1 \oplus S_2)$ □

6.10 Translating Computational Element Operators

In this section, we summarise the syntax of the computational element operations defined in this chapter, and we give some useful facts about the operations. The grammar *ceop*, in Definition 6.22, gives the syntax of computational element operators; token m (possibly subscripted) stands for *ceop* terms. Note that *ceop* includes *comp* which is the grammar for canonical computational elements. We assume that *ceop* is added to *Exp* (i.e. *Exp* ::= *ceop*).

Definition 6.22 (Syntax of Computational element Operators)

$$ceop ::= comp \mid m[p\backslash q] \mid m\backslash i \mid m_1 \oplus m_2$$

□

We obtain the following facts from the definition of the translation mapping on canonical computational elements, which is $[\![m]\!] = \langle [\![m]\!]_{dom}, [\![m]\!]_{val} \rangle$ for any $m : comp$; recall that $[\![\_]\!]_{dom}$ and $[\![\_]\!]_{val}$ translate canonical computational elements into their domains and value-tuples, respectively.

Fact 6.26 $[\![m[p\backslash q]]\!]_{dom} = [\![m]\!]_{dom}[[\![p]\!]_{Id} \backslash\!\backslash [\![q]\!]_{Id}]$ □

Fact 6.27 $[\![m\backslash i]\!]_{dom} = [\![m]\!]_{dom} \backslash\!\backslash [\![i]\!]$ $\square$

Fact 6.28 $[\![m_1 \oplus m_2]\!]_{dom} = [\![m_1]\!]_{dom} +\!\!+ [\![m_2]\!]_{dom}$ $\square$

Fact 6.29 $[\![m[p\backslash q]]\!]_{val} = [\![m]\!]_{val}$ $\square$

Fact 6.30 $[\![m\backslash i]\!]_{val} = [\![m]\!]_{val}$ $\square$

Fact 6.31 $[\![m_1 \oplus m_2]\!]_{val} = [\![m_1]\!]_{val} \odot [\![m_2]\!]_{val}$ $\square$

Fact 6.32 $[\![m[p\backslash q]]\!] = [\![m]\!][[\![p]\!]_{Id}\backslash[\![q]\!]_{Id}]$ $\square$

Fact 6.33 $[\![m\backslash i]\!] = [\![m]\!]\backslash[\![i]\!]$ $\square$

Fact 6.34 $[\![m_1 \oplus m_2]\!] = [\![m_1]\!] +\!\!+ [\![m_2]\!]$ $\square$

6.11 Inverse Operators

In this section, we define the inverses of the computational element renaming and hiding operators. These inverse operators will be needed, in Chapter 7, to define specification operators that rename and hide components in specifications. The inverse operators will also be used in Chapter 8 to define some refinement laws.

6.11.1 Inverse of Computational Element Renaming

Given any signature S and component names $p, q \in \mathbb{S}$, then the inverse of the computational element renaming operator $\_[p\backslash q] \in S \to S[p\backslash q]$ is written as $\_[p\backslash q]_S^{-1}$ and has the type $S[p\backslash q] \to S$. Note that $\_[p\backslash q]_S^{-1}$ is subscripted by its range S. The following facts say that $\_[p\backslash q]_S^{-1}$ is the inverse of $\_[p\backslash q]$:

Fact 6.35 $(m[p\backslash q])[p\backslash q]_S^{-1} = m \in S$ where S **sig** and $m \in S$ $\square$

Fact 6.36 $(m[p\backslash q]_S^{-1})[p\backslash q] = m \in S[p\backslash q]$ where S **sig** and $m \in S[p\backslash q]$ $\square$

We give a formal definition of $\_[p\backslash q]_S^{-1}$ in Definition 6.23. We justify Definition 6.23 by proving that $(m[p\backslash q])[p\backslash q]_S^{-1} = m$ (i.e. Fact 6.35), and the proof follows Definition 6.23:

Definition 6.23
Given any S **sig**, names $p, q \in \mathbb{S}$ and $m \in S[p\backslash q]$ then $\_[p\backslash q]_S^{-1} \in S[p\backslash q] \to S$ is defined as follows:

$$m[p\backslash q]_S^{-1} = \langle Dom(S), val(m)\rangle \in S$$

$\square$

Proof (of Fact 6.35)
Consider any $m \in S$ where S is any signature. By Definition 4.8, the domains of m and S are identical, i.e. $dom(m) = Dom(S) \in \mathbb{D}$. By Definition 6.16, the value tuple of $m[p\backslash q]$ is identical to the value tuple of m, i.e. $val(m) = val(m[p\backslash q]) \in Lsg(S)$. The proof of Fact 6.35 is:

$$\begin{aligned}
& (m[p\backslash q])[p\backslash q]_S^{-1} \in S \\
= & \text{ “Definition 6.23”} \\
& \langle Dom(S), val(m[p\backslash q])\rangle \in S \\
= & \text{ “}dom(m) = Dom(S),\ val(m) = val(m[p\backslash q])\text{”} \\
& \langle dom(m), val(m)\rangle \in S \\
= & \text{ “Fact 4.5”} \\
& m \in S
\end{aligned}$$

□

6.11.2 Inverse of Computational Element Hiding

Given any signature S and set of names $i \in Set(\mathbb{S})$, then the inverse of $\_\backslash i \in S \to S\backslash i$ is written as $\_\backslash_S^{-1} i$ and has the type $S\backslash i \to S$. Note that $\_\backslash_S^{-1} i$ is subscripted by its range S. The following facts say that $\_\backslash_S^{-1} i$ is the inverse of $\_\backslash i$:

Fact 6.37 $(m\backslash i)\backslash_S^{-1} i = m \in S$ where S **sig** and $m \in S$ □

Fact 6.38 $(m\backslash_S^{-1} i)\backslash i = m \in S\backslash i$ where S **sig** and $m \in S\backslash i$ □

Definition 6.24 contains a formal definition of $\_\backslash_S^{-1} i$. A justification of the correctness of Definition 6.24 is similar to the justification of the inverse computational element renaming operator (Definition 6.23).

Definition 6.24
Given any S **sig**, a set of names $i \in Set(\mathbb{S})$, and $m \in S\backslash i$ then $\_\backslash_S^{-1} i \in S\backslash i \to S$ is defined as follows:

$$m\backslash_S^{-1} i = \langle Dom(S), val(m)\rangle \in S$$

□

We finish this section by giving two properties of $\_\backslash_S^{-1} i$. The first property is that the inverse operation to hiding an empty set of components is an identity operation:

Theorem 6.41 $m\backslash_S^{-1}\{\} = m$ where S **sig** and $m \in S = S\backslash\{\}$ □

Proof Omitted □

The second property is that consecutive applications of $\_\backslash_S^{-1} i$ can be expressed as a single application of $\_\backslash_S^{-1} i$:

Theorem 6.42
$(m\backslash_{S\backslash i}^{-1} j)\backslash_S^{-1} i = m\backslash_S^{-1}(i \cup j) \in S$ where S **sig**, $i, j \in \mathit{Set}(\mathbb{S})$ and $m \in S\backslash i\backslash j$
□

Proof

$$\begin{aligned}
&\quad (m\backslash_{S\backslash i}^{-1} j)\backslash_S^{-1} i \in S \\
&= \text{“definition of } m\backslash_{S\backslash i}^{-1} j\text{”} \\
&\quad \langle \mathit{Dom}(S\backslash i), \mathit{val}(m)\rangle\backslash_S^{-1} i \in S \\
&= \text{“definition of } m\backslash_{S\backslash i}^{-1} j\text{”} \\
&\quad \langle \mathit{Dom}(S), \mathit{val}(m)\rangle \in S \\
&= \text{“definition of } m\backslash_{S\backslash (i\cup j)}^{-1} j\text{”} \\
&\quad m\backslash_S^{-1}(i \cup j) \in S
\end{aligned}$$

□

6.12 Discussion and Conclusion

In the course of developing the operators given in this chapter, we have considered several alternative definitions for some of our operators. In particular, we have considered several solutions to resolving the name-clashes that arise when concatenating signatures. The most promising alternative solution involves merging components that have the same name. For example, consider the following signatures:

$S_1 = y_1 \in \mathbb{N}, y_2 \in \mathbb{B}$ and $S_2 = y_3 \in \mathbb{N} \times \mathbb{N}, y_2 \in \mathbb{B}$

A signature combinator that merges common names might combines S_1 and S_2 to produce the following signature:

$S_3 = y_1 \in \mathbb{N}, y_2 \in \mathbb{B}, y_3 \in \mathbb{N} \times \mathbb{N}$

Note that the components named y_2 from S_1 and S_2 have been merged to a single component in S_3. In contrast, the signature concatenation of S_1 and S_2 contains two distinct components named y_2:

$S_1 \oplus S_2 = y_1 \in \mathbb{N}, y_2 \in \mathbb{B}, y_3 \in \mathbb{N} \times \mathbb{N}, y_2 \in \mathbb{B}$

Many specification languages, such as ASL and Z, resolve name-clashes by merging components in the style illustrated above. The advantage of merging is that components with the same name are automatically identified as being equal. So, y_2 values from S_3 can be used in place of y_2 values from S_1 and S_2. However, there are problems with merging: if the types of two components with the same name are not equal, then they cannot always be merged. For example, if the type of y_2, in S_2, is changed to $\mathbb{N}$ then y_2 cannot be given a type in S_3, as we cannot merge $\mathbb{N}$ and $\mathbb{B}$. In Z, components are only merged if they have the same name and type, and an attempt to compose two schemas in which the name-clashes cannot be resolved by merging is ill-defined.

In contrast, our signature concatenation operation is a total operation which can combine any two signatures.

In order to merge components that have differing types, we have considered introducing an intersection type $(*)$ that combines any two types P and Q to produce a type $P * Q$ containing the elements common to both P and Q. However, even with such an intersection type, the dependencies within a signature may still prevent components being merged. For example, consider the following signatures:

$$S_4 = y_1 \in A_1, y_2 \in A_2(y_1) \text{ and } S_5 = y_2 \in B_2, y_1 \in B_1(y_2)$$

Note that in S_4, the type of y_2 is dependent on y_1; and in S_5, the type of y_1 is dependent on y_2. If we use $(*)$ to merge the y_1 components in S_4 and S_5 then we get $y_1 \in A_1 * B_1(y_2)$, so that y_1 is dependent on y_2. Similarly, merging both y_2 components gives us $y_2 \in B_2 * A_2(y_1)$ which is dependent on y_1. However, there is no way of ordering the new definitions of y_1 and y_2 so that y_1 is dependent on y_2, and y_2 is dependent on y_1. Therefore, even with an intersection type, it is not always possible to merge the types of two components in a signature.

We have shown that there is an interesting relationship between each signature operator, and the corresponding operator on computational elements: given a computational element $m \in S$, if f is a computational element operator, and F is the corresponding signature operator, then $f(m) \in F(S)$. In Chapter 9, we will show that this relationship allows specifications that are made using the signature operators to be implemented using the corresponding computational element operators; and this property will allow the piecewise implementation of structured specifications.

One disadvantage of using MLT to define signature and computational element operators is that the semantics of these operators is complicated. One reason for this complexity is the very simplicity of MLT, and in particular, its lack of pre-defined operations. Consequently, we have defined many auxiliary operators—such as tuple and loose signature concatenation—in order to define our signature and computational element operators. The need to define so many auxiliary operators is irritating, but as a library of such operators is developed, so the need to define more auxiliary operations will be reduced. The complexity of the semantics is exacerbated by the need to define operations twice—once for signatures, and once for computational elements. This is a consequence of specifications and modules having different semantics in MLT.

It is possible to describe the signature and computational element operators at a purely syntactic level. Such a description has the same complexity as the one given in this chapter, and still requires us to define the auxiliary operators given in this chapter so that we can reason about our signature and computational element operators.

In summary, we have presented the definition of several signature and computational element operators. These operators will be used to define specification and module structuring operators. We have also supplied laws concerning the basic properties of each operator, and these laws will be used to justify laws about specification and module operators.

Chapter 7

Structured Specifications

7.1 Introduction

Large specifications are as complex in their structure as programs. It is convenient to construct them, methodically, in small pieces which we then combine to make larger specifications. Such a style supports the modular decomposition of specifications into manageable pieces, and also helps to control the complexity arising from a large number of type, value and function symbols. Structuring a specification also helps to isolate its individual parts from changes to other parts of the specification; this minimises the amount of re-specification required when we change part of a specification. It is also more likely that we can reuse the individual parts of a structured specification, and this encourages the building of libraries of reusable specifications. In order to support the construction of structured specifications, a specification language requires structuring operators which can both modify and combine specifications to make new specifications. Our purpose in this chapter is to define some structuring operations on specifications, and consider their role in making structured specifications.

Our choice of specification operators is determined by how we intend to make specifications. We take the view that, where possible, specifications should be constructed in an incremental manner where we continually add detail to a specification until we are satisfied it specifies what we want. For example, we often wish to add new components and restrictions to an existing specification; so, we will require some form of enrichment operator. We may wish to make a specification by combining several independently developed specifications, so we will require some form of sum operator to combine the components of two specifications. We may also wish to reuse existing specifications to make new specifications; so, we will require operators to "customise" existing specifications to their new role by renaming, or hiding, some of their components, for example.

Several specification languages, such as CLEAR [5], ASL [51] and Z [52], supply specification operators that support the construction of structured specifications; readers familiar with such languages should find some of our specification operators familiar. However, unlike the specification languages mentioned above, our specification operators are defined in a type-theoretic framework. This approach confers certain benefits when implementing and deducing properties from structured specifications. For example, we can use the constructive logic available in type theory to prove properties of the operators. In Chapter 9, we will show that for some specification operators there exist corresponding module operators that we can use to implement structured specifications; this fact is a consequence of the fact that specifications are types, and their implementations are members of these types.

We will give some basic laws of the specification operators. These laws can be used to deduce the behaviour of a software system from its specification: we can formulate some required properties of a system and then use the laws to prove that its specification implies these properties. The laws can also be used to change the structure of a specification prior to, or during, its refinement and implementation: we may have to rearrange a specification if it has a structure that, if carried over to an implementation, yields an unacceptably inefficient implementation.

The formal definitions of some of the specification operators are more complicated than we might expect. We will show that the complexity of these definitions is necessary to ensure that the operators are well-typed.

This chapter is organised as follows. We begin by introducing our specification operators, and then give an example that shows how they can be used to incrementally construct specifications. Next, we give the formal semantics of each operator. Finally, we state and prove some basic properties of the specification operators.

7.2 The Specification Operators

In this section, we introduce a small collection of specification operators. Figure 7.1 contains a syntax summary of the specification operators. Some of the operators, such as *rename*, *hide* and *sum*, should be familiar to reader acquainted with algebraic specification languages, or Z, while other operators, such as *derive* and *translate*, may be less familiar. Note that we overload the signature hiding and renaming notation and use it to denote the *rename* and *hide* operators, respectively. We will give an informal description of the properties of each specification operator and consider how they address some of the issues discussed in the introduction. The operators defined here are not the only operators possible; however, we believe them to be a useful collection and they are sufficient for our purposes.

Rename.	$SP[p\backslash q]$	$p, q \in \mathbb{S}$
Hide.	$SP\backslash i$	$i \in Set(\mathbb{S})$
Union.	$SP_1 \cup SP_2$	$Sig(SP_1) = Sig(SP_2)$
Sum.	$SP_1 + SP_2$	
Enrich.	$SP \lhd \Lambda\langle S \mid R\rangle$	$Sig(SP) \vdash S$ **sig**; $[\![m \in Sig(SP) \oplus S \rhd R(m)$ **type**$]\!]$
Translate.	$SP \uparrow f$	$f \in S \to Sig(SP)$ and S **sig**
Derive.	$SP \downarrow f$	$f \in Sig(SP) \to S$ and S **sig**

Figure 7.1: The Specification Operators

In what follows, it may be helpful to think of specification operators as functions that take a number of arguments, including specifications, and return specifications as result. Examples of using the operators are left until Section 7.3; and the formal semantics of the specification operators are left until Section 7.4.

Renaming The *rename* operator allows a specifier to rename the components supplied by the signature of a specification. Renaming a component p to q in specification SP is written as $SP[p\backslash q]$. Renaming is a surprisingly useful operation. For example, renaming allows us to reuse an existing specification whose component names are not meaningful in the context in which it is reused. Renaming can also be used to prevent name-clashes when combining specifications—we shall see examples of this latter. Renaming is defined using signature renaming, and has similar properties to signature renaming.

Hiding The *hide* operator allows a specifier to modify an existing specification by making visible components local to the signature of a specification. An application of *hide* has the form $SP\backslash i$ for any specification SP and i a set of component names. $SP\backslash i$ is SP with all visible components named in i made local in SP. Intuitively, $SP\backslash i$ puts a bullet (•) in front of the declaration of any visible component of SP named in i. *hide* is defined using signature hiding.

Union The union operator merges two specifications, provided they have the same signature. The union of two specifications SP_1 and SP_2 is written as $SP_1 \cup SP_2$, assuming $Sig(SP_1) = Sig(SP_2)$. The specification $SP_1 \cup SP_2$ has the same signature as SP_1 and SP_2, but its restriction is the conjunction of the restrictions of SP_1 and SP_2. Consequently, the implementations of $SP_1 \cup SP_2$ are exactly those modules that satisfy both SP_1 and SP_2.

Sum The sum of two arbitrary specifications SP_1 and SP_2 is written as $SP_1 + SP_2$. The *sum* operator allows us to make individual specifications in isolation and then combine them to make a new specification. $SP_1 + SP_2$ supplies all the components supplied by SP_1 and SP_2: the signature of $SP_1 + SP_2$ is the signature concatenation of $Sig(SP_1)$ and $Sig(SP_2)$. The restriction of $SP_1 + SP_2$ is the conjunction of the restrictions of SP_1 and SP_2. Note that *sum* differs from *union* as it allows us to combine specifications with differing signatures. One of the issues that arises when defining *sum* is resolving the name-clashes that can occur if some component names in SP_1 are also used as component names in SP_2. *sum* resolves name-clashes in the same way as signature concatenation, by overloading any component name that appears in both its arguments. For example, if SP_1 and SP_2 both specify a component named y, then $SP_1 + SP_2$ specifies both those components; in other words, $SP_1 + SP_2$ supplies two, distinct, components named y.

Enrichment The *enrich* operator allows us to modify a specification by adding new components and restrictions. Enriching a specification SP with the components supplied by a signature S, and adding the restriction R, is written as $SP \lhd \Lambda\langle S \mid R\rangle$. The expression $\Lambda\langle S \mid R\rangle$ is called an *enrichment*. S is dependent on $Sig(SP)$ (i.e. $Sig(SP) \vdash S$ **sig** holds), so the components supplied by S may be dependent on the components of SP. The signature of $SP \lhd \Lambda\langle S \mid R\rangle$ is the signature concatenation of $Sig(SP)$ and S (i.e. $Sig(SP) \oplus S$). R is dependent on any $m \in Sig(SP) \oplus S$, so R may be defined using any components from SP and S. $\Lambda\langle S \mid R\rangle$ is not a specification since R depends on computational elements that meet $Sig(SP) \oplus S$, instead of those that meet S; this is the main difference between *sum* and *enrich*. We will sometimes write enrichements of the form $\Lambda\langle S \mid R\rangle$ as Λ**Elements** S **Restrictions** R **End**.

Derive An application of *derive* has the form $SP \downarrow f$ for any $f \in Sig(SP) \rightarrow S$ and S any signature. Functions such as f, that are used by *derive* are called *derive* maps. If we consider $SP \downarrow f$ as a type, its members are all $m \in SP$, but with f applied to the computational element of each m (i.e. $SP \downarrow f$ specifies any module whose computational element is in the set $\{f(ce(m)) | m \in SP\}$). For example, we may use the computational element renaming operator $\_[p\backslash q] \in Sig(SP) \rightarrow Sig(SP)[p\backslash q]$ as a *derive* map, to produce the specification $SP \downarrow (\_[p\backslash q])$ that is equivalent to $SP[p\backslash q]$. *derive* is a versatile operator: with appropriate choices of *derive* map, *derive* can be used to permute, rename, and remove components from the signature of a specification. *derive* can also be used to change the type of components in the signature of a specification. The only conditions on f are that its domain is $Sig(SP)$ and its range is a signature; the range of f is the signature of $SP \downarrow f$. In Section 7.3, we will describe how *derive* can be used to remove components from a specification. For readers familiar with algebraic specifications, *derive* is a generalisation of the Derive operation in the specification language ASL [51].

Translate An application of *translate* has the form $SP \uparrow f$ for any $f \in S \to Sig(SP)$ and S any signature. Functions such as f, that are used by *translate* are called *translate* maps. $SP \uparrow f$ specifies any module whose computational element meets S and maps, under f, to a computational element that satisfies the restriction of SP (i.e. $SP \uparrow f$ specifies any module whose computational element is in the set $\{m \in S | Ax(SP)(f(m))\}$). For example, if f is the inverse computational element renaming operator $\_[p\backslash q]^{-1}_{Sig(SP)} \in Sig(SP)[p\backslash q] \to Sig(SP)$ then $SP \uparrow f$ is equivalent to $SP[p\backslash q]$. With appropriate choices of *translate* map, *translate* can be used to permute, rename, or insert components into a specification. The reader might be forgiven for thinking that *derive* and *translate* are so similar that we can do without one. However, there are many transformations that can be defined by *derive*, but not *translate*, and vice-versa: we illustrate this in an example, later. For those readers familiar with algebraic specifications, *translate* is a generalisation of the operation Translate defined in [50].

7.3 An Example Structured Specification

In this section, we give an example of constructing a structured specification. As our example, we specify some modules that might be used as part of a book library system. We intend to use the specifications *Catalogue* and *BookSpec* given previously in Figures 1.2 and 1.4, respectively. Recall that *Catalogue* supplies a type named *Stock* which is used to represent collections of books; *Catalogue* also supplies operations to add books to *Stock* values; remove books from *Stock* values; and also includes operations to query the availability of a book. *BookSpec* supplies a type named *Book*, together with operations on *Book* values; for example, there is an operation to create new book values, as well as operations to query the title and author of a book. We show how the structuring operators can be used to construct a new specification of a book catalogue in an incremental style. We also show how the specification operators facilitate the reuse of *Catalogue* and *BookSpec* to specify a module for a register of library users. Our aim is only to illustrate the use of the specification operators and we do not present a complete specification of a library system.

7.3.1 A Book Catalogue

We begin by considering how to combine *Catalogue* and *BookSpec* to produce a new catalogue specification supplying the components in *Catalogue* and *BookSpec*. Inspection of *Catalogue* and *BookSpec* reveals that they both supply a type named *Book*. When we combine *Catalogue* and *BookSpec*, we would like to ensure that the values of the type *Book* specified by *Catalogue* are compatible with the values of *Book* specified by *BookSpec*. Let us consider trying to combine *Catalogue* and *BookSpec* using the *sum* operator. If we sum *Catalogue* and *BookSpec* directly, the resultant specification will contain all the components of *Catalogue* and *BookSpec*, including two versions of the type *Book* (i.e. *Book* is overloaded). We avoid overloading *Book*—for reasons that will be made clear below—by renaming the component *Book* in *BookSpec* to $Book_2$ prior to summing *Catalogue* and *BookSpec*:

$$Catalogue_2 = Catalogue + BookSpec[Book \backslash Book_2]$$

$Catalogue_2$ does not quite specify the catalogue module we desire, as it does not specify that values of type *Book* and $Book_2$ are compatible: compatibility means that *Book* values can be used in place of $Book_2$ values, and vice versa. Therefore, we use the *enrich* operator to add the constraint that *Book* and $Book_2$ should be equal:

$$Catalogue_3 = Catalogue_2 \lhd \Lambda\langle \Phi \mid Book =_{U_1} Book_2 \rangle$$

Note that the signature of the enrichment is Φ as we do not wish to add new components to $Catalogue_2$. If we had not renamed *Book* to $Book_2$ then we would be unable to specify the enrichment above. This is because *Book* is overloaded in *Catalogue* + *BookSpec* and, hence, the version of *Book* from *Catalogue* is not in scope in the restriction of *Catalogue* + *BookSpec*, or any enrichment of the restriction (the scope rules for overloaded names establish that *Book*, from *BookSpec*, is in scope, rather than *Book* from *Catalogue*).

Since $Book_2$ is equivalent to *Book*, we choose to hide $Book_2$ so that users of $Catalogue_3$ can only use *Book* as the type of books; $Book_2$ is relegated to an internal role.

$$Catalogue_4 = Catalogue_3 \backslash \{Book_2\}$$

$Book_2$ is a local component in $Catalogue_4$, but this does not cause any problems with the use of $Catalogue_4$. $Book_2$ can be removed entirely, and the removal of components is discussed in Section 7.3.3.

$Catalogue_4$ concludes the specification of the book catalogue. Using the specification operators, we have been able to construct the book catalogue in an incremental style, starting with *Catalogue*, and finishing with $Catalogue_4$—we say more about the incremental construction of specifications in Section 7.3.4.

Although $Catalogue_4$ is a structured specification, it can still be "flattened" into a canonical specification. In practice, the structure of a specification should always be maintained, as it makes the specification easier to understand. However, to help confirm the readers intuition about the specification operators used in $Catalogue_4$, we unfold $Catalogue_4$ to an equivalent—under specification equality ($\Leftrightarrow$)—canonical specification named $Catalogue_5$ (i.e. $Catalogue_4 \Leftrightarrow Catalogue_5$). $Catalogue_5$ is given in Figure 7.2. We omit the details of producing $Catalogue_5$, but the reader should be aware that work needs to be done to unfold $Catalogue_4$ to $Catalogue_5$-we unfold $Catalogue_4$ to $Catalogue_5$ using theorems given later, in Section 7.5.

$Catalogue_5 \equiv$

Elements

$Book \in U_1,$

$Stock \in U_1,$

$empty \in Stock,$

$add \in Book \rightarrow Stock \rightarrow Stock,$

$remove \in Book \rightarrow Stock \rightarrow Stock,$

$instock \in Book \rightarrow Stock \rightarrow \mathbb{B},$

$isempty \in Stock \rightarrow \mathbb{B},$

$\bullet Book_2 \in U_1,$

$mkBook \in Author \rightarrow Title \rightarrow Id \rightarrow Book_2,$

$author \in Book_2 \rightarrow Author,$

$title \in Book_2 \rightarrow Title,$

$bookId \in Book_2 \rightarrow Id$

Restrictions

$\forall s \in Stock.\forall m, n \in Book.$

$isempty(empty) =_{\mathbb{B}} \text{true} \quad \wedge$

$isempty(add(m, s)) =_{\mathbb{B}} \text{false} \quad \wedge$

$instock(m, empty) =_{\mathbb{B}} \text{false} \quad \wedge$

$instock(m, add(m, s)) =_{\mathbb{B}} \text{true} \quad \wedge$

$\neg(m =_{Book} n) \Rightarrow instock(n, add(m, s)) =_{\mathbb{B}} instock(n, s) \quad \wedge$

$remove(m, empty) =_{Stock} empty \quad \wedge$

$remove(m, add(m, s)) =_{Stock} remove(m, s) \quad \wedge$

$\neg(m =_{Book} n) \Rightarrow remove(n, add(m, s)) =_{Stock} add(m, remove(n, s))$

$\wedge$

$\forall a \in Author.\forall t \in Title.\forall i \in Id.$

$author(mkBook(a, t, i)) =_{Author} a \wedge$

$title(mkBook(a, t, i)) =_{Title} t \wedge$

$bookId(mkBook(a, t, i)) =_{Id} i$

$\wedge$

$Book =_{U_1} Book_2$

End

Figure 7.2: Expansion of $Catalogue_4$

7.3.2 A Register of Library Users

If we continue to develop a library specification, then we might wish to specify a module for a register of library users. Such a specification would be very similar to *Catalogue*, as we would require operations to add and remove library users to and from a register. We would also require operations to query the status of a register and library users. One possible specification of a register module may be defined by simply renaming some of the components of *Catalogue*, giving the following specification:

$$\begin{aligned} Register = Catalogue[\ & Book\backslash Person, Stock\backslash Users, add\backslash addUsers, \\ & remove\backslash RemoveUsers, instock\backslash registered\ ] \end{aligned}$$

In practice, when we reuse a specification, such as *Catalogue*, it may contain components that we don't require, and these may be hidden using *hide*, or *derive*. Additionally, we may want to add some new components defined in terms of existing components, and these may be added using the *enrich* operator. For example, suppose we require an extra operation within a register module, so that we can calculate the number of registered users. Such an operation may be specified as an enrichment of *Register*:

```
Register_1 = Register ◁ ΛElements
                          size ∈ Users → ℕ
                        Restrictions
                          ∀s ∈ Users.∀m ∈ Person.
                          size(empty) = 0 ∧
                          size(addUser(m, s)) = if registered(m, s) then
                                                  size(s)
                                                else
                                                  size(s) + 1
                        End
```

In any register, we might keep the name, address and personal identifier of each library user. So we reuse *BookSpec* to specify a module containing a type, named *Person*, that is used to represent library users:

$$\begin{aligned} PersonSpec = BookSpec[\ & Book\backslash Person, mkBook\backslash mkPerson, \\ & author\backslash name, title\backslash address, bookId\backslash personId\ ] \end{aligned}$$

We now aim to produce a specification that specifies all the components supplied by *Register* and *PersonSpec*, such that the type *Person* in *Register* is compatible with the type *Person* in *PersonSpec*, and vice versa. There are several ways of combining *Register* and *PersonSpec* to specify a register of library users, and in the following we consider two possible solutions.

$$
\begin{aligned}
S = {} & \mathit{Person} \in U_1, \\
& \mathit{Users} \in U_1, \\
& \mathit{empty} \in \mathit{Users}, \\
& \mathit{addUsers} \in \mathit{Person} \to \mathit{Users} \to \mathit{Users}, \\
& \mathit{removeUsers} \in \mathit{Person} \to \mathit{Users} \to \mathit{Users}, \\
& \mathit{registered} \in \mathit{Person} \to \mathit{Users} \to \mathbb{B}, \\
& \mathit{isempty} \in \mathit{Users} \to \mathbb{B} \\
& \mathit{mkPerson} \in \mathit{Name} \to \mathit{Address} \to \mathit{PId} \to \mathit{Person}, \\
& \mathit{name} \in \mathit{Person} \to \mathit{Name}, \\
& \mathit{address} \in \mathit{Person} \to \mathit{Address}, \\
& \mathit{personId} \in \mathit{Person} \to \mathit{PId}
\end{aligned}
$$

Figure 7.3: A Signature S

The task of combining *Register* and *PersonSpec* is similar to that of combining *Catalogue* and *BookSpec*. So, one solution is to use the same style of combination used to construct $\mathit{Catalogue}_3$:

$$
\begin{aligned}
& \mathit{PersonRegister}_1 \equiv \\
& \quad (\mathit{Register} + \mathit{PersonSpec}[\mathit{Person} \backslash \mathit{Person}_2]) \lhd \Lambda\langle \Phi \mid \mathit{Person} =_{U_1} \mathit{Person}_2 \rangle
\end{aligned}
$$

Our main criticism of $\mathit{PersonRegister}_1$ is that Person_2 is redundant since it is equivalent to *Person*. If we use this style of combination too often then we can end up with specifications that are "cluttered" with many redundant components.

It is possible to combine *Register* and *PersonSpec* to produce a specification in which Person_2 and *Person* are merged into a single component, and we now describe how to make such a specification. We will use *translate* to transform *Register* and *PersonSpec* into specifications with a common signature and then merge those specifications using the *union* operator. The common signature is called S, and is given in Figure 7.3. $[\![S]\!]$ **sig** holds, and we use S as a type (i.e. as $[\![S]\!]$). S contains all the components of *Register* and *PersonSpec*, but *Person* only appears once.

