THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Operational Semantics and Polymorphic Type Inference

Mads Tofte

Ph. D.
University of Edinburgh
1987

Abstract

Three languages with polymorphic type disciplines are discussed, namely the λ-calculus with Milner's polymorphic type discipline; a language with imperative features (polymorphic references); and a skeletal module language with structures, signatures and functors. In each of the two first cases we show that the type inference system is consistent with an operational dynamic semantics.

On the module level, polymorphic types correspond to signatures. There is a notion of principal signature. So-called signature checking is the module level equivalent of type checking. In particular, there exists an algorithm which either fails or produces a principal signature.

Contents

1 Introduction 1

I An Applicative Language 8

2 Milner's Polymorphic Type Discipline 9
- 2.1 Notation 10
- 2.2 Dynamic Semantics 11
- 2.3 Static Semantics 13
- 2.4 Principal Types 20
- 2.5 The Consistency Result 21

II An Imperative Language 26

3 Formulation of the Problem 27
- 3.1 A simple language 29
- 3.2 The Problem is Generalization 31

4 The Type Discipline 36
- 4.1 The Inference system 36
- 4.2 Examples of Type Inference 37
- 4.3 A Type Checker 45

5 Proof of Soundness 47
- 5.1 Lemmas about Substitutions 47
- 5.2 Typing of Values using Maximal Fixpoints 51
- 5.3 The Consistency Theorem 62

6 Comparison with Damas' Inference System 70
- 6.1 The Inference System 70
- 6.2 Comparison 76
- 6.3 Pragmatics 77

III A Module Language 79

7 Typed Modules 80

8 The Language ModL 83
8.1 Syntax 84
8.2 Semantic Objects 88
8.3 Notation 94
8.4 Inference Rules 95
8.4.1 Declarations and Structure Expressions 95
8.4.2 Specifications and Signature Expressions 97
8.4.3 Programs 98

9 Foundations of the Semantics 100
9.1 Coherence and Consistency 100
9.1.1 Coherence 102
9.1.2 Consistency 106
9.1.3 Coherence Only 109
9.2 Well-formedness and Admissibility 110
9.3 Two Lemmas about Instantiation 112

10 Robustness Results 116
10.1 Realisation and Instantiation 116
10.2 The Strict Rules Preserve Admissibility 119
10.3 Realisation and Structure Expressions 121

11 Unification of Structures 123
11.1 Soundness of *Unify* 128
11.2 Completeness of *Unify* 133
11.3 Comparison With Term Unification 140
11.4 Comparison with Aït–Kaci's Type Discipline 141

12 Principal Signatures 145
12.1 The Signature Checker, *SC* 148
12.2 Soundness of *SC* 150
12.3 Completeness of SC 159

13 Conclusion 166
13.1 Summary 166
13.2 How the ML Modules Evolved 167
13.3 The Experience of Using Operational Semantics 169
13.4 Future Work 172

Appendix A: Robustness Proofs 174

Bibliography 201

Chapter 1

Introduction

Since early days in programming the concept of *type* has been important. The basic idea is simple; the values on which a program operates have types and whenever the program performs an operation on a value the operation must be consistent with the type of the value. For example, the operation "multiply x by 7" makes sense if x is a value of type integer or real, but not if x is a day in the week, say.

Any programming language comes with some *typing rules*, or a *type discipline*, if you like, that the programmer keeps in mind when using the language. This is true even of so-called "untyped" languages.[1]

There is an overwhelming variety of programming languages with different notions of values and types and consequently also a great variety of type disciplines.

Some languages have deliberately been designed so that a machine by just examining the program text can determine whether the program complies with the typing rules of the language. In our example, "multiply x by 7", it is a simple matter, even for a computer, to check the well-typedness of the expression assuming that x has type *int*, say, without ever doing multiplications by 7. This kind of textual analysis, *static type checking*, has two advantages: firstly, one can discover programming mistakes before the program is executed; and secondly, it can help a compiler generate code.

For these languages one can factor out the *static semantics* from the *dynamic semantics*. The former deals with type checking and possibly other things a compiler can handle, while the latter deals with the evaluation of programs provided

[1] Every LISP programmer knows that one should not attempt to add an integer and a list — although this is not always conceived as a typing rule

they are legal according to the static semantics. Let us refer to this class of languages as the *statically typed* languages. It includes ALGOL [4], PASCAL [44], and Standard ML [18, 20].

But there are also languages where such a separation is impossible. One example is LISP [26] which has basically one data type. The "type errors" one can commit are of such a kind that they cannot in general be discovered statically by a mere textual analysis.[2] Similarly, there can be no separation in languages where types can be manipulated as freely as values. Let us call such languages *dynamically typed.*

At first sight it seems a wonderful idea that a type checker can find programming mistakes even before the program is executed. The catch is, of course, that the typing rules have to be simple enough that we humans can understand them and make the computers enforce them. Hence we will always be able to come up with examples of programs that are perfectly sensible and yet illegal according to the typing rules. Some will be quick to say that far from having been offered a type discipline they have been lumbered with a type bureaucracy.

The introduction of *polymorphism* [27] in functional programming must have been extremely welcome news to people who believed in statically typed languages because the polymorphic type checker in some ways was a lot more tolerant than what had previously been seen. A *monomorphic* type system is one where every expression has at most one type. By contrast, in Milner's polymorphic type discipline there is the distinction between a *generic type*, or *type scheme*, and all the *instances* of the generic type. One of the typing rules is that if an expression has a generic type, then it also has all instances of the generic type. So the empty list, for example, has the generic type $\forall\alpha.\alpha\ list$ and all its instances $int\ list$, $(bool\ list)\ list$, $(int \rightarrow bool)\ list$, and so on. In fact, $\forall\alpha.\alpha\ list$ is said to be the *principal* type of the empty list, because all other types of the empty list can be obtained from it by instantiation.

Polymorphism occurs naturally in programming in many situations. For example the function that reverses lists is logically the same regardless of the type of list it reverses. Indeed, if all lists are represented in a uniform way, one compiled version of the reverse function will suffice.

Milner's polymorphic type discipline has been used in designing the functional

[2] Even in statically typed languages there will normally be kinds of "type errors" that cannot be discovered. Taking the head of the empty list is one example; index errors in arrays is another.

language ML, first in "Old ML", which was a "meta language" for the proof system LCF [15], and later in Standard ML [18].

The main purpose of this thesis is to demonstrate that the basic ideas in Milner's polymorphism carry over from the purely applicative setting to two quite different languages. Because these languages have different notions of values and types, the objects we reason about are different. But there is still the idea of types that are instances of type schemes, there are still notions of principal types, and using different kinds of unification algorithm one can get new type checkers that greatly resemble the one for the applicative case. Perhaps some category theorist will tell us that these different type systems are all the same, but I find it interesting that this kind of polymorphism "works" in languages that are not the same at all.

Unfortunately, polymorphism does not come for free. The price we have to pay is that we have to think pretty hard when we lay down the typing rules. First and foremost, typing rules must be *sound* in the sense that if they admit a program then (in some sense) that program must not be bad. Whereas in monomorphic type disciplines one would not feel compelled to invest lots of energy in investigating soundness, one has to be extremely careful when considering polymorphism. Indeed Part II of this thesis has its root in the historical fact that people have worked very hard to extend the purely applicative type discipline to one that handles *references* (pointers) as values. Polymorphic *exceptions* were in ML for years before it quite recently was discovered that the "obvious" typing rules are unsound.

Clearly we do not want to launch unsound type inference systems. Perhaps we do not want to undertake the task of proving the soundness of typing rules for a big programming language, such as ML, but the least we can do is to make sure that the big language rests on a sound base. Then we want to formulate the typing rules for the small language in a way that is convenient for stating and proving theorems.

It so happens that there is a notation that is extremely convenient for defining these polymorphic type disciplines (and others as well, I trust). It started as "Structural Operational Semantics" [34], the French call it "Natural Semantics" [9] — I shall use the term "operational semantics". The idea is to borrow the concept of *inference rule* and *proof* from formal logic. The typing rules are

then expressed as *type inference rules.* For example the rule

$$\frac{e_1 : \tau' \rightarrow \tau \quad e_2 : \tau'}{e_1\ e_2 : \tau}$$

can be read: " if you can prove that the expression e_1 has type $\tau' \rightarrow \tau$ and that e_2 has type τ' then you may infer that the application of e_1 to e_2 has type τ." Such rules need not be deterministic (τ is not a function of e, not even when e contains no free variables) and that makes them very suitable for defining polymorphism.