In order to use *translate* to transform the signatures of *Register* and *PersonSpec* to S, we require the functions

$$
\begin{aligned}
f & \in S \to \mathit{Sig}(\mathit{Register}) \text{ and} \\
g & \in S \to \mathit{Sig}(\mathit{PersonSpec})
\end{aligned}
$$

given in Figures 7.4 and 7.5, respectively. Function f takes any computational element $m \in S$ and removes all the components of m that are not specified by *Register*; the result is a computational element of type $\mathit{Sig}(\mathit{Register})$. Similarly, g throws away any components not specified by *PersonSpec*, to produce a computational element of type $\mathit{Sig}(\mathit{PersonSpec})$. We use f and g to make the following specifications:

$$
\begin{aligned}
\mathit{Register}_2 & = \mathit{Register} \uparrow f \\
\mathit{PersonSpec}_2 & = \mathit{PersonSpec} \uparrow g
\end{aligned}
$$

```
f = λm ∈ S.module
                Person = m.Person,
                Users = m.Users,
                addUsers = m.addUsers,
                removeUsers = m.removeUsers,
                isempty = m.isempty,
                registered = m.registered
            end
```

Figure 7.4: A function $f \in S \rightarrow Sig(Register)$

```
g = λm ∈ S.module
                Person = m.Person,
                mkPerson = m.mkPerson,
                name = m.name,
                address = m.address,
                personId = m.personId
            end
```

Figure 7.5: A function $g \in S \rightarrow Sig(PersonSpec)$

Let us consider *Register*$_2$. The application of *translate* to *Register* changes the signature of *Register* to S by adding extra components *mkPerson*, *name*, *address* and *personId*. The restrictions of *Register*$_2$ and *Register* are equivalent. Consequently, the components of *Register*$_2$ that are common to *Register* have the same behaviour as the components of *Register*. The components of *Register*$_2$ which are common to *PersonSpec* are left unconstrained. A more technical description of *Register*$_2$ is that it is a type containing modules whose computational elements are in the set $\{m \in S \mid Ax(Register)(f(m))\}$; f "translates" each $m \in S$ to $f(m) \in Register$ in order to verify that the components of m that are specified by *Register* satisfy the restriction of *Register*. *PersonSpec*$_2$ is defined in a similar style to *Register*$_2$. *PersonSpec*$_2$ has signature S, and has the same restriction as *PersonSpec*.

Finally, we take the union of *Register*$_2$ and *PersonSpec*$_2$—both have signature S—to get a specification with signature S, and with a restriction that is equivalent to the conjunction of the restrictions of *Register* and *PersonSpec*:

$$PersonRegister_2 = Register_2 \cup PersonSpec_2$$

PersonRegister$_2$ specifies all the components of *Register* and *PersonSpec*, but it only contains a single version of the type *Person*.

The above method of combining specifications is more complicated than using the *sum* operator which is usually more appropriate for combining specifications. Some algebraic specification languages provide an operator, called amalgamated sum, that combines specifications in a similar manner to that described above.

To make a complete specification of a library system, we would expect to combine $\mathit{Catalogue}_3$ with either $\mathit{PersonRegister}_1$ or $\mathit{PersonRegister}_2$. We would also expect to add other specifications such as a loans module, for example. However, we do not pursue any of these additions.

7.3.3 The Book Catalogue Revisited

We may regain a specification similar to *Catalogue* from $\mathit{Catalogue}_3$ by using the *hide* operator ($\backslash$). For example, consider the following specification:

$$\mathit{Catalogue}_6 = \mathit{Catalogue}_3 \backslash \{\mathit{mkBook}, \mathit{author}, \mathit{title}, \mathit{bookId}\}$$

The visible signature of $\mathit{Catalogue}_6$ contains the components defined in *Catalogue*. However, $\mathit{Catalogue}_6$ is not equal to *Catalogue*: the signature of $\mathit{Catalogue}_6$ contains local components and its restriction is still the conjunction of the restrictions of *Catalogue* and *BookSpec*.

The signature of $\mathit{Catalogue}_6$ contains local components because when we hide components they are not removed, but become local components. However, we can use the *derive* ($\downarrow$) operator to remove components from specifications. For example, we will describe how to use *derive* to remove *mkBook*, *author*, *title* and *bookId* from $\mathit{Catalogue}_3$. We require the function

$$h \in \mathit{Sig}(\mathit{Catalogue}_3) \nrightarrow \mathit{Sig}(\mathit{Catalogue})$$

defined in Figure 7.6. Function h takes any computational element satisfying the signature of $\mathit{Catalogue}_3$, and removes the components *mkBook*, *author*, *title* and *bookId* to yield a computational element satisfying $\mathit{Sig}(\mathit{Catalogue})$. We use h, in conjunction with *derive*, to make $\mathit{Catalogue}_7$ (below) which is $\mathit{Catalogue}_3$ with the components *mkBook*, *author*, *title* and *bookId* removed:

$$\mathit{Catalogue}_7 = \mathit{Catalogue}_3 \downarrow h$$

The signature of $\mathit{Catalogue}_7$ is identical to the signature of *Catalogue*, and the restriction of $\mathit{Catalogue}_7$ is equivalent to the restriction of $\mathit{Catalogue}_3$. A more technical description of $\mathit{Catalogue}_7$ is that it is a type containing all modules satisfying $\mathit{Catalogue}_3$, but with h applied to their computational elements to remove the components *mkBook*, *author*, *title* and *bookId* (i.e. $\mathit{Catalogue}_7$ is a type containing modules whose computational elements are in the set $\{h(ce(m)) \mid m \in \mathit{Catalogue}_3\}$).

$\mathit{Catalogue}_6$ and $\mathit{Catalogue}_7$ illustrate the choice we have when discarding components: components can be made local, or they can be removed. We prefer to hide components by using *hide* rather than remove them using *derive*. Firstly, *hide* is easier to use than *derive*, as *hide* does not require us to supply an extra function, such as h. Secondly, if a component is made local then it can still be used to help specify other components in a specification. In contrast, we cannot remove a component from a specification if

```
h = λm ∈ Sig(Catalogue_3).
      module
         Book = m.Book,
         Stock = m.Stock,
         empty = m.empty,
         remove = m.remove,
         instock = m.instock,
         isempty = m.isempty
      end
```

Figure 7.6: A function $h \in \mathit{Sig}(\mathit{Catalogue}_3) \rightarrow \mathit{Sig}(\mathit{Catalogue})$

the component is used to give the type of other components in the signature of the specification. However, if a component is only used in the restriction of a specification, then it can be removed using *derive* without affecting the behaviour of the remaining components in the specification; this fact may seem surprising, but it is justified by the semantics of *derive*.

7.3.4 Discussion of the Example

We have shown how a suitable collection of specification operators can be used to systematically construct specifications. We have employed an incremental style of specification construction to make specifications for both a book catalogue and a library register. An incremental style of specification construction is particularly suited for the construction of large specifications. The operators *enrich* and *sum* play a key role in allowing us to make specifications in an incremental style. For example, we used *Catalogue* as an initial framework for $\mathit{Catalogue}_4$, and most of the other parts of $\mathit{Catalogue}_4$ were added using *enrich* and *sum*. The *rename* operator also played a useful role by allowing us to reuse *Catalogue* to obtain a starting point for the construction of $\mathit{PersonRegister}_1$ and $\mathit{PersonRegister}_2$.

There are several advantages to an incremental approach to specification construction. Firstly, an incremental approach simplifies the specification task. Secondly, it is easier to modify a specification if its requirements change. For example, if we decide to change $\mathit{Catalogue}_4$ then the steps $\mathit{Catalogue}$–$\mathit{Catalogue}_3$ can still be used to make a new version of $\mathit{Catalogue}_4$. Changing $\mathit{Catalogue}_4$ may still require us to re-specify $\mathit{Catalogue}_6$ and $\mathit{Catalogue}_7$ which depend on $\mathit{Catalogue}_4$, but this is preferable to completely re-specifying the catalogue. The individual steps in the development of a specification can also be reused to make other specifications. Typically, we only reuse complete specifications such as $\mathit{Catalogue}_4$. However, incremental construction also makes the intermediate steps $\mathit{Catalogue}$–$\mathit{Catalogue}_3$ available for reuse.

The library example also raises other issues. These include the way in which we choose the structure of a specification; and the use of parameterisation to construct specifications. For example, combining *Catalogue* with *BookSpec* may have been more suited to a parameterised approach in which we parameterised *Catalogue* with respect to *BookSpec*. We discuss these issues later.

In summary, we have shown how a small collection of specification operators can be used to construct specifications in an incremental style.

7.4 The Semantics of Specification Operators

In this section, we give the formal definitions of the specification operators. Semantically, specification operators are transformations that take a list of arguments, including specifications, and return a specification as result. The specification operators are defined in terms of the signature and computational element operators defined in the previous chapter. In the following, we will write specifications in MLT, such as $\sum m \in S.R(m)$, in the form $\langle S \mid (m)R \rangle$.

7.4.1 Rename

The definition of the renaming operation on specifications is given in Definition 7.1, below, with an explanation following. $S[p\backslash q]$ is signature S with component p renamed to q. Note that the restriction $R(m[p\backslash q]_S^{-1})$ is expressed using the inverse computational element renaming operator $\_[p\backslash q]_S^{-1} \in S[p\backslash q] \to S$. The restriction $R(m[p\backslash q]_S^{-1})$ expresses the requirement that $m \in S[p\backslash q]$ must satisfy the restriction of $\langle S \mid (m)R \rangle$ if we rename component q (in m) back to p. In other words, the components supplied by $\langle S \mid (m)R \rangle[p\backslash q]$ and $\langle S \mid (m)R \rangle$ must be identical except that component p in $\langle S \mid (m)R \rangle$ is named q in $\langle S \mid (m)R \rangle[p\backslash q]$.

Definition 7.1 (*Rename*)
Given any $\langle S \mid (m)R \rangle$ **spec**, and component names $p, q \in \mathbb{S}$ then

$$\langle S \mid (m)R \rangle[p\backslash q] = \langle S[p\backslash q] \mid (m)R(m[p\backslash q]_S^{-1}) \rangle$$

□

At first sight, $\langle S[p\backslash q] \mid (m)R(p\backslash q) \rangle$ seems to be an alternative definition of $\langle S \mid (m)R \rangle[p\backslash q]$ where $R(p\backslash q)$ denotes the textual substitution of component name p by q, in R. However, the substitution $R(p\backslash q)$ may do more than just rename p to q; it may actually change the intended behaviour of the specification. For example, consider the specification $SP = \langle S \mid (m)\text{“}p\text{”} =_{\mathbb{S}} m.\text{“}p\text{”} \rangle$; we assume SP supplies a component $p \in \mathbb{S}$. If we rename p to q using the alternative definition of renaming, proposed above, we get the specification $SP^* = \langle S[p\backslash q] \mid (m)\text{“}q\text{”} =_{\mathbb{S}} m.\text{“}q\text{”}\rangle$. Note that the restriction of SP^* expresses that the component q is the string "q". This is wrong, since renaming a component must not change its behaviour, i.e. component q should be specified to be the string "p". The example is contrived, but it does illustrates that the alternative definition of renaming is flawed.

Remark If $(m)R(p\backslash q)$ is equivalent to $(m)R(m[p\backslash q]_S^{-1})$ with respect to weak type equality (i.e. $(m)R(p\backslash q) \Leftrightarrow (m)R(m[p\backslash q]_S^{-1}))$ then we can use $(m)R(p\backslash q)$ in place of the restriction of $\langle S \mid (m)R\rangle[p\backslash q]$. We discuss this property in Section 7.5. □

7.4.2 Hide

We define *hide* in the same style as we defined *rename*. The definition of *hide* is given in Definition 7.2 with an explanation following.

Definition 7.2 (*Hide*)
Given any $\langle S \mid (m)R\rangle$ **spec** and a set of identifiers $i \in Set(\mathbb{S})$ then

$$\langle S \mid (m)R\rangle\backslash i = \langle S \backslash i \mid (m)R(m\backslash_S^{-1} i)\rangle$$

□

The expression $S\backslash i$ denotes the application of signature hiding to S. The expression $m\backslash_S^{-1} i$ denotes the application of the inverse computational element hiding operator $\_\backslash_S^{-1} i \in S\backslash i \to S$ to $m \in S\backslash i$. Viewing $\langle S \mid (m)R\rangle\backslash i$ as a type, its members are modules whose computational elements are in the set $\{ce(x) \backslash i \mid x \in \langle S \mid (m)R\rangle\}$.

It might seem reasonable to define $\langle S \mid (m)R\rangle \backslash i$ as $\langle S \backslash i \mid (m)R\rangle$. However, $R(m)$ may be ill-typed since $R(m)$ is defined under the assumption that $m \in S$, so it may not be well-typed for $m \in S\backslash i$. If $R(m)$ is well-typed for all $m \in S\backslash i$, and is equivalent to the restriction of $\langle S \mid (m)R\rangle \backslash i$ then we can use R in place of the restriction of $\langle S \mid (m)R\rangle \backslash i$; this property is discussed in Section 7.5.

7.4.3 Union

The *union* operator is defined in terms of the product type constructor ($\times$). *Union* uses $\times$ to combine the restrictions of two specifications:

Definition 7.3 (*Union*)
Given any $\langle S_1 \mid (m)R_1\rangle$ **spec** and $\langle S_2 \mid (m)R_2\rangle$ **spec** such that $S_1 = S_2$, then

$$\langle S_1 \mid (m)R_1\rangle \cup \langle S_2 \mid (m)R_2\rangle = \langle S_1 \mid (m)R_1(m) \times R_2(m)\rangle$$

□

Note that we apply R_2 to $m \in S_1$ even though R_2 is dependent on all $m \in S_2$: we can apply R_2 to all $m \in S_1$ since $S_1 = S_2$.

7.4.4 Derive

The definition of *derive* is given in Definition 7.4 with justification following. $\langle S_1 \mid (m_1)R\rangle \downarrow f$ (where $f \in S_1 \to S_2$) is a type whose members are all modules $x \in \langle S_1 \mid (m_1)R\rangle$ with f applied to their computational elements. In other words, the members of $\langle S_1 \mid (m_1)R\rangle \downarrow f$ are modules whose computational elements are in the set $\{f(ce(x)) \mid x \in \langle S_1 \mid (m_1)R\rangle\}$. This set can also be expressed as:

$$\{m_2 \in S_2 \mid \exists x \in \langle S_1 \mid (m_1)R\rangle . f(ce(x)) = m_2\}$$

This set, when expressed as a type, justifies the definition of *derive* given below.

Definition 7.4 (*Derive*)
Given any $\langle S_1 \mid (m_1)R\rangle$ **spec**, S_2 **sig**, and a function $f \in S_1 \to S_2$, then

$$\langle S_1 \mid (m_1)R\rangle \downarrow f = \langle S_2 \mid (m_2)\exists x \in \langle S_1 \mid (m_1)R\rangle . [f(ce(x)) =_{S_2} m_2]\rangle$$

□

7.4.5 Translate

The formal definition of *translate* is given in Definition 7.5 with explanation following. $\langle S_1 \mid (m)R\rangle \uparrow f$ (where $f \in S_2 \to S_1$) is a type whose members are modules, such that for all $x \in \langle S_1 \mid (m)R\rangle \uparrow f$, x can be transformed into a module satisfying $\langle S_1 \mid (m)R\rangle$ by applying f to the computational element of x. In other words, for all $x \in \langle S_1 \mid (m)R\rangle \uparrow f$, the restriction $R(f(ce(x)))$ is true where $ce(x) \in S_2$ and $f(ce(x)) \in S_1$. The set $\{m \in S_2 \mid R(f(m))\}$ contains the computational elements of all modules satisfying $\langle S_1 \mid (m)R\rangle \uparrow f$. This set, when expressed as a type, justifies the definition of *translate* given below.

Definition 7.5 (*Translate*)
Given any $\langle S_1 \mid (m)R\rangle$ **spec**, S_2 **sig** and a function $f \in S_2 \to S_1$, then

$$\langle S_1 \mid (m)R\rangle \uparrow f = \langle S_2 \mid (m)R(f(m))\rangle$$

□

As a point of interest, both *rename* and *hide* are special cases of *translate*. *rename* can be defined using *translate* by choosing the inverse renaming operator on computational elements as a *translate* map; and *hide* can be defined by choosing the inverse hiding operator on computational elements as a *translate* map:

Fact 7.1 $\langle S \mid (m)R\rangle[p\backslash q] = \langle S \mid (m)R\rangle \uparrow (\lambda m \in S[p\backslash q].\ m[p\backslash q]_S^{-1})$ □

Fact 7.2 $\langle S \mid (m)R\rangle \setminus i = \langle S \mid (m)R\rangle \uparrow (\lambda m \in S\backslash i.\ m\backslash_S^{-1} i)$ □

7.4.6 Sum

The definition of the *sum* operator is given in Definition 7.6 with explanation following. The signature of $\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle$ is $S_1 \oplus S_2$ which is the signature concatenation of S_1 and S_2. Consequently, the restriction of $\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle$ depends on computational elements with type $S_1 \oplus S_2$. Note that the restriction of $\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle$ is defined using the projection functions ${}^{S_1S_2}_{\Pi_1}$ and ${}^{S_1S_2}_{\Pi_2}$. Recall that given any $m = (m_1 \oplus m_2) \in S_1 \oplus S_2$ where $m_1 \in S_1$ and $m_2 \in S_2$,

$$\begin{array}{lllll} {}^{S_1S_2}_{\Pi_1}(m) & = & m_1 & \in & S_1 \text{ and ;} \\ {}^{S_1S_2}_{\Pi_2}(m) & = & m_2 & \in & S_2 \end{array}$$

Let us consider the restriction of $\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle$:

$$(m)R_1({}^{S_1S_2}_{\Pi_1}(m)) \times R_2({}^{S_1S_2}_{\Pi_2}(m)) \;\text{ where } m \in (S_1 \oplus S_2)$$

${}^{S_1S_2}_{\Pi_1}(m) \in S_1$ is a computational element containing the components in m that are specified by S_1; ${}^{S_1S_2}_{\Pi_2}(m) \in S_2$ is a computational element containing the components in m that are specified by S_2. Therefore, $R_1({}^{S_1S_2}_{\Pi_1}(m))$ expresses the requirement that the components in m that are specified by S_1 must satisfy R_1; $R_2({}^{S_1S_2}_{\Pi_2}(m))$ expresses the requirement that the components in m that are specified by S_2 must satisfy R_2.

Definition 7.6 (Sum)
Given any $\langle S_1 \mid (m)R_1\rangle$ **spec** and $\langle S_2 \mid (m)R_2\rangle$ **spec** then

$$\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle = \langle (S_1 \oplus S_2 \mid (m)R_1({}^{S_1S_2}_{\Pi_1}(m)) \times R_2({}^{S_1S_2}_{\Pi_2}(m))\rangle$$

□

Remark It might seem that we can give a simpler definition of *sum* by defining the restriction of $\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle$ to be $R_1(m) \times R_2(m)$ ($m \in S_1 \oplus S_2$). However, this restriction may be ill-typed for some restrictions R_1 and R_2. This is because R_1 and R_2 depend on computational elements of type S_1 and S_2, respectively, so both $R_1(m)$ and $R_2(m)$ may be ill-typed for some, or all, $m \in S_1 \oplus S_2$. If $R_1(m) \times R_2(m)$ is well-typed for all $m \in S_1 \oplus S_2$ then we can use it to simplify the restriction of $\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle$; this point is discussed in Section 7.5 □

As a point of interest, *sum* can be expressed using the *union* and *translate* operations:

Theorem 7.1 $\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle = \langle S_1 \mid (m)R_1\rangle \uparrow {}^{S_1S_2}_{\Pi_1} \cup \langle S_2 \mid (m)R_2\rangle \uparrow {}^{S_1S_2}_{\Pi_2}$ □

Proof

$$\begin{array}{ll} & \langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle \\ = & \text{“definition of } \textit{sum}\text{”} \\ & \langle (S_1 \oplus S_2 \mid (m)R_1({}^{S_1S_2}_{\Pi_1}(m)) \times R_2({}^{S_1S_2}_{\Pi_2}(m))\rangle \\ = & \text{“definition of } \textit{union}\text{”} \\ & \langle (S_1 \oplus S_2 \mid R_1({}^{S_1S_2}_{\Pi_1}(m))\rangle \cup \langle (S_1 \oplus S_2 \mid (m)R_2({}^{S_1S_2}_{\Pi_2}(m))\rangle \\ = & \text{“type of } {}^{S_1S_2}_{\Pi_1} \text{ and } {}^{S_1S_2}_{\Pi_2}\text{, definition of } \textit{translate}\text{”} \\ & \langle S_1 \mid (m)R_1\rangle \uparrow {}^{S_1S_2}_{\Pi_1} \cup \langle S_2 \mid (m)R_2\rangle \uparrow {}^{S_1S_2}_{\Pi_2} \end{array}$$

□

7.4.7 Enrich

The formal definition of *enrich* is given in Definition 7.7 with an explanation following. Consider the specification $\langle S_1 \mid (m)R_1\rangle \lhd \Lambda\langle S_2 \mid R_2\rangle$. S_2 is a dependent signature that supplies new components to be added to $\langle S_1 \mid (m)R_1\rangle$. The components supplied by S_2 may be dependent on the components of S_1, i.e. $S_1 \vdash S_2$ **sig** holds. The signature of $\langle S_1 \mid (m)R_1\rangle \lhd \Lambda\langle S_2 \mid R_2\rangle$ is $S_1 \oplus S_2$; recall that $\oplus$ allows S_2 to be dependent on S_1. The restriction R_2 depends on the components supplied by S_1 and S_2, i.e. R_2 depends on any $m \in S_1 \oplus S_2$. Let us consider the restriction of $\langle S_1 \mid (m)R_1\rangle \lhd \Lambda\langle S_2 \mid R_2\rangle$:

$$(m)R_1(\overset{S_1 S_2}{\Pi_1}(m)) \times R_2(m) \text{ where } m \in S_1 \oplus S_2$$

$R_1(\overset{S_1 S_2}{\Pi_1}(m))$ expresses that the components of m that are specified by S_1 (i.e. the components in computational element $\overset{S_1 S_2}{\Pi_1}(m)$) must satisfy R_1. $R_2(m)$ expresses that all the components in m must satisfy R_2.

Definition 7.7 (Enrich)
Given any $\langle S_1 \mid (m)R_1\rangle$ **spec**, a dependent signature S_2 such that $S_1 \vdash S_2$ **sig** holds, and a restriction R_2 such that $|[m \in (S_1 \oplus S_2) \rhd R_2(m)\ \mathbf{type}]|$ then

$$\langle S_1 \mid (m)R_1\rangle \lhd \Lambda\langle S_2 \mid R_2\rangle = \langle S_1 \oplus S_2 \mid (m)R_1(\overset{S_1 S_2}{\Pi_1}(m)) \times R_2(m)\rangle$$

□

The definitions of *enrich* and *sum* differ only because *enrich* defines R_2 to be dependent on all $m \in S_1 \oplus S_2$ rather than all $m \in S_2$. We can express *sum* in terms of *enrich*, and *enrich* can be expressed in terms of the *union* and *translate* operators:

Fact 7.3 $\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle = \langle S_1 \mid (m)R_1\rangle \lhd \Lambda\langle S_2 \mid R_2(\overset{S_1 S_2}{\Pi_2}(m))\rangle$ □

Fact 7.4 $\langle S_1 \mid (m)R_1\rangle \lhd \Lambda\langle S_2 \mid R_2\rangle = \langle S_1 \mid (m)R_1\rangle \uparrow \overset{S_1 S_2}{\Pi_1} \cup \langle S_1 \oplus S_2 \mid (m)R_2(m)\rangle$ □

7.4.8 Summary

We have defined the formal semantics of each specification operator. We note that the definitions of some of the operators are more complicated than we might have expected.

7.5 Unfolding Specification Operators

In Section 7.4, we remarked that we can sometimes simplify the restrictions of specifications that are made using *rename*, *hide* or *sum*, and in this section we give some laws to do just that. In the following, if we say that two specifications are equivalent, we mean that they are equivalent with respect to specification equality ($\Leftrightarrow$);

for example, if we say SP_1 is equivalent to SP_2 we mean $SP_1 \Leftrightarrow SP_2$. If we say two restrictions (or propositions) are equivalent, we mean equivalence with respect to weak type equality $\Leftrightarrow$.

We begin by considering Theorem 7.2 which shows how to simplify the restrictions of specifications that are made using *rename*. The pre-condition in Theorem 7.2 expresses the requirement that the textual substitution $R(p\backslash q)$ must not change the behaviour of p—and the other components supplied by $\langle S \mid (m)R\rangle$—after p is renamed to q in R. In Section 7.4.1, we showed that the pre-condition may not hold if q is used as a string in the restriction $(m)R$. Apart from that situation, the pre-condition is true for all specifications that only use m in $(m)R$ to refer to components in m by dot notation. Such specifications include all canonical specifications, and all structured specifications that can be unfolded to canonical specifications.

Theorem 7.2 (Unfolding Rename)
Given $\langle S \mid (m)R\rangle$ **spec** and $\langle S[p\backslash q] \mid (m)R(p\backslash q)\rangle$ **spec** where $p, q \in \mathbb{S}$, if

$$\forall m \in S.\ R(m) \Leftrightarrow R(p\backslash q)(m[p\backslash q])$$

then

$$\langle S \mid (m)R\rangle[p\backslash q] \Leftrightarrow \langle S[p\backslash q] \mid (m)R(p\backslash q)\rangle$$

□

Proof The proof uses Lemma 7.1 which says that the precondition in Theorem 7.2 implies that the restriction of $\langle S \mid (m)R\rangle[p\backslash q]$ is equivalent to $R(p\backslash q)$. The proof is

$$\begin{aligned}
&\quad \langle S \mid (m)R\rangle[p\backslash q] \\
= &\ \text{“definition of } \textit{rename}\text{”} \\
&\quad \langle S[p\backslash q] \mid (m)R(m[p\backslash q]_S^{-1})\rangle \\
\Leftrightarrow &\ \text{“precondition, Lemma 7.1, definitions of } \Leftrightarrow\text{”} \\
&\quad \langle S[p\backslash q] \mid (m)R(p\backslash q)\rangle
\end{aligned}$$

□

Lemma 7.1 If $\forall m \in S.\ R(m) \Leftrightarrow R(p\backslash q)(m[p\backslash q])$ then

$$\forall m \in S[p\backslash q].\ Ax(\langle S \mid (m)R\rangle[p\backslash q])(m) \Leftrightarrow R(p\backslash q)(m))$$

□

Proof

$$\begin{aligned}
&\quad \forall m \in S.\ R(m) \Leftrightarrow R(p\backslash q)(m[p\backslash q]) \\
\Rightarrow &\ \text{“} m \in S[p\backslash q] \text{ implies } m[p\backslash q]_S^{-1} \in S\text{”} \\
&\quad \forall m \in S[p\backslash q].\ R(m[p\backslash q]_S^{-1}) \Leftrightarrow R(p\backslash q)((m[p\backslash q]_S^{-1})[p\backslash q]) \\
= &\ \text{“}\_[p\backslash q]_S^{-1} \text{ is inverse of } \_[p\backslash q]\text{”} \\
&\quad \forall m \in S[p\backslash q].\ R(m[p\backslash q]_S^{-1}) \Leftrightarrow R(p\backslash q)(m) \\
= &\ \text{“definition of } \textit{rename}\text{”} \\
&\quad \forall m \in S[p\backslash q].\ Ax(\langle S \mid (m)R\rangle[p\backslash q])(m) \Leftrightarrow R(p\backslash q)(m)
\end{aligned}$$

□

$Circ$
$= \textbf{Elements}$
$\quad sqr \in \mathbb{N} \to \mathbb{N},$
$\quad pi \in \mathbb{N},$
$\quad Rect \in U_1$
$\textbf{Restrictions}$
$\quad \forall x \in \mathbb{N}.$
$\quad sqr(x) = x * x \ \wedge$
$\quad pi = 31415 \ \wedge$
$\quad Rect = \mathbb{N} \times \mathbb{N}$
$\textbf{End}$

Figure 7.7: Specification *Circ*

$Circ_2$
$= \textbf{Elements}$
$\quad sqr \in \mathbb{N} \to \mathbb{N},$
$\quad pi \in \mathbb{N},$
$\quad Square \in U_1$
$\textbf{Restrictions}$
$\quad \forall x \in \mathbb{N}.$
$\quad sqr(x) = x * x \ \wedge$
$\quad pi = 31415 \ \wedge$
$\quad Square = \mathbb{N} \times \mathbb{N}$
$\textbf{End}$

Figure 7.8: Specification $Circ_2$

Although the pre-condition in Theorem 7.2 seems complicated, in practice, it is usually easy to discharge. For example, consider the specifications *Circ* and $Circ_2$ given in Figure 7.7 and Figure 7.8, respectively. $Circ_2$ is *Circ* with *Rect* renamed to *Square*. We can easily verify that the signature of $Circ_2$ equals $Sig(Circ)[Rect \backslash Square]$, and the restriction of $Circ_2$ equals $Ax(Circ)(m)(Rect \backslash Square)$ for all $m \in Sig(Circ)$. Figure 7.9 contains a proof that $Ax(Circ)(m) = Ax(Circ_2)(m[Rect \backslash Square])$ for all $m \in Sig(Circ)$. Hence, by Theorem 7.2 we get that $Circ[Rect \backslash Square] \Leftrightarrow Circ_2$.

Theorem 7.3 is used to simplifying the restrictions of specifications constructed with a *hide* operation. The pre-condition in Theorem 7.3 ensures that the restrictions of $\langle S \backslash i \mid (m)R \rangle$ and $\langle S \mid (m)R \rangle \backslash i$ are logically equivalent. In practice, the pre-condition is true for all canonical specifications, and any structured specification that can be unfolded to a canonical specification.

Theorem 7.3 (Unfolding Hide)
Given $\langle S \mid (m)R \rangle$ **spec** and $\langle S \backslash i \mid (m)R \rangle$ **spec** where $i \in Set(\mathbb{S})$,

$$\text{if } \forall m \in S.\ R(m) \Leftrightarrow R(m \backslash i) \text{ then } \langle S \mid (m)R \rangle \backslash i \Leftrightarrow \langle S \backslash i \mid (m)R \rangle$$

□

$$
\begin{array}{ll}
 & Ax(Circ_2)(m[Rect \backslash Square]) \\
= & \text{“definition of } Ax\text{, definition of } Circ_2\text{”} \\
 & \forall x \in \mathbb{N}. \\
 & m[Rect \backslash Square].sqr(x) = x * x \ \wedge \\
 & m[Rect \backslash Square].pi = 31415 \ \wedge \\
 & m[Rect \backslash Square].Square = \mathbb{N} \times \mathbb{N} \\
= & \text{“definition of } m[Rect \backslash Square]\text{”} \\
 & \forall x \in \mathbb{N}. \\
 & m.sqr(x) = x * x \ \wedge \\
 & m.pi = 31415 \ \wedge \\
 & m.Rect = \mathbb{N} \times \mathbb{N} \\
= & \text{“definition of } Ax\text{, definition of } Circ\text{”} \\
 & Ax(Circ)(m)
\end{array}
$$

Figure 7.9: Proof that $Ax(Circ)(m) = Ax(Circ_2)(m[Rect \backslash Square])$

Proof Similar to that of Theorem 7.2 above □

Finally in this section, we consider Theorem 7.4 which shows how to simplify the restrictions of specifications that are made using *sum*. The pre-condition in Theorem 7.4 says that the restrictions of $\langle S_1 \mid (m)R_1 \rangle + \langle S_2 \mid (m)R_2 \rangle$ and $\langle S_1 \oplus S_2 \mid (m)R_1(m) \times R_2(m) \rangle$ must be logically equivalent. In practice, the pre-condition is true if the arguments to *sum* (+) are canonical specifications, or are structured specifications that can be unfolded to canonical specifications. Therefore, the pre-condition can often be discharged by inspection.