Given a type inference system in the form of a collection of type inference rules, how do we investigate soundness?

The first soundness proofs used denotational semantics [14, 13]. Types are seen as ideals of values in a model of the lambda calculus and it is proved that if by the type inference system an expression has a type then the value denoted by the expression is a member of the ideal that models the type.

It seems a bit unfortunate that we should have to understand domain theory to be able to investigate whether a type inference system admits faulty programs. The approach to soundness I take is different in that I also use operational semantics to define the dynamic semantics and then prove that, in a sense that is made precise later on, the two inference systems are consistent. This only requires elementary set theory plus a simple powerful technique for proving properties of maximal fixpoints of monotonic operators.

1.1 Related Work

Let us briefly review some of the contributions related to typing of programs or, more generally, typing of formal expressions.

Hindley's seminal paper [22] concerns the typing of objects in combinatory logic. The expressions studied by Hindley can be thought of as the lambda calculus [5] with constants. He considers type expressions built from base types using the type constructor for function space. In Hindley's terminology, a *type scheme* is a type in which type variables can occur. Hindley gives type inference rules allowing inferences of the form $\mathcal{A} \vdash \alpha X$ where X is an object (expression), α is a type scheme and $\mathcal{A}$ is a set of statements assigning one (and only one) type scheme to every variable that occurs free in X. He proves the existence of *principal type schemes* in the following strong sense: for all X, if for some α and $\mathcal{A}$ one has $\mathcal{A} \vdash \alpha X$ then there exists a type scheme, α_0, with the property that for all $\mathcal{A}'$, α', if $\mathcal{A}' \vdash \alpha' X$ then α' is a substitution instance of α_0.

Milner [27] independently discovered essentially the same result but he was able to extend the type inference rules so that a function, once declared, can be applied to objects of different types. For instance, the expression

```
let I = λ x.x in (I 3, I true)        (1.1)
```

is typable in Milner's system. By contrast, the semantically equivalent

```
(λ I. (I 3, I true))(λ x.x)        (1.2)
```

is typable neither in Hindley's system nor in Milner's system. The essential innovation in Milner's system is the introduction of *bound* type variables. More precisely, trying to maintain the above notation, Milner's notion of a *type scheme* is an expression, α, of the form $\forall a_1 \cdots a_n.\beta$, where β is what Hindley called a type scheme, and $a_1, \ldots, a_n$ are type variables bound in α. In (1.1), when I is declared it can be ascribed the type scheme $\forall a.a \rightarrow a$ which in the two applications can be instantiated to $int \rightarrow int$ and $bool \rightarrow bool$, respectively. This extension can be done without the loss of principal typing, although the notion of principal typing is slightly different from the one stated above.

Milner's extension gives the ability to type a very large class of programs. Type checking can be done effectively by a type checker [27, 14]. Milner's type discipline is used in the language Standard ML.

Another extension of Hindley's work is the introduction of *intersection types* by Coppo *et al.* [10, 11, 12, 38]. For instance, at the binding of I in (1.2), I can be ascribed the type $(int \rightarrow int) \cap (bool \rightarrow bool)$ making the two applications of I typable. The use of intersection types replaces the binding of type variables in Milner's scheme, in fact every expression that is well-typed in Milner's system is also well-typed using intersection types. The price for the greater power is that the well-typedness of expressions is only semi-decidable. There is an algorithm which produces principal types for well-typed programs, but it is not always able to detect that an ill-typed program is ill-typed.

Common to the above three approaches is that type expressions are inferred from expressions that do not contain type expressions. Thus, in Milner's system, the type checker infers the types of formal parameters of functions and of functions that are declared in the program.

A different approach to polymorphism is Reynolds' second order typed lambda calculus [35, 36]. Here one can form functional abstractions, the formal parameter

being a type variable. Such an abstraction can be applied to a type giving an object the type of which depends on the actual type argument.

Finally there is work on *type inheritance* or *subtyping*. This is a form of polymorphism which is natural in connection with typing of labelled records. If, for example, function `f` selects the component labelled `L` from its argument, then it should be possible to apply `f` to all arguments that have an `L` component. Cardelli [8] introduced one system for subtyping and, more recently, Wand [42] has given a similar type inference system. Principal types do not exist for Cardelli's system. Due to a mistake principal types do not exist in Wand's system either, but I understand that Wand will publish a corrected version. Although developed independently, there are strong similarities between Wand's unification algorithm and the one presented in Part III of this thesis.

A more detailed overview of the above approaches to typing of expressions can be found in [36]. See also Chapter 6 for a comparison of our imperative type discipline with that of Luis Damas, and Section 11.4 for a comparison between structure unification in ML and the similar unification algorithm due to H. Aït–Kaci.

The three type disciplines we study are all related to the programming language ML, although they are not applicable to ML only. Wikström's textbook on ML [43] describes the the core language at length. Harper's shorter report [16] explains the full language, including modules.

1.2 Outline

There are three parts. In Part I we review Milner's polymorphic type discipline for a purely applicative language and we formulate and prove a consistency result using operational semantics.

In Part II we extend the language to have references (pointers) as values. We present a new polymorphic type discipline for this language and compare it with one due to Luis Damas [13]. The soundness of the new type inference system is proved using operational semantics.

Part III is concerned with a polymorphic type discipline for modules. The language has structures, signatures and functors as in ML. Signatures will be seen to correspond to structures as type schemes correspond to types in the applicative setting. We present a unification algorithm for structures that generalizes the ordinary first order unification algorithm used for types, and we present a "sig-

nature checker" (the analogue of a type checker) and prove that it finds *principal signatures* of all well-formed signature expressions.

In the conclusion I shall comment on the role of operational semantics partly based on the proofs I have done, and partly based on the experience we as a group had of using operational semantics to write the full ML semantics [20].

Part I

An Applicative Language

Chapter 2

Milner's Polymorphic Type Discipline

We define a dynamic and a static semantics for a little functional language and show that, in a sense to be made precise below, they are consistent.

The type discipline is essentially Milner's discipline for polymorphism in functional languages [27, 14]. However, we formulate and prove soundness of the type inference system with respect to an operational dynamic semantics instead of a denotational semantics.

I apologize to readers who already know this type discipline for bothering them with the differences between types and type schemes, substitution and instantiation, the role of free type variables in the type environment, and so on. However, I cannot bring myself to copy out the technical definitions without some examples and comments, mostly because I have a feeling that many have some experience of polymorphism through the use of ML without being used to thinking of the semantics in terms of inference rules. Moreover, a good understanding of the inference rules is essential for the understanding of the type disciplines in Part II and III.

The soundness of the type inference system is not hard to establish using operational semantics. The proof therefore serves as a simple illustration of a proof method that will be put to more heavy use in the later parts.

We are considering the following little language, Exp. Assuming a set, Var, of program variables

$$x \in \mathrm{Var} = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{x}, \mathtt{y}, \ldots\}.$$

The language Exp of expressions, ranged over by e, is defined by the syntax

$e ::= x$	variable
$\mid\ \lambda x.e_1$	lambda abstraction
$\mid\ e_1\ e_2$	application
$\mid$ **let** $x = e_1$ **in** e_2	let expression

2.1 Notation

Throughout this thesis we shall give inference rules that allow us to infer sequents of the form

$$A \vdash \textit{phrase} \longrightarrow B$$

where *phrase* is a syntactic object and A and B are so-called *semantic* objects.

Semantic objects are always defined as the least solution to a collection of set equations involving Cartesian product ($\times$), disjoint union ($+$), and finite subsets and maps. When A is a set then $\mathrm{Fin}(A)$ denotes the set of finite subsets of A. A *finite map* from a set A to a set B is a partial map with finite domain. The set of finite maps from A to B is denoted

$$A \stackrel{\text{fin}}{\to} B.$$

The domain and range of any function, f, is denoted $\mathrm{Dom}(f)$ and $\mathrm{Rng}(f)$, and $f \downarrow A$ means the restriction of f to A. When f and g are (perhaps finite) maps then $f \pm g$, called f *modified* by g, is the map with domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values

$$(f \pm g)(a) = \text{if } a \in \mathrm{Dom}(g) \text{ then } g(a) \text{ else } f(a).$$

The symbol $\pm$ is a reminder that $f \pm g$ may have a larger domain than f (hence the $+$) and also that some of the values of f may "disappear" because they are superseded by the values of g (hence the $-$).