Theorem 7.4 (Unfolding Sum)
Given any $\langle S_1 \mid (m)R_1 \rangle$ **spec**, $\langle S_2 \mid (m)R_2 \rangle$ **spec** and $\langle S_1 \oplus S_2 \mid (m)R_1 \times R_2 \rangle$ **spec**, if

$$\forall m \in S_1 \oplus S_2.\ R_1(m) \times R_2(m) \Leftrightarrow R_1({}^{S_1 S_2}_{\Pi_1}(m)) \times R_2({}^{S_1 S_2}_{\Pi_2}(m))$$

then

$$\langle S_1 \mid (m)R_1 \rangle + \langle S_2 \mid (m)R_2 \rangle \Leftrightarrow \langle S_1 \oplus S_2 \mid (m)R_1(m) \times R_2(m) \rangle.$$

□

Proof Follows immediately from the definition of specification equality ($\Leftrightarrow$) □

In Theorem 7.4, the assumption $\langle S_1 \oplus S_2 \mid (m)R_1 \times R_2 \rangle$ **spec** implies that $(m)R_1(m) \times R_2(m)$ is well typed for all $m \in S_1 \oplus S_2$.

To summarise this section: we have given some laws that allow us to simplify the restrictions of structured specifications. Such laws can be used to unfold applications of the specification operators.

7.6 The Properties of Specification Operators

In this section, we give some properties of the specification operators. The properties that we give are intended to be used to rearrange, and simplify, structured specifications. For example, when we implement a structured specification, we usually need to rearrange it to make it easier to refine and implement. This is often necessary because specifications are not always in a form that is suited to program development; when we construct specifications our main concern is that they are easy to make and understand. We also need a basic collection of properties about the specification operators in order to reason about structured specifications.

In Chapter 4, we stated that constructive logics do not enjoy many of the algebraic properties that classical logics do; for example, combinators such as $\wedge$ and $\vee$ are neither commutative nor associative with respect to the extensional, or intensional, type equality provided by MLT. Therefore, most of the properties of the specification operators are defined using specification equality ($\Leftrightarrow$), rather than type equality ($=$).

It is not our intention to give a complete set of properties of the specification operators, but rather to show that they have useful properties, and that these properties can be used to simplify and rearrange structured specifications.

7.6.1 Properties of *Rename* and *Hide*

Most properties of *rename* and *hide* are dependent on the properties of signature renaming and signature hiding, respectively, since *rename* and *hide* are defined using these signature operators. We begin by stating some facts about *rename*:

Fact 7.5 $SP[p\backslash p] \Leftrightarrow SP$ □

Fact 7.6 $SP[p\backslash q] \Leftrightarrow SP$, if p is not in the domain of SP. □

Fact 7.7 $SP[p\backslash q][q\backslash p] \Leftrightarrow SP$, if p and q are not in the domain of SP. □

The above facts should not be surprising, since textual substitution has analogous properties. Like textual substitution, the order in which we rename components is important; for example, given any specification SP, and any components names p, q, x and y, then $(SP[p\backslash q])[x\backslash y]$ is not always equivalent to $(SP[x\backslash y])[p\backslash q]$.

In the remainder of this section, we consider some properties of *hide*. One simple property of *hide* is that hiding an empty set of component names is equivalent to an identity operation on specifications:

Theorem 7.5 Given SP **spec** then $SP\backslash\{\} \Leftrightarrow SP$. □

Proof

$$
\begin{aligned}
& \langle S \mid (m)R \rangle \backslash \{\} \\
= {} & \text{“definition of } \mathit{hide}\text{”} \\
& \langle S \backslash \{\} \mid (m)R(m \backslash_S^{-1} \{\}) \rangle \\
= {} & \text{“Theorem 6.11 } (S\backslash\{\} = S)\text{, Theorem 6.41 } (m\backslash_S^{-1}\{\} = m)\text{”} \\
& \langle S \mid (m)R \rangle
\end{aligned}
$$

□

Fact 7.8 $SP\backslash i \Leftrightarrow SP$, if the names in i are not in the domain of SP □

We can express the composition of several consecutive applications of *hide* as a single application of *hide*:

Theorem 7.6 Given SP **spec** then $SP\backslash i\backslash j \Leftrightarrow SP\backslash(i \cup j)$ □

Proof

$$
\begin{aligned}
& \langle S \mid (m)R \rangle \backslash i \backslash j \\
= {} & \text{“definition of } \mathit{hide}\text{”} \\
& \langle S \backslash i \mid (m)R(m \backslash_S^{-1} i) \rangle \backslash j \\
= {} & \text{“definition of } \mathit{hide}\text{”} \\
& \langle S \backslash i \backslash j \mid (m)R((m \backslash_{S\backslash i}^{-1} j) \backslash_S^{-1} i) \rangle \\
= {} & \text{“Theorem 6.12, Theorem 6.42 } ((m\backslash_{S\backslash i}^{-1} j)\backslash_S^{-1} i = m\backslash_S^{-1}(i \cup j))\text{”} \\
& \langle S \backslash (i \cup j) \mid (m)R(m \backslash_S^{-1} (i \cup j)) \rangle \\
= {} & \text{“definition of } \mathit{hide}\text{”} \\
& \langle S \mid (m)R \rangle \backslash (i \cup j)
\end{aligned}
$$

□

Theorem 7.6 justifies the fact that the order in which we hide the components of a specification is not important:

Fact 7.9 $(SP\backslash i)\backslash j \Leftrightarrow (SP\backslash j)\backslash i$ □

The properties of *rename* and *hide* given above are in no way complete, but do illustrate that specifications of the form $SP[p\backslash q]$, and $SP\backslash i$, can be simplified and rearranged. In this section, we have not considered the relationship between *rename* and the other specification operators such as *sum* and *union*; nor have we considered the relationship between *hide* and the other specification operators. Such relationships will be discussed when we consider the properties of the other specification operators.

7.6.2 Properties of Union

In the following, whenever we write $(SP_1 \cup SP_2)$ we assume that $Sig(SP_1) = Sig(SP_2)$. Many of the properties of *union* are determined by the properties of the $(\times)$ type constructor, since *union* is defined in terms of $(\times)$. Consequently, *union* is both associative and commutative since $(\times)$ is associative and commutative if we use weak type equality $(\Leftrightarrow)$ to compare propositions:

Theorem 7.7 $SP_1 \cup SP_2 \Leftrightarrow SP_2 \cup SP_1$ □

Proof Omitted □

Theorem 7.8 $(SP_1 \cup SP_2) \cup SP_3 \Leftrightarrow SP_1 \cup (SP_2 \cup SP_3)$ □

Proof Omitted □

The *union* operator is also idempotent since $(\times)$ is idempotent with respect to weak type equality:

Fact 7.10 $SP \cup SP \Leftrightarrow SP$ □

Let us consider the relationship between *union* and the other specification operators. All the specification operators distribute over *union*:

Theorem 7.9 $(SP_1 \cup SP_2)[p\backslash q] \Leftrightarrow (SP_1[p\backslash q]) \cup (SP_2[p\backslash q])$ □

Proof Omitted □

Theorem 7.10 $(SP_1 \cup SP_2)\backslash i \Leftrightarrow (SP_1\backslash i) \cup (SP_2\backslash i)$ □

Proof Omitted □

Theorem 7.11 $(SP_1 \cup SP_2) \downarrow f \Leftrightarrow (SP_1 \downarrow f) \cup (SP_2 \downarrow f)$ □

Proof Omitted □

Theorem 7.12 $(SP_1 \cup SP_2) \uparrow f \Leftrightarrow (SP_1 \uparrow f) \cup (SP_2 \uparrow f)$ □

Proof

$$
\begin{aligned}
& (\langle S_1 \mid (m)R_1\rangle \cup \langle S_2 \mid (m)R_2\rangle) \uparrow f \\
\Leftrightarrow\ & \text{“}S_1 = S_2\text{, definition of } \textit{union}\text{”} \\
& \langle S_1 \mid (m)R_1 \times (m)R_2\rangle \uparrow f \\
\Leftrightarrow\ & \text{“definition of } \textit{translate}\text{”} \\
& \langle S_1 \mid (m)R_1(f(m)) \times (m)R_2(f(m))\rangle \\
\Leftrightarrow\ & \text{“}S_1 = S_2\text{, definition of } \textit{union}\text{”} \\
& \langle S_1 \mid (m)R_1(f(m))\rangle \cup \langle S_2 \mid (m)R_2(f(m))\rangle \\
\Leftrightarrow\ & \text{“definition of } \textit{translate}\text{, twice”} \\
& \langle S_1 \mid (m)R_1\rangle \uparrow f \cup \langle S_2 \mid (m)R_2\rangle \uparrow f
\end{aligned}
$$

□

Theorem 7.13 $(SP_1 \cup SP_2) + SP_3 \Leftrightarrow (SP_1 + SP_3) \cup (SP_2 + SP_3)$ □

Proof Omitted □

Theorem 7.14 $(SP_1 \cup SP_2) \lhd \Lambda\langle S \mid R\rangle \Leftrightarrow (SP_1 \lhd \Lambda\langle S \mid R\rangle) \cup (SP_2 \lhd \Lambda\langle S \mid R\rangle)$ □

Proof Omitted □

Given a specification that has a large number of axioms in its restriction, we can use the distributive properties, given above, to express such a specification as the union of several specifications, each of which have fewer axioms in their restriction than the original specification. For example, given a specification $SP = \langle S \mid (m)R\rangle[p\backslash q]$, we can usually express R in the form $R_1 \times R_2$ where R_1 and R_2 are propositions. Consequently, by the definition of *union*, and by Theorem 7.9, we can express SP as

$$(\langle S \mid (m)R_1\rangle[p\backslash q]) \cup (\langle S \mid (m)R_2\rangle[p\backslash q])$$

The above rearrangement of SP allows us to reason about the properties of SP by considering the specifications $\langle S \mid (m)R_1\rangle[p\backslash q]$ and $\langle S \mid (m)R_2\rangle[p\backslash q]$ in relative isolation. For example, we will see later that we can decompose the task of refining specifications of the form $SP_1 \cup SP_2$ to that of refining SP_1 and SP_2.

7.6.3 Properties of *Sum* and *Enrich*

The properties of *sum* and *enrich* are dependent on the properties of signature concatenation and the $(\times)$ type constructor, since we use them to define *sum* and *enrich*.

We begin by considering the properties of *sum*. The *sum* operator is associative, and this fact follows from the fact that signature concatenation is associative:

Theorem 7.15 $(SP_1 + SP_2) + SP_3 \Leftrightarrow SP_1 + (SP_2 + SP_3)$ □

Proof Given in Figure 7.10 □

Unlike *union*, the *sum* operator is neither commutative nor idempotent since signature concatenation is neither commutative nor idempotent.

The *rename* operator distributes over *sum* provided that the name of the component being renamed does appear in the domain of both the arguments of *sum*:

Theorem 7.16 if p is not in the domains of SP_1 and SP_2 then

$$(SP_1 + SP_2)[p\backslash q] \Leftrightarrow (SP_1[p\backslash q]) + (SP_2[p\backslash q])$$

□

Proof Omitted □

$$(\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle) + \langle S_3 \mid (m)R_3\rangle$$

$\Leftrightarrow$ "definition of *sum*"

$$(\langle S_1 \oplus S_2 \mid (m)(R_1 \circ \overset{S_1S_2}{\Pi_1})(m) \times (m)(R_2 \circ \overset{S_1S_2}{\Pi_2})(m)\rangle) + \langle S_3 \mid (m)R_3\rangle$$

$\Leftrightarrow$ "definition of *sum*, Let $S_L = S_1 \oplus S_2$"

$$\left\langle (S_1 \oplus S_2) \oplus S_3 \middle| \begin{array}{l} ((m)(R_1 \circ \overset{S_1S_2}{\Pi_1} \circ \overset{S_LS_3}{\Pi_1})(m) \times (m)(R_2 \circ \overset{S_1S_2}{\Pi_2} \circ \overset{S_LS_3}{\Pi_1})(m)) \\ \times (m)(R_3 \circ \overset{S_LS_3}{\Pi_2})(m) \end{array} \right\rangle$$

$\Leftrightarrow$ "Theorem 6.19 (Associativity of $\oplus$), Let $S_R = S_2 \oplus S_3$"

$$\left\langle S_1 \oplus (S_2 \oplus S_3) \middle| \begin{array}{l} (m)(R_1 \circ \overset{S_1S_R}{\Pi_1})(m) \times \\ ((m)(R_2 \circ \overset{S_2S_3}{\Pi_1} \circ \overset{S_1S_R}{\Pi_2})(m) \times (m)(R_3 \circ \overset{S_2S_3}{\Pi_2} \circ \overset{S_1S_R}{\Pi_2})(m)) \end{array} \right\rangle$$

$\Leftrightarrow$ "definition of *sum*"

$$\langle S_1 \mid (m)R_1\rangle + \langle S_2 \oplus S_3 \mid (m)(R_2 \circ \overset{S_2S_3}{\Pi_1})(m) \times (m)(R_3 \circ \overset{S_2S_3}{\Pi_2})(m)\rangle$$

$\Leftrightarrow$ "definition of *sum*"

$$\langle S_1 \mid (m)R_1\rangle + (\langle S_2 \mid (m)R_2\rangle + \langle S_3 \mid (m)R_3\rangle)$$

Figure 7.10: Proof of Theorem 7.15 (*sum* associativity)

Let us consider the precondition in Theorem 7.16. If p appears in the domains of SP_1 and SP_2 then it is overloaded in $SP_1 + SP_2$. Consequently, the component p supplied by SP_1 is renamed in $(SP_1 + SP_2)[p\backslash q]$, but not the component p supplied by SP_2; recall that an attempt to rename a component that has an overloaded name only renames the first component with that overloaded name. So, the name q appears once in the domain of $(SP_1 + SP_2)[p\backslash q]$. In contrast, the component p supplied by SP_1, and the component p supplied by SP_2, are renamed in $(SP_1[p\backslash q]) + (SP_2[p\backslash q])$. So, the name q appears twice in the domain of $(SP_1[p\backslash q]) + (SP_2[p\backslash q])$. Hence, $(SP_1 + SP_2)[p\backslash q]$ is not equivalent to $(SP_1[p\backslash q]) + (SP_2[p\backslash q])$ if p appears in the domain of SP_1 and SP_2.

The *hide* operator distributes over the *sum* operator:

Theorem 7.17 $(SP_1 + SP_2)\backslash i \Leftrightarrow (SP_1\backslash i) + (SP_2\backslash i)$ □

Proof Omitted □

The *derive* and *translate* operators rarely distribute over the sum of two specifications. If they do, then the *derive* map and *translate* map used by *derive* and *translate*, respectively, have a special form:

Fact 7.11 Let $f = \lambda m.f_1(\overset{S_1S_2}{\Pi_1}(m)) \oplus f_2(\overset{S_1S_2}{\Pi_2}(m)) \in (S_1 \oplus S_2) \rightarrow (S_3 \oplus S_4)$ where $f_1 \in S_1 \rightarrow S_3$ and $f_2 \in S_2 \rightarrow S_4$.

$$(\langle S_1 \,|\, (m)R_1\rangle + \langle S_2 \,|\, (m)R_2\rangle) \downarrow f \Leftrightarrow \langle S_1 \,|\, (m)R_1\rangle \downarrow f_1 + \langle S_2 \,|\, (m)R_2\rangle \downarrow f_2$$

□

Fact 7.12 Let $f = \lambda m.f_1(\overset{S_3S_4}{\Pi_1}(m)) \oplus f_2(\overset{S_3S_4}{\Pi_2}(m)) \in (S_3 \oplus S_4) \to (S_1 \oplus S_2)$ where $f_1 \in S_3 \to S_1$ and $f_2 \in S_4 \to S_2$.

$$(\langle S_1 \,|\, (m)R_1\rangle + \langle S_2 \,|\, (m)R_2\rangle) \uparrow f \Leftrightarrow \langle S_1 \,|\, (m)R_1\rangle \uparrow f_1 + \langle S_2 \,|\, (m)R_2\rangle \uparrow f_2$$

□

In practice, *derive* maps rarely have the same form as f in Theorem 7.11; similarly, *translate* maps rarely have the same form as f in Theorem 7.12. However, if we regard the equivalences given in Theorem 7.11 and 7.12 as rewrite rules, then these equivalences can always be applied in the right-to-left direction to transform specifications of the form $(SP_1 \downarrow f_1) + (SP_2 \downarrow f_2)$ into equivalent specifications of the form $(SP_1 + SP_2) \downarrow f$.

In the remainder of this section, we consider a few properties of *enrich*. The most common rearrangement of a specification of the form $\langle S_1 \mid (m)R_1\rangle \lhd \Lambda\langle S_2 \mid R_2\rangle$ is simply to unfold the application of *enrich* to get an identical specification of the form $\langle S_1 \oplus S_2 \mid (m)(R_1 \circ \overset{S_1S_2}{\Pi_1})(m) \times R_2(m)\rangle$. The unfolded specification can then be rearranged further, if required. We can use Fact 7.13, below, to unfold an application of *enrich* that adds new components to a specification, but no new restrictions. We can use Fact 7.14, below, to unfold an application of *enrich* that adds new restrictions to a specification, but no new components.

Fact 7.13 $\langle S_1 \mid (m)R_1\rangle \lhd \Lambda\langle S_2 \mid T\rangle \Leftrightarrow \langle S_1 \oplus S_2 \mid (m)(R_1 \circ \overset{S_1S_2}{\Pi_1})(m)\rangle$ □

Fact 7.14 $\langle S_1 \mid (m)R_1\rangle \lhd \Lambda\langle \Phi \mid R_2\rangle \Leftrightarrow \langle S_1 \mid (m)R_1(m) \times R_2(m)\rangle$ □

The *enrich* operator enjoys a kind of associativity, whereby consecutive enrichments to a specification can be combined into a single enrichment:

Theorem 7.18 Let $L = Sig(SP) \oplus S_2$.
$(SP \lhd \Lambda\langle S_2 \,|\, R_2\rangle) \lhd \Lambda\langle S_3 \,|\, R_3\rangle \Leftrightarrow SP \lhd \Lambda\langle S_2 \oplus S_3 \,|\, (m)(R_2 \circ \overset{LS_3}{\Pi_1})(m) \times R_3\rangle$
□

Proof

$$(\langle S_1 \,|\, (m)R_1\rangle \lhd \Lambda\langle S_2 \,|\, R_2\rangle) \lhd \Lambda\langle S_3 \,|\, R_3\rangle$$

$\Leftrightarrow$ "definition of *enrich*"

$$\langle S_1 \oplus S_2 \mid (m)(R_1 \circ \overset{S_1S_2}{\Pi_1})(m) \times R_2(m)\rangle \lhd \Lambda\langle S_3 \,|\, R_3\rangle$$

$\Leftrightarrow$ "definition of *enrich*, Let $L = S_1 \oplus S_2$"

$$\langle (S_1 \oplus S_2) \oplus S_3 \mid (m)((R_1 \circ \overset{S_1S_2}{\Pi_1} \circ \overset{LS_3}{\Pi_1})(m) \times (R_2 \circ \overset{LS_3}{\Pi_1})(m)) \times R_3(m)\rangle$$

$\Leftrightarrow$ "associativity of $\oplus$, associativity of $\times$, Let $R = S_2 \oplus S_3$"

$$\langle S_1 \oplus (S_2 \oplus S_3) \mid (m)(R_1 \circ \overset{S_1R}{\Pi_1})(m) \times ((R_2 \circ \overset{LS_3}{\Pi_1})(m) \times R_3(m))\rangle$$

$\Leftrightarrow$ "definition of *enrich*"

$$\langle S_1 \mid (m)R_1\rangle \lhd \Lambda\langle S_2 \oplus S_3 \mid (m)(R_2 \circ \overset{LS_3}{\Pi_1})(m) \times R_3(m)\rangle$$

□

Both the *rename* and *hide* operators distribute over applications of *enrich*:

Fact 7.15 If name p is not in the domains of SP and S then:

$$(SP \lhd \Lambda\langle S \,|\, R\rangle)[p\backslash q] \Leftrightarrow (SP[p\backslash q]) \lhd \Lambda\langle S[p\backslash q] \,|\, (m)R(m[p\backslash q]^{-1}_{S_1 \oplus S_2})\rangle$$

□

Fact 7.16 $(SP \lhd \Lambda\langle S \,|\, R\rangle)\backslash i \Leftrightarrow (SP\backslash i) \lhd \Lambda\langle S\backslash i \,|\, (m)R(m\backslash^{-1}_{S_1 \oplus S_2} i)\rangle$ □

Remark The precondition in Fact 7.15 is needed for the same reasons as we gave for requiring a similar precondition in Theorem 7.16—recall that Theorem 7.16 says that *rename* distributes over *sum* □

Apart from *rename* and *hide*, the other specification operators rarely distribute over the enrichment of a specification. However, this is not a problem as, in practice, we usually only require to distribute *rename* and *hide* over *enrich*.

7.6.4 Properties of *Derive* and *Translate*

We begin by giving some properties of *derive*. The first property we consider is that an application of *derive*, using an identity function as a *derive* map, is equivalent to an identity operation on specifications:

Theorem 7.19
Let $Id = \lambda x.x \in Sig(SP) \to Sig(SP)$. $SP \downarrow Id \Leftrightarrow SP$.
□

Proof Omitted □

We can express two, or more, consecutive applications of *derive* as a single application of *derive*:

Theorem 7.20
$(SP \downarrow f) \downarrow g \Leftrightarrow SP \downarrow (g \circ f)$ where SP **spec**, S_2 **sig**, S_3 **sig**, $f \in Sig(SP) \to S_2$ and $g \in S_2 \to S_3$.
□

Proof Omitted □

We now consider some properties of *translate*. Many of the properties of *translate* are similar to those of *derive*. For example, Theorem 7.21 (below) says that an application of *translate*, using an identity function as a *translate* map, is equivalent to an identity operation on specifications. Note that the equivalence expressed in Theorem 7.21 is a type equality (=).

Theorem 7.21 Let $Id_S = \lambda m.m \in S \to S$. $\langle S \,|\, (m)R\rangle \uparrow Id_S = \langle S \,|\, (m)R\rangle$ □

Proof $\langle S \,|\, (m)R\rangle \uparrow Id_S = \langle S \,|\, (m)R(Id_S(m))\rangle = \langle S \,|\, (m)R\rangle$ □

Several consecutive applications of *translate* can be expressed as a single application of *translate*:

Theorem 7.22
Given any $\langle S_1 \,|\, (m)R_1 \rangle$ **spec**, S_1 **sig**, S_2 **sig**, S_3 **sig**, $f \in S_2 \to S_1$ and $g \in S_3 \to S_2$,

$$((\langle S_1 \,|\, (m)R_1 \rangle \uparrow f) \uparrow g = \langle S_1 \,|\, (m)R_1 \rangle \uparrow (f \circ g)$$

□

Proof

$$
\begin{aligned}
& ((\langle S_1 \,|\, (m)R_1 \rangle \uparrow f) \uparrow g \\
= \; & \text{“definition of } \textit{translate}\text{”} \\
& \langle S_2 \,|\, (m)(R_1 \circ f)(m) \rangle \uparrow g \\
= \; & \text{“definition of } \textit{translate}\text{”} \\
& \langle S_2 \,|\, (m)((R_1 \circ f) \circ g)(m) \rangle \\
= \; & \text{“associativity of function composition”} \\
& \langle S_2 \,|\, (m)(R_1 \circ (f \circ g))(m) \rangle \\
= \; & \text{“definition of } \textit{translate}\text{”} \\
& \langle S_1 \,|\, (m)R_1 \rangle \uparrow (f \circ g)
\end{aligned}
$$

□

The following fact is justified by Theorems 7.22 and by Theorem 7.21:

Fact 7.17 Given S_2 **sig** and $f \in Sig(SP) \to S_2$ then $(SP \uparrow f) \uparrow f^{-1} = SP$ □

The *translate* operator is an inverse for *derive*, but *derive* is only an inverse for *translate* under certain conditions:

Fact 7.18 $(SP \downarrow f) \uparrow f \Leftrightarrow SP$ where $f \in Sig(SP) \to S_2$ and S_2 **sig** □

Fact 7.19 If f^{-1} exists then $(SP \uparrow f) \downarrow f \Leftrightarrow SP$ where $f \in S_2 \to Sig(SP)$ and S_2 **sig** □

7.7 Discussion and Conclusion

Several of our specification operators can be defined differently. For example, both *rename* and *hide* can be defined in terms of *derive*:

$$
\begin{aligned}
SP[p \backslash q] \quad & \Leftrightarrow \quad SP \downarrow (\lambda m \in Sig(SP).\, m[p \backslash q]) \\
SP \backslash i \quad & \Leftrightarrow \quad SP \downarrow (\lambda m \in Sig(SP).\, m \backslash i)
\end{aligned}
$$

However, the disadvantage of using *derive* to define *rename* and *hide* is that the restrictions of $SP \downarrow (\lambda m \in Sig(SP).\, m[p\backslash q])$ and $SP \downarrow (\lambda m \in Sig(SP).\, m\backslash i)$ are more complicated than those of $SP[p\backslash q]$ and $SP\backslash i$. Consequently, constructive proofs concerning specifications of the form $SP[p\backslash q]$ and $SP\backslash i$ are easier to make than those concerning $SP \downarrow (\lambda m \in Sig(SP).\, m[p\backslash q])$ and $SP \downarrow (\lambda m \in Sig(SP).\, m\backslash i)$.

One of the issues raised when making the specifications $Catalogue_6$ and $Catalogue_7$ is that of eliminating redundant components from specifications: we illustrated that redundant components can be hidden using *hide*, or removed using *derive*. In the Z specification language, the schema hiding operator removes components from the signature of a schema since Z does not allow local components. The use of *derive* to remove components from a specification can be shown to be similar, in function, to the use of schema hiding in Z. Z does not allow dependencies between components in a schema signature. Therefore, we can remove components from the signature of a schema without making it ill-defined. In contrast, if we remove a component from one of our specifications, then there is a possibility that the specification may become ill-defined since the type of other components in the specification may depend on the component that we remove. Hence, we prefer to hide components, rather than remove them, since hidden components remain available for use in the signature of a specification.

Some specification languages, such as Z and ASL, do not permit component names to be overloaded, and they resolve name-clashes—that can arise when two specifications are combined—by merging components that have the same name into a single component. One advantage of merging components is that if we intended the components to be equal then we do not have to explicitly specify that fact. However, the disadvantage of merging components is that we can accidentally merge two components that are intended to be different, as it is easy to loose track of the component names in a large specification. With our *sum* operator there is no danger of accidentally merging component names; if we wish to specify two components as being equal, then we can add that requirement, as an extra restriction, to a specification.

In summary, we have developed a collection of specification operators which combine, and modify specifications within our type-theoretic framework. We have shown that these specification operators facilitate the incremental construction of specifications. We have also shown that the specification operators enjoy some useful algebraic properties that can be used to change the structure of structured specifications.

Chapter 8

Refinement

8.1 Introduction

The design and construction of specifications is only one half of the programming task. The other half is the development of programs that satisfy our specifications. It is now well known that specifications can play an active role in programming: it is possible, at least in principle, to derive a program from its specification by mathematical transformation. The process of mathematically transforming a specification towards a program is called *refinement*. In this chapter, we give a type-theoretic definition of refinement for specifications, and to use it to develop a collection of refinement laws.

Refining a specification may be regarded as the task of making concrete design decisions about properties of the specification that were left open by the specifier—the choice of data structures and error messages for example. The definition of refinement given here is one where we add more detail to a specification to get a refinement. In the context of program development in type theory, we regard refinement as being different from implementation. Specifications are types, and any refinement of a specification is a specification and, hence, a type. However, implementations are values of a type, so that implementation is the task of finding values of a type. We don't implement specifications in one go, instead we proceed to an implementation by refining a specification. Implementations become easier the more we refine a specification, and so we keep refining a specification until we arrive at a specification that strongly suggests an implementation. Then we proceed by finding an implementation of the refined specification by using a constructive proof. This last step is usually clerical and so we do not discuss it in much detail here. We refer the reader interested in deriving programs via constructive proofs to [3, 44].

Our definition of refinement in type theory is similar to the notion of subtyping in programming languages such as Quest [7]. Indeed, a subtype of a specification is a refinement. However, using subtyping as a definition of refinement is overly restrictive, as it fails to meet the following two requirements. Firstly, we want to be able to refine specifications by adding, or removing components from their signature; for example, we often add extra components that we use to refine other components in a specification. Secondly, we want to be able to refine a specification by changing the type of some of its components, as this often leads to more efficient implementations; such refinements are sometimes called *data refinements.* The definition of refinement that we will propose allows both kinds of refinements mentioned above. Our definition of refinement is similar to the *deliverables* approach of Burstall et al [6], and to refinement in ECC [27]; we discuss this related work later.

To be useful, refinement must proceed in a piecewise manner. In other words we want to refine a specification by refining its individual parts relatively independently of each other, and then combine its refined parts to get a refinement of the whole specification. A piecewise approach has several advantages. Firstly, we avoid the work of reshaping a specification before we refine it. Secondly, we can more easily reuse refinements when part of a specification is reused. Thirdly, if we change some parts of a specification after it is refined, then only the changed parts need to be re-refined to get a refinement of the new specification. A piecewise approach to refinement may not always produce the most efficient implementation of a specification. Nevertheless, it is a systematic approach to tackling the refinement of large specification.

8.2 Refinement of Types

In many program development formalisms—such as algebraic specifications [50, 56], and the refinement calculus [37]—a specification P is refined by a specification Q whenever all the implementations of Q are also implementations of P. If we were to adopt a similar definition of refinement for type theory, then we might say that a type P is refined by a type Q whenever all the members of Q are also members of P; in other words, Q is a subtype of P. In practice, using subtyping as a definition of refinement is of limited use in type theory. For example, when we refine a specification, we often add extra components to its signature, which play an intermediary role in helping to make the refinement. However, if we use subtyping as the definition of refinement, then we cannot refine a specification by adding extra components to its signature: the implementations of the refinement will contain extra components, and so, will no longer be implementations of the original specification.

We propose a more flexible definition of refinement which allow the refinements of a type to be more than just subtypes. Using our proposed definition of refinement, if a type P is refined by a type Q, then the values of Q need not have the same 'shape' as the values of P. Central to our proposal is the introduction of a *refinement map* that translates values of type Q to values of type P. We write that type P is refined by type Q, with respect to a refinement map f, as $P \sqsubseteq^f Q$. We propose the following definition of refinement:

Definition 8.1 (General Refinement)
Given P **type** and Q **type**, then P is refined by Q with respect to f (written $P \sqsubseteq^f Q$) iff we have a proof of the judgement $f \in Q \rightarrow P$.
□

The intuition behind the definition of general refinement is that if we have a refinement map $f \in Q \rightarrow P$, then given any implementation q of Q (i.e. $q \in Q$) we can always guarantee that $f(q)$ is an implementation of the original specification P (i.e. $f(q) \in P$). For example, if Q is a specification that supplies all the components that P does, plus some extra components, then we would choose f to be a function that takes a module $q \in Q$, and throws away the extra components in q to get a module satisfying P. If we view P and Q as propositions, then the requirement that there exists some $f \in Q \rightarrow P$ says that Q implies P; this implication corresponds with the intuition that refinement strengthens the constraints on a specification.