When $\mathrm{Dom}(f) \cap \mathrm{Dom}(g) = \emptyset$ we write $f \mid g$ for $f \pm g$. We say that $f \mid g$ is *the simultaneous composition* of f and g. Note that for every $a \in \mathrm{Dom}(f \mid g)$ we have that either $(f \mid g)a = f(a)$ or $(f \mid g)a = g(a)$.

We say that g *extends* f, written $f \subseteq g$ if $\mathrm{Dom}(f) \subseteq \mathrm{Dom}(g)$ and for all x in the domain of f we have $f(x) = g(x)$.

Any $f \in A \stackrel{\text{fin}}{\to} B$ can be written in the form

$$\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}.$$

In particular, the empty map is written $\{\}$.

$$\begin{array}{rcl} b & \in & \text{BasVal} = \{\texttt{true}, \texttt{false}, 1, 2, \ldots\} \\ v & \in & \text{Val} = \text{BasVal} + \text{Clos} \\ [x, e, E] & \in & \text{Clos} = \text{Var} \times \text{Exp} \times \text{Env} \\ E & \in & \text{Env} = \text{Var} \xrightarrow{\text{fin}} \text{Val} \\ r & \in & \text{Results} = \text{Val} + \{\mathit{wrong}\} \end{array}$$

Figure 2.1: Semantic Objects (Dynamic Semantics)

2.2 Dynamic Semantics

The semantic objects for the dynamic semantics of Exp are defined in Figure 2.1. We assume a set, BasVal, of basic values. The basic values can be thought of either as syntactic objects (numerals, constants, constructors) or as the mathematical objects they denote (the integers, the booleans, and so on) but the other semantic objects should be seen as mathematical objects so that we have the usual operations (e.g., application) at our disposal. The object *wrong* is not a value; *wrong* is the result of a nonsense evaluation such as an attempt to apply a non-function to an argument.

The inference rules appear in Figure 2.2. The notation $\boxed{E \vdash e \longrightarrow r}$ means that they define a ternary relation between members of Env, Exp, and Results — this one is read *e evaluates to r in E*. Here, and everywhere else, the relation defined by the inference rules is the smallest relation closed under the rules. Variable names are used to avoid explicit injections and projections so that for instance rule 2.7 and rule 2.8 are mutually exclusive (since *wrong* is kept disjoint from the values). The **or** in rule 2.6 is used to collapse what is really two rules into one.

In general, every inference rule has the form

$$\frac{P_1 \cdots P_n}{C} \quad (n \geq 0)$$

and allows us from the *premises* $P_1, \ldots, P_n$ of the rule to *infer* the *conclusion*, C. Each premise is either a sequent, $A \vdash \mathit{phrase} \longrightarrow B$, or some *side condition* expressed using standard mathematical concepts. The conclusion is always a sequent.

For example, rule 2.6 can be read: "*if e_1 evaluates to a basic value or to wrong in E then the application $e_1\ e_2$ evaluates to wrong in E*".

$$\boxed{E \vdash e \longrightarrow r}$$

$$\frac{x \in \mathrm{Dom}\, E}{E \vdash x \longrightarrow E(x)} \tag{2.1}$$

$$\frac{x \notin \mathrm{Dom}\, E}{E \vdash x \longrightarrow \mathit{wrong}} \tag{2.2}$$

$$\frac{}{E \vdash \lambda x.e_1 \longrightarrow [x, e_1, E]} \tag{2.3}$$

$$\frac{\begin{array}{rcl} E & \vdash & e_1 \longrightarrow [x_0, e_0, E_0] \\ E & \vdash & e_2 \longrightarrow v_0 \\ E_0 \pm \{x_0 \mapsto v_0\} & \vdash & e_0 \longrightarrow r \end{array}}{E \vdash e_1\ e_2 \longrightarrow r} \tag{2.4}$$

$$\frac{E \vdash e_1 \longrightarrow [x_0, e_0, E_0] \quad E \vdash e_2 \longrightarrow \mathit{wrong}}{E \vdash e_1\ e_2 \longrightarrow \mathit{wrong}} \tag{2.5}$$

$$\frac{E \vdash e_1 \longrightarrow b \text{ or } \mathit{wrong}}{E \vdash e_1\ e_2 \longrightarrow \mathit{wrong}} \tag{2.6}$$

$$\frac{E \vdash e_1 \longrightarrow v_1 \quad E \pm \{x \mapsto v_1\} \vdash e_2 \longrightarrow r}{E \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \longrightarrow r} \tag{2.7}$$

$$\frac{E \vdash e_1 \longrightarrow \mathit{wrong}}{E \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \longrightarrow \mathit{wrong}} \tag{2.8}$$

Figure 2.2: Dynamic Semantics

By using the conclusion of one rule as the premise of another one can build complex evaluations out of simpler ones. More precisely, an evaluation is regarded as a special case of a proof tree in formal logic.

2.3 Static Semantics

We start with an infinite set, TyVar, of *type variables* and a set, TyCon, of nullary *type constructors.*

$$\begin{array}{rcl} \pi & \in & \mathrm{TyCon} = \{int, bool, \ldots\} \\ \alpha & \in & \mathrm{TyVar} = \{\mathrm{t}, \mathrm{t}', \mathrm{t}_1, \mathrm{t}_2, \ldots\} \end{array}$$

Then the set of *types*, Type, ranged over by τ and the set of *type schemes*, TypeScheme, ranged over by σ are defined by

$$\begin{array}{rcl} \tau & ::= & \pi \mid \alpha \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \tau \mid \forall\alpha.\sigma_1 \end{array}$$

The $\to$ is right associative. Note that types contain no quantifiers and that type schemes contain outermost quantification only. This is necessary to get a type checking algorithm based on first order term unification. A *type environment* is a finite map from program variables to type schemes:

$$TE \in \mathrm{TyEnv} = \mathrm{Var} \xrightarrow{\mathrm{fin}} \mathrm{TypeScheme}$$

A type scheme $\sigma = \forall\alpha_1. \ \ldots \ .\forall\alpha_n.\tau$ is written $\forall\alpha_1 \ldots \alpha_n.\tau$. We say that $\alpha_1, \ldots, \alpha_n$ are *bound* in σ and that a type variable is *free* in σ if it occurs in τ and is not bound. Moreover, we say that a type variable is free in TE if it is free in a type scheme in the range of TE.

The map tyvars : Type $\to$ Fin(TyVar) maps every type τ to set set of type variables that occur in τ. More generally, tyvars(σ) and tyvars(TE) means the set of type variables that occur *free* in σ and TE, respectively. Also, σ and TE are said to be *closed* if $\mathrm{tyvars}\,\sigma = \emptyset$ and $\mathrm{tyvars}\,TE = \emptyset$ and τ is said to be a *monotype* if $\mathrm{tyvars}(\tau) = \emptyset$.

A *total substitution* is a total map S : TyVar $\to$ Type. A *finite substitution* is a finite map from type variables to types. Every finite substitution can be extended to a total substitution by letting it be the identity outside its domain, and we shall often not bother to destinguish between the two. However, to deal

with renaming of bound variables, I like to think of the *region* of a substitution,

$$\mathrm{Reg}(S) = \bigcup_{\alpha \in \mathrm{Dom}(S)} \mathrm{tyvars}(S(\alpha))$$

and this is only relevant when S is a finite substitution.

By natural extension, substitutions can be applied to types. This gives composition of substitutions with identity ID. As usual, $(S_2 \circ S_1)\tau$ means $S_2(S_1\tau)$, which we often shall write simply $S_2 S_1 \tau$.

A substitution is *ground* if every type in its range is a monotype.

Substitution on type schemes and type environments is defined as follows.

Definition 2.1 *Let* $\sigma_1 = \forall\alpha_1 \ldots \alpha_n.\tau_1$ *and* $\sigma_2 = \forall\beta_1 \ldots \beta_m.\tau_2$ *be type schemes and* S *be a substitution. We write* $\sigma_1 \xrightarrow{S} \sigma_2$ *if*

1. $m = n$, *and* $\{\alpha_i \mapsto \beta_i \mid 1 \leq i \leq n\}$ *is a bijection, no* β_i *is in* $\mathrm{Reg}(S_0)$, *and*

2. $(S_0 \mid \{\alpha_i \mapsto \beta_i\})\tau_1 = \tau_2$

where $S_0 \stackrel{\text{def}}{=} S \downarrow \mathrm{tyvars}\,\sigma_1$. *Moreover, we write* $TE \xrightarrow{S} TE'$ *if* $\mathrm{Dom}\,TE = \mathrm{Dom}\,TE'$ *and for all* $x \in \mathrm{Dom}\,TE$, $TE(x) \xrightarrow{S} TE'(x)$.