Refinement maps are similar, in function, to *retrieve* functions used in VDM data reification [21], and to functional *abstraction invariants* for data refinement in the refinement calculus [36, 38], and to *constructors* used in the constructor implementation of algebraic specifications [50]. A discussion of the relationship between our definition of refinement, and those cited above, is given later.

One problem with general refinement is that its very simplicity often leads to very complicated refinement maps, especially when refining complex types such as specifications. It is more convenient to define specialised definitions of refinement for types such as specifications, under the condition that the specialised definitions are special cases of general refinement. This is not to say general refinement is not needed; it is, but mainly for refining types within specifications. For example, we would use general refinement to refine a function type that occurs in the signature of a specification. Our main interest is the refinement and development of specifications, and in the following section we give a specialised definition of refinement for specifications.

8.3 Refinement of Specifications

In this section we define the notion of specification refinement which is a special case of general refinement. We motivate the need for specification refinement as follows. Suppose $SP_1 \sqsubseteq^f SP_2$ where SP_1 and SP_2 are specifications and $f \in SP_2 \rightarrow SP_1$. In mapping each $m \in SP_2$ to $f(m) \in SP_1$, the refinement map f does two things. Firstly, it maps the witness of each $m \in SP_2$ to a witness of an implementation of

SP_1, thus proving that the restriction of SP_2 implies the restriction of SP_1. Secondly, it maps the computational element of each $m \in SP_2$ to a computational element of type $Sig(SP_1)$. The fact that the refinement map has two roles when refining specifications can make the refinement map difficult to construct and understand. Therefore, we propose a specialised definition of refinement for specifications which, instead of requiring a single refinement map $f \in SP_2 \rightarrow SP_1$, requires the existence of two maps: one maps between witnesses, and the other between computational elements. A formal definition of specification refinement is given in Definition 8.2 with a justification of the definition following.

Definition 8.2 (Specification Refinement) Given SP_1 **spec** and SP_2 **spec**, an abstraction map between SP_1 and SP_2 is any function

$$h \in Sig(SP_2) \rightarrow Sig(SP_1)$$

such that there exists some witness p that satisfies the following proposition:

$$p \in \forall x \in Sig(SP_2).\ Ax(SP_2)(x) \Rightarrow Ax(SP_1)(h(x))$$

We call p a proof map. If h is an abstraction map between SP_2 and SP_1 then we say that SP_1 is refined to SP_2 with respect to h, written $SP_1 \sqsubseteq_h SP_2$
□

Notation For any specifications SP_1 and SP_2, and function $h \in Sig(SP_2) \rightarrow Sig(SP_1)$ we will often represent $\forall x \in Sig(SP_2).\ Ax(SP_2)(x) \Rightarrow Ax(SP_1)(h(x))$ by $Res(SP_1, SP_2, h)$ for convenience of presentation □

Notation (Simple Refinement) If $Sig(SP_1) = Sig(SP_2)$ and $Id \in Sig(SP_2) \rightarrow Sig(SP_1)$ is the identity function then we abbreviate $SP_1 \sqsubseteq_{Id} SP_2$ to $SP_1 \sqsubseteq SP_2$. The relation $\sqsubseteq$ is called the *simple refinement* relation. □

The definition of specification refinement is justified as follows. The existence of a proof map p proves that the restriction of SP_2 implies the restriction of SP_1. Proof map p also proves that abstraction map h maps any computational element of a module satisfying SP_2 to a computational element of a module satisfying SP_1.

If $SP_1 \sqsubseteq_h SP_2$ than any implementation m of SP_2 (i.e. $m \in SP_2$) gives an implementation of SP_1:

Fact 8.1 If there exists some proof map p that witnesses that $SP_1 \sqsubseteq_h SP_2$ then given any $m \in SP_2$:

$$\langle h(ce(m)), p(ce(m))(pf(m))\rangle \in SP_1$$

where

$$\begin{array}{lll} p & \in & Res(SP_1, SP_2, h) \\ h(ce(m)) & \in & Sig(SP_1);\ \text{and} \\ p(ce(m))(pf(m)) & \in & Ax(SP_1)(h(ce(m))) \end{array}$$

□

```
0.0      |[   $SP_1$ spec ; $SP_2$ spec ;
0.1           $h \in Sig(SP_2) \to Sig(SP_1)$ ;
0.2           $p \in \forall x \in Sig(SP_2).\ Ax(SP_2)(x) \Rightarrow Ax(SP_1)(h(x))$ ;
0.3.0  ▷      |[    $m \in SP_2$
              ▷       "0.0, 0.3.0, × - elimination"
0.3.1               $ce(m) \in Sig(SP_2)$
                      "0.1, 0.3.1, × - elimination"
0.3.2               $h(ce(m)) \in Sig(SP_1)$
                      "0.0, 0.3.0, × - elimination"
0.3.3               $pf(m) \in Ax(SP_2)(ce(m))$
                      "0.2, 0.3.1, → - elimination"
0.3.4               $p(ce(m)) \in Ax(SP_2)(ce(m)) \Rightarrow Ax(SP_1)(h(ce(m)))$
                      "0.3.3, 0.3.4, → - elimination"
0.3.5               $p(ce(m))(pf(m)) \in Ax(SP_1)(h(ce(m)))$
                      "0.3.2, 0.3.5, × - introduction"
0.3.6               $\langle h(ce(m)), (p(ce(m)))(pf(m))\rangle \in SP_1$
              ]|
                "0.3.0, 0.3.6, → - introduction"
0.4           $\lambda m.\langle h(ce(m)), p(ce(m))(pf(m))\rangle \in SP_2 \to SP_1$
         ]|
```

Figure 8.1: Proof of Theorem 8.1

If a specification SP_1 can be refined to some specification SP_2 using specification refinement, then SP_1 can also refined to SP_2 using general refinement:

Theorem 8.1
If $SP_1 \sqsubseteq_h SP_2$ then $SP_1 \sqsubseteq^f SP_2$; here $f = \lambda m.\langle h(ce(m)), p(ce(m))(pf(m))\rangle \in SP_2 \to SP_1$ where $p \in Res(SP_1, SP_2, h)$ is a proof map. □

Proof The proof is given in Figure 8.1. □

The fact that a specification SP_1 can be refined to SP_2 under general refinement does not guarantee that SP_1 refines to SP_2 under specification refinement. For example, consider the following specifications

SP_1 = **Elements** $P \in U_1$ **Restrictions** $P =_{U_1} \mathbb{N}$ **end**
SP_2 = **Elements** $P \in \{\mathbb{N}\}_{U_1}$ **Restrictions** T **end**

Under general refinement, $SP_1 \sqsubseteq^f SP_2$ for $f = \lambda m.\langle h(ce(m), eq\rangle \in SP_2 \to SP_1$. However, SP_1 does not refine to SP_2 under specification refinement since the restriction of SP_2 does not imply the restriction of SP_1. In practice, we rarely make refinements by weakening a restriction, but if we do, then we need to use general refinement, as exemplified above. Most refinements needed for program development can be made using specification refinement. As we will see later, specification refinement possesses some useful properties that general refinement does not.

8.3.1 An Example of a Specification Refinement

In this section, we give a small example of a specification refinement. Consider the following specification of a module containing two functions f and g:

$SP_0 \equiv$
Elements
$f \in \mathbb{N} \to \mathbb{N}$,
$g \in \mathbb{N} \to \mathbb{N}$
Restrictions
$\forall a \in \mathbb{N}.$
$[(f(a) > a) =_{\mathbb{B}} true] \quad \wedge$ (1)
$[(g(a) < f(a)) =_{\mathbb{B}} true]$ (2)
End

One possible refinement of SP_0 is given by the specification SP_1:

$SP_1 \equiv$
Elements
$f \in \mathbb{N} \to \mathbb{N}$,
$c \in \mathbb{N}$,
$g \in \mathbb{N} \to \mathbb{N}$
Restrictions
$\forall a \in \mathbb{N}.$
$[(f(a) = a + 27) =_{\mathbb{B}} true] \quad \wedge$ (3)
$[(g(a) = f(a) - c) =_{\mathbb{B}} true] \quad \wedge$ (4)
$[(c = 3) =_{\mathbb{B}} true]$ (5)
End

Note the addition of an extra component c in the signature of SP_1. New components, such as c, often play an intermediary role during refinement; for example, they may be used to break up the specification of complicated operations. Although the signature of SP_1 differs from the signature of SP_0, we can relate the two signatures by the following abstraction map:

$$h = \lambda m.\ \textbf{module}\ f = m.f, g = m.g\ \textbf{end} \in \mathit{Sig}(SP_1) \to \mathit{Sig}(SP_0)$$

Function h returns a computational element with signature $\mathit{Sig}(SP_0)$ by removing the constant component c from a computational element with signature $\mathit{Sig}(SP_1)$. We may observe that, for all f, g and c, the conjunction of (4) and (5) implies (2); and (3) implies (1). Hence, the restriction of SP_1 implies the restriction of SP_0, and proof object p, below, witnesses this implication:

$$p = \lambda m.\lambda x.\lambda a.\langle eq, \langle eq, eq\rangle\rangle \in \mathit{Res}(SP_0, SP_1, h)$$

Therefore, by the definition of specification refinement, we get that $SP_0 \sqsubseteq_h SP_1$. For an intuitive understanding, it is enough to observe that any computational element satisfying the restriction of SP_1 must also satisfy the restriction of SP_0 since the restriction of SP_1 is stronger than that of SP_0. Therefore, every implementation of SP_1 (ignoring component c) is an implementation of SP_0.

The form of SP_1, above, strongly suggests module m, below, as a possible implementation for SP_1 (i.e. $m \in SP_1$). It is easily shown, by constructive proof, that m is indeed an implementation for SP_1.

$$
\begin{array}{l}
m = \textbf{module} \\
\qquad \begin{array}{lcl} f & = & \lambda a \in \mathbb{N}.a + 27, \\ c & = & 3, \\ g & = & \lambda a \in \mathbb{N}.f(a) - c \end{array} \\
\quad \textbf{proof} \\
\qquad \lambda a \in \mathbb{N}.\langle eq, \langle eq, eq\rangle\rangle \\
\quad \textbf{end} \in SP_1
\end{array}
$$

We observe that module m does not quite satisfy of the original specification SP_0, since m contains the extra component c. However, by appealing to Fact 8.1, we can use the abstraction function h and proof map p to get the module

$$\langle h(ce(m)), p(ce(m))(pf(m))\rangle \in SP_0$$

Here,

$$
\begin{array}{l}
h(ce(m)) = \textbf{module} \\
\qquad\qquad \begin{array}{lcl} f & = & \lambda a \in \mathbb{N}.a + 27, \\ g & = & \lambda a \in \mathbb{N}.f(a) - 3 \end{array} \\
\qquad\quad \textbf{end} \in Sig(SP_0)
\end{array}
$$

and

$$p(ce(m))(pf(m)) = \lambda a \in \mathbb{N}.\langle eq, eq\rangle \in Ax[SP_0](h(ce(m)))$$

Remark In practice, we do not bother calculating the new witness $p(ce(m))(pf(m))$ since, as far as program development is concerned, we are usually only interested in the computational element of an implementation □

8.4 Basic Properties of Refinement

In general, for two arbitrary specifications SP_1 and SP_2, verifying that SP_2 is a refinement of SP_1 is difficult. However, since we have a formal definition of refinement, we can give a collection of useful laws that can help make refinements. It is important to be aware that refinement laws must be proved, otherwise there is no guarantee that refinement yields acceptable specifications and implementations. The laws given below are only the most basic laws of refinement, and are in no way a complete collection. If a sufficient number of useful laws can be developed, they can be built up into a calculus. Rather than guess a refinement, and then verify it afterwards, the motivation behind a calculus is that we refine a specification by systematically applying the laws of the calculus until we arrive at an acceptable refinement.

The first property we give is that the specification refinement relation is transitive:

Theorem 8.2 (Transitivity of $\sqsubseteq_h$)
If $SP_1 \sqsubseteq_{h_1} SP_2$ and $SP_2 \sqsubseteq_{h_2} SP_3$ then $SP_1 \sqsubseteq_{h_1 \circ h_2} SP_3$
□

Proof Given in Figure 8.2 □

The transitivity property of $\sqsubseteq$ is particularly useful as it allows us to refine a specification via a series of intermediate refinements. For example, we might refine a specification SP_0 to SP_n as follows:

$$SP_0 \sqsubseteq_{h_1} SP_1 \sqsubseteq_{h_2} SP_2 \ldots \sqsubseteq_{h_{n-1}} SP_{n-1} \sqsubseteq_{h_n} SP_n$$

where $SP_1, \ldots, SP_{n-1}$ are intermediate refinements; and $h_1, \ldots, h_n$ are abstraction maps. Given such a series of refinements, we can use Theorem 8.2 to justify that $SP_0 \sqsubseteq_h SP_n$ where $h = h_1 \circ h_2 \circ \ldots \circ h_n$.

Fact 8.2 If $SP_1 \sqsubseteq SP_2$ and $SP_2 \sqsubseteq SP_3$ then $SP_1 \sqsubseteq SP_3$. □

If two specifications are equivalent under specification equality $\Leftrightarrow$—they need not be identical—then they are also refinements of each other:

Theorem 8.3 If $SP_1 \Leftrightarrow SP_2$ then $SP_1 \sqsubseteq SP_2$ and $SP_2 \sqsubseteq SP_1$ □

Proof If $SP_1 \Leftrightarrow SP_2$ then by the definition of $\Leftrightarrow$ we can deduce that $Sig(SP_1) = Sig(SP_1)$ and that the restrictions of SP_1 and SP_2 are equivalent:

$$(1).\ \forall x \in Sig(SP_1).Ax(SP_1)(x) \Leftrightarrow Ax(SP_2)(x)$$

From (1) we may deduce that there exists two proof maps p_1 and p_2 such that,

$$\begin{array}{lll} p_1 & \in & \forall x \in Sig(SP_2).\ Ax(SP_2)(x) \Rightarrow Ax(SP_1)(x) \\ p_2 & \in & \forall x \in Sig(SP_1).\ Ax(SP_1)(x) \Rightarrow Ax(SP_2)(x) \end{array}$$

By the definition of specification refinement, proof maps p_1 and p_2 immediately give us that $SP_1 \sqsubseteq SP_2$ and $SP_2 \sqsubseteq SP_1$.
□

Theorem 8.3 is useful. It allows us to conclude that if we rearrange the structure of a specification to produce an equivalent specification, then the rearranged specification and the original specification are refinements of each other. Consequently, many of the properties of structured specifications given in Chapter 7 can also be used to help make refinements. For example, we may conclude that $SP\backslash i\backslash j \sqsubseteq SP\backslash(i \cup j)$ and $SP\backslash(i \cup j) \sqsubseteq SP\backslash i\backslash j$ since we have already shown that $SP\backslash i\backslash j \Leftrightarrow SP\backslash(i \cup j)$ in Theorem 7.6.

Given $SP_1 \sqsubseteq_{h_1} SP_2$ and $SP_2 \sqsubseteq_{h_2} SP_3$ we can deduce that there exists two proof maps p_1 and p_2 such that:

$p_1 \in Res(SP_1, SP_2, h_1)$ where $h_1 \in Sig(SP_2) \to Sig(SP_1)$
$p_2 \in Res(SP_2, SP_3, h_2)$ where $h_2 \in Sig(SP_3) \to Sig(SP_2)$

The proof proceeds as follows:

```
0.0  |[   $m \in Sig(SP_3)$
     ▷      "0.0 and type of $p_2$"
0.1       $p_2(m) \in Ax(SP_3)(m) \Rightarrow Ax(SP_2)(h_2(m))$
            "0.0, type of $h_2$"
0.2       $h_2(m) \in Sig(SP_2)$
            "0.2 and type of $p_1$"
0.3       $p_2(h_2(m)) \in Ax(SP_2)(h_2(m)) \Rightarrow Ax(SP_1)((h_1 \circ h_2)(m))$
            "0.1, 0.3 and function composition"
0.4       $(p_2(h_2(m))) \circ (p_2(m)) \in Ax(SP_3)(m) \Rightarrow Ax(SP_1)((h_1 \circ h_2)(m))$
     ]|
       "0.0, 0.4, $\to$ - introduction"
1    $\lambda m \in Sig(SP_3).\ (p_2(h_2(m))) \circ (p_2(m)) \in Res(SP_1, SP_3, h_1 \circ h_2)$
       "1, definition of refinement"
2    $SP_1 \sqsubseteq_{h_1 \circ h_2} SP_3$
```

Figure 8.2: A proof of Theorem 8.2

We can also use Theorem 8.3 to conclude that any specification is a refinement of itself. That is to say, the refinement relation is reflexive:

Theorem 8.4 (Reflexivity) $SP \sqsubseteq SP$. □

Proof Follows immediately by Theorem 8.3 since $SP = SP$ implies $SP \Leftrightarrow SP$ □

Simple refinement enjoys a kind of anti-symmetry:

Fact 8.3 If $SP_1 \sqsubseteq SP_2$ and $SP_2 \sqsubseteq SP_1$ then $SP_1 \Leftrightarrow SP_2$. □

As an aside, general refinement is transitive and reflexive, just like specification refinement:

Fact 8.4 Let P, Q, and R be types. If $P \sqsubseteq^f Q$ and $Q \sqsubseteq^g R$ then $P \sqsubseteq^{f \circ g} R$. □

Fact 8.5 Let $Id_P \in P \to P$ be an identity function. $P \sqsubseteq^{Id_P} P$ □

To summarise this section: we have given a collection of laws that allow us to refine specifications via a series of intermediate refinements.

8.5 Refinement of Canonical Specifications

In this section, we consider the refinement of canonical specifications; that is, specifications that do not use structuring operations. Most refinements of canonical specifications fall into three broad categories: strengthening restrictions, adding components to signatures, and decomposing canonical specifications into structured specifications. In the following we give refinement laws for all three categories mentioned above. We will also show that we can refine a specification by refining its signature using general refinement.

8.5.1 Strengthening Restrictions

Theorem 8.5, below, says that a specification is refined by strengthening its restriction. Typically, a restriction is strengthened by adding more axioms, and this fact gives us Fact 8.6 below. We can also use the *enrich* and *union* operators to add new restrictions to a specification. Consequently, Fact 8.6 justifies the refinement laws given in Facts 8.7 and 8.8.

Theorem 8.5 If $\forall m \in S.\ R_2(m) \Rightarrow R_1(m)$ then $\langle S \mid (m)R_1 \rangle \sqsubseteq \langle S \mid (m)R_2 \rangle$ □

Proof Follow immediately from the definition of specification refinement. □

Fact 8.6 $\langle S \mid (m)R_1 \rangle \sqsubseteq \langle S \mid (m)R_1 \times R_2 \rangle$ where $[\![m \in S \rhd R_2(m)\ \mathbf{type}]\!]$ □

Fact 8.7 $SP \sqsubseteq SP \lhd \Lambda\langle \Phi \mid R_2 \rangle$ where $[\![m \in Sig(SP) \rhd R_2(m)\ \mathbf{type}]\!]$ □

Fact 8.8 $SP_1 \sqsubseteq SP_1 \cup SP_2$ assuming $Sig(SP_1) = Sig(SP_2)$ □

8.5.2 Refinement by Adding Components

When making refinements, we often add auxiliary components which only play an intermediate role in helping to make refinements; for example, auxiliary components are often used to simplify the definitions of other components in a specification. It turns out that adding extra components to a specification yields a refinement. We can use signature concatenation to add new components to a specification, and this gives us the following refinement law:

Fact 8.9 $\langle S_1 \mid (m)R \rangle \sqsubseteq_{\substack{S_1 S_2 \\ \Pi_1}} \langle S_1 \oplus S_2 \mid (R \circ {}^{S_1 S_2}_{\Pi_1})(m) \rangle$ where $S_1 \vdash S_2$ **sig** □

Recall that the projection function ${}^{S_1 S_2}_{\Pi_1} \in (S_1 \oplus S_2) \rightarrow S_1$ takes any computational element $m \in S_1 \oplus S_2$ and removes the components of m that are specified in S_2. Note that we compose R with ${}^{S_1 S_2}_{\Pi_1}$ to ensure that the restriction of the refinement is dependent on all $m \in S_1 \oplus S_2$ since R is only dependent on all $m \in S_1$.

0.0 $[\![$ $x \in Sig(\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle)$;

0.1 $r \in Ax(\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle)(x)$;

$\rhd$ "0.0, definition of $(+)$"

0.2 $x \in S_1 \oplus S_2$

"0.1, definition of $(+)$"

0.3 $r \in (R_1 \circ {}^{S_1S_2}_{\Pi_1})(x) \times (R_2 \circ {}^{S_1S_2}_{\Pi_2})(x)$

"0.3, $\times$ - elimination"

0.4 $fst(r) \in R_1({}^{S_1S_2}_{\Pi_1}(x))$

"0.4, definition of $(+)$"

0.5 $fst(r) \in Ax(\langle S_1 \mid (m)R_1\rangle)({}^{S_1S_2}_{\Pi_1}(x))$

$]\!]$

"0.0, 0.5, $\rightarrow$ - introduction"

1 $\lambda x.\lambda r.fst(r) \in \forall x \in S_1 \oplus S_2.Ax(\langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle)(x) \Rightarrow Ax(\langle S_1 \mid (m)R_1\rangle)({}^{S_1S_2}_{\Pi_1}(x))$

"1, definition of refinement"

2 $\langle S_1 \mid (m)R_1\rangle \sqsubseteq_{{}^{S_1S_2}_{\Pi_1}} \langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle$

Figure 8.3: Proof of Theorem 8.6

The *sum* and *enrich* operators may also be used to add extra components, and restrictions, to a specification. In fact, it can be shown that a specification can be refined by summing it with any specification:

Theorem 8.6 $\langle S_1 \mid (m)R_1\rangle \sqsubseteq_{{}^{S_1S_2}_{\Pi_1}} \langle S_1 \mid (m)R_1\rangle + \langle S_2 \mid (m)R_2\rangle$ □

Proof Given in Figure 8.3. □

Fact 8.10 $\langle S_1 \mid (m)R_1\rangle \sqsubseteq_{{}^{S_2S_1}_{\Pi_2}} \langle S_2 \mid (m)R_2\rangle + \langle S_1 \mid (m)R_1\rangle$ □

Fact 8.11 $\langle S_1 \mid (m)R_1\rangle \sqsubseteq_{{}^{S_1S_2}_{\Pi_1}} \langle S_1 \mid (m)R_1\rangle \lhd \Lambda\langle S_2 \mid R_2\rangle$ where $S_1 \vdash S_2$ **sig** □

8.5.3 Refining Signatures

A signature is a type, and can therefore be refined using general refinement. Moreover, any refinement of the signature of a specification gives a refinement of that specification:

Theorem 8.7 if $S_1 \sqsubseteq^f S_2$ then $\langle S_1 \mid (m)R\rangle \sqsubseteq_f \langle S_2 \mid (m)(R \circ f)(m)\rangle$ □

Proof From the assumption $S_1 \sqsubseteq^f S_2$ we may deduce that $f \in S_2 \rightarrow S_1$. By the definition of specification refinement, $\langle S_1 \mid (m)R\rangle \sqsubseteq_f \langle S_2 \mid (m)(R \circ f)(m)\rangle$ holds given any proof map $p \in \forall x \in S_2.(R \circ f)(x) \Rightarrow (R \circ f)(x)$. It is easy to verify that $\lambda x \in S_2.\lambda r \in (R \circ f)(x).r$ is a suitable candidate for p. □

Note that in Theorem 8.7 the restriction of the refinement is $R \circ f$: composing R with f ensures that the restriction of the refinement is well typed for all $m \in S_2$.

Theorem 8.8, below, says that a signature is refined by adding new components to it. This fact, together with Theorem 8.7 justifies Fact 8.9, given earlier, which says that a specification is refined by adding new components to its signature.

Theorem 8.8 $S_1 \sqsubseteq^{\overset{S_1 S_2}{\Pi_1}} S_1 \oplus S_2$ where S_1 **sig** and $S_1 \vdash S_2$ **sig** $\square$

Proof ${}^{S_1 S_2}_{\Pi_1} \in (S_1 \oplus S_2) \rightarrow S_1$ $\square$

A signature may be refined by refining the types of the components that appear in it. Such a refinement of the signature of a specification gives a refinement of the specification. For example, the type U_1 can be refined to $\{P\}_{U_1}$ for any $P \in U_1$ since $\{P\}_{U_1}$ is a subtype of U_1; formally, $U_1 \sqsubseteq^{Id} \{P\}_{U_1}$ where $Id \in \{P\}_{U_1} \rightarrow U_1$ is an identity function. From this fact we get the following refinement law:

Theorem 8.9 $\langle y \in U_1, S \mid (m)R\rangle \sqsubseteq \langle y \in \{P\}_{U_1}, S \mid (m)R\rangle$ for any $P \in U_1$ $\square$

Proof $(y \in U_1, S) \sqsubseteq^{Id} (y \in \{P\}_{U_1}, S)$ where $Id \in (y \in \{P\}_{U_1}, S) \rightarrow (y \in U_1, S)$ is an identity function, and $P \in U_1$. The proof then follows by Theorem 8.7 $\square$

The above refinement law is particularly useful since many specifications contain a type component such as $y \in U_1$, and we usually refine it by choosing an actual implementation for y such as $\mathbb{N}$, $\mathbb{B}$ etc.

The development of refinement laws for types, other than specifications, is beyond the scope of this thesis. Nevertheless, Theorem 8.9 illustrates that, at least in principle, a specification can be refined by refining the types within its signature. It also illustrates the usefulness of having a notion of general refinement that is a generalisation of specification refinement.

8.5.4 Refinement by Structuring Canonical Specifications

If a canonical specification contains many components in its signature, or many axioms in its restriction, then it can be difficult to refine. In particular, we can be overwhelmed by the clerical task of managing large numbers of axioms and components. In such cases, refinement can be made easier by decomposing a large canonical specification into smaller, more manageable, pieces. Typically, the decomposition takes the form of expressing the original specification as the composition of several, smaller, specification that are 'glued' together using the specification operators. Once a canonical specification is decomposed into separate specifications, we can refine each specification in isolation. The refinements of each piece of the decomposed specification can then be combined to produce a refinement of the original canonical specification. In this section, we consider some laws that allow specifications to be refined by decomposing them using the structuring operators.

Typically, we decompose a canonical specification by splitting its axioms and/or signature into separate pieces. For example, in Chapter 7 we gave Theorem 7.4 that says that a specification can, under certain conditions, be expressed as the sum of, some, two specifications. That sum is a refinement of the original specification:

Theorem 8.10 If $\forall m \in (S_1 \oplus S_2).\ R_1(m) \times R_2(m) \Leftrightarrow (R_1 \circ {}^{S_1 S_2}_{\Pi_1})(m) \times (R_2 \circ {}^{S_1 S_2}_{\Pi_2})(m)$ then $\langle S_1 \oplus S_2 \mid (m)R_1 \times R_2 \rangle \sqsubseteq \langle S_1 \mid (m)R_1 \rangle + \langle S_2 \mid (m)R_2 \rangle$.
□

Proof $\langle S_1 \oplus S_2 \mid (m)R_1 \times R_2 \rangle \Leftrightarrow \langle S_1 \mid (m)R_1 \rangle + \langle S_2 \mid (m)R_2 \rangle$ by Theorem 7.4. Hence, the proof of the theorem follows immediately by Theorem 8.3. □

Fact 8.12 $\langle S \mid (m)R_1 \times R_2 \rangle \sqsubseteq \langle S \mid (m)R_1 \rangle \cup \langle S \mid (m)R_2 \rangle$ □

Fact 8.12, above, says that we can refine a specification by using the *union* operator to split its restriction into separate pieces; this fact follows immediately from the definition of *union*.

We finish this section by giving some refinement laws, below, that are used to refine canonical specifications to structured specifications. Strictly speaking, the rules given below can also be applied to structured specifications, but since they are mostly used to refine canonical specifications, we give them here. Note that Facts 8.13 and 8.14 use the inverses of the computational element renaming and hiding operators, respectively; these inverse operators were given, previously, in Section 6.11.

Fact 8.13 Let $h = \_[p\backslash q]^{-1}_{Sig(SP)} \in Sig(SP)[p\backslash q] \rightarrow Sig(SP)$. $SP \sqsubseteq_h SP[p\backslash q]$ □

Fact 8.14 Let $h = \_\backslash^{-1}_{Sig(SP)} i \in Sig(SP)\backslash i \rightarrow Sig(SP)$. $SP \sqsubseteq_h SP\backslash i$ □

Fact 8.15 $SP \sqsubseteq_f SP \uparrow f$ where $f \in S \rightarrow Sig(SP)$ and S **sig** □

8.6 Refinement of Structured Specifications

To be useful, the refinement of specifications must proceed in a stepwise manner. In other words, we want to refine a specification by refining its individual parts in relative isolation, and then combine its refined parts to get a refinement of the original specification. We will give refinement laws that show that if $SP_1 \sqsubseteq_h SP_2$ then there exists some abstraction map g such that $F(SP_1) \sqsubseteq_g F(SP_2)$ for all specifications SP_1 and SP_2, and specification operators F. In other words, the structure of a specification carries over to its refinement.

In practice, most refinements are made using the simple refinement relation ($\sqsubseteq$), and we will also show that the specification operators are monotonic with respect to $\sqsubseteq$:

$$SP_1 \sqsubseteq SP_2 \Rightarrow F(SP_1) \sqsubseteq F(SP_2)$$

for all specifications SP_1 and SP_2, and specification operators F. The transitivity of $\sqsubseteq$, and the monotonicity of the specification operators with respect to $\sqsubseteq$, are the two properties that make stepwise refinement an effective notion for structured specifications. The laws given in this section are in no way complete, but do serve to illustrate that structured specifications can be refined in a stepwise manner.

Some of the proofs of the refinement laws presented in this section are rather large, and for presentation, they are given in figures at the end of this chapter.

8.6.1 Refinement of Rename and Hide

We begin by considering the refinement of specifications produced using the *rename* operator. The structure of a specification of the form $SP[p\backslash q]$ carries over to a refinement of such a specification:

Theorem 8.11
Let $h = (\_[p\backslash q]) \circ g \circ (\_[p\backslash q]^{-1}_{Sig(SP_2)}) \in Sig(SP_2)[p\backslash q] \to Sig(SP_1)[p\backslash q]$ where

$$\begin{array}{lll} g & \in & Sig(SP_2) \to Sig(SP_1)\ ; \\ \_[p\backslash q] & \in & Sig(SP_1) \to Sig(SP_1)[p\backslash q]\ ;\ \text{and} \\ \_[p\backslash q]^{-1}_{Sig(SP_2)} & \in & Sig(SP_2)[p\backslash q] \to Sig(SP_2) \end{array}$$

If $SP_1 \sqsubseteq_g SP_2$ then $SP_1[p\backslash q] \sqsubseteq_h SP_2[p\backslash q]$.
□

Proof Given in Figure 8.5 □

Note that in the above theorem, the abstraction map h is defined using the renaming operator $\_[p\backslash q]$, and the inverse renaming operator $\_[p\backslash q]^{-1}_{Sig(SP_2)}$. To see how h arises observe that g does not have a suitable type to be an abstraction map between $SP_1[p\backslash q]$ and $SP_2[p\backslash q]$, but composing g with $\_[p\backslash q]$ and $\_[p\backslash q]^{-1}_{Sig(SP_2)}$ produces a suitably typed abstraction map h.

We may deduce, as a special case of Theorem 8.11, that the *rename* operator is monotonic with respect to the simple refinement relation $\sqsubseteq$:

Theorem 8.12 if $SP_1 \sqsubseteq SP_2$ then $SP_1[p\backslash q] \sqsubseteq SP_2[p\backslash q]$ □

Proof Let $Sig(SP_1) = Sig(SP_2)$ and g be an identity function in Theorem 8.11. Then h is an identity function □

We now consider the refinement of specifications that are produced by applying the *hide* operator. The structure of a specification of the form $SP\backslash i$ carries over to a refinement of such a specification:

Theorem 8.13
Let $h = (\_\backslash i) \circ g \circ (\_\backslash^{-1}_{Sig(SP_2)} i) \in Sig(SP_2)\backslash i \rightarrow Sig(SP_1)\backslash i$ where

$$\begin{array}{lcl} g & \in & Sig(SP_2) \rightarrow Sig(SP_1)\ ; \\ \_\backslash i & \in & Sig(SP_1) \rightarrow Sig(SP_1)\backslash i\ ;\ \text{and} \\ \_\backslash^{-1}_{Sig(SP_2)} i & \in & Sig(SP_2)\backslash i \rightarrow Sig(SP_2) \end{array}$$

If $SP_1 \sqsubseteq_g SP_2$ then $SP_1\backslash i \sqsubseteq_h SP_2\backslash i$.
□

Proof Similar to that of Theorem 8.11 above □

Note that in the above theorem, the abstraction map h is defined using the hiding operator $\_\backslash i$, and the inverse hiding operator $\_\backslash^{-1}_{Sig(SP_2)} i$. To see how h arises observe that g does not have a suitable type to be an abstraction map between $SP_1\backslash i$ and $SP_2\backslash i$, but composing g with $\_\backslash i$ and $\_\backslash^{-1}_{Sig(SP_2)} i$ produces a suitably typed abstraction map h.

We may deduce, as a special case of Theorem 8.13, that the *hide* operator is monotonic with respect to the simple refinement relation $\sqsubseteq$:

Theorem 8.14 if $SP_1 \sqsubseteq SP_2$ then $SP_1\backslash i \sqsubseteq SP_2\backslash i$ □

Proof Let $Sig(SP_1) = Sig(SP_2)$ and g be an identity function in Theorem 8.13. Then h is an identity function □

8.6.2 Refinement of Union

Given a specification of the form $SP_1 \cup SP_2$, if we refine SP_1 and SP_2 to some specifications SP_3 and SP_4, respectively, then the signatures of SP_3 and SP_4 must be equal for $SP_3 \cup SP_4$ to be a specification. If SP_1 and SP_2 are refined using the same abstraction map then the signatures of SP_3 and SP_4 are equal . In that case $SP_3 \cup SP_4$ is a refinement of $SP_1 \cup SP_2$:

Theorem 8.15
If $SP_1 \sqsubseteq_h SP_3$ and $SP_2 \sqsubseteq_h SP_4$ then $SP_1 \cup SP_2 \sqsubseteq_h SP_3 \cup SP_4$ where $h \in Sig(SP_3) \rightarrow Sig(SP_1)$ assuming $Sig(SP_1) = Sig(SP_2)$ and $Sig(SP_3) = Sig(SP_4)$
□

Proof The proof is given in Figure 8.6. □

As a special case of Theorem 8.15, we may deduce that the *union* operator is monotonic with respect to the simple refinement relation $\sqsubseteq$:

Theorem 8.16 if $SP_1 \sqsubseteq SP_3$ and $SP_2 \sqsubseteq SP_4$ then $SP_1 \cup SP_2 \sqsubseteq SP_3 \cup SP_4$ assuming that the signatures of SP_1, SP_2, SP_3 and SP_4 are all equal. □

Proof Let the signatures of SP_1, SP_2, SP_3 and SP_4 be equal in Theorem 8.15, and let h be an identity function □

8.6.3 Refinement of Sum and Enrich

We may refine the sum of any two specifications be refining them independently of each other and then summing their refinements:

Theorem 8.17
Let $S_i = Sig(SP_i)$ for $1 \leq i \leq 4$, and $h = \lambda m.\ f({}^{S_3S_4}\Pi_1(m)) \oplus g({}^{S_3S_4}\Pi_2(m)) \in (S_3 \oplus S_4) \rightarrow (S_1 \oplus S_2)$ where $f \in S_3 \rightarrow S_1$ and $g \in S_4 \rightarrow S_2$. If $SP_1 \sqsubseteq_f SP_3$ and $SP_2 \sqsubseteq_g SP_4$ then $SP_1 + SP_2 \sqsubseteq_h SP_3 + SP_4$
□

Proof The proof is given in Figure 8.7. □

Note that in the above theorem, the abstraction function h is defined using the projection functions ${}^{S_3S_4}\Pi_1 \in (S_3 \oplus S_4) \rightarrow S_3$ and ${}^{S_3S_4}\Pi_2 \in (S_3 \oplus S_4) \rightarrow S_4$; and the concatenation operator $\oplus$. To see how h arises observe that h takes any $m \in S_3 \oplus S_4$ and applies f to that part of m specified by S_3 (i.e. to ${}^{S_3S_4}\Pi_1(m)$); and h applies g to that part of m specified by S_4 (i.e. to ${}^{S_3S_4}\Pi_2(m)$).

We may deduce, as a special case of Theorem 8.17, that *sum* is monotonic with respect to simple refinement:

Theorem 8.18 if $SP_1 \sqsubseteq SP_3$ and $SP_2 \sqsubseteq SP_4$ then $SP_1 + SP_2 \sqsubseteq SP_3 + SP_4$ □

Proof Let f and g be identity functions in Theorem 8.17. Then h is an identity function □

We now consider refining specifications that are produced by the *enrich* operator. Such specifications have the form $SP \lhd \Lambda\langle S \mid R\rangle$, and are refined by refining SP:

Theorem 8.19
Let $S_i = Sig(SP_i)$ for $1 \leq i \leq 2$, and $h = \lambda m.\ g({}^{S_2S}\Pi_1(m)) \oplus {}^{S_2S}\Pi_2(m) \in S_2 \oplus S \rightarrow S_1 \oplus S$ where $g \in S_2 \rightarrow S_1$ and $S_2 \vdash S$ **sig**. If $SP_1 \sqsubseteq_g SP_2$ then $SP_1 \lhd \Lambda\langle S \mid R\rangle \sqsubseteq_h SP_2 \lhd \Lambda\langle S \mid R \circ h\rangle$.
□

Proof Similar to that of Theorem 8.17 above. □

The following is an explanation of Theorem 8.19 above. The abstraction map h arises for a similar reason as the abstraction map h in Theorem 8.17 (see the discussion after Theorem 8.17). The assumption $S_2 \vdash S$ **sig** is necessary to ensure that the refinement is well typed. We also compose R with h, in the enrichment of the refinement, to make the refinement well typed: this composition makes the restriction of the enrichment dependent on all $m \in S_2 \oplus S$.

We may deduce, as a special case of Theorem 8.19, that the *enrich* operator is monotonic with respect to the simple refinement relation $\sqsubseteq$:

Theorem 8.20 if $SP_1 \sqsubseteq SP_2$ then $SP_1 \lhd \Lambda\langle S \mid R\rangle \sqsubseteq SP_2 \lhd \Lambda\langle S \mid R\rangle$ □

Proof Let g be an identity function in Theorem 8.19. Then h is also an identity function. □

Strengthening the restriction of an enrichment also yields a refinement of an *enrich* operation:

Theorem 8.21
$SP \lhd \Lambda\langle S \mid R_1\rangle \sqsubseteq SP \lhd \Lambda\langle S \mid R_2\rangle$ if $\forall m \in Sig(SP) \oplus S.\ R_2(m) \Rightarrow R_1(m)$
□

Proof Apply Theorem 8.5 □

8.6.4 Refinement of Translate and Derive

Given any specification of the form $SP_1 \uparrow f$, and any refinement SP_2 of SP_1, then $SP_2 \uparrow f$ is not always a refinement of $SP_1 \uparrow f$. This is because for $SP_2 \uparrow f$ to be a valid specification the signature of SP_2 must be equal to the range of f. This requirement is equivalent to saying that the signatures of SP_1 and SP_2 must be equal. In that case, we can show that *translate* is monotonic with respect to $\sqsubseteq$:

Theorem 8.22
if $SP_1 \sqsubseteq SP_2$ then $SP_1 \uparrow f \sqsubseteq SP_2 \uparrow f$ for any S **sig** and $f \in S \rightarrow Sig(SP_2)$ □

Proof Let g be an identity function in Theorem 8.23 below □

Theorem 8.23
If $SP_1 \sqsubseteq_g SP_2$ then $SP_1 \uparrow (g \circ f) \sqsubseteq SP_2 \uparrow f$ for any S **sig**, $f \in S \rightarrow Sig(SP_2)$ and $g \in Sig(SP_2) \rightarrow Sig(SP_1)$ □

Proof The proof is given in Figure 8.8 □

Theorem 8.22 is a special case of Theorem 8.23. Theorem 8.23 has limited applications since it is only applicable to *translate* expressions whose *translate* maps have the form $g \circ f$. In practice, most refinements are made using the simple refinement relation $\sqsubseteq$, so that Theorem 8.22 is sufficient to refine most applications of *translate*.

We now consider refinements involving the *derive* operator. To begin with, the structure of a specification of the form $SP \downarrow f$ carries over to a refinement of such a specification:

Theorem 8.24
If $SP_1 \sqsubseteq_g SP_2$ then $SP_1 \downarrow f \sqsubseteq SP_2 \downarrow (f \circ g)$ for any S **sig**, $f \in Sig(SP_1) \rightarrow S$, and $g \in Sig(SP_2) \rightarrow Sig(SP_1)$ □

Proof The proof is given in Figure 8.9. □

Note that in the above theorem we use $f \circ g$ as a *derive* map for SP_2. To see how $f \circ g$ arises observe that f is not a suitably typed *derive* map for SP_2—since the domain of f is not equal to $Sig(SP_2)$—but composing f with g gives a suitably typed *derive* map for SP_2.

We can deduce, as a special case of Theorem 8.24, that the *derive* operator is monotonic with respect to the simple refinement relation $\sqsubseteq$:

Theorem 8.25 if $SP_1 \sqsubseteq SP_2$ then $SP_1 \downarrow f \sqsubseteq SP_2 \downarrow f$ for any S **sig** and $f \in Sig(SP_1) \rightarrow S$ □

Proof Let $Sig(SP_1) = Sig(SP_2)$ and g be an identity function in Theorem 8.24 □

8.6.5 Summary

To summarise Section 8.6: the structure of a structured specification carries over to its refinements and, as a special case of this property, the structuring operators are monotonic with respect to the simple refinement relation.

8.7 An Example Refinement

We present an example to illustrate the use of the refinement laws given in this chapter. We will refine the specification *Mean* given, previously, in Figure 3.1; the refinement is based on an example in [36]. Recall that *Mean* supplies a type, named *Data*, for samples of naturals. *Mean* also supplies operations to calculate the sum, size and mean of members of *Data*, and it supplies an operation to add naturals to members of *Data*. *Mean* is a canonical specification, and we shall refine it to a structured specification. Consequently, the example will illustrate the refinement of canonical and structured specifications.

We aim for a refinement of *Mean* that is defined in terms of the specification *PointSpec* given, previously, in Figure 4.7. *PointSpec* specifies a product-like type, named *Point*, for Cartesian points. We refine *Mean* by choosing to implement *Data* by *Point*, such that *Data* values are pairs that contain the sum and size of a sample of naturals. Our refinement of *Mean* is named *Mean*$_3$, and is given in Figure 8.4. An energetic reader would not find it intellectually taxing to verify that the restriction of *Mean*$_3$ implies the restriction of *Mean*, but it would be long and tedious work. In practice, *Mean*$_3$ is arrived at via a series of intermediate refinements. We give an outline of the refinement of *Mean*. The refinement is made in several stages, and we give a commentary on the refinement steps as we proceed.

8.7.1 The Refinement of *Mean*

Stage 1 The first move in the refinement of *Mean* is to combine *PointSpec* with *Mean*, and rename the type *Point*, in *PointSpec*, to *Pair*. The renaming simply gives *Point* a more meaningful name in the context in which it is used. We let $S_1 = Sig(PointSpec[Point\backslash Pair])$ and $S_2 = Sig(Mean)$:

$$
\begin{array}{ll}
 & Mean \\
\underset{\Pi_2}{\sqsubseteq_{S_1 S_2}} & \text{“Fact 8.10”} \\
 & PointSpec[Point\backslash Pair] + Mean
\end{array}
$$

The above refinement supplies the operations supplied by *PointSpec* and *Mean*. We aim to use the components in *PointSpec* to refine the components and restriction of *Mean*. We rewrite the above refinement in a form that reveals the operations and restrictions supplied by *Mean*:

$\sqsubseteq$ “Theorem 7.3 (transform *sum* to *enrich*), Theorem 8.3”

$$
\begin{array}{l}
(PointSpec[Point\backslash Pair]) \lhd \\
\Lambda\textbf{Elements} \\
\quad Data \in U_1 \\
\quad clear \in Data \\
\quad \bullet sum \in Data \to \mathbb{N} \\
\quad size \in Data \to \mathbb{N} \\
\quad enter \in \mathbb{N} \to Data \to Data \\
\quad mean \in \prod d \in Data.(\textbf{if } size(d) \neq 0 \textbf{ then } \mathbb{N} \textbf{ else } \mathbb{S}) \\
\textbf{Restrictions} \\
\quad \forall d \in Data. \forall n \in \mathbb{N}. \\
\quad sum(clear) =_{\mathbb{N}} 0 \quad \wedge \qquad\qquad (1) \\
\quad sum(enter(n,d)) =_{\mathbb{N}} sum(d) + n \quad \wedge \qquad (2) \\
\quad size(clear) =_{\mathbb{N}} 0 \quad \wedge \qquad\qquad (3) \\
\quad size(enter(n,d)) =_{\mathbb{N}} size(d) + 1 \quad \wedge \qquad (4) \\
\quad mean(d) = (\textbf{if } size(d) \neq 0 \textbf{ then } sum(d) \text{ div } size(d) \textbf{ else } \text{“error”}) \\
\textbf{End}
\end{array}
$$

Next, we add our main design decision that *Data* is to be implemented by *Pair*. We do this in two refinement steps. Firstly, we refine the type of *Data* (i.e. U_1) to $\{Pair\}_{U_1}$; such a refinement says that *Data* equals *Pair*. Consequently, operations on *Pair* values are valid on *Data* values, and vice versa. By Theorem 8.9, replacing the type of *Data* with $\{Pair\}_{U_1}$ results in a refinement of the above refinement. We call the resultant refinement $Mean_2$, and omit writing it out as it is identical to the above refinement, except that $Data \in U_1$ is replaced by $Data \in \{Pair\}_{U_1}$ in $Mean_2$.

$\sqsubseteq$ “Theorem 8.9”

$\quad Mean_2$

The second step in implementing *Data* by *Pair* is to specify that *Data* values are pairs that contain the size and sum of a sample of naturals:

$\sqsubseteq$ "Fact 8.7 (any enrichment is a refinement)"

$$
\begin{array}{l}
Mean_2 \lhd \\
\Lambda\textbf{Elements} \\
\quad \Phi \\
\quad \textbf{Restrictions} \\
\quad\quad \forall d \in Data. \\
\quad\quad sum(d) = X(d) \quad \wedge \qquad\qquad\qquad\qquad (a) \\
\quad\quad size(d) = Y(d) \qquad\qquad\qquad\qquad\quad\; (b) \\
\quad \textbf{End}
\end{array}
$$

Note that the signature of the above enrichment of $Mean_2$ is empty (Φ). We use axioms (a) and (b) to calculate new restrictions that subsume the old restrictions (1)-(4). We first consider axioms (2) and (4). By substituting X for *sum*, and Y for *size*, in the left hand side of (2) and (4), together with straightforward logical manipulation and case analysis, we may calculate that (c), below, implies (2) and (4). Similarly, we may deduce that (d), below, implies (1) and (3).

$$
\begin{array}{lcll}
enter(n,d) & = & mkPoint(sum(d)+n, size(d)+1) & (c) \\
clear & = & mkPoint(0,0) & (d)
\end{array}
$$

Hence, we may show that $(a) \wedge (b) \wedge (c) \wedge (d)$ implies $(1) \wedge (2) \wedge (3) \wedge (4)$. By this fact, and Theorem 8.21, we deduce the following refinement step:

$\sqsubseteq$ "Theorem 8.21 (strengthening the restriction of an enrichment)"

$$
\begin{array}{l}
(PointSpec[Point \backslash Pair]) \lhd \\
\Lambda\textbf{Elements} \\
\quad\quad Data \in \{Pair\}_{U_1} \\
\quad\quad clear \in Data \\
\quad\quad \bullet sum \in Data \rightarrow \mathbb{N} \\
\quad\quad size \in Data \rightarrow \mathbb{N} \\
\quad\quad enter \in \mathbb{N} \rightarrow Data \rightarrow Data \\
\quad\quad mean \in \prod d \in Data.(\textbf{if } size(d) \neq 0 \textbf{ then } \mathbb{N} \textbf{ else } String) \\
\quad \textbf{Restrictions} \\
\quad\quad \forall n \in \mathbb{N}.\forall d \in Data. \\
\quad\quad enter(n,d) = mkPoint(sum(d)+n, size(d)+1) \quad \wedge \\
\quad\quad clear = mkPoint(0,0) \quad \wedge \\
\quad\quad sum(d) = X(d) \quad \wedge \\
\quad\quad size(d) = Y(d) \quad \wedge \\
\quad\quad mean(d) = (\textbf{if } size(d) \neq 0 \textbf{ then } sum(d) \text{ div } size(d) \textbf{ else } \text{"error"}) \\
\quad \textbf{End}
\end{array}
$$

For ease of reference, we give the above enrichment of $PointSpec[Point \backslash Pair]$ the name $Mean^+$:

= "Introduce the name $Mean^+$ for the above enrichment"

$$(PointSpec[Point \backslash Pair]) \lhd Mean^+$$

The enrichment Mean^+ suggests an implementation for *Mean* based on the operations from *PointSpec*. Therefore, we shall not refine Mean^+ any more. We now consider the refinement of $(\mathit{PointSpec}[\mathit{Point}\backslash\mathit{Pair}])$.

Stage 2 We will refine $(\mathit{PointSpec}[\mathit{Point}\backslash\mathit{Pair}])$ by refining *PointSpec*. We refine *PointSpec* by refining the type *Point* to $\mathbb{N} \times \mathbb{N}$, and strengthening the restriction of *PointSpec* by specifying the components X and Y to be *fst* and *snd*, respectively:

$$
\begin{array}{ll}
 & \mathit{PointSpec} \\
\sqsubseteq & \text{“Theorem 8.9, Theorem 8.5 (strengthening restrictions)”} \\
 & \quad \textbf{Elements} \\
 & \quad\quad \mathit{Point} \in \{\mathbb{N} \times \mathbb{N}\}_{U_1}, \\
 & \quad\quad \mathit{mkPoint} \in \mathbb{N} \to \mathbb{N} \to \mathit{Point}, \\
 & \quad\quad X \in \mathit{Point} \to \mathbb{N}, \\
 & \quad\quad Y \in \mathit{Point} \to \mathbb{N} \\
 & \quad \textbf{Restrictions} \\
 & \quad\quad \forall x, y \in \mathbb{N}. \\
 & \quad\quad \mathit{mkPoint}(x,y) = \langle x, y\rangle \quad \wedge \\
 & \quad\quad X = \mathit{fst} \quad \wedge \\
 & \quad\quad Y = \mathit{snd} \\
 & \quad \textbf{End}
\end{array}
$$

We give the above refinement of *PointSpec* the name $\mathit{PointSpec}_2$:

$$
\begin{array}{ll}
= & \text{“introducing the name } \mathit{PointSpec}_2\text{”} \\
 & \mathit{PointSpec}_2[\mathit{Point}\backslash\mathit{Pair}]
\end{array}
$$

We do not refine $\mathit{PointSpec}_2$ further since its restriction is in a form that immediately suggests an implementation based on representing *Point* values as pairs.

Stage 3 By the refinement step in Stage 2, and the fact that *rename* is monotonic with respect to $\sqsubseteq$, we can refine $(\mathit{PointSpec}[\mathit{Point}\backslash\mathit{Pair}])$ as follows:

$$
\begin{array}{ll}
 & \mathit{PointSpec}[\mathit{Point}\backslash\mathit{Pair}] \\
\sqsubseteq & \text{“Stage 2, Theorem 8.12 (monotonicity of } \mathit{rename}\text{)”} \\
 & \mathit{PointSpec}_2[\mathit{Point}\backslash\mathit{Pair}]
\end{array}
$$

Stage 4 Now we take the refinement in Stage 1 one step further by replacing *PointSpec* with $\mathit{PointSpec}_2$:

$$
\begin{array}{ll}
 & \mathit{PointSpec}[\mathit{Point}\backslash\mathit{Pair}] \lhd \mathit{Mean}^+ \\
\sqsubseteq & \text{“Stage 3, Theorem 8.20 (monotonicity of } \mathit{enrich}\text{)”} \\
 & \mathit{PointSpec}_2[\mathit{Point}\backslash\mathit{Pair}] \lhd \mathit{Mean}^+
\end{array}
$$

The above refinement is our final refinement of *Mean*. All that remains is to combine the refinements steps in Stages 1 and 4 by appealing to the transitivity of specification refinement:

$$\begin{array}{ll} & \mathit{Mean} \\ \sqsubseteq_{\overset{s_1 s_2}{\Pi_2}} & \text{“Theorem 8.2 (transitivity of refinement)”} \\ & \mathit{PointSpec}_2[\mathit{Point}\backslash\mathit{Pair}] \lhd \mathit{Mean}^{+} \end{array}$$

In the above refinement step the abstraction map is ${}^{s_1 s_2}_{\Pi_2}$ since this is the composition of all the abstraction maps used in the refinement steps in Stages 1 and 4. For ease of reference, we give our final refinement the name Mean_3:

$$\mathit{Mean}_3 = \mathit{PointSpec}_2[\mathit{Point}\backslash\mathit{Pair}] \lhd \mathit{Mean}^{+}$$

A full expansion of Mean_3 is given in Figure 8.4.

To summarise this section: we have shown that *Mean* refines to Mean_3 with respect to the abstraction map ${}^{s_1 s_2}_{\Pi_2}$, i.e. $\mathit{Mean} \sqsubseteq_{\overset{s_1 s_2}{\Pi_2}} \mathit{Mean}_3$.

8.7.2 Discussion of Example

For our purposes, we assume that Mean_3 is the final refinement of *Mean*—we use Mean_3 as the starting point for an implementation of *Mean* in the next chapter. However, some readers may feel unhappy that the final refinement contain some auxiliary components; namely, the components from *PointSpec*. However, by Fact 8.1, any implementation of Mean_3 can be transformed to an implementation of *Mean*.

The final form of the refinement $\mathit{PointSpec}_2[\mathit{Point}\backslash\mathit{Pair}] \lhd \mathit{Mean}^{+}$ is not accidental. We deliberately aimed to refine *Mean* to a structured specification. There are several reason why we should do so. One reason is that this encourages the breaking up of the refinement task into several smaller, and more manageable, refinement tasks that can proceed more or less independently of each other. For example, the refinement of *PointSpec* in Stage 2 proceeded independently of Stage 1. As we shall see in the next chapter, structured specification are also easier to implement than unstructured specifications—we illustrate this point in the next chapter by implementing Mean_3.

Another feature of the refinement of *Mean* is that it makes use of *PointSpec*. The reuse of *PointSpec* illustrates that the reuse of specifications is not limited to making new specifications, but that specifications can also be reused in the refinement process. Reusing specifications in refinements has some useful consequences. For example, we can reuse previous refinements of a specification if it is reused to refine another specification. Furthermore, when we come to implement a refinement, we can reuse implementations of any reused specification.

```
$Mean_3 \equiv$
  $(PointSpec_2[Point \backslash Pair]) \lhd$
  $\Lambda$**Elements**
      $Data \in \{Pair\}_{U_1}$,
      $clear \in Data$,
      $\bullet sum \in Data \to \mathbb{N}$,
      $size \in Data \to \mathbb{N}$,
      $enter \in \mathbb{N} \to Data \to Data$,
      $mean \in \prod d \in Data.(\textbf{if } size(d) \neq 0 \textbf{ then } \mathbb{N} \textbf{ else } \mathbb{S})$
    **Restrictions**
      $\forall n \in \mathbb{N}. \forall d \in Data.$
      $enter(n, d) = mkPoint(sum(d) + n, size(d) + 1) \quad \wedge$
      $clear = mkPoint(0, 0) \quad \wedge$
      $sum(d) = X(d) \quad \wedge$
      $size(d) = Y(d) \quad \wedge$
      $mean(d) = (\textbf{if } size(d) \neq 0 \textbf{ then } sum(d) \text{ div } size(d) \textbf{ else } \text{“error”})$
    **End**
```

here

```
$PointSpec_2 \equiv$
  **Elements**
    $Point \in \{\mathbb{N} \times \mathbb{N}\}_{U_1}$,
    $mkPoint \in \mathbb{N} \to \mathbb{N} \to Point$,
    $X \in Point \to \mathbb{N}$,
    $Y \in Point \to \mathbb{N}$
  **Restrictions**
    $\forall x, y \in \mathbb{N}.$
    $mkPoint(x, y) = \langle x, y \rangle \quad \wedge$
    $X = fst \quad \wedge$
    $Y = snd$
  **End**
```

Figure 8.4: A refinement of *Mean*

8.8 Discussion and Summary

The notion of refinement is well known in many non type-theoretic program development formalisms. Early work on refinement includes [19]. More recently, work has been done on refining algebraic specifications [50, 56], and model-oriented specifications in the refinement calculus [37, 38, 36] and VDM [21]. In such non type-theoretic formalisms, our work on refinement is closely related to the refinement of algebraic specifications in [50]. In [50], the notion of constructors correspond to our abstraction maps. In type theory, our work on refinement is closely related to the work of Burstall et al [6] on deliverables, and Luo's work on specification and refinement in ECC [27]. In particular, [27] gives a definition of refinement, similar to ours, that includes a notion of an abstraction map.

In our experience of refining specifications, we have observed that most refinement steps are made using the simple refinement relation ($\sqsubseteq$). Abstraction maps are only used for major structural changes to a specification; such as adding new components, or changing the type of a component. In many cases, the modules satisfying a refinement of a specification are not implementations of the original specification; although, by Fact 8.1, an abstraction map can be used to produce an implementation of a specification from an implementation of its refinement. However, in many cases we are happy to accept an implementation of a refinement even if it does not satisfy our original specification. For example, suppose *Mean* is defined so that *Data* has the type $\{Set(\mathbb{N})\}_{U_1}$, instead of U_1, and the axioms of *Mean* are given in terms of set operators. Furthermore, suppose that we refine *Data* to $\{\mathbb{N} \times \mathbb{N}\}_{U_1}$, and then refine *Mean* to $Mean_3$. Then an implementation of $Mean_3$ is likely to be more efficient than an implementation of *Mean*, since an implementation of *Mean* will use set operations, and these are usually very inefficient. Consequently, we are unlikely to want to transform an implementation of $Mean_3$ to an implementation of *Mean*.

It might be useful to classify abstraction maps into those that can be ignored, and those that we wish to apply to an implementation of a refinement. By only using particular classes of abstraction maps, we can obtain more specialised definitions of refinement. For example, if we use retract functions as abstraction maps—retract functions throw-away, or permute, components in computational elements—then we get a refinement relation that is equivalent to a Quest [7] style subtype relation.

One of the issues still to be addressed properly is that of managing the refinement task. When we refine a large specification, we can be overwhelmed by the number of steps involved and the clerical task of organising, and keeping track of, these steps. However, type theories promise some solutions to this problem as they lend themselves to the implementation of mechanised proof development systems [8, 23]. In [8, 23], it is argued that mechanised systems can assist in choosing appropriate refinement steps. If nothing else, mechanisation may help to manage the clerical task of organising the many steps involved in a refinement proof.

Another issue that we have not considered is the development of a refinement calculus for types other than module specifications. Clearly, if we are to refine types in signatures then refinement laws for function types , product types, type universes etc., would be highly desirable. Such types can be refined using general refinement, and a refinement calculus for these types promises to integrate well with specification refinement; such a conclusion is partially justified by our laws for refining signatures.