We write $\sigma_1 \underset{\alpha}{=} \sigma_2$ as a shorthand for $\sigma_1 \xrightarrow{ID} \sigma_2$. Note that this is the familiar notion of α-conversion.

The operation of putting $\forall\alpha.$ in front of a type or a type scheme is called *generalization (on α)*, or *quantification (of α)*, or simply *binding (of α)*. Conversely, τ' is an *instance* of $\sigma = \forall\alpha_1 \ldots \alpha_n.\tau$, written

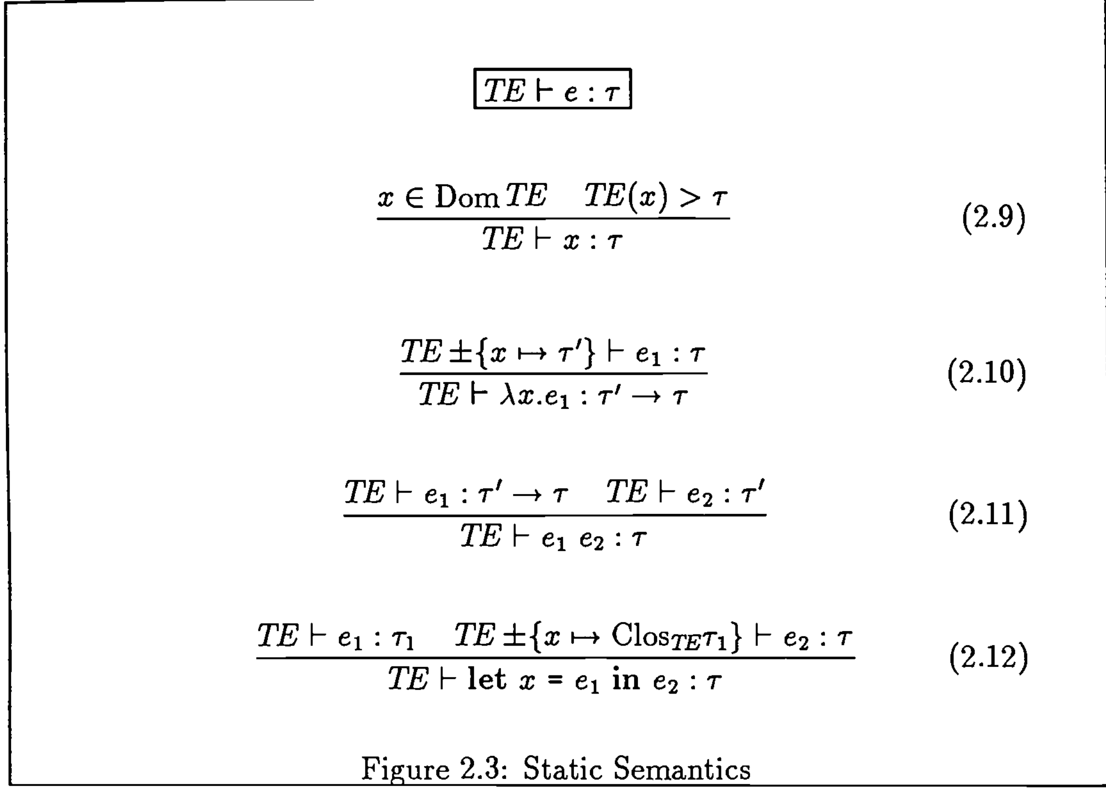
$$\sigma > \tau',$$

if there exists a finite substitution, S, with domain $\{\alpha_1, \ldots, \alpha_n\}$ and $S(\tau) = \tau'$. The operation of substituting types for *bound* variables is called *instantiation*. Instantiation is extended to type schemes as follows: σ_2 is an *instance* of σ_1, written $\sigma_1 \geq \sigma_2$, if for all types τ, if $\sigma_2 > \tau$ then $\sigma_1 > \tau$. Write $\sigma_2 = \forall\beta_1 \ldots \beta_m.\tau_2$. One can prove that $\sigma_1 \geq \sigma_2$ if and only if $\sigma_1 > \tau_2$ and no β_j is free in σ_1. (This, in turn, is equivalent to demanding that $\sigma_1 > \tau_2$ and $\mathrm{tyvars}(\sigma_1) \subseteq \mathrm{tyvars}(\sigma_2)$).

Finally,

$$\mathrm{Clos}_{TE}\tau$$

means $\forall\alpha_1 \ldots \alpha_n.\tau$, where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{tyvars}\,\tau \setminus \mathrm{tyvars}\,TE$.[1]



$$\boxed{TE \vdash e : \tau}$$

$$\frac{x \in \mathrm{Dom}\, TE \quad TE(x) > \tau}{TE \vdash x : \tau} \tag{2.9}$$

$$\frac{TE \pm \{x \mapsto \tau'\} \vdash e_1 : \tau}{TE \vdash \lambda x.e_1 : \tau' \to \tau} \tag{2.10}$$

$$\frac{TE \vdash e_1 : \tau' \to \tau \quad TE \vdash e_2 : \tau'}{TE \vdash e_1\ e_2 : \tau} \tag{2.11}$$

$$\frac{TE \vdash e_1 : \tau_1 \quad TE \pm \{x \mapsto \mathrm{Clos}_{TE}\tau_1\} \vdash e_2 : \tau}{TE \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau} \tag{2.12}$$

Figure 2.3: Static Semantics

This is all we need to give the static semantics, see Figure 2.3.[2]

The following examples illustrate the use of the system. Skip them if you know the type inference system.

Example 2.2 (Shows a simple monomorphic type inference). Consider the expression

$$((\lambda \mathrm{x}.\lambda \mathrm{y}.\mathrm{y}\ \mathrm{x})\mathrm{z})(\lambda \mathrm{x}.\mathrm{x}).$$

Assume $TE_0(\mathrm{z}) = int$, let $TE_1 = TE_0 \pm \{\mathrm{x} \mapsto int\}$ and $TE_2 = TE_1 \pm \{\mathrm{y} \mapsto int \to int\}$. We then have the inference

$$\cfrac{\cfrac{\cfrac{TE_2 \vdash \mathrm{y} : int \to int \quad TE_2 \vdash \mathrm{x} : int}{TE_2 \vdash \mathrm{y}\ \mathrm{x} : int}}{TE_1 \vdash \lambda \mathrm{y}.\mathrm{y}\ \mathrm{x} : (int \to int) \to int}}{TE_0 \vdash \lambda \mathrm{x}.\lambda \mathrm{y}.\mathrm{y}\ \mathrm{x} : int \to ((int \to int) \to int)}$$

[1]Throughout, the symbol $\setminus$ is used for set difference, i.e., $A \setminus B$ means $\{x \in A \mid x \notin B\}$.

[2]The reader familiar with the type inference system of [14] will note that our version does not have an instantiation and a generalization rule Instead instantiation is done precisely when variables are typed and generalization is done explicitly by the closure operation in the let rule Also note that the result of a typing is a type rather than a general type scheme. Although it is not trivial to prove it, the two systems admit exactly the same expressions. Our system has the advantage that whenever $TE \vdash e : \tau$, the form of e uniquely determines what rule was applied.

where the form of the expressions always tells us which rule is used. Thus

$$\frac{TE_0 \vdash \lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{y}\ \mathtt{x} : int \to (int \to int) \to int \quad TE_0 \vdash \mathtt{z} : int}{(\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{y}\ \mathtt{x})\mathtt{z} : (int \to int) \to int}. \tag{2.13}$$

We have

$$\frac{TE_1 \vdash \mathtt{x} : int}{TE_0 \vdash \lambda\mathtt{x}.\mathtt{x} : int \to int}$$

which with (2.13) gives

$$\frac{TE_0 \vdash (\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{y}\ \mathtt{x})\mathtt{z} : (int \to int) \to int \quad TE_0 \vdash \lambda\mathtt{x}.\mathtt{x} : int \to int}{TE_0 \vdash ((\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{y}\ \mathtt{x})\mathtt{z})(\lambda\mathtt{x}.\mathtt{x}) : int}$$

as we suspected. Notice that we had to "guess" the right types for the lambda bound variables x and y, but there exists an algorithm that can do this job (see Section 2.4).

Example 2.3 (Illustrates instantiation of type schemes). For the sake of this and following examples let us extend Type with lists:

$$\tau ::= \ldots \mid \tau_1\ list$$

and let us assume that

$$\begin{aligned} TE_0(\mathtt{nil}) &= \forall \mathrm{t}.\mathrm{t}\ list \\ TE_0(\mathtt{hd}) &= \forall \mathrm{t}.\mathrm{t}\ list \to \mathrm{t} \\ TE_0(\mathtt{tl}) &= \forall \mathrm{t}.\mathrm{t}\ list \to \mathrm{t}\ list \\ TE_0(\mathtt{cons}) &= \forall \mathrm{t}.\mathrm{t} \to \mathrm{t}\ list \to \mathrm{t}\ list \\ TE_0(\mathtt{rev}) &= \forall \mathrm{t}.\mathrm{t}\ list \to \mathrm{t}\ list\,. \end{aligned}$$

Let $TE_1 = TE_0 \pm \{\mathtt{x} \mapsto (int\ list)\ list\}$ and $TE_2 = TE_1 \pm \{\mathtt{g} \mapsto int \to int \to int\}$ and consider

$$e = \mathtt{g(hd(rev(hd\ x)))}.$$

Here hd is a polymorphic function used once to take the head of a list of integer lists and then once to get the head of an integer list. Thus in the following type inference, two different instances of the type scheme of hd are used:

$$\frac{TE_2 \vdash \mathtt{hd} : (int\ list)\ list \to int\ list \quad TE_2 \vdash \mathtt{x} : (int\ list)\ list}{TE_2 \vdash \mathtt{hd\ x} : int\ list} \tag{2.14}$$

$$\frac{TE_2 \vdash \mathtt{rev} : int\ list \to int\ list \quad TE_2 \vdash \mathtt{hd\ x} : int\ list \quad \text{by (2.14)}}{TE_2 \vdash \mathtt{rev(hd\ x)} : int\ list} \tag{2.15}$$

$$\frac{TE_2 \vdash \mathtt{hd} : \mathit{int\ list} \to \mathit{int} \quad TE_2 \vdash \mathtt{rev(hd\ x)} : \mathit{int\ list} \quad \text{by (2.15)}}{TE_2 \vdash \mathtt{hd(rev(hd\ x))} : \mathit{int}} \tag{2.16}$$

$$\frac{TE_2 \vdash \mathtt{g} : \mathit{int} \to \mathit{int} \to \mathit{int} \quad TE_2 \vdash \mathtt{hd(rev(hd\ x))} : \mathit{int} \quad \text{by (2.16)}}{TE_2 \vdash \mathtt{g(hd(rev(hd\ x)))} : \mathit{int} \to \mathit{int}} \tag{2.17}$$

Here it was the *instantiations* we had to guess — because of the type of g the instantiations we chose are the only ones that make the term well-typed.

Example 2.4 (Illustrates free type variables in the type environment). Take TE_0 as in the previous example, but let $TE_1 = TE_0 \pm \{\mathtt{x} \mapsto (\mathtt{t}'\,\mathit{list})\,\mathit{list}\}$ and $TE_2 = TE_1 \pm \{\mathtt{g} \mapsto \mathtt{t}' \to \mathtt{t}\}$. Thus t and t′ are free in TE_2. Now the inference for the same expression becomes

$$\frac{TE_2 \vdash \mathtt{hd} : (\mathtt{t}'\,\mathit{list})\,\mathit{list} \to \mathtt{t}'\,\mathit{list} \quad TE_2 \vdash \mathtt{x} : (\mathtt{t}'\,\mathit{list})\,\mathit{list}}{TE_2 \vdash \mathtt{hd\ x} : \mathtt{t}'\,\mathit{list}} \tag{2.18}$$

$$\frac{TE_2 \vdash \mathtt{rev} : \mathtt{t}'\,\mathit{list} \to \mathtt{t}'\,\mathit{list} \quad TE_2 \vdash \mathtt{hd\ x} : \mathtt{t}'\,\mathit{list} \quad \text{by (2.18)}}{TE_2 \vdash \mathtt{rev(hd\ x)} : \mathtt{t}'\,\mathit{list}} \tag{2.19}$$

$$\frac{TE_2 \vdash \mathtt{hd} : \mathtt{t}'\,\mathit{list} \to \mathtt{t}' \quad TE_2 \vdash \mathtt{rev(hd\ x)} : \mathtt{t}'\,\mathit{list} \quad \text{by (2.19)}}{TE_2 \vdash \mathtt{hd(rev(hd\ x))} : \mathtt{t}'} \tag{2.20}$$

$$\frac{TE_2 \vdash \mathtt{g} : \mathtt{t}' \to \mathtt{t} \quad TE_2 \vdash \mathtt{hd(rev(hd\ x))} : \mathtt{t}' \quad \text{by (2.20)}}{TE_2 \vdash \mathtt{g(hd(rev(hd\ x)))} : \mathtt{t}} \tag{2.21}$$

Note that if we substitute *int* for t′ and $\mathit{int} \to \mathit{int}$ for t throughout, the proof we get is precisely the proof from Example 2.3.

So we notice that a type variable free in the type environment behaves like a type constant different from all other type constants (*int*, *bool*, etc).

Note that it is the rule for lambda abstraction, and only this rule, by which new type variables can become free in the type environment.

Example 2.5 (Illustrates the let rule, in particular the role of type variables free in the type environment when types are generalized). We continue the previous example. From (2.21) we get

$$TE_1 \vdash \lambda\mathtt{g}.\mathtt{g(hd(rev(hd\ x)))} : (\mathtt{t}' \to \mathtt{t}) \to \mathtt{t}.$$

Note that t′ is free in TE_1 and in the resulting type, whereas t is not free in TE_1. Now consider the expression (where `first` has type $\forall \mathtt{t}_1\mathtt{t}_2.\mathtt{t}_1 \to \mathtt{t}_2 \to \mathtt{t}_1$)

let f= λg.g(hd(rev(hd x)))
in

```
··· (f first) 7···
··· (f first) true···
··· (f cons) nil···
```

in TE_1. The let rule (rule 2.12) requires that the body be checked in the type environment

$$TE_1' = TE_1 \pm \{\mathtt{f} : \forall \mathrm{t}.(\mathrm{t}' \to \mathrm{t}) \to \mathrm{t}\}$$

since $\mathrm{Clos}_{TE_1}((\mathrm{t}' \to \mathrm{t}) \to \mathrm{t}) = \forall \mathrm{t}.(\mathrm{t}' \to \mathrm{t}) \to \mathrm{t}$. That we must not generalize on t' is not too surprising, if we think of t' as a type constant (c.f. Example 2.4).

In the body of the above expression we use the following instantiations for `f`:

$$\begin{array}{lcl} \forall \mathrm{t}.(\mathrm{t}' \to \mathrm{t}) \to \mathrm{t} & > & (\mathrm{t}' \to (\mathit{int} \to \mathrm{t}')) \to (\mathit{int} \to \mathrm{t}') \\ \forall \mathrm{t}.(\mathrm{t}' \to \mathrm{t}) \to \mathrm{t} & > & (\mathrm{t}' \to (\mathit{bool} \to \mathrm{t}')) \to (\mathit{bool} \to \mathrm{t}') \\ \forall \mathrm{t}.(\mathrm{t}' \to \mathrm{t}) \to \mathrm{t} & > & (\mathrm{t}' \to (\mathrm{t}'\,\mathit{list} \to \mathrm{t}'\,\mathit{list})) \to (\mathrm{t}'\,\mathit{list} \to \mathrm{t}'\,\mathit{list}) \end{array}$$

The following two lemmas are essential for the later proofs:

Lemma 2.6 *For any S, if $\sigma_1 > \tau'$ and $\sigma_1 \xrightarrow{S} \sigma_2$ then $\sigma_2 > S\,\tau'$.*