In summary, we have given a definition of refinement for specifications in the framework of Martin-Löf's Type Theory. We have also given a collection of refinement laws that can be used to refine canonical and structured specifications in a piecewise manner.

```
0       Let $S_1 = Sig(SP_1)$ ; Let $S_2 = Sig(SP_2)$ ;
1       $g \in S_2 \rightarrow S_1$ ;
2       $\_[p\backslash q] \in S_1 \rightarrow S_1[p\backslash q]$ ;
3       $\_[p\backslash q]^{-1}_{S_2} \in S_2[p\backslash q] \rightarrow S_2$ ;
4       Let $h = (\_[p\backslash q] \circ g \circ \_[p\backslash q]^{-1}_{S_2}) \in S_2[p\backslash q] \rightarrow S_1[p\backslash q]$
          "Assumption: $p$ witnesses that $SP_1 \sqsubseteq_g SP_2$"
5       $p \in \forall x \in S_2.\ Ax(SP_2)(x) \Rightarrow Ax(SP_1)(g(x))$
          "5, $(\_[p\backslash q]^{-1}_{S_1} \circ \_[p\backslash q])$ is an identity function (Fact 6.35)"
6       $p \in \forall x \in S_2.\ Ax(SP_2)(x) \Rightarrow Ax(SP_1)((\_[p\backslash q]^{-1}_{S_1} \circ \_[p\backslash q] \circ g)(x))$
          "6, definition of the *rename* operator"
7       $p \in \forall x \in S_2.\ Ax(SP_2)(x) \Rightarrow Ax(SP_1[p\backslash q])((\_[p\backslash q] \circ g)(x))$
8.0     |[   $x \in Sig(SP_2[p\backslash q])$
8.1.0   ▷    |[   $y \in Ax(SP_2[p\backslash q])(x)$
             ▷       "8.1.0, definition of *rename*"
8.1.1             $y \in Ax(SP_2)(x[p\backslash q]^{-1}_{S_2})$
                    "8.0, 3, $\rightarrow$ - elimination"
8.1.2             $x[p\backslash q]^{-1}_{S_2} \in S_2$
                    "8.1.2, 7, $\rightarrow$ - elimination"
8.1.3             $p(x[p\backslash q]^{-1}_{S_2}) \in Ax(SP_2)(x[p\backslash q]^{-1}_{S_2}) \Rightarrow$
                          $Ax(SP_1[p\backslash q])((\_[p\backslash q] \circ g \circ \_[p\backslash q]^{-1}_{S_2})(x))$
                    "8.1.3, definition of $h$ in 4"
8.1.4             $p(x[p\backslash q]^{-1}_{S_2}) \in Ax(SP_2)(x[p\backslash q]^{-1}_{S_2}) \Rightarrow Ax(SP_1[p\backslash q])(h(x))$
                    "8.1.4, 8.1.1, $\rightarrow$ - elimination"
8.1.5             $(p(x[p\backslash q]^{-1}_{S_2}))(y) \in Ax(SP_1[p\backslash q])(h(x))$
             ]|
        ]|
          "8.0, 8.1.0, 8.1.5, $\rightarrow$ - introduction twice"
9       $\lambda x.\lambda y.(p(x[p\backslash q]^{-1}_{S_2}))(y) \in \forall x \in S_2[p\backslash q].\ Ax(SP_2[p\backslash q])(x) \Rightarrow$
                                                              $Ax(SP_1[p\backslash q])(h(x))$
          "9, definition of specification refinement"
10      $SP_1[p\backslash q] \sqsubseteq_h SP_2[p\backslash q]$
```

Figure 8.5: Proof of Theorem 8.11

0 $h \in Sig(SP_3) \to Sig(SP_1)$;

"assumption"

1 $Sig(SP_1) = Sig(SP_2)$; $Sig(SP_3) = Sig(SP_4)$;

"assumption: p witnesses that $SP_1 \sqsubseteq_h SP_3$"

2 $p \in \forall x \in Sig(SP_3).\ Ax(SP_3)(x) \Rightarrow Ax(SP_1)(h(x))$;

"assumption: q witnesses that $SP_2 \sqsubseteq_h SP_4$"

3 $q \in \forall x \in Sig(SP_4).\ Ax(SP_4)(x) \Rightarrow Ax(SP_2)(h(x))$

4.0 |[$x \in Sig(SP_3)$

4.1.0 $\triangleright$ |[$y \in Ax(SP_3 \cup SP_4)(x)$

$\triangleright$ "4.1.0, definition of *union*"

4.1.1 $y \in Ax(SP_3)(x) \times Ax(SP_4)(x)$

"4.1.1, $\times$ - elimination"

4.1.2 $fst(y) \in Ax(SP_3)(x)$

"2, 4.0, 4.1.2, $\to$ - elimination twice"

4.1.3 $(p(x))(fst(y)) \in Ax(SP_1)(h(x))$

"4.1.1, , $\times$ - elimination"

4.1.4 $snd(y) \in Ax(SP_4)(x)$

"3, 4.0, 4.1.4, $\to$ - elimination, $x \in Sig(SP_3) = Sig(SP_4)$"

4.1.5 $(q(x))(snd(y)) \in Ax(SP_2)(h(x))$

"4.1.3, 4.1.5, $\times$ - introduction"

4.1.6 $\langle (p(x))(fst(y)), (q(x))(snd(y)) \rangle \in Ax(SP_1)(h(x)) \times Ax(SP_2)(h(x))$

"4.1.6, definition of *union*"

4.1.7 $\langle (p(x))(fst(y)), (q(x))(snd(y)) \rangle \in Ax(SP_1 \cup SP_2)(h(x))$

]|

]|

"4.0, 4.1.0, 4.1.7, $\to$ - introduction twice"

5 $\lambda x.\lambda y.\langle (p(x))(fst(y)), (q(x))(snd(y)) \rangle \in \forall x \in Sig(SP_3).\ Ax(SP_3 \cup SP_4)(x) \Rightarrow Ax(SP_1 \cup SP_2)(h(x))$

"5, definition of specification refinement"

6 $SP_1 \cup SP_2 \sqsubseteq_h SP_3 \cup SP_4$

Figure 8.6: Proof of Theorem 8.15

Let $S_i = Sig(SP_i)$ for $1 \leq i \leq 4$, and $h = \lambda m.\ f({}^{S_3S_4}_{\Pi_1}(m)) \oplus g({}^{S_3S_4}_{\Pi_2}(m)) \in (S_3 \oplus S_4) \to (S_1 \oplus S_2)$ where $f \in S_3 \to S_1$ and $g \in S_4 \to S_2$. We are given

1. $SP_1 \ \sqsubseteq_f \ SP_3$
2. $SP_2 \ \sqsubseteq_g \ SP_4$

We shall require the following facts:

3. $SP_1 \uparrow {}^{S_1S_2}_{\Pi_1} \quad \sqsubseteq_h \quad (SP_1 \uparrow {}^{S_1S_2}_{\Pi_1}) \uparrow h$
4. $SP_2 \uparrow {}^{S_1S_2}_{\Pi_2} \quad \sqsubseteq_h \quad (SP_2 \uparrow {}^{S_1S_2}_{\Pi_1}) \uparrow h$
5. $SP_1 \uparrow (f \circ {}^{S_3S_4}_{\Pi_1}) \quad \sqsubseteq \quad SP_3 \uparrow {}^{S_3S_4}_{\Pi_1}$
6. $SP_2 \uparrow (g \circ {}^{S_3S_4}_{\Pi_2}) \quad \sqsubseteq \quad SP_4 \uparrow {}^{S_3S_4}_{\Pi_2}$

Facts (3) and (4) follow immediately by Theorem 8.15; fact (5) follows by Theorem 8.23 and assumption (1); and fact (6) follows by Theorem 8.23 and assumption (2). The proof now proceeds as follows:

$SP_1 + SP_2$

$=$ "Theorem 7.1 (*sum* expressed in terms of *translate* and *union*)"

$(SP_1 \uparrow {}^{S_1S_2}_{\Pi_1}) \cup (SP_2 \uparrow {}^{S_1S_2}_{\Pi_2})$

$\sqsubseteq_h$ "(3),(4) and Theorem 8.15 (refinement of *union*)"

$((SP_1 \uparrow {}^{S_1S_2}_{\Pi_1}) \uparrow h) \cup ((SP_2 \uparrow {}^{S_1S_2}_{\Pi_2}) \uparrow h)$

$=$ "Theorem 7.22 (*translate* composition)"

$(SP_1 \uparrow ({}^{S_1S_2}_{\Pi_1} \circ h)) \cup (SP_2 \uparrow ({}^{S_1S_2}_{\Pi_2} \circ h))$

$=$ "$({}^{S_1S_2}_{\Pi_1} \circ h) = (f \circ {}^{S_3S_4}_{\Pi_1})$ and $({}^{S_1S_2}_{\Pi_2} \circ h) = (g \circ {}^{S_3S_4}_{\Pi_2})$"

$(SP_1 \uparrow (f \circ {}^{S_3S_4}_{\Pi_1})) \cup (SP_2 \uparrow (g \circ {}^{S_3S_4}_{\Pi_2}))$

$\sqsubseteq$ "(5), (6), monotonicity of *union* w.r.t $\sqsubseteq$"

$(SP_3 \uparrow {}^{S_3S_4}_{\Pi_1}) \cup (SP_4 \uparrow {}^{S_3S_4}_{\Pi_2})$

$=$ "Theorem 7.1 (*sum* expressed in terms of *translate* and *union*)"

$SP_3 + SP_4$

By the transitivity of specification refinement, we compose the abstraction maps used in each step of the proof—recalling that $\sqsubseteq = \sqsubseteq_{Id}$—to get $SP_1 + SP_2 \sqsubseteq_h SP_3 + SP_4$.

Figure 8.7: Proof of Theorem 8.17

```
0       $S$ sig; $f \in S \to Sig(SP_2)$ ; $g \in Sig(SP_2) \to Sig(SP_1)$ ;
          "assumption: $p$ witnesses that $SP_1 \sqsubseteq_g SP_2$"
1       $p \in \forall x \in Sig(SP_2).Ax(SP_2)(x) \Rightarrow Ax(SP_1)(g(x))$
2.0     |[   $x \in S$
2.1.0   ▷    |[   $y \in Ax(SP_2 \uparrow f)(x)$
             ▷      "2.1.0, definition of translate"
2.1.1             $y \in (Ax(SP_2))(f(x))$
                    "0, 2.0, $\to$ - elimination"
2.1.2             $f(x) \in Sig(SP_2)$
                    "1, 2.1.1, 2.1.2, $\to$ - elimination twice"
2.1.3             $(p(f(x)))(y) \in Ax(SP_1)(g(f(x)))$
                    "2.1.3, definition of translate"
2.1.4             $(p(f(x)))(y) \in Ax(SP_1 \uparrow (g \circ f))(x)$
             ]|
        ]|
          "2.0, 2.1.0, 2.1.4, $\to$ - introduction twice"
3       $\lambda x.\lambda y.(p(f(x))(y) \in \forall x \in S.\ Ax(SP_2 \uparrow f)(x) \Rightarrow$
                                              $Ax(SP_1 \uparrow (g \circ f))(x)$
          "3, definition of specification refinement"
4       $SP_1 \uparrow (g \circ f) \sqsubseteq SP_2 \uparrow f$
```

Figure 8.8: Proof of Theorem 8.23

```
0        $S$ sig; $f \in Sig(SP_2) \to S$ ; $g \in Sig(SP_2) \to Sig(SP_1)$ ;
           "assumption: $p$ witnesses that $SP_1 \sqsubseteq_g SP_2$"
1        $p \in \forall x \in Sig(SP_2).Ax(SP_2)(x) \Rightarrow Ax(SP_1)(g(x))$ ;
2.0      |[    $x \in S$
2.1.0    ▷    |[    $y \in Ax(SP_2 \downarrow (f \circ g))(x)$
               ▷       "2.1.0, definition of derive"
2.1.1                $y \in \exists z \in SP_2.[f(g(ce(z))) =_S x]$
                       "2.1.1, $\times$ - elimination"
2.1.2                $fst(y) \in SP_2$
                       "2.1.2, $\times$ - elimination"
2.1.3                $ce(fst(y)) \in Sig(SP_2)$
                       "2.1.3, 0, $\to$ - elimination"
2.1.4                $g(ce(fst(y))) \in Sig(SP_1)$
                       "2.1.2, $\times$ - elimination"
2.1.5                $pf(fst(y)) \in Ax(SP_2)(ce(fst(y)))$
                       "1, 2.1.3, 2.1.5, $\to$ - elimination twice"
2.1.6                $(p(ce(fst(y))))(pf(fst(y))) \in Ax(SP_1)(g(ce(fst(y))))$
                       "2.1.4, 2.1.6, $\times$ - introduction, introduce name $m$"
2.1.7                Let $m = \langle g(ce(fst(y))), (p(ce(fst(y))))(pf(fst(y)))\rangle \in SP_1$
                       "2.1.1, $\times$ - elimination, $Eq$ - elimination"
2.1.8                $snd(y) = eq \in [f(g(ce(fst(y)))) =_S x]$
                       "2.1.7, $\times$ - elimination"
2.1.9                $ce(m) = g(ce(fst(y))) \in Sig(SP_1)$
                       "2.1.8, 2.1.9, substitution"
2.1.10               $snd(y) = eq \in [f(ce(m)) =_S x]$
                       "2.1.7, 2.1.10, $\times$ - introduction"
2.1.11               $\langle m, eq\rangle \in \exists z \in SP_1.[f(ce(z)) =_S x]$
                       "2.1.11, definition of derive"
2.1.12               $\langle m, eq\rangle \in Ax(SP_1 \downarrow f)(x)$
               ]|
         ]|
           "2.0, 2.1.0, 2.1.12, $\to$ - introduction twice"
3        $\lambda x.\lambda y.\langle m, eq\rangle \in \forall x \in S.\ Ax(SP_2 \downarrow (f \circ g))(x) \Rightarrow Ax(SP_1 \downarrow f)(x)$
           "3, definition of specification refinement"
4        $SP_1 \downarrow f \sqsubseteq SP_2 \downarrow (f \circ g)$
```

Figure 8.9: Proof of Theorem 8.24

Chapter 9

Implementation

9.1 Introduction

In our type-theoretic specification framework, the development of implementations from specifications is a two step process. First, we refine a specification and then we implement the refined specification. Implementation is the task of finding a module that satisfies a specification. In other words, implementation is the task of constructing a member of the type a specification denotes. To be useful, implementation must proceed in a piecewise manner. That is to say, we want to implement a specification by implementing its constituent pieces, so that the implemented pieces may be glued together to form an implementation of the specification. The "glue" is a collection of module operators that we call *implementors.* In this chapter, we define several implementors, and use them to give some implementation laws that aid the piecewise implementation of specifications.

Each implementor corresponds to a specification operator, and is used to implement specifications made using its corresponding specification operator. For example, the *sum* implementor $(+)$ corresponds to the *sum* specification operator; note that we overload $+$, so that it denotes both the *sum* specification operator and the *sum* implementor. Given any two specifications SP_1 and SP_2, and any two modules $m_1 \in SP_1$ and $m_2 \in SP_2$, then $m_1 + m_2 \in SP_1 + SP_2$. Some specification operators, such as *Union*, do not have corresponding implementors, but specifications that use such specification operators can, under certain conditions, be implemented in a piecewise manner.

There are several advantages to the piecewise implementation of specifications. Firstly, we avoid the need to rearrange a specification prior to implementation. Secondly, it puts a tight upper-bound on the amount of recoding we have to do if we change part of a specification. Thirdly, it is more likely that we can reuse implementations if a specification is used more than once. This last point is important as it encourages us to make libraries of useful specifications, and their implementations, that can be used in other applications. The piecewise implementation of specifications may not always yield an efficient implementation. However, it is a systematic approach to implementing large specifications, and helps to decompose the implementation task into a number of smaller, more manageable, tasks.

In the following, we ignore the implementation of canonical specifications. We assume that specifications have already been refined prior to implementation, so that the restrictions of canonical specifications suggest possible implementations. Therefore, we assume that canonical specifications can be implemented by a simple constructive proof. Alternatively, if we have a large canonical specification it may first be structured and then implemented using the techniques described in this chapter. Wherever we require the implementation of a canonical specification, we simply state the implementation, although the reader should be aware that work needs to be done to produce such implementations.

9.2 Implementors

In this section, we give a formal definition of implementors. Implementors are module operators that correspond to some specification operators. The implementors we define correspond to the specification operators *rename*, *hide*, *sum* and *derive*. Implementors are used to implement specifications of the form $F(SP)$ in a piecewise manner, where F is any one of the specification operators mentioned above, and SP any specification. That is to say, $F(SP)$ may be implemented by replacing SP with any implementation $m \in SP$, and replacing F by its corresponding implementor f:

$$m \in SP \Rightarrow f(m) \in F(SP)$$

We show that the piecewise implementation property, described above, follows immediately from the definition of each of the implementors we define.

To avoid the proliferation of operator symbols, we will overload the operator symbols for the specification operators *rename*, *hide*, *sum* and *derive*, and use them to denote their corresponding implementors.

9.2.1 The Rename and Hide Implementors

An application of the *rename* implementor has the form $m[p\backslash q]$ for m any module, and p and q any component names. $m[p\backslash q]$ is module m with component p renamed to q. An application of the *hide* implementor has the form $m\backslash i$ for m any module, and i a set of component names. $m\backslash i$ is module m with all components named in i made local in m. The formal definitions of the *rename* and *hide* implementors are given below with an explanation following the definitions.

Definition 9.1 (The Rename Implementor)
Let SP be any specification, and $p, q \in \mathbb{S}$. Given any $m \in SP$, the *rename* implementor $\_[p\backslash q] \in SP \to SP[p\backslash q]$ is defined as follows:

$$m[p\backslash q] = \langle ce(m)[p\backslash q], pf(m)\rangle \in SP[p\backslash q]$$

□

Definition 9.2 (The Hide Implementor)
Let SP be any specification, and $i \in Set(\mathbb{S})$. Given any $m \in SP$, the *hide* implementor $\_\backslash i \in SP \to SP\backslash i$ is defined as follows:

$$m\backslash i = \langle ce(m)\backslash i, pf(m)\rangle \in SP\backslash i$$

□

Let us consider the definition of the *rename* implementor. The expression $ce(m)[p\backslash q]$ is the computational element of module $m[p\backslash q]$, and expresses the application of the computational element renaming operator to the computational element of m. The witness of $m[p\backslash q]$ is $pf(m)$, which is just the witness of m.

From the definition of the *rename* implementor, we may deduce that a specification of the form $SP[p\backslash q]$ may be implemented in a piecewise manner: if $m \in SP$ then $m[p\backslash q] \in SP[p\backslash q]$. A proof of this property is given in Figure 9.1.

The *hide* implementor is defined using the computational element hide operator, and its definition is similar, in style, to that of the *rename* implementor. From the definition of the *hide* implementor, we may deduce that specifications of the form $SP\backslash i$ may be implemented in a piecewise manner: if $m \in SP$ then $m\backslash i \in SP\backslash i$. The proof of this property is similar to the proof in Figure 9.1.

9.2.2 The Sum Implementor

An application of the *sum* implementor has the form $m_1 + m_2$ where m_1 and m_2 are any modules. The module $m_1 + m_2$ supplies all the components in m_1 and m_2. A formal definition of the *sum* implementor is given below, with an explanation following the definition.

Definition 9.3 (Sum Implementation)
Let SP_1 and SP_2 be any specifications. Given any $m_1 \in SP_1$ and $m_2 \in SP_2$, the *sum* implementor $\_ + \_ \in SP_1 \to SP_2 \to SP_1 + SP_2$ is defined as follows:

$$m_1 + m_2 = \langle ce(m_1) \oplus ce(m_2), \langle pf(m_1), pf(m_2)\rangle\rangle \in SP_1 + SP_2$$

□

```
0.0   |[  $SP$ **spec** ; $p, q \in \mathbb{S}$ ;
0.1       $m \in SP$
      ▷     “0.1, definition of $ce$”
0.2       $ce(m) \in Sig(SP)$
            “0.2, definition of computational element renaming”
0.3       $ce(m)[p\backslash q] \in Sig(SP)[p\backslash q] = Sig(SP[p\backslash q])$
            “0.1, definition of $pf$”
0.4       $pf(m) \in Ax(SP)(ce(m))$
            “Fact 6.35”
0.5       $ce(m) = (ce(m)[p\backslash q])[p\backslash q]^{-1}_{Sig(SP)} \in Sig(SP)$
            “0.4, 0.5, substitution”
0.6       $pf(m) \in Ax(SP)((ce(m)[p\backslash q])[p\backslash q]^{-1}_{Sig(SP)})$
            “definition of *rename*”
0.7       $Ax(SP)((ce(m)[p\backslash q])[p\backslash q]^{-1}_{Sig(SP)}) = Ax(SP[p\backslash q])(ce(m)[p\backslash q])$
            “0.6, 0.7, substitution”
0.8       $pf(m) \in Ax(SP[p\backslash q])(ce(m)[p\backslash q])$
            “0.3, 0.8”
0.9       $\langle ce(m)[p\backslash q], pf(m)\rangle \in SP[p\backslash q]$
            “0.9, notation for the *rename* implementor”
0.10      $m[p\backslash q] \in SP[p\backslash q]$
      ]|
```

Figure 9.1: A Proof of the Type Correctness of the Rename Implementor

0.0 $\llbracket$ SP_1 **spec** ; SP_2 **spec** ;

0.1 Let $S_1 = Sig(SP_1)$ and $S_2 = Sig(SP_2)$;

0.2 $m_1 \in SP_1$; $m_2 \in SP_2$

$\triangleright$ "0.2, definition of *ce*, twice"

0.3 $ce(m_1) \in Sig(SP_1)$; $ce(m_2) \in Sig(SP_2)$

"0.3, definition of computational element concatenation"

0.4 $ce(m_1) \oplus ce(m_2) \in Sig(SP_1) \oplus Sig(SP_2)$

"0.4, definition of *sum*"

0.5 $ce(m_1) \oplus ce(m_2) \in Sig(SP_1 + SP_2)$

"0.2, definition of *pf*, twice"

0.6 $pf(m_1) \in Ax(SP_1)(ce(m_1))$; $pf(m_2) \in Ax(SP_2)(ce(m_2))$

"0.6 ${}^{S_1S_2}_{\Pi_1}$-elimination"

0.7 $pf(m_1) \in Ax(SP_1)({}^{S_1S_2}_{\Pi_1}(ce(m_1) \oplus ce(m_2)))$

"0.6 ${}^{S_1S_2}_{\Pi_2}$-elimination"

0.8 $pf(m_2) \in Ax(SP_2)({}^{S_1S_2}_{\Pi_2}(ce(m_1) \oplus ce(m_2)))$

"0.7, 0.8, definition of *sum*"

0.9 $\langle pf(m_1), pf(m_2)\rangle \in Ax(SP_1 + SP_2)(ce(m_1) \oplus ce(m_2))$

"0.5, 0.9"

0.10 $\langle ce(m_1) \oplus ce(m_2), \langle pf(m_1), pf(m_2)\rangle\rangle \in SP_1 + SP_2$

"0.10, notation for the *sum* implementor"

0.11 $m_1 + m_2 \in SP_1 + SP_2$

$\rrbracket$

Figure 9.2: A Proof of the Type Correctness of the Sum Implementor

The *sum* implementor is defined using the computational element concatenation operator ($\oplus$). The expression $ce(m_1) \oplus ce(m_2)$ is the computational element of $m_1 + m_2$, and satisfies the signature of $SP_1 + SP_2$. The expression $\langle pf(m_1), pf(m_2)\rangle$ is the witness of $m_1 + m_2$, and satisfies the restriction of $SP_1 + SP_2$ which is the product of the restrictions of SP_1 and SP_2.

From the definition of the *sum* implementor we get that specifications of the form $SP_1 + SP_2$ may be implemented in a piecewise manner: given any $m_1 \in SP_1$ and $m_2 \in SP_2$ then $m_1 + m_2 \in SP_1 + SP_2$. In other words, $SP_1 + SP_2$ is implemented by implementing SP_1 and SP_2 independently of each other, and then combining their implementations using the *sum* implementor. A proof of this property is given in Figure 9.2.

0.0 $\llbracket$ SP **spec** ; S **sig** ;

0.1 $f \in Sig(SP) \to S$

0.2 $m \in SP$

$\rhd$ "0.2, definition of *ce*"

0.3 $ce(m) \in Sig(SP)$

"0.3, 0.1, $\to$-elimination"

0.4 $f(ce(m)) \in S$

"0.4, 0.1, definition of *derive* "

0.5 $f(ce(m)) \in Sig(SP \downarrow f)$

"0.4, = is reflexive, Eq-introduction"

0.6 $eq \in [f(ce(m)) =_S f(ce(m))]$

"0.2, 0.6, $\times$-introduction"

0.7 $\langle m, eq\rangle \in \exists x \in SP.[f(ce(x)) =_S f(ce(m))]$

"0.7, definition of *derive*"

0.8 $\langle m, eq\rangle \in Ax(SP \downarrow f)(f(ce(m)))$

"0.5, 0.8"

0.9 $\langle f(ce(m)), \langle m, eq\rangle\rangle \in SP \downarrow f$

"0.9, notation for the *derive* implementor"

0.10 $m \downarrow f \in SP \downarrow f$

$\rrbracket$

Figure 9.3: A proof of the Type Correctness of the Derive Implementor

9.2.3 The Derive Implementor

Let SP be any specification, and S any signature. An application of the *derive* implementor has the form $m \downarrow f$ for any module $m \in SP$ and any *derive* map $f \in Sig(SP) \to S$. The module $m \downarrow f$ satisfies the specification $SP \downarrow f$. A formal definition of the *derive* implementor is given below:

Definition 9.4 (The Derive Implementor)
Given SP **spec**, any $m \in SP$, and $f \in Sig(SP) \to S$ where S **sig**, the *derive* implementor $\_ \downarrow f \in SP \to SP \downarrow f$ is defined as follows:

$$m \downarrow f = \langle f(ce(m)), \langle m, eq\rangle\rangle \in SP \downarrow f$$

□

The computational element, $f(ce(m))$, of $m \downarrow f$ satisfies the signature, S, of $SP \downarrow f$. The expression $\langle m, eq\rangle$ is the witness of $m \downarrow f$. This witness satisfies the restriction of $SP \downarrow f$ which is $\exists x \in SP.[f(ce(x)) =_S ce(m)]$. Figure 9.3 contains a proof that $m \downarrow f \in SP \downarrow f$.

9.3 Some Implementation Laws

Specifications that take the form of a *union*, *enrich* or *translate* operation cannot always be implemented in a piecewise fashion. In this section, we highlight the problem of implementing such specifications, and give some laws that can be used to implement them.

9.3.1 Implementing Enrich

Consider the specification $SP = SP_1 \lhd \Lambda\langle S \mid R\rangle$. To implement SP in a piecewise manner, we would like to develop some implementation $m_1 \in SP_1$, and then add to m_1 an implementation for the components specified by the enrichment $\Lambda\langle S \mid R\rangle$. Our aim is to use m_1 to make an implementation for the components specified by $\Lambda\langle S \mid R\rangle$. However, it is possible that some implementations of SP_1 may make the restriction R inconsistent, as they may be implemented without considering the extra restrictions on SP_1 introduced by R. Consequently, an unfortunate choice of implementation for SP_1 may prevent the implementation of the components specified by $\Lambda\langle S \mid R\rangle$. To show that an implementation of SP_1 avoids such "dead-end" developments requires proving that the remaining implementation task is consistent. It can be shown—and this is one of the interesting features of using type-theory to develop programs—that to prove that R is consistent, for an arbitrary implementation of SP_1, is equivalent to making an implementation for the components specified by $\Lambda\langle S \mid R\rangle$.

Given an implementation of SP_1, the next stage in implementing SP is to implement the components specified by $\Lambda\langle S \mid R\rangle$. $\Lambda\langle S \mid R\rangle$ can almost be viewed as a specification; although, it is not a specification since it is dependent on the components specified by SP_1. However, $\Lambda\langle S \mid R\rangle$ can be transformed into a specification by replacing the free names of components from SP_1 by actual implementations of such components taken from any implementation of SP_1. Given any $m_1 \in SP_1$, we may implement the components of $\Lambda\langle S \mid R\rangle$ by implementing the specification $SP_2 = \langle S \mid R\rangle(y_1\backslash m.y_1, \ldots, y_n\backslash m.y_n)$ where $y_1, \ldots, y_n$ are the names of the components in m_1 (and SP_1). Given any $m_2 \in SP_2$, we can show that combining m_1 and m_2, using the *sum* implementor, gives $m_1 + m_2 \in SP$. We state this property, formally, in the following theorem, with an explanation following the theorem:

Theorem 9.1
Let $S_1 = Sig(SP_1)$. If $m_1 \in SP_1$ and $m_2 \in \langle S(\!|ce(m_1)|\!)_{S_1} \mid (m)R(ce(m_1) \oplus m)\rangle$ then:

$$m_1 + m_2 \in SP_1 \lhd \Lambda\langle S \mid R\rangle$$

□

Proof Given in Figure 9.4 □

0.0 $|[$ SP_1 **spec** ; Let $S_1 = Sig(SP_1)$;
0.1 $m_1 \in SP_1$;
0.2 $S_1 \vdash S$ **sig** ; $|[m \in S_1 \oplus S \rhd R(m)$ **type**$]|$;
0.3 Let $SP_2 = \langle S(\!| ce(m_1) |\!)_{S_1} \mid (m)R(ce(m_1) \oplus m)\rangle$;
0.4 $m_2 \in SP_2$
$\rhd$ "0.1, 0.4, definition of the *sum* implementor"
0.5 $m_1 + m_2 \in SP_1 + SP_2$
"0.5, 0.2, 0.3, definition of *sum*"
0.6 $ce(m_1 + m_2) \in S_1 \oplus S(\!| ce(m_1) |\!)_{S_1}$
"0.6, 0.2, definition of $\oplus$ and the *sum* implementor"
0.7 $ce(m_1 + m_2) = ce(m_1) \oplus ce(m_2) \in S_1 \oplus S$
"0.7, definition of *enrich*"
0.8 $ce(m_1 + m_2) \in Sig(SP_1 \lhd \Lambda\langle S \mid R\rangle)$
"0.7, $\pi_2^{S_1 S}$-elimination"
0.9 $\pi_2^{S_1 S}(ce(m_1 + m_2)) = ce(m_2) \in S(\!| ce(m_1) |\!)_{S_1}$
"0.5, definition of *sum*. Let m $=$ $ce(m_1 + m_2)$"
0.10 $pf(m_1 + m_2) \in Ax(SP_1)(\pi_1^{S_1 S}(m)) \times R(ce(m_1) \oplus \pi_2^{S_1 S}(m))$
"0.10, 0.9, 0.7 and substitution"
0.11 $pf(m_1 + m_2) \in Ax(SP_1)(\pi_1^{S_1 S}(ce(m_1 + m_2))) \times R(ce(m_1 + m_2))$
"0.11, definition of *enrich*"
0.12 $pf(m_1 + m_2) \in Ax(SP_1 \lhd \Lambda\langle S \mid R\rangle)(ce(m_1 + m_2))$
"0.8, 0.12"
0.13 $m_1 + m_2 \in SP \lhd \Lambda\langle S \mid R\rangle$
$]|$

Figure 9.4: A proof of Theorem 9.1

In Theorem 9.1, the specification $\langle S(\!| ce(m_1) |\!)_{S_1} \mid (m)R(ce(m_1) \oplus m)\rangle$, which we will refer to as SP^+, is equivalent to SP_2 above. In SP^+, the substitutions described in SP_2 are performed using the signature instantiation notation (Section 6.2.2), i.e. $S(\!| ce(m_1) |\!)_{S_1}$ is S with free occurrences of component names from S_1 replaced by implementations of such components taken from $ce(m_1) \in S_1$. By the definition of *enrich*, R is dependent on computational elements with signature $S_1 \oplus S$. The expression $R(ce(m_1) \oplus m)$, where $ce(m_1) \oplus m \in S_1 \oplus S$, is R with free occurrences of component names from S_1 replaced by implementations of such components taken from $ce(m_1) \in S_1$. Therefore, $(m)R(ce(m_1) \oplus m)$ is dependent on $m \in S(\!| ce(m_1) |\!)_{S_1}$.

Sometimes, the enrichment $SP = SP_1 \lhd \Lambda\langle S \mid R\rangle$ can be implemented by a function that takes any implementation of SP_1 and returns an implementation of the components specified by $\Lambda\langle S \mid R\rangle$. The type of such a function is a dependent function type that we name F:

$$F = \prod x \in SP_1.\langle S(\!| ce(x) |\!)_{S_1} \mid (m)R(ce(x) \oplus m)\rangle$$

F may be regarded as the specification of a parameterised module; in other words, the members of F are functions that return modules. Given a member of F, it can be used to make an implementation of SP:

Theorem 9.2
Let $S_1 = Sig(SP_1)$ and $F = \prod x \in SP_1.\langle S(\!| ce(x) |\!)_{S_1} \mid (m)R(ce(x) \oplus m)\rangle$. If $f \in F$ and $m_1 \in SP_1$ then:

$$m_1 + f(m_1) \in SP_1 \lhd \Lambda\langle S \mid R\rangle$$

□

Proof Similar to that of Theorem 9.1 above □

In practice, F is often inconsistent (i.e. has no members) since, as discussed earlier, some $m_1 \in SP_1$ may make R inconsistent, so that F becomes inconsistent. Therefore, it is often easier to borrow a specific $m_1 \in SP_1$—as in Theorem 9.1, above—to implement an enrichment.

In summary, we can implement specifications of the form $SP_1 \lhd \Lambda\langle S \mid R\rangle$ in a piecewise manner, provided that we keep one eye on the restriction R when implementing SP_1, to ensure that the implementation of SP_1 keeps R consistent.

9.3.2 Implementing Union

Consider the specification $SP = SP_1 \cup SP_2$ where we assume that $Sig(SP_1) = Sig(SP_2)$. By the definition of *union*, the signature of SP is $Sig(SP_1)$, and implementations of SP satisfy the restrictions of SP_1 and SP_2. Suppose we develop implementations $m_1 \in SP_1$ and $m_2 \in SP_2$. m_1 and m_2 both satisfy the signature of SP, but m_1 is not guaranteed to satisfy the restriction of SP_2; similarly, m_2 is not guaranteed to satisfy the restriction of SP_1. In general, there is no way to combine m_1 and m_2 to produce a module satisfying the restrictions of SP_1 and SP_2. Consequently, there is no point in implementing SP in a piecewise manner.