Proof. Let $\sigma_1 = \forall \alpha_1 \ldots \alpha_n.\tau$ and let I be the instantiation substitution,

$$I = \{\alpha_i \mapsto \tau_i \mid 1 \leq i \leq n\}$$

with $I\,\tau = \tau'$. Now σ_2 is of the form $\forall \beta_1 \ldots \beta_n.(S_0 \,|\{\alpha_i \mapsto \beta_i\})\tau$ where $\{\alpha_i \mapsto \beta_i\}$ is a bijection and no β_i is in $\mathrm{Reg}(S_0)$, where $S_0 = S \downarrow \mathrm{tyvars}\,\sigma_1$. Let $J = \{\beta_i \mapsto S\,\tau_i\}$ and let us first show that

$$J((S_0 \,|\{\alpha_i \mapsto \beta_i\})\tau) = S(I\,\tau). \tag{2.22}$$

This we show by proving that for each α occurring in τ,

$$J((S_0 \,|\{\alpha_i \mapsto \beta_i\})\alpha) = S(I\,\alpha). \tag{2.23}$$

If α is bound in σ_1, i.e. $\alpha = \alpha_j$, say, then

$$J((S_0 \,|\{\alpha_i \mapsto \beta_i\})\alpha_j) = J\,\beta_j = S\,\tau_j = S(I\,\alpha_j).$$

Otherwise α is free in σ_1, so

$$J((S_0 \,|\{\alpha_\imath \mapsto \beta_\imath\})\alpha) = J(S_0\,\alpha) = J(S\,\alpha) = S\,\alpha,$$

since $\text{Reg}\,S_0 \cap \{\beta_1, \ldots, \beta_n\} = \emptyset$. But $S\,\alpha = S(I\,\alpha)$, since α is not in $\{\alpha_i, \ldots, \alpha_n\}$, showing that (2.23) holds in this case as well. Thus we have (2.22) and since $I\,\tau = \tau'$, this gives the desired $\sigma_2 > S\,\tau'$ ■

Lemma 2.7 *If $TE \vdash e : \tau$ and $TE \xrightarrow{S} TE'$ then $TE' \vdash e : S\,\tau$.*

We have already seen one application of this lemma, namely that the proof in Example 2.3 could be obtained from the proof in Example 2.4.

Proof. By structural induction on e. The case where e is a variable follows from Lemma 2.6. Of the remaining cases, only the case for let expressions is interesting.

$\boxed{e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2}$ Here $TE \vdash e : \tau$ must have been inferred by application of

$$\frac{TE \vdash e_1 : \tau_1 \quad TE \pm \{x \mapsto \text{Clos}_{TE}\tau_1\} \vdash e_2 : \tau}{TE \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau} \tag{2.24}$$

Let $\{\alpha_1, \ldots, \alpha_n\}$ be $\text{tyvars}\,\tau_1 \setminus \text{tyvars}\,TE$. Let $\{\beta_1, \ldots, \beta_n\}$ be such that $\{\alpha_i \mapsto \beta_\imath\}$ is a bijection and no $\beta_\imath$ is in $\text{Reg}(S_0)$, where $S_0 = S \downarrow \text{tyvars}\,TE$. Let $S' = S_0 \,|\{\alpha_\imath \mapsto \beta_\imath\}$. Then

$$\begin{aligned} \text{Clos}_{TE}\tau_1 &= \forall\alpha_1 \ldots \alpha_n.\tau_1 \\ &\xrightarrow{S} \forall\beta_1 \ldots \beta_n.\,S'\,\tau_1 \end{aligned} \tag{2.25}$$

No $\beta_\imath$ is free in TE' (since $TE \xrightarrow{S} TE'$ and $\beta_\imath \neq \text{Reg}(S_0)$). Moreover, any type variable that occurs in $S'\,\tau_1$ and is not a $\beta_\imath$ must be free in TE' (since every type variable free in $\forall\alpha_1 \ldots \alpha_n.\tau_1$ is free in TE and $TE \xrightarrow{S} TE'$). Therefore, by (2.25), we have

$$\text{Clos}_{TE}\tau_1 \xrightarrow{S} \text{Clos}_{TE'}\,S'\,\tau_1. \tag{2.26}$$

Since $TE \xrightarrow{S} TE'$ and no $\alpha_\imath$ is free in TE we have $TE \xrightarrow{S'} TE'$. Thus by induction on e_1 and the first premise of (2.24) we have

$$TE' \vdash e_1 : S'\,\tau_1. \tag{2.27}$$

Moreover, we have

$$TE \pm \{x \mapsto \text{Clos}_{TE}\tau_1\} \xrightarrow{S} TE' \pm \{x \mapsto \text{Clos}_{TE'}\,S'\,\tau_1\}$$

by (2.26). Thus by induction on e_2 and the second premise of (2.24) we get

$$TE' \pm \{x \mapsto \mathrm{Clos}_{TE'}\, S'\tau_1\} \vdash e_2 : S\,\tau$$

which with (2.27) gives the desired $TE' \vdash e : S\,\tau$ by the let rule. ∎

2.4 Principal Types

The type inference rules are non-deterministic; for instance the expression $\lambda \mathbf{x}.\mathbf{x}$ can get type $\mathrm{t} \to \mathrm{t}$, $int \to int$, and $bool \to bool$. However, among the types that can be inferred some are principal in that all the others can be obtained from them:

Definition 2.8 *A type τ is* principal *for e in TE if $TE \vdash e : \tau$ and moreover, whenever $TE \vdash e : \tau'$ then $\mathrm{Clos}_{TE}\tau > \tau'$.*

It can be proved that if an expression has a type in a type environment then it has a principal type in that environment. More precisely, Milner devised a *type checker*, i.e., an algorithm which given TE and e determines whether there exists a τ such that $TE \vdash e : \tau$. We repeat it in Figure 2.4. The algorithm, W, uses Robinson's unification algorihm [37] to unify types. As in [13] it can be proved that W is "sound" in the following sense:

Theorem 2.9 (Soundness of W) *If $(S, \tau) = W(TE, e)$ succeeds and $TE \xrightarrow{S} TE'$ then $TE' \vdash e : \tau$.*

Moreover, W is "complete" in the following sense:

Theorem 2.10 (Completeness of W) *If $TE \xrightarrow{S_1} TE_1$ and $TE_1 \vdash e : \tau_1$ then*

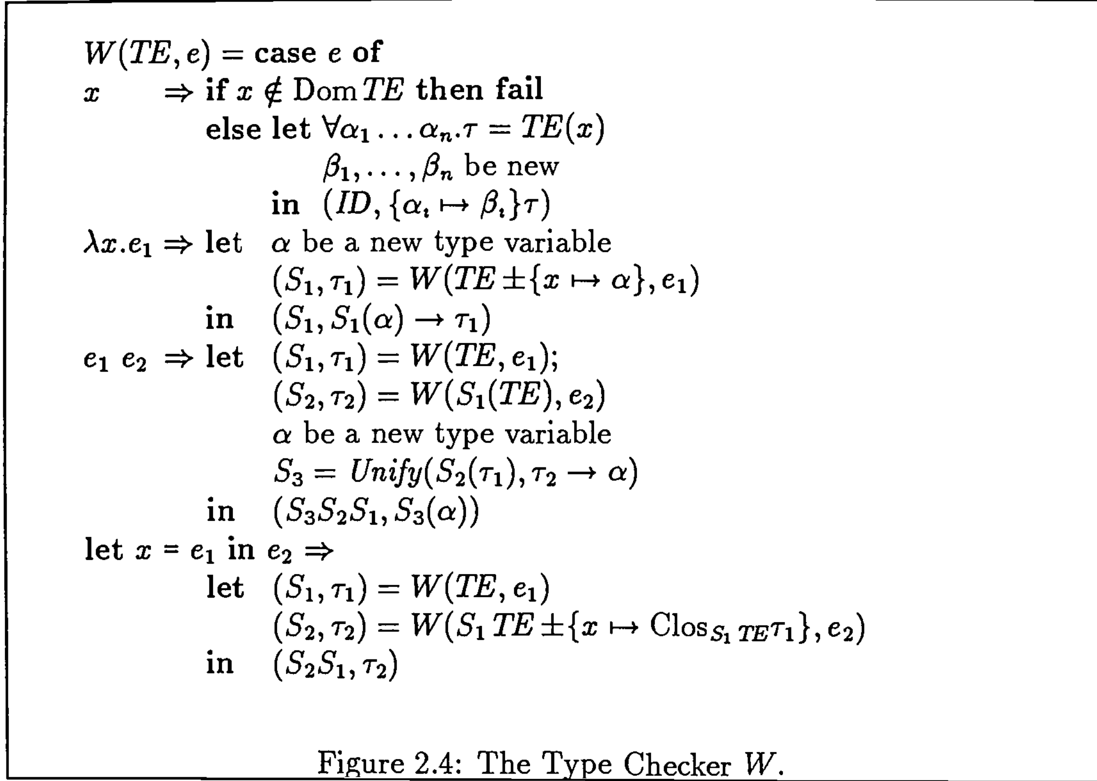
$$(S_0, \tau_0) = W(TE, e)$$

succeeds and there exists a substitution S_0' and a type environment TE_0 such that

$$TE \xrightarrow{S_0} TE_0 \text{ and } TE \xrightarrow{S_0' \circ S_0} TE_1 \text{ and } S_0'\,\mathrm{Clos}_{TE_0}\tau_0 > \tau_1.$$

Moreover, if $W(TE, e)$ does not succeed then it stops with fail.

The proofs of Theorems 2.9 and 2.10 are by structural induction on e and use Lemma 2.7.



$W(TE, e) =$ **case** e **of**
$x \Rightarrow$ **if** $x \notin \mathrm{Dom}\, TE$ **then fail**
  else let $\forall\alpha_1 \ldots \alpha_n.\tau = TE(x)$
    $\beta_1, \ldots, \beta_n$ be new
  in $(ID, \{\alpha_i \mapsto \beta_i\}\tau)$
$\lambda x.e_1 \Rightarrow$ **let** α be a new type variable
    $(S_1, \tau_1) = W(TE \pm \{x \mapsto \alpha\}, e_1)$
  in $(S_1, S_1(\alpha) \rightarrow \tau_1)$
$e_1\ e_2 \Rightarrow$ **let** $(S_1, \tau_1) = W(TE, e_1)$;
    $(S_2, \tau_2) = W(S_1(TE), e_2)$
    α be a new type variable
    $S_3 = \mathit{Unify}(S_2(\tau_1), \tau_2 \rightarrow \alpha)$
  in $(S_3 S_2 S_1, S_3(\alpha))$
let $x = e_1$ **in** $e_2 \Rightarrow$
  let $(S_1, \tau_1) = W(TE, e_1)$
    $(S_2, \tau_2) = W(S_1 TE \pm \{x \mapsto \mathrm{Clos}_{S_1 TE}\tau_1\}, e_2)$
  in $(S_2 S_1, \tau_2)$

Figure 2.4: The Type Checker W.

2.5 The Consistency Result

We shall now show the consistency of the dynamic and the static semantics. The result will imply that a well-typed program cannot evaluate to *wrong*.

To this end we define a relation between the objects of the dynamic and the static semantics.

Assume a basic relation between basic values and type constants

$$R_{\mathrm{Bas}} \subseteq \mathrm{BasVal} \times \mathrm{TyCon}$$

relating 5 to *int*, *true* to *bool*, and so on. In order to ensure that the definitions really define something, we define the relation between values, v, and types τ in stages. We start with monotypes, μ:

Definition 2.11 *We say that v has monotype μ, written $\models v : \mu$ if*

either $v = b$ *and* $(b, \mu) \in R_{\mathrm{Bas}}$,
or $v = [x, e, E]$ *and* $\mu = \mu_1 \rightarrow \mu_2$, *for some* μ_1, μ_2, *and for all* v_1, r
if $\models v_1 : \mu_1$ *and* $E \pm \{x \mapsto v_1\} \vdash e \longrightarrow r$ *then* $r \neq \mathit{wrong}$ *and* $\models r : \mu_2$.

This is well-defined on the structure of types. The definition is extended to types with free type variables, type schemes, and type environments as follows (where TE° ranges over closed type environments)

Definition 2.12 *We define:*
$\models v : \tau$ *if for all total, ground* S *we have* $\models v : S\,\tau$;
$\models v : \forall\alpha_1 \ldots \alpha_n.\tau$ *if* $\models v : \tau$;
$\models E : TE^\circ$ *if* $\mathrm{Dom}\,E = \mathrm{Dom}\,TE^\circ$ *and* $\models E(x) : TE^\circ(x)$ *for all* $x \in \mathrm{Dom}\,E$;
$\models E : TE$ *if* $\models E : TE^\circ$ *whenever* S *is total and ground and* $TE \xrightarrow{S} TE^\circ$.

We expect the static semantics to be consistent with the dynamic semantics in the following sense: no matter which type τ we can infer for e using the static semantics, and no matter which result, r, we get by dynamic evaluation of e, r is not *wrong*, in fact r is a value of type τ. If we can prove this then, in particular, if e can be typed then the dynamic evaluation of e will never lead to an attempt to apply a non-function to an argument or to looking in vain for a variable in the environment.

Assuming for the moment that e contains no free variables this can be formulated very easily as follows using the relation $\models$ defined above:

$$\text{if} \vdash e : \tau \text{ and} \vdash e \longrightarrow r \text{ then } r \neq \mathit{wrong} \text{ and} \models r : \tau.$$

To prove this, we need to consider open expression as well. When considering the more general $TE \vdash e : \tau$ we are only interested in dynamic environments whose values have the types assumed in TE. Thus we arrive at the main theorem which states the "soundness" of the polymorphic type discipline in just one line:

Theorem 2.13 (Consistency of Static and Dynamic Semantics)
If $\models E : TE$ *and* $TE \vdash e : \tau$ *and* $E \vdash e \longrightarrow r$ *then* $r \neq \mathit{wrong}$ *and* $\models r : \tau$.

Proof. By structural induction on e.

$\boxed{e = x}$ Here $x \in \mathrm{Dom}\,TE$ and $TE(x) > \tau$. Since $\models E : TE$ we have $x \in \mathrm{Dom}\,E$. Thus $r \neq \mathit{wrong}$, in fact $r = E(x)$. As $\models E(x) : TE(x)$ and $TE(x) > \tau$ we have $\models E(x) : \tau$ using Definition 2.12.

$\boxed{e = \lambda x.e_1}$ Here the type inference must have been of the form

$$\frac{TE \pm \{x \mapsto \tau'\} \vdash e_1 : \tau}{TE \vdash \lambda x.e_1 : \tau' \to \tau} \tag{2.28}$$

and the evaluation must have been

$$\overline{E \vdash \lambda x.e_1 \longrightarrow [x, e_1, E]}.$$

Thus $r = [x, e_1, E]$ which clearly is different from *wrong*. To prove $\models r : \tau' \to \tau$, let S be any total, ground substitution. There is a TE° such that $TE \xrightarrow{S} TE^\circ$. Thus $TE \pm \{x \mapsto \tau'\} \xrightarrow{S} TE^\circ \pm \{x \mapsto S\,\tau'\}$. This, with the premise of (2.28) gives

$$TE^\circ \pm \{x \mapsto S\,\tau'\} \vdash e_1 : S\,\tau \tag{2.29}$$

by Lemma 2.7. Now let v' be such that $\models v' : S\,\tau'$ and assume

$$E \pm \{x \mapsto v'\} \vdash e_1 \longrightarrow r_1. \tag{2.30}$$

Since $\models E : TE$ we have $\models E : TE^\circ$ and therefore

$$\models E \pm \{x \mapsto v'\} : TE^\circ \pm \{x \mapsto S\,\tau'\} \tag{2.31}$$

By induction on e_1, using (2.29), (2.30), and (2.31) we have $r_1 \neq \mathit{wrong}$ and $\models r_1 : S\,\tau$.

This proves $\models [x, e_1, E] : S\,\tau' \to S\,\tau$, i.e., $\models r : S\,\tau' \to S\,\tau$.

Since this holds for any S, we have proved $\models r : \tau' \to \tau$ as desired.

$\boxed{e = e_1\ e_2}$ Here the type inference must have been of the form

$$\frac{TE \vdash e_1 : \tau' \to \tau \quad TE \vdash e_2 : \tau'}{TE \vdash e_1\ e_2 : \tau}. \tag{2.32}$$

Let S be any total, ground substitution. There exists a TE° such that $TE \xrightarrow{S} TE^\circ$. By Lemma 2.7 on the premises of (2.32) we have

$$TE^\circ \vdash e_1 : S\,\tau' \to S\,\tau \tag{2.33}$$

$$TE^\circ \vdash e_2 : S\,\tau'. \tag{2.34}$$

Since $\models E : TE$ we have $\models E : TE^\circ$. By assumption, $E \vdash e_1\ e_2 \longrightarrow r$. Looking at the evaluation rules we see that there must exist an r_1 such that $E \vdash e_1 \longrightarrow r_1$. Thus, by induction on e_1, using (2.33), we get

$$r_1 \neq \mathit{wrong} \text{ and } \models r_1 : S\,\tau' \to S\,\tau.$$

By Definition 2.11, r_1 must be a closure, $[x_0, e_0, E_0]$, say. Thus $E \vdash e_1\ e_2 \longrightarrow r$ was not by rule (2.6) i.e., it must have been by rule (2.4) or (2.5). Therefore, there exists an r_2 so that $E \vdash e_2 \longrightarrow r_2$.

Using induction on e_2 together with (2.34) we get that

$$r_2 \neq \mathit{wrong} \text{ and } \models r_2 : S\,\tau'.$$

Thus r_2 is a value, v_0, say. In particular, it must have been rule (2.4) that was used.

Hence $E_0 \pm \{x_0 \mapsto v_0\} \vdash e_0 \longrightarrow r$. Since $\models [x_0, e_0, E_0] : S\,\tau' \to S\,\tau$ and $\models v_0 : S\,\tau'$, we must therefore have that $r \neq \mathit{wrong}$ and $\models r : S\,\tau$.

Since this holds for any S, we have proved $\models r : \tau$.

$\boxed{e = \textbf{let } x = e_1 \textbf{ in } e_2}$ The type inference must have been of the form

$$\frac{TE \vdash e_1 : \tau_1 \quad TE \pm \{x \mapsto \mathrm{Clos}_{TE}\tau_1\} \vdash e_2 : \tau}{TE \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau}. \tag{2.35}$$

Let S be any total, ground substitution. Let $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{tyvars}\,\tau_1 \setminus \mathrm{tyvars}\,TE$, let $S_1 = S \downarrow \mathrm{tyvars}\,TE$ and let $S_0 = S \downarrow \mathrm{tyvars}\,\mathrm{Clos}_{TE}\tau_1$. Let $\{\beta_1, \ldots, \beta_n\}$ be such that $\{\alpha_i \mapsto \beta_i\}$ is a bijection and no β_i is in $\mathrm{Reg}\,S_1$. Then no β_i is in $\mathrm{Reg}\,S_0$, so

$$\begin{aligned} \mathrm{Clos}_{TE}\tau_1 &= \forall\alpha_1 \ldots \alpha_n.\tau \\ &\xrightarrow{S} \forall\beta_1 \ldots \beta_n.(S_0 \,|\{\alpha_i \mapsto \beta_i\})\tau_1 \\ &= \forall\beta_1 \ldots \beta_n.(S_1 \,|\{\alpha_i \mapsto \beta_i\})\tau_1 \\ &= \forall\beta_1 \ldots \beta_n.\, S'\,\tau_1 \end{aligned} \tag{2.36}$$

where $S' = S_1 \,|\{\alpha_i \mapsto \beta_i\}$.

There exists a TE° such that $TE \xrightarrow{S} TE^\circ$. Surely $\forall\beta_1 \ldots \beta_n.\, S'\,\tau_1 = \mathrm{Clos}_{TE^\circ}\, S'\,\tau_1$ so by (2.36) we have

$$\mathrm{Clos}_{TE}\tau_1 \xrightarrow{S} \mathrm{Clos}_{TE^\circ}\, S'\,\tau_1. \tag{2.37}$$

Since $TE \xrightarrow{S} TE^\circ$ and no α_i is free in TE we have $TE \xrightarrow{S'} TE^\circ$. This with Lemma 2.7 on the first premise of (2.35) gives

$$TE^\circ \vdash e_1 : S'\,\tau_1. \tag{2.38}$$

Since $TE \xrightarrow{S} TE^\circ$ and (2.37) we have

$$TE \pm \{x \mapsto \mathrm{Clos}_{TE}\tau_1\} \xrightarrow{S} TE^\circ \pm \{x \mapsto \mathrm{Clos}_{TE^\circ}\, S'\,\tau_1\}.$$

This with Lemma 2.7 on the second premise of (2.35) gives

$$TE^{\circ} \pm \{x \mapsto \mathrm{Clos}_{TE^{\circ}}\, S'\,\tau_1\} \vdash e_2 : S\,\tau. \tag{2.39}$$

Inspecting the evaluation rules we see that there must exist an r_1 such that $E \vdash e_1 \longrightarrow r_1$. Since $\models E : TE$ we have $\models E : TE^{\circ}$ so by induction on e_1, using (2.38), we get

$$r_1 \neq \mathit{wrong} \text{ and } \models r_1 : S'\,\tau_1.$$

Thus r_1 is a value, v_1, say. Then it must have been rule 2.7 that was applied so $E \pm \{x \mapsto v_1\} \vdash e_2 \longrightarrow r$.

Since $\models v_1 : S'\,\tau_1$ we have $\models v_1 : \mathrm{Clos}_{TE^{\circ}}\, S'\,\tau_1$ by Definition 2.12. Thus

$$\models E \pm \{x \mapsto v_1\} : TE^{\circ} \pm \{x \mapsto \mathrm{Clos}_{TE^{\circ}}\, S'\,\tau_1\}.$$

Hence by induction on e_2, using (2.39), we get $r \neq \mathit{wrong}$ and $\models r : S\,\tau$.

Since we can do this for any S, we have proved $r \neq \mathit{wrong}$ and $\models r : \tau$. ∎

Part II

An Imperative Language

Chapter 3

Formulation of the Problem

The problem that is the topic of this part is: how can we extend the polymorphic type discipline to a language with imperative features?

The reason for studying this problem is that it has proved itself to be an obstacle (if not *the* obstacle) to the harmonic integration of imperative language features (e.g., assignment, pointers and arrays) as we know them from languages like ALGOL and PASCAL and functional language features (e.g., functions as first class objects, polymorphism) as we know them from ML. Some people take such obstacles as evidence that imperative and functional languages should be kept apart.[1] I shall first argue that we should try to take the best of both worlds and then give a concrete piece of evidence to the feasibility of the task, namely a new type discipline that integrates polymorphism with imperative features. The discipline seems to be sufficiently simple that programmers can use it with confidence — the point being that a type inference system should never be so advanced that programmers only have a vague feeling for which programs are well-typed. Moreover, the discipline seems to admit enough programs to be of practical use.

What is to gain for functional languages? Firstly, there are good algorithms and techniques that are based on imperative features (sorting, graph algorithms, dynamic programming, table driven algorithms, etc.) and it is desirable that these can be used essentially as they are. Secondly, it is sometimes possible to obtain greater execution speed with the use of imperative features without sacrificing the clarity of the algorithm. One example is the type checker discussed in the last chapter; there it was presented as a functional program computing substitutions, but a much more efficient and equally natural type checker is obtained

[1] Let alone those who believe that one kind of language is superior to the other in all respects.

by actually doing the substitutions by assignment statements.

What is to gain for imperative languages? Primarily a type system with the advantages of polymorphism. For example, we shall admit

```
fun fast_reverse(l)=
let left = ref l;
    right = ref nil
in  while !left <> nil do
    begin
      right := hd(!left) :: (!right);
      left := tl(!left)
    end;
    !right
end
```

where the evaluation of `ref` e dynamically creates a new reference to the value of e and `!` stands for dereferencing. Notice that none of the variables have had to be given explicit types. More importantly, this is an example of a polymorphic function that uses imperative features; intuitively, the most general type of `fast_reverse` is $\forall t.t\ list \to t\ list$.

The first ML [15] had typing rules for so-called `letref` bound variables. (Like a PASCAL variable, a `letref` bound variable can be updated with an assignment operation but, unlike a PASCAL variable, a `letref` bound variable is bound to a permanent address in the store). The rules admitted some polymorphic fuctions that used local `letref` bound variables.

Damas [13] went further in allowing references (or pointers, as other people call them) as first order values and he gave an impressive extension of the polymorphic type discipline to cope with this situation.

Yet, many more have thought about references and polymorphism without publishing anything. Many, including the author, know all too well how easy it is to guess some plausible typing rules that later turn out to be plain wrong.

Guessing and verifying are inseparable parts of developing a new theory. None is prior to the other, neither in time nor in importance. I believe the reason why the guessing has been so hard is precisely that the verifying has been hard. The soundness of the LCF rules was stated informally, but no proof was published.[2] In his thesis Damas did give a soundness proof for his system; it was based on

[2]There is some uncertainty as to whether a proof was carried out

denotational semantics and involved a very difficult domain construction. More seriously, although his soundness theorem is not known to be false, there appears to be a fatal mistake in the soundness proof.[3]

The good news is that we now do have a tractable way of proving soundness theorems. The basic idea is to prove consistency theorems