In practice, SP can be implemented by first unfolding it to eliminate the union operator, and then implementing the resultant specification. An alternative implementation strategy is to make an implementation $m_1 \in SP_1$ and verify that it satisfies the restriction of SP_2 (i.e. verify that there exists some witness $p \in Ax(SP_2)(ce(m_1))$):

0.0 |[SP_1 **spec** ; SP_2 **spec** ; $Sig(SP_1) = Sig(SP_2)$;
0.1 $m_1 \in SP_1$;
0.2 $p \in Ax(SP_2)(ce(m_1))$
$\triangleright$ "0.1, definition of *ce*"
0.3 $ce(m_1) \in Sig(SP_1)$
"0.3, definition of *Union*"
0.4 $ce(m_1) \in Sig(SP_1 \cup SP_2)$
"0.1, definition of *pf*"
0.5 $pf(m_1) \in Ax(SP_1)(ce(m_1))$
"0.1, 0.2, $\times$-introduction"
0.6 $\langle pf(m_1), p\rangle \in Ax(SP_1)(ce(m_1)) \times Ax(SP_2)(ce(m_1))$
"0.6, definition of *Union*"
0.7 $\langle pf(m_1), p\rangle \in Ax(SP_1 \cup SP_2)(ce(m_1))$
"0.4, 0.7"
0.8 $\langle ce(m_1), \langle pf(m_1), p\rangle\rangle \in SP_1 \cup SP_2$
]|

Figure 9.5: A Proof of Theorem 9.3

Theorem 9.3
Given any $m_1 \in SP_1$, and any $p \in Ax(SP_2)(ce(m_1))$ then:

$$\langle ce(m_1), \langle pf(m_1), p\rangle\rangle \in SP_1 \cup SP_2$$

□

Proof Given in Figure 9.5 □

We can also implement *SP* by making an implementation $m_2 \in SP_2$ and verifying that it satisfies the restriction of SP_1.

It could be argued that the *union* operator is of limited use since specifications that use it cannot be implemented in a piecewise manner. However, the *union* operator is extremely useful for making specifications and we should continue to use it in such a role. We should not let concerns about the implementation task interfere with the specification task. However, when we develop an implementation from a specification, the evidence above suggests that occurrences of the union operator should be eliminated during the refinement task. Applying such a heuristic makes it more likely that refinements can be implemented in a piecewise manner.

```
0.0  |[  $SP$ spec ; $S$ sig ; $f \in S \to Sig(SP)$ ; $f^{-1} \in Sig(SP) \to S$ ;
0.1      $|[m \in Sig(SP) \rhd f(f^{-1}(m)) = m \in Sig(SP)]|$ ;
0.2      $m \in SP$
     $\rhd$    "0.2, definition of $ce$"
0.3      $ce(m) \in Sig(SP)$
           "0.0, 0.3, $\to$-elimination, definition of $Translate$"
0.4      $f^{-1}(ce(m)) \in S = Sig(SP \uparrow f)$
           "0.3, 0.1 inverse property"
0.5      $f(f^{-1}(ce(m))) = ce(m) \in Sig(SP)$
           "0.2, definition of $pf$"
0.6      $pf(m) \in Ax(SP)(ce(m))$
           "0.6, 0.5, substitution"
0.7      $pf(m) \in Ax(SP)(f(f^{-1}(ce(m))))$
           "0.7, definition of $translate$"
0.8      $pf(m) \in Ax(SP \uparrow f)(f^{-1}(ce(m)))$
           "0.4, 0.8"
0.9      $\langle f^{-1}(ce(m)), pf(m)\rangle \in SP \uparrow f$
     ]|
```

Figure 9.6: A proof of Theorem 9.4

9.3.3 Implementing Translate

Consider a specification $SP = SP_1 \uparrow f$ where $f \in S \to Sig(SP_1)$ and S **sig**. By the definition of *translate*, any implementation $m \in SP$ meets the signature S, and satisfies the restriction $Ax(SP_1)(f(ce(m)))$. We can implement SP in a piecewise manner provided that the *translate* map f has an inverse:

Theorem 9.4
Given any $m \in SP$ and $f \in S \to Sig(SP)$ where S **sig**, if f^{-1} exists then:

$$\langle f^{-1}(ce(m)), pf(m)\rangle \in SP \uparrow f$$

□

Proof Given in Figure 9.6 □

Not all *translate* maps have an inverse, so specifications such as SP cannot always be implemented in a piecewise manner. If *translate* map f has no inverse, SP can be implemented by first unfolding it, to eliminate the *translate* operator, and then implementing the resultant specification. When refining a specification, we should attempt to eliminate all uses of the *translate* operation, except for those uses whose associated *translate* map has an inverse. Such a refinement heuristic makes it more likely that refinements can be implemented in a piecewise manner.

9.4 An Example Implementation

In this section, we give an example of implementing a specification. We choose to implement the specification Mean_3 which we developed in Section 8.7 as a refinement of the specification *Mean*—*Mean* and Mean_3 are given in Figures 3.1 and 8.4, respectively. Recall that Mean_3 specifies operations to calculate the mean of a sample of naturals, and is defined as follows:

$$\mathit{Mean}_3 = \mathit{PointSpec}_2[\mathit{Point}\backslash\mathit{Pair}] \lhd \mathit{Mean}^+$$

where the details of $\mathit{PointSpec}_2$ and the enrichment Mean^+ can be found in Figure 8.4. We will show that the implementors and implementation laws given in this chapter can be used to implement Mean_3 in a piecewise manner.

In the following, we will abbreviate $\mathit{PointSpec}_2[\mathit{Point}\backslash\mathit{Pair}]$ as PS_2, and the signature and restriction of Mean^+ as S and R, respectively.

9.4.1 The Implementation of Mean_3

The structure of Mean_3 suggests using Theorems 9.1 or 9.2 as a starting point for its implementation. In fact, we will use Theorem 9.2 and implement Mean^+ by a function developed independently of any implementation of PS_2. We will implement the enrichment Mean^+ first, and then implement PS_2.

To implement Mean^+ by Theorem 9.2, we require to construct a function that has the following type:

$$F = \prod x \in PS_2.\langle S(\!| ce(x) |\!)_{PS_2} \mid (m)R(ce(x) \oplus m)\rangle$$

We omit the details of making a function of type F, but the restriction, R, of Mean^+ strongly suggests the function f, given in Figure 9.7, as a member of F. f may be developed through a constructive proof from its type F; the proof is largely a clerical task.

Next, we consider implementing the specification $\mathit{PointSpec}_2[\mathit{Point}\backslash\mathit{Pair}]$. We will implement $\mathit{PointSpec}_2$, and then apply the *rename* implementor to its implementation to produce an implementation of $\mathit{PointSpec}_2[\mathit{Point}\backslash\mathit{Pair}]$. The restriction of $\mathit{PointSpec}_2$ suggests the module *PointModule*, in Figure 9.8, as an implementation of $\mathit{PointSpec}_2$. We can verify that

$$\mathit{PointModule} \in \mathit{PointSpec}_2$$

by a constructive proof; this task is a largely clerical and is omitted. By applying the *rename* implementor to *PointModule* we get:

$$\mathit{PointModule}[\mathit{Point}\backslash\mathit{Pair}] \in \mathit{PointSpec}_2[\mathit{Point}\backslash\mathit{Pair}]$$

$$
\begin{array}{l}
f = \lambda m \in PS_2. \\
\quad \textbf{module} \\
\quad\quad \begin{array}{lcl}
Data & = & m.Pair, \\
clear & = & m.mkPoint(0,0), \\
\bullet sum & = & m.X, \\
size & = & m.Y, \\
enter & = & \lambda n \in \mathbb{N}.\lambda d \in Data.\ m.mkPoint(sum(d)+n, size(d)+1), \\
mean & = & \lambda d \in Data.\textbf{if } size(d) \neq 0 \textbf{ then } sum(d)\ \underline{\text{div}}\ size(d) \textbf{ else } \text{“error”}
\end{array} \\
\quad \textbf{proof} \\
\quad\quad \lambda n \in \mathbb{N}.\lambda d \in Data.\langle eq, \langle eq, \langle eq, \langle eq, eq\rangle\rangle\rangle\rangle \\
\quad \textbf{end} \in F
\end{array}
$$

Figure 9.7: A parameterised module of type F

$$
\begin{array}{l}
PointModule \equiv \\
\quad \textbf{module} \\
\quad\quad \begin{array}{lcl}
Point & = & \mathbb{N} \times \mathbb{N}, \\
mkPoint & = & \lambda x \in \mathbb{N}.\lambda y \in \mathbb{N}.\langle x, y+100\rangle, \\
X & = & \lambda p \in Point.\ \text{fst}(p), \\
Y & = & \lambda p \in Point.\ \text{snd}(p) - 100,
\end{array} \\
\quad \textbf{proof} \\
\quad\quad \lambda x \in \mathbb{N}.\lambda y \in \mathbb{N}.\langle eq, eq\rangle \\
\quad \textbf{end} \in PointSpec_2
\end{array}
$$

Figure 9.8: An implementation of $PointSpec_2$

We put all the implemented pieces together by applying Theorem 9.2 to get the following implementation, named *MeanModule*, of $Mean_3$:

$$
\begin{array}{l}
MeanModule = \\
\quad PointModule[Point \backslash Pair] + f(PointModule[Point \backslash Pair]) \in Mean_3
\end{array}
$$

9.4.2 A Discussion of the Example

We have deliberately used the structure of $Mean_3$ to guide its implementation, and have succeeded in our goal of developing the implementation in a piecewise manner. Using the structure of a specification to guide its implementation often simplifies the implementation task. This statement is justified, to some extent, by our example. Of course, we assume that the structured specifications we implement have already been refined, so the task of implementing its constituent canonical specifications is relatively straightforward; implementing un-refined specifications is usually more difficult than implementing refined specifications.

Making implementations through implementors and functions has some interesting consequences. One of these is that if we reuse existing specifications to make a new specification, then it may also be possible to reuse their implementations to implement the new specification. This is illustrated in our example, which shows how reusing *PointSpec* to make $Mean_3$ allows us to reuse the implementation *PointModule* to implement $Mean_3$. This property is a direct consequence of having laws to refine and implement specifications in a piecewise manner.

The observant reader will have noticed that *PointModule* appears twice in *MeanModule*. By abstracting over *PointModule*, we can develop a parameterised module that returns different implementations of $Mean_3$. For example, function g, below, takes any implementation of $PointSpec_2$ and returns an implementation of $Mean_3$:

$$g = \lambda m \in PointSpec_2.\ m[Point \backslash Pair] + f(m[Point \backslash Pair])$$

where

$$g \in PointSpec_2 \rightarrow Mean_3$$

The type of g is justified by the definition of the *rename* implementor and Theorem 9.2. The function g is a parameterised version of *MeanModule*, and it can be used to express *MeanModule* as follows:

$$MeanModule = g(PointModule) \in Mean_3$$

However, the main advantage of defining g is that any implementation of $PointSpec_2$ may be used to produce an implementation of $Mean_3$. In other words, if we subsequently choose to re-implement $PointSpec_2$—maybe for reasons of efficiency—then g allows us to reuse f to make a new implementation of $Mean_3$. The fact that we can reuse implementations in this way is a direct consequence of using a structured approach to construct, refine and implement specifications.

9.5 Discussion and Summary

The stepwise implementation of specifications of modules is also encouraged in the refinement calculus [39] and algebraic specifications [50, 56]. It is worth pointing out that our notion of implementation differs from that commonly used by algebraic specifications. Typically, algebraic specifications are implemented by other algebraic specifications whose axioms look like programs. However, there is work on implementing algebraic specifications as programs. In particular, [50] implements algebraic specifications via functions called constructors. Our use of implementors is similar to the notion of constructors in [50]. In Extended ML [48], algebraic specifications are implemented as programs in the functional programming language Standard ML.

There are several advantages to a stepwise approach to implementation. We avoid rearranging a specification prior to, and during, implementation. It also makes it more likely that we can reuse implementations of existing specifications. For example, consider the specification $SP_1 + SP_1[p\backslash q]$. If we make an implementation $m_1 \in SP_1$, we can reuse m_1 to make the implementation $m_1[p\backslash q] \in SP_1[p\backslash q]$ and, by the *sum* implementor, we get $m_1 + m_1[p\backslash q] \in SP_1 + SP_1[p\backslash q]$. Piecewise implementation also helps to reduce the amount of recoding necessary if we change part of a specification. For example, consider the implementation $m_1 + m_2 \in SP_1 + SP_2$. If we change SP_2 to any specification SP_3 then we only have to recode SP_3 to get an implementation of $SP_1 + SP_3$; for any $m_3 \in SP_3$ we get that $m_1 + m_3 \in SP_1 + SP_3$.

Piecewise implementation does have limitations. Making an implementation with the same structure as its specification may not lead to an efficient implementation. If we implement part of a specification in isolation, without considering how it interacts with the rest of the specification, then we could preclude efficient implementations that might be obtained by rearranging the specification prior to implementation. However, large specifications can be difficult to rearrange, and it is usually easier to implement them directly in a piecewise manner rather than rearrange them. In [13]—which discusses the implementation of modules in VDM—it is argued that the structure of a specification should be designed to make a specification easier to understand, rather than easier to implement. We agree with this argument, and it does not preclude the piecewise implementation of specifications.

We have only given the most basic theorems about the piecewise implementation of structured specifications, and they are in no way complete. In particular, each implementor has many algebraic properties that may help in making implementations. In general, the algebraic properties of each implementor are similar to the properties of the computational element operator used to define the implementor. For example, the sum implementor is associative since computational element concatenation is associative. We have not defined the algebraic properties of implementors since we have not required their use, but such properties are not difficult to prove.

In summary, we have shown that in type theory specification can be implemented in a piecewise manner. We have defined some module operators, called implementors, and used them to give some implementation laws that aid the piecewise implementation of specifications.

Chapter 10

Conclusions

The need to apply formal specification and development of programs to large problems has highlighted a need for methods to support the modular construction of specifications and their implementations. Martin-Löf's Type Theory supports both these activities. Type theory allows us to express specifications, modules and specification building operators in a single framework. Furthermore, we can incrementally construct and implement modular specifications in the single framework of type theory.

The main conclusions of this thesis are that, in principle, Martin-Löf's Type Theory can be used to express a rich specification and implementation language for modules, and it also provides a framework for the incremental construction and implementation of modular specifications.

In this chapter, we summarise the results of the main chapters in this thesis: Chapters 3, 4, 6, 7, 8 and 9. We also give a global critical review of Martin-Löf's Type Theory for specification and programming, and our use of the theory in this thesis.

10.1 Summary

10.1.1 Typeful Specification

Adopting a type-theoretic view of modules and their specifications allows us to exploit many of the features of type theory when making specifications. In Chapter 3 we described the advantages of typeful specification. The main conclusion of Chapter 3 is that typeful specification simplifies the specification task; it produces more concise specifications, and it allows specifications to be made in a hierarchical manner that makes specifications easier to understand.

It was shown that we can use the built-in types in Martin-Löf's Type Theory as models for data-types specified in our specifications. This approach contrasts with the purely property-based style of specification encouraged by algebraic specification languages, in which we may not assume any model for data-types (sorts). We gave empirical evidence that specifying models for data-types results in specifications with fewer axioms, compared with equivalent property-based specifications. Consequently, it is easier to verify that a model-based specification specifies the system we intended it to specify, compared to an equivalent property-based specification.

The fact that specifications are types allows us to use specifications as types for components in specifications. This allows us to nest one specification inside another. We say that such a style of nesting is hierarchical as it introduces different scopes, and so, components with the same name can co-exist in the same specification.

Interpreting specifications as types allows us to use specifications as record types, and create multiple instances of a module inside a specification. This is shown to be useful when specifying complicated data-structures; we gave an example of a book library that uses multiple instances of a book module to represent the stock of a library.

We defined parameterised specifications as functions that return specifications. By specifying the parameter of a parameterised specification to be a type, we can specify generic specifications that can be instantiated with different types. We may also use modules as parameters, so that we can use the operations in a module to make a specification; in other words, we can mix specifications and implementations to make specifications.

It was shown that by using the fact that propositions are types, we can add equality relations—called observational equalities—as components in specifications, and that these relations can be used to solve the problem of over-specifying modules. Adding observational equalities to a specification adds extra restrictions to it, but allows more implementations to be derived from a specification.

10.1.2 The Semantics of Specifications

To be useful, a specification language must have a formal semantics. A formal semantics allows us to reason about specifications, and to justify laws for developing implementations from specifications. In Chapter 4, we defined a semantics for canonical specifications, and modules, in Martin-Löf's Type Theory. A specification is a type, and its members are modules that satisfy it.

We reviewed Nordström and Petersson's (NPS) approach to specifying modules in type theory, and contrasted this with Burstall's Deliverables approach to making specifications in type theory. We concluded that both the above approaches were deficient since they do not include a formal semantics for the name-space of components in modules and specifications.

We proposed, and defined, a semantics for specifications and modules which is a fusion of the NPS and Deliverables approaches, but where component names have a formal semantics. We also gave local components a formal semantics in specifications and modules.

Many specifications that we would expect to be equal to each other are not equal in type theory. It was shown that we can define a specification equality, based on a weak equality on types, that admits equalities not allowed by type equality.

10.1.3 Structured Specifications

One way of making large specifications is to construct them by combining smaller specifications. Such a style encourages us to make specifications incrementally, as well as helping to decompose the specification task into more manageable pieces. Chapters 6 and 7 are concerned with defining operators that allow us to combine and modify specifications to produce structured specifications. The main conclusion of Chapters 6 and 7 is that it is possible to define a useful collection of specification operators for a modular specification language in the framework of Martin-Löf's Type Theory.

In Chapter 6, we defined renaming, hiding and concatenation operators on signatures and computational elements; these operators were used in Chapter 7 to define structuring operators on specifications. An important achievement in Chapter 6 was the definition of a signature concatenation operator that resolved the name-clashes that can arise when combining specifications; we showed that allowing overloading of component names provided a simple mechanism for resolving name-clashes.

An important result in Chapter 6 was the relationship shown to exist between each signature operator and the corresponding computational element operator: given any signature S and computational element $m \in S$, $f(m) \in F(S)$ for any signature operator F and its corresponding computational element operator f. This relationship was exploited in Chapter 9 to give laws for implementing structured specifications.

In Chapter 7, we proposed a collection of specification operators that can be used to construct structured specifications. It was shown, by example, that the specification operators can be used to construct specifications in an incremental style, by adding more and more detail to a specification until we are satisfied that it specifies what we want. It was also shown that an incremental style of construction confers some benefits in the area of reusing specifications.

We formulated some laws that, under certain conditions, allow us to simplify the semantics of structured specifications; these laws aid reasoning about structured specifications. It was shown that the specification operators enjoy a collection of algebraic properties that can be used to reason about, and rearrange, structured specifications.

10.1.4 Refinement

One of the attractions of using type theory to construct specifications is that it can also be used to make implementations from specifications. Since specifications and modules are defined in the single framework of type theory, we can give a precise meaning to a notion of refinement that supports the implementation of specifications by mathematical transformation. In Chapter 8, we gave a formal definition of refinement for specifications in type theory, together with a collection of refinement laws. The main conclusion of Chapter 8 is that type theory provides an excellent framework for the systematic refinement of large structured specifications.

We have shown that our definition of refinement allows the piecewise refinement of specifications. That is to say, we can refine a specification by refining its individual parts more or less independently of each other, and then combine its refined parts to get a refinement of the whole specification. The piecewise refinement property is due to the monotonicity of our specification operators; we gave proofs of the monotonicity of each of our specification operators. We have also shown that refinement is transitive. The transitivity of refinement, together with the monotonicity of our operators, allow us to refine a specification in small steps, such that a derived specification is always a refinement of the original.

We stated, and proved, some refinement laws that allow large unstructured specifications to be refined by structuring them using the specification operators.

We showed that all types, not just specifications, can be refined in type theory. Moreover, we have shown that the types of the individual components in the signature of a specification can be refined to produce a refinement of the whole specification.

10.1.5 Implementation

Once we have refined a specification, we then implement the specification. The restriction of a canonical specification usually suggests an implementation, and making such an implementation is usually straightforward. However, implementing structured specifications is more complicated, and is discussed in Chapter 9.

We defined a collection of module operators, called implementors, that correspond to some of our specification operators. It was shown that we can often implement a structured specification by implementing its constituent canonical specifications, and replacing the specification operators used in the specification with their corresponding implementors. In other words, we can use implementors to implement structured specifications in a piecewise manner.

Piecewise implementation is shown to have its limitations; for example, we show that *enrich* and *union* operations cannot always be implemented in a piecewise manner. Furthermore, making an implementation with the same structure as its specification may not always lead to an efficient implementation. However, piecewise implementation is still a useful strategy, as it decomposes the implementation task into more manageable pieces. It was also shown that piecewise implementation encourages the reuse of implementations; and this helps to simplify the implementation task.

The main conclusion of Chapter 9 is that defining specification and module operators in the single framework of type theory allows us to implement structured specifications in a piecewise manner.

10.2 A Critical Review of MLT

In this thesis, we have shown that Martin-Löf's Type Theory (MLT) provides a more than adequate framework for constructing modular specifications and programs. We have found MLT pleasurable to use. The rules of the type theory have a logical structure that makes them easy to master, and the formal language of the type theory is very expressive as a specification and programming language. However, MLT has some drawbacks that limit its use as a specification and program development framework. In this section, we address some of the drawbacks of MLT. We give a critical appraisal of MLT for specification and programming in general, and of the particular approach taken in this thesis.

10.2.1 Specification and Programming in MLT

We begin by considering the intuitionistic logic used by Martin-Löf's Type Theory. We have found intuitionistic logic more difficult to use and reason about compared to classical logic. The fact that many propositions that are equal to each other in classical logic are not equal under intuitionistic logic means that specifications that we would expect, and desire, to be equivalent to each other are not equal in type theory. We overcame this problem by defining weak type equality ($\Leftrightarrow$) to compare propositions. However, the introduction of weak type equality imposes an overhead on specifiers and implementors, as they must reason about normal type equality and weak type equality in order to compare specifications.

The need for weak type equality and specification equality is an indication that in many instances the identification of propositions with types is far-fetched and awkward. This identification may result in computationally irrelevant proof objects appearing in programs derived from specifications in MLT. To reason about the computationally relevant parts of such programs, it is often necessary to separate the computational parts and proof objects. This is an irritating overhead when reasoning about programs in MLT. Consequently, there has been a trend towards separating propositions from types when making specifications in MLT and other type theories. Indeed, the Calculus of Constructions, and the Extended Calculus of Constructions both make a formal distinction between types and propositions. Our specifications of modules also separate data-types from propositions, by putting data-types in the signature, and propositions in the restriction of a specification. We have found that with such a separation, proof objects do not interfere with the derivation of modules, or reasoning about them. However, ensuring such a separation still adds to the work of a specifier, and we can make some specifications more easily by mixing propositions and data-types.

Subset types have been introduced into MLT in order to avoid computationally irrelevant proof objects appearing in programs. However, subset types do not integrate well with the principle of propositions as types because they throw away proof objects. The loss of proof objects makes it difficult to deduce properties about specifications that use subset types; we discussed this in Section 4.10.2. It seems that if we want a subset type that will work in practice, it must be possible to say that a proposition is true without explicitly showing a proof object. Nordström et al have obtained this by defining the *subset theory* [44]. The subset theory is an extension of MLT with two new forms of judgement P **prop** and P **true**, which mean that P is a proposition and P is true, respectively. In the subset theory, the logical constants—such as $\wedge$ and $\vee$—are no longer abbreviations for type constructors, but are introduced as new primitive constants. Moreover, propositions are no longer identified with types. It seems that the price we have to pay to introduce usable subset types is a more complicated theory.

Related to the identification of propositions with types is a potential to confuse propositions with Boolean values in MLT. This is not a serious problem, but it can make it hard to express some propositions succinctly. For example, $x < y$ $(x, y \in \mathbb{N})$ is a Boolean expression where $\_ < \_ \in \mathbb{N} \to \mathbb{N} \to \mathbb{B}$, but to use $x < y$ as a proposition we have to write $[x < y =_{\mathbb{B}} \mathit{true}]$ which is rather cumbersome. Moreover, the effective use of Boolean values requires the definition of Boolean connectives, and these can be confused with propositional connectives. For example, $[x =_{\mathbb{N}} y]$ is a proposition (and type), but the expression $x = y$ is a Boolean expression when used in the expression **if** $x = y$ **then** 0 **else** 1, where $\_ = \_ \in \mathbb{N} \to \mathbb{N} \to \mathbb{B}$ is a Boolean connective.

There are several problems with MLT that concern the expression of functions and types in the formal language of MLT. Firstly, general recursion is not available. Secondly, many useful functions that are pre-defined in most programming languages are not pre-defined in MLT. Thirdly, the theory does not include partial functions.

Many programming problems can be solved elegantly using recursive solutions, and this suggests that the introduction of general recursion to MLT is an urgent problem. The unavailability of general recursion means that the definitions of many recursive programs in MLT are more complicated than the definitions of the same programs in programming languages that allow general recursion. At present, we can only define recursive programs by using the primitive recursive operators, such as *natrec* and *listelim*, together with higher order functions. In our experience, such functions are difficult to define and reason about. The introduction of general recursion would not only simplify the definition of such functions, but would also make it easier to derive recursive programs from specifications in MLT. However, the introduction of general recursion to the theory is still an open problem, and is discussed in [45] and [42].

Some functions that we commonly use to make specifications and programs are not pre-defined in MLT. Such "auxiliary" functions include arithmetic operators such as sum, subtraction, *div*, *mod*; Boolean functions such as equality tests between naturals, ordering relations such as $\_ < \_$; and list operations such as head, tail, concatenation, etc. A programmer must define and, if necessary, deduce the properties of any auxiliary function he or she needs to use. This can add extra work to a specification or implementation task. We admit that defining most auxiliary functions is not intellectually taxing to the practising programmer; although some functions may be difficult to define due to the lack of general recursion. Nevertheless, the need to give the definitions of auxiliary functions can clutter specifications and programs in MLT.

The requirement that all functions be total in MLT has only been a problem to us when defining functions on types. Most of the functions we have defined on types are applied to $\sum$-types. To make such functions total, we have to define their domain to be U_1, even though we only apply the functions to $\sum$-types; this is because there is no kind in MLT that contains only $\sum$-types. It would be easier to define such functions as partial functions, if the theory allowed them. Constable et al [8] have introduced partial functions to the type theory used in the NuPRL system, and we regard that as a useful innovation to type theory.

The fact that specifications and programs inhabit different worlds in MLT causes some problems in using MLT as a program development framework. Firstly, it means we have to develop separate notations for specifications and programs; in this thesis, we have had to develop a specification language and an associated implementation language for modules. Secondly, it can affect the smooth derivation of programs since we have to jump from the world of types to the world of programs when making programs from specifications.

Although MLT suits the development of sequential functional programs, it is unsuited to the development of some other styles of programs that arise in the real world, such as imperative or concurrent programs. There is no formal notion of a state in MLT. Constable [9] suggests that state can be understood in type theory by manipulating n-tuples of values as if they were the values stored in a state. For example, if a state contains the values $x \in \mathbb{N}$ and $y \in \mathbb{N}$ then the statement $y := y+1$ is represented by the function $f(\langle x, y\rangle) = \langle x, y+1\rangle$. In practice, passing the state from one operation to the next is unwieldy, and it does not capture the idea of values being updated in place. Exceptions are another important programming language feature not available in MLT. However, Murthy [41] has shown that we can introduce control operators similar to exceptions into a constructive type theory. Therefore, in principle, exceptions could be added as an extension to MLT. We cannot make concurrent programs in MLT since all the operators in MLT are sequential, and functions can only be composed sequentially. This is a significant limitation to the usefulness of type theory for making programs in the real world, as the development of parallel computers has increased interest in making concurrent programs that exploit such hardware. Nondeterministic programs cannot be defined in MLT since all its operators are deterministic; the evaluation of terms in MLT is defined under a lazy and deterministic evaluation order. The lack of nondeterminism is a loss as it useful for making programs concerned with systems programming and scheduling resources. We can make nondeterministic specifications in MLT, but these must be implemented by deterministic programs. For example, the following specification does not determine the value of function f, but any implementation of f will be deterministic:

Elements $f \in \mathbb{N} \to \mathbb{N}$ **Restrictions** $\forall x \in \mathbb{N}.0 < f(x) < 100$ **end**

10.2.2 A Critical Appraisal of Our Approach

We now give an overall critical appraisal of the particular approach to making specifications and programs taken in this thesis. Our main criticism concerns the complexity of the semantics of our specification language and its associated implementation language. The semantics are complicated to understand and reason about because we have had to introduce so much notation to handle them. One reason for so much notation is the lack of pre-defined operations in MLT. The unavailability of general recursion has added to the complexity of the notation. The complexity of the semantics has been exacerbated by the fact that we have had to define many operations twice–once for specifications and once for modules. This is a direct consequence of the fact that specifications and programs inhabit different worlds in MLT.

The complexity of our semantics for specifications and modules adds to the difficulty of reasoning about, and proving properties of, specifications and programs. In particular, we found it hard to prove some of the algebraic properties of the specification operators. Indeed, we found it necessary to define some laws to simplify specifications made using the specification operators. We also found it necessary to give laws to simplify the restrictions of structured specifications because of the difficulties, discussed earlier, of reasoning in intuitionistic logic.

The fact that MLT distinguishes between specifications and programs has lead to our choice of a two-stage strategy for implementing specifications: first, we refine a specification, and then we find a module that implements its refinement. Therefore, we have developed two calculi for program development: one for refining specifications, and one for implementing refined specifications using our "implementor" functions. Consequently, a programmer is burdened by having to use, and master, two different calculi. To add to the programmer's burden, the two calculi employ different proof techniques. Refinement proofs employ equational reasoning, whereas proofs concerning the implementation of refined programs employ a constructive proof technique.

So far in this review, we have mentioned some of the limitations of using MLT to make specifications. However, in this thesis, we have also highlighted some special advantages of using type theory to make specifications. For example, dependent functions make it easy to return error messages. The theory also allows us the freedom to make specifications in an algebraic or model-oriented style as suits a specification problem. Furthermore, the use of types and specifications as first-class values make it easy to define parameterised specifications. There are other features of type theory that we have not exploited for making specifications. These include the use of higher order functions to describe restrictions; and the use of polymorphic types [3] to make specifications and programs.

We now compare the utility and elegance of our approach to specification and program development with other approaches such as Z, CLEAR, VDM, etc. The overall utility of our approach compares favourably against other modular approaches. Like Z and CLEAR, our approach encourages an incremental style of specification construction that is more than adequately supported by the operators we supply. Moreover, compared with Z or CLEAR, our approach offers more flexibility and ease in making structured specifications, because it allows us to structure specifications hierarchically by nesting specifications.

When it comes to specifying the individual structural units that make up a structured specification (i.e. canonical specifications in our approach, Schemas in Z, Theories in CLEAR, etc.) state based languages, like Z and VDM, have the advantage of allowing us to specify operations in an equational or imperative style; our approach only supports an equational style. Apart from our specification language, all the other specification languages mentioned above use classical logic. Therefore, because of the problems with intuitionistic logic that we discussed earlier, we sometimes find it harder to reason about specifications made in our specification language as compared to reasoning about similar specifications made in the other specification languages. However, it is often easier to rearrange and modify specifications in our approach because of the extra structuring techniques it offers.

It can be argued that VDM, Extended ML, CLEAR and the refinement calculus provide a more elegant approach to program development compared to ours, because in these languages the programming language is an implementable subset of the specification language. With such an approach, we can implement specifications by refinement alone; there is no need to jump abruptly from the world of specifications to the world of programs, as in our approach. Nevertheless, we have found our approach to implementing programs from specifications to be perfectly adequate.

10.3 Specification in ECC and MLT: a comparison

Luo has shown, in [27], that the Extended Calculus of Constructions (ECC) [24, 25, 26] provides an excellent framework for the modular specification and development of programs. He has shown this by defining specifications and specification building operators in ECC, as well as defining an effective notion of specification refinement in ECC. Luo's work tackles many of the issues discussed in this thesis. However, Luo's work is more general than ours: our work deals with many detailed issues such as component namespaces and local components, whereas Luo's work is more concerned with showing how the higher-order features of type theory provide useful mechanisms for structuring and implementing specifications. In addition, Luo investigates some issues, such as parameterisation, in more depth than we have discussed in this thesis. In this section, we discuss the points in common, as well as the differences, between Luo's work and ours.

In the following, we assume the reader is familiar with the basic features of ECC; recall that Section 1.1.3 contains a brief description of the Calculus of Constructions and ECC. For the purpose of comparison, ECC may be seen as an extension of MLT with an impredicative universe *Prop* of all propositions. There are minor differences in syntax between ECC and MLT. For example, type universes in ECC are written as $Type_i$ $(i > 0)$ and not U_i as in MLT; in fact, Luo omits the universe level i altogether and writes *Type* instead of $Type_i$ since i can be inferred from the context in which *Type* is used in ECC.

A specification in ECC is a pair, and not a type as in our work. The first component of a specification in ECC is any type $S \in Type$, called the *structure type* of the specification, and the second component is any predicate $P \in S \rightarrow Prop$ on S. The structure type of a specification *SP* plays the same role as a signature: it gives the type of the possible programs that may realise *SP*. The predicate part of *SP* specifies the properties that programs realising *SP* must satisfy. Note that, in common with our specifications in MLT, specifications in ECC keep their computational part (expressed by the structure type) and axiomatic part separate.

Notation For any specification *SP* in ECC, we may write $Str(SP)$ to mean the structure type of *SP*, and $Ax(SP)$ to mean the predicate part of *SP*. □

The decision to define specifications in ECC as pairs rather than $\sum$-types (as we do) is intentional. There are two reason for the decision. Firstly, it permits the definition of operations that decompose specifications in ECC into their structure type and axiomatic part (by using the functions *fst* and *snd*). Such operations could not be defined in ECC if specifications in ECC were $\sum$-types since ECC does not have an operator like *urec* in MLT that defines operations over the structure of all types. Secondly, the definition of specifications as pairs means that it is possible to define a type, called *SPEC*, of all specifications in ECC: $SPEC = \sum S \in Type.S \rightarrow Prop$. In principal, we can define a type of all specifications in MLT by using the subset type constructor; but such a definition is of limited use due to the limitations of subset types discussed earlier (Section 4.10.2). *SPEC* is useful for defining specification operators and parameterised specifications, and we say more about it later.

Luo shows how to specify modules like ours in MLT, but without component names and local components, by making the structure type of a specification in ECC a loose signature. Unlike specifications in MLT, there is no formal mechanism for naming components or introducing local components in specifications in ECC. This is reflected by the fact that a structure type can be any type, whereas a signature in MLT is defined to include a semantics for the namespace of components. Consequently, Luo's work is free of much of the detail that the inclusion of a formal semantics for namespace introduces to our work. The fact that ECC and MLT supply the same logical connectives means that the predicates used in the axiomatic parts of specifications in ECC are similar in style to the predicates used as restriction types by our specifications. In common with our work, Luo introduces observational equalities and demonstrates their usefulness in specifying abstract data types.

The definition of specifications in ECC as pairs means that the implementation relationship between specifications and implementations in MLT (i.e. that a specification is a type and its implementations are all the members of the type) does not hold between specifications and implementations in ECC. Nevertheless, implementations of specifications in ECC have a similar definition to modules in MLT: an implementation of a specification *SP* in ECC is defined as a pair consisting of any member m of the structure type of *SP*, together with a witness that m satisfies $Ax(SP)$ (i.e.

that $Ax(SP)(m)$ is inhabited). The set of all implementations of SP is given by the type $\sum m \in Str(SP).Ax(SP)(m)$; note the similarity of this type to our notion of specification in MLT.

We now compare the specification operators in our specification language with those defined by Luo. Luo does not define any specification operators that are similar in function to our *rename* or *hide* operators since the components of specifications in ECC are not named, and there is no mechanism for specifying local components. However, the remaining specification operators in our specification language correspond to specification operators defined by Luo. Luo defines an infix binary specification operator $\otimes$ which is similar in function to our *sum* operator: for any two specifications SP_1 and SP_2, $SP_1 \otimes SP_2$ specifies the components specified by SP_1 and SP_2. The structure type of $SP_1 \otimes SP_2$ is defined as $Str(SP_1) \times Str(SP_1)$, using the product type constructor ($\times$). Luo does not need an operator as sophisticated as signature concatenation—which we use to combine signatures in our definition of *sum*—to combine structure types since there is no formal namespace for components in structure types, and therefore the issue of combining namespaces does not arise.

Luo also defines an operator called *Extend* which extends a specification by adding some extra components and/or axioms. *Extend* is similar in function to our *enrich* operator. For example, the specification $Extend(SP, S, E)$ is specification SP extended by some extra components given by an extension S, which is function of type $S \in Str(SP) \rightarrow Type$, and some axioms E over the extended structure type ($E \in (\sum x \in Str(SP).S(x)) \rightarrow Prop$). The extension S can be seen as a structure type that is dependent on the components supplied by $Str(SP)$; extensions like S play the same role as dependent signatures in our work.

Luo defines the operator *Join* which is identical in function to our *union* operator. Given two specifications SP_1 and SP_2 with the same structure type—S, say—the specification $Join_S(SP_1, SP_2)$ has structure type S, and its axiomatic part is the conjunction of the axiomatic parts of SP_1 and SP_2. Luo also defines two operators *Con* (called constructor) and *Sel* (called selector) which are identical in function to our *derive* and *translate* operations, respectively. An application of *Con* has the form $Con_f(SP)$ for any specification SP, $f \in Str(SP) \rightarrow S$ and S any structure type. The function f plays the same role as a *derive* map. An application of *Sel* has the form $Sel_g(SP)$ for any specification SP, $g \in S \rightarrow Str(SP)$ and S any structure type. The function g plays the same role as a *translate* map.

Specifications in ECC are implemented through stepwise refinement just like our specifications in MLT: a specification is refined through a sequence of refinement steps until its axioms strongly suggest an implementation which can then be constructed via a constructive proof. Luo's definition of refinement is very similar to our definition of specification refinement in MLT:

Definition 10.1 (Specification Refinement in ECC)
A specification SP_1 refines to a specification SP_2 through a refinement map

$$h \in Str(SP_2) \rightarrow Str(SP_1)$$

(written $SP_1 \Longrightarrow_h SP_2$) if the following proposition is provable:

$$\forall x \in Str(SP_2).Ax(SP_2)(x) \Rightarrow Ax(SP_1)(h(x)) \qquad (1)$$

□

The refinement map in the above definition plays the same role as an abstraction map in our definition of refinement in MLT; and a witness to proposition (1) is equivalent to a proof map in our definition of refinement. Like our refinement relation $\sqsubseteq$, the refinement relation $\Longrightarrow$ is transitive. $\Longrightarrow$ is also monotonic with respect to the specification operators defined by Luo.

Luo's work does not address the issue of implementors (as we do in Chapter 9) for implementing structured specifications. However, unlike our work, Luo's work addresses the issue of implementing structured specifications whose individual sub-specifications share common data-types and/or operations. Such "sharing" may prevent a programmer from implementing the sub-specifications of a structured specification independently of each other, but Luo shows how a programmer can overcome this problem by using $\sum$-types to restructure such specifications.

Parameterised specifications in ECC are functions that return specifications as their result; as are parameterised specifications in our work. The parameters of a parameterised specification in ECC (and in our specification language) can be any values in type theory, including types, functions, program modules and specifications. Unlike our work, Luo addresses the interesting issue of implementing parametersised specifications. In particular, Luo gives a definition of refinement for parameterised specifications, together with some laws for refining parameterised specifications.

Finally, we return to the issue of the type *SPEC* of all specifications in ECC, described earlier in this section. The type *SPEC* allows Luo to define parameterised specifications—including specification operators, which are a special case of parameterised specifications—more concisely than we can in our specification language. For example, Luo may use *SPEC* to define parameterised specifications of the form $\lambda x \in SPEC.SP(x)$ (with type $SPEC \rightarrow SPEC$) which are parameterised over specifications. In our specification language, we define such parameterised specifications in the form $\lambda x \in U_1.SP(x)$ (with type $U_1 \rightarrow U_1$); note that we can only give specifications the type U_1. Consequently, in our specification language, we have to define the body, $SP(x)$, of such parameterised specifications as being well-typed for all types $x \in U_1$, and not just specifications, since functions in MLT (and ECC) must be total. This adds to the work of making such parameterised specifications in our specification language; although the additional work is not technically challenging.

In conclusion, Luo's work is an extremely important contribution to the study of constructing and refining modular specifications in type theory. It is interesting to note that there are many points in common between Luo's work and ours, even though they are set in different type thoeries.

10.4 A Final Comment

This thesis has demonstrated that type theory provides an excellent framework for both the specification and development of program modules. We have shown how type theory can be used to give a formal semantics to a module specification language, and an associated implementation language. We have also shown that implementations of modular specifications may be calculated by systematic refinement.

Appendix A

A.1 Dot Notation

The semantics of specifications and modules have been defined to include names for components. The names are intended to allow us to refer to the individual components of modules by using a Pascal style dot notation. In this appendix, we give a formal definition of dot notation on computational elements and modules.

Dot notation is defined using *value selector* functions on tuples; tuples are nested pairs of the form $\langle e_0, \langle e_2, \ldots \langle e_{n-1}, e_n \rangle \ldots \rangle \rangle$. Value selectors return the values of components in tuples via their position in the tuple. For example, the component at position i ($i \in \mathbb{N}$) in a tuple $b \in B$ is referred to as $b_B.i$. To calculate the type of any component referred to by dot notation on computational elements and modules, we will also define a form of dot notation that returns the types of components in signatures or specifications. Dot notation on signatures and specifications is defined using *type selector* functions on tuple types; tuple types are nested $\sum$-types of the form $\sum y_0 \in A_0 \ldots \sum y_{n-1} \in A_{n-1}.A_n$. Type selectors return the type of any component $b_B.i$ in a tuple $b \in B$ where B is a tuple type.

We begin by defining value and type selector functions on tuples. We then define dot notation on computational elements and signatures. Then we define dot notation on modules and specifications.

A.1.1 Selector Functions

In this section, we define functions that return the values and types of individual components within tuples. Functions that return the values in tuples are called *value selectors*, and functions that return the types of the values in tuples are called *type selectors*. Previously, tuples have been assumed to be nested pairs, so that tuple types are nested $\sum$-types. However, selector functions are generalised so that they can be applied to non-pair values. A non-pair value is any value whose type is not a $\sum$-type. Non-pair values are thought of as "tuples" containing a single value.

Value Selectors

Every tuple type has its own value selector function; just as every pair type has its own projection functions. Consider the tuple $b = \langle e_0, \langle e_2, \ldots \langle e_{n-1}, e_n \rangle \ldots \rangle \rangle$ that is assumed to have type $B = \sum y_0 \in A_0 \ldots \sum y_{n-1} \in A_{n-1}.A_n$. An application of value selection to b has the form $b_B.i$ for any index $i \in \mathbb{N}$. $b_B.i$ is the component at position i in tuple b. The value of $b_B.i$ is given by:

$$\begin{aligned} b_B.i &= e_i \quad , \; i \leq n \\ b_B.i &= e_n \quad , \; i > n \end{aligned}$$

Selector indices are assumed to range from 0 onwards, so that $b_B.0$ is the first component in b. Note that for any $i > n$, the expression $b_B.i$ always equals the final component in b; so a selector index is never out of range. Selector functions are defined in terms of the projection functions *fst* and *snd*. Figure A.1 contains a definition of $b_B.i$, for all $i \in \mathbb{N}$, in terms of *fst* and *snd*.

For any non $\sum$-type R, the value selector function on members of R is defined such that for any $r \in R$, $r_R.i = r$. In other words, a value selector function on members of a non $\sum$-type acts like an identity function, and ignores the selector index.

Type Selectors

To calculate the type of any component referred to by a value selector, we will define functions, called type selectors, that return the type of a component in a tuple. Consider the tuple type $B = \sum y_0 \in A_0 \ldots \sum y_{n-1} \in A_{n-1}.A_n$. An application of type selection has the form $B_b.i$ for any tuple $b \in B$ and index $i \in \mathbb{N}$. $B_b.i$ is the type of the component at position i in tuple b. The type of each component in any $b \in B$ is given as follows:

$$\begin{aligned} b_B.i \in B_b.i &= A_i(y_0 \backslash b_B.0, \ldots, y_{i-1} \backslash b_B.(i-1)) \quad , \; i \leq n \\ b_B.i \in B_b.i &= A_n(y_0 \backslash b_B.0, \ldots, y_{n-1} \backslash b_B.(n-1)) \quad , \; i > n \end{aligned}$$

Note that the type of a component $b_B.i$ may be dependent on the components $b_B.0, \ldots, b_B.(i-1)$ in b.

Given any non-tuple type R, type selection on any $r \in R$ is defined so that $R_r.i = R$ for any index $i \in \mathbb{N}$. Defining type selection functions over all types is mathematically appealing since it yields the following relationship between value and type selectors:

$$\forall B \in U_1.\forall b \in B.\forall i \in \mathbb{N}.(b_B.i \in B_b.i)$$

$$b_B.i = \begin{cases} fst(b) & , \ i = 0 \\ fst(snd(b)) & , \ i = 1 \\ \vdots & \\ fst(snd(\ldots snd(b)\ldots)) & , \ i = n-1 \\ snd(snd(\ldots snd(b)\ldots)) & , \ i \geq n \end{cases}$$

Figure A.1: Value Selection on Tuples of Type B

A Technical Definition of Value Selectors

A formal definition of value selection is given by defining a function VS (*value select*) that takes a type and returns its value selector function. The type of VS is given below:

$$VS \in \prod B \in U_1.\prod b \in B.\prod i \in \mathbb{N}.(B_b.i)$$

Using VS, the expression $b_B.i$ is defined as

$$b_B.i \equiv VS(B)(b)(i).$$

Function VS is defined in terms of *fst* and *snd*. The definition of VS is presented in a clausal form , and is given in Definition A.1.

Definition A.1 (Value Selectors)
Given any type B, $b \in B$ and index $i \in \mathbb{N}$ then $b_B.i \equiv VS(B)(b)(i)$. VS is defined inductively over the structure of type B as follows:

$$\begin{array}{lcll} VS(R)(b)(i) & = & b, & (R \text{ not a } \sum\text{-type}) \\ VS(\sum x \in P.Q(x))(b)(i) & = & \textbf{if } i = 0 \textbf{ then } fst(b) \textbf{ else } VS(Q(fst(b)))(snd(b))(i-1) & \end{array}$$

□

In the above definition, the expression $VS(Q(fst(b)))(snd(b))(i-1)$ is the component at position $(i-1)$ in $snd(b)$ where $snd(b) \in Q(fst(b))$ since $b \in \sum x \in P.Q(x)$.

A Technical Definition of Type Selectors

A formal definition of type selector functions is given in terms of a function TS (*type select*) that takes a type and returns its type selector function. The type of TS is given below:

$$TS \in \prod B \in U_1.\prod b \in B.\prod i \in \mathbb{N}.(U_1)$$

Using TS, the type $B_b.i$ is defined as:

$$B_b.i \equiv TS(B)(b)(i).$$

Function TS is defined over type terms and is similar, in style, to the definition of VS. The definition of TS is given in Definition A.2.

Definition A.2 (Type Selectors)
Given any type B, $b \in B$ and index $i \in \mathbb{N}$ then $B_b.i \equiv TS(B)(b)(i)$. TS is defined inductively over the structure of type B as follows:

$$\begin{array}{lcll} TS(R)(b)(i) & = & R, & (R \text{ not a } \sum\text{-type}) \\ TS(\sum x \in P.Q(x))(b)(i) & = & \textbf{if } i = 0 \textbf{ then } A \textbf{ else } TS(Q(fst(b)))(snd(b))(i-1) & \end{array}$$

□

In the above definition, the expression $TS(Q(fst(b)))(snd(b))(i-1)$ is the type of the component at position $(i-1)$ in $snd(b) \in Q(fst(b))$.

Theorem A.1 Given any B **type**, $b \in B$ and $i \in \mathbb{N}$ then $b_B.i \in B_b.i$. □

Proof By structural induction on B. □

A.1.2 Dot Notation on Computational Elements

In this section, we define a Pascal style dot notation on computational elements and signatures. Given any signature S, computational element $m \in S$, and a component name $n \in \mathbb{S}$, the expression $m.n$ is the value of a component named n in m. The expression $S_m.n$ is the type of $m.n$, i.e. $m.n \in S_m.n$. Dot notation on computational elements and signatures may refer to both local and visible components. In the event that n is overloaded in m (and S), then $m.n$ returns the value of the last (right-most) version of a component named n in m.

Dot notation on computational elements is defined using the value selector function VS. An expression $m.n$ is defined by first seeking the position—i, say—at which n occurs in the domain of m, and then using VS to select the component at position i in the value tuple of m. Similarly, $S_m.n$ is defined by using the type selector TS to calculate the type of the component at position i in the value tuple of m. In the event that there is no visible or local component named n in m, the expression $m.n$ returns the unit value tt; and $S_m.n$ returns the unit type T.

The definition of dot notation on computational elements and signatures is given in Definition A.3 with an explanation following. The expression

$$index(dom(m))(n)(equalAll)$$

is the index of the last occurrence of name n in the domain of m ($dom(m)$); n can be either visible or local. If n is not found in $dom(m)$ then the index of the last element, tt, in the value tuple of m ($val(m)$) is returned. $Lsg(S)$ is the loose signature of S, so that $val(m) \in Lsg(S)$.

Definition A.3 (Dot notation on computational elements)
For all signatures S, $m \in S$ and $n \in \mathbb{S}$,

$$\begin{array}{lcl} m.n & = & VS(Lsg(S))(val(m))(index(dom(m))(n)(equalAll)) \in S_m.n \\ S_m.n & = & TS(Lsg(S))(val(m))(index(dom(m))(n)(equalAll)) \in U_1 \end{array}$$

□

The functions *index* and *equalAll* are given in Definitions A.4 and A.6, respectively, with an explanation following. The expression $index(l)(n)(equal)$ returns the position of the last tagged name—*e*, say—in domain $l \in \mathbb{D}$, that makes *equal*(e,n) true. The expression $equalAll(e,n)$ compares a tagged name, *e*, with an un-tagged name, *n*, and returns true if the two names are equal when the tag on *e* is ignored. The expression $isin(n)(l)(equal)$ is true if the name *n* appears as a tagged name in l. Consequently, $index(l)(n)(equalAll)$ expresses the position, in *l*, of the last occurrence of component name *n*.

Definition A.4 (*index*)
For all $e : l \in \mathbb{D}$, $n \in \mathbb{S}$ and $equal \in (\mathbb{S}+\mathbb{S}) \rightarrow \mathbb{S} \rightarrow \mathbb{B}$, the function

$$index \in \mathbb{S} \rightarrow \mathbb{D} \rightarrow ((\mathbb{S}+\mathbb{S}) \rightarrow \mathbb{S} \rightarrow \mathbb{B}) \rightarrow \mathbb{N},$$

is defined as follows:

$$\begin{array}{lcl} index(n)(nil)(equal) & = & 0 \\ index(n)(e:l)(equal) & = & \textbf{if } equal(e)(n) \wedge \neg(isin(n)(l)(equal)) \textbf{ then} \\ & & \quad 0 \\ & & \textbf{else} \\ & & \quad index(n)(l)(equal) + 1 \end{array}$$

□

Definition A.5 (*isin*)
For all $e : l \in \mathbb{D}$, $n \in \mathbb{S}$ and $equal \in (\mathbb{S}+\mathbb{S}) \rightarrow \mathbb{S} \rightarrow \mathbb{B}$, $isin \in \mathbb{S} \rightarrow \mathbb{D} \rightarrow \mathbb{B}$ is defined as follows:

$$\begin{array}{lcl} isin(n)(nil)(equal) & = & \text{false} \\ isin(n)(e:l)(equal) & = & \textbf{if } equal(e)(n) \textbf{ then } \text{true} \textbf{ else } isin(n)(l) \end{array}$$

□

Definition A.6 (*equalAll*)
For all $e \in \mathbb{S}+\mathbb{S}$ and $n \in \mathbb{S}$, $equalAll \in (\mathbb{S}+\mathbb{S}) \rightarrow \mathbb{S} \rightarrow \mathbb{B}$ is defined as follows:

$$equalAll(e)(n) = ((e = inl(n)) \text{ or } (e = inr(n)))$$

□

Restricted Dot Notation on Computational Elements

In this section, we define a restricted form of dot notation that cannot refer to local components. This restricted form of dot notation will be used later to define dot notation on modules. For any signature S, computational element $m \in S$ and name $m \in \mathbb{S}$, the restricted form of dot notation on computational elements has the form $m.[n]$. If n is a visible component name in m, or if n is not a component name in m, then $m.[n]$ is $m.n$. But, if n is a local component name in m then $m.[n]$ is the unit value $tt \in T$. The restricted form of dot notation on signatures has the form $S_m.[n]$; $S_m.[n]$ is the type of $m.[n]$.

The definition of restricted dot notation is almost identical to the definition of unrestricted dot notation given in the previous section, except that the function *equalAll* is replaced by the function *equalVis* given in Definition A.7. An expression $\mathit{equalVis}(e, n)$ is true iff e is a visible name which, when we strip its tag, is equal to the un-tagged name n. The definition of restricted dot notation is given in Definition A.8.

Definition A.7 (*equalVis*)
For all $e \in \mathbb{S} + \mathbb{S}$ and $n \in \mathbb{S}$, $\mathit{equalVis} \in (\mathbb{S} + \mathbb{S}) \to \mathbb{S} \to \mathbb{B}$ is defined as follows:

$$\mathit{equalVis}(e)(n) = (e = \mathit{inl}(n))$$

□

Definition A.8 (Restricted Dot extraction)
For all S **sig**, $m \in S$ and $n \in \mathbb{S}$,

$$\begin{aligned} m.[n] &= \mathit{VS}(\mathit{Lsg}(S))(\mathit{val}(m))(\mathit{index}(\mathit{dom}(m))(n)(\mathit{equalVis})) \in S_m.[n] \\ S_m.[n] &= \mathit{TS}(\mathit{Lsg}(S))(\mathit{val}(m))(\mathit{index}(\mathit{dom}(m))(n)(\mathit{equalVis})) \in U_1 \end{aligned}$$

□

A.1.3 Dot Notation on Modules and Specifications

Dot notation on modules allows us to refer to any visible components in modules; and dot notation on specifications allows us to refer to the type of any visible component in a specification. Any dot references to local components in modules and specifications returns the unit value *tt* and the unit type T, respectively. Dot notation on modules and signatures is defined using the restricted form of dot notation on computational elements and signatures.

Definition A.9 (Dot notation on modules and specifications)
For all specification SP, modules $m \in SP$ and name $n \in \mathbb{S}$,

$$\begin{aligned} m.n &= \mathit{ce}(m).[n] \in SP_m.n \\ SP_m.n &= \mathit{Sig}(SP)_{\mathsf{ce}(m)}.[n] \in U_1 \end{aligned}$$

□

Bibliography

[1] S.N. Ahmed and J.M. Morris. Constructing and Refining Modules in a Type Theory. In J.M. Morris, editor, *Proceedings of the 4th Workshop on Refinement*, Springer-Verlag (BCS Workshop series), 1991.

[2] R.J.R. Back. Correctness preserving program refinements: proof theory and applications. Mathematical Centre Tracts 131, Mathematisch Centrum, Amsterdam, 1980.

[3] R.C. Backhouse et al. Do-it-yourself type theory. *Formal Aspects of Computing*, Vol.1, No. 1, January-March 1989.

[4] R.M. Burstall, J.A Goguen. The semantics of CLEAR, a specification language. In *Proc. of Advanced Course on Abstract Software Specifications*, LNCS 86, Springer, 1981.

[5] R.M. Burstall, J.A Goguen. An Informal Introduction to Specifications Using CLEAR. In N. Gehani, A.D. McGettrick, editors, *Software Specification Techniques*, Addison-Wesley, 1986.

[6] R.M. Burstall, J. McKinna. Deliverables: an approach to program development in the Calculus of Constructions. In the preliminary *Proceedings of the 1st Workshop on Logical Frameworks*, 1990.

[7] L. Cardelli. Typeful Programming. Technical Report 45, DEC Systems Research Center, 130 Lytton Ave., Palo Alto, 1989.

[8] R.L. Constable et al. *Implementing Mathematics With The NuPRL Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[9] R.L. Constable. Lectures on: Classical Proofs as Programs. In F. L. Bauer et al, editors, *Logic and Algebra of Specification*, NATO ASI Series, Springer-Verlag, Berlin, 1991.

[10] T. Coquand, G. Huet. The Calculus of Constructions. *Information and Computation*, 76 (2/3), pages 96-120, 1988.

[11] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J, 1976.

[12] H. Ehrig, H-J. Kreowski, B. Mahr, P. Padawitz. Algebraic implementation of abstract types. *Theoretical Computer Science*, 20, 1982.

[13] J.S. Fitzgerald, C.B. Jones. Modularizing a Formal Description of a Database System. In D. Bjorner et al, editors, *VDM '90 VDM and Z*, LNCS 428, Springer, Berlin, 1990.

[14] J.A. Goguen, J.J. Tardo. An Introduction to OBJ: a language for writing and testing formal algebraic program specifications. In N. Gehani, A.D. McGettrick, editors, *Software Specification Techniques*, Addison-Wesley, 1986.

[15] J.A. Goguen, J.W. Thatcher, E.G. Wagner. Initial algebra semantics of continuous algebras. *Journal of the ACM*, 24, 1977.

[16] J.V. Guttag. *The specification and application to programming of abstract data types.* Ph.D. Thesis, University of Toronto, 1975.

[17] J.V. Guttag, J.J. Horning. Report on the LARCH shared language. *Science of Computer Programming*, 6, pages 103-134, 1986.

[18] R. Hennicker. Observational Implementations. In B. Moniens, R. Cori, editors, *Proc. STACS 89*, LNCS 349, Springer, Berlin, 1989.

[19] C.A.R. Hoare. Proofs of Correctness of Data Representations. *Acta Informatica*, 1, pages 271-281, 1972.

[20] W.A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479-490. Academic Press, London, 1980.

[21] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[22] S. King. Z and the Refinement Calculus. In D. Bjorner et al, editors, *VDM '90 VDM and Z*, LNCS 428, Springer, Berlin, 1990.

[23] Z. Luo, R. Pollack, and P. Taylor. How to Use LEGO: a preliminary user's manual. LFCS Technical Note LFCS-TN-27, Dept. of Computer Science, Edinburgh University, 1989.

[24] Z. Luo. ECC, an extended Calculus of Constructions. In *Proc. of the Fourth Annual Symposium on Logic in Computer Science*, Asilomar, California, U.S.A, June 1989.

[25] Z. Luo. A Higher-Order Calculus and Theory Abstraction. *Information and Computation*, 90, pages 107-137, 1991.

[26] Z. Luo. *An Extended Calculus of Constructions.* Ph.D. Thesis, University of Edinburgh, 1990.

[27] Z. Luo. Program Specification and Data Refinement in Type Theory. LFCS Technical Notes ECS-LFCS-91-131, Dept. of Computer Science, Edinburgh University, 1991.

[28] P. Martin-Löf. A Theory of Types. Technical report 71-3, University of Stockholm, 1971.

[29] P. Martin-Löf. An Intuitionistic Theory of Types. Technical report, University of Stockholm, 1972.

[30] P. Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science, 6, 1979*, North-Holland, Amsterdam, 1982.

[31] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Naples, 1984.

[32] C.A. Middelburg. VVSL: A Language for Structured VDM Specifications. *Formal Aspects of Computing*, Vol.1, No. 1, January-March 1989.

[33] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[34] J. Mitchell, G. Plotkin. Abstract types have existential type. In *Proc. of the 12th ACM Symposium on Principles of Programming Languages*, New York, 1985.

[35] C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, Vol.10, No.3, pages 403-419, July 1988.

[36] C. Morgan and P.H.B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, 27, pages 481-503, 1990.

[37] J.M. Morris. A Theoretical Basis For Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9, pages 287-306, 1987.

[38] J.M. Morris. The Laws of Data Refinement. *Acta Informatica*, 26, pages 287-308, 1989.

[39] J.M. Morris and S.N. Ahmed. Designing and Refining Specifications with Modules. In J. Woodcock , editor, *Proceedings of the 3rd Workshop on Refinement*, Springer-Verlag (BCS Workshop series), 1990.

[40] P. Mosses. *Action Semantics.* Cambridge Tracts in Theoretical Computer Science, 26, Cambridge University Press, 1992.

[41] C. Murthy. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, Department of Computer Science, 1990.

[42] B. Nordström. Terminating General Recursion. *BIT*, 28 (3), pages 605-619, October 1988.

[43] B. Nordström and K. Petersson. The Semantics of Module Specifications in Martin-Löf's Type Theory. Report 36, Programming methodology group, Department of computer science, University of Gothenburg, 1985.

[44] B. Nordström, K. Petersson and J.M. Smith. *Programming in Martin-Löf's Type Theory.* Oxford Clarenden Press, Oxford, 1990.

[45] L. C. Paulson. Constructing Recursion Operators in Intuitionistic Type Theory. *Journal of Symbolic Computation*, 2, pages 325-355, 1986.

[46] A. Salvesen and J.M. Smith. The Strength of the Subset Type in Martin-Löf's Type Theory. Report 45, Programming methodology group, Department of computer science, University of Gothenburg, 1985.

[47] A. Sampaio and S. Meira. Modular Extensions to Z. In D. Bjorner et al, editors, *VDM '90 VDM and Z*, LNCS 428, Springer, Berlin, 1990.

[48] D. Sannella. Formal Program Development in Extended ML for the Working Programmer. In J. Woodcock, editor, *Proceedings of the 3rd Workshop on Refinement*, Springer-Verlag (BCS Workshop series), 1990.

[49] D.T. Sannella and A. Tarlecki. Program specification and development in Standard ML. In *Proc. 12th Annual ACM Symposium on Principles of Programming Languages*, 1985.

[50] D.T. Sannella and A. Tarlecki. Towards formal development of programs from algebraic specifications: implementations revisited. In H. Ehrig et al, editors, *TAPSOFT '87*, LNCS 249, Springer, Berlin, 1987.

[51] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specifications and Implementations. Technical Report CSR-155-83, Dept. of Computer Science, Edinburgh University, 1983.

[52] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge Tracts in Theoretical Computer Science, 3, Cambridge University Press, 1988.

[53] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* Cambridge, Massachusetts, MIT Press, 1977.

[54] M. Wand. Final algebra semantics and data type extensions. *Journal of Computer System Science*, 19, 1981.

[55] M. Wirsing. Structured Algebraic Specifications: A Kernel Language. *Theoretical Computer Science*, 42, pages 123-249, 1986.

[56] M. Wirsing and M. Broy. A Modular Framework for Specification and Implementation. Invited paper in J. Diaz, F. Orejas, editors, *TAPSOFT 89*, LNCS 351, Springer, Berlin, 1989.

[57] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14, pages 221-227, April 1971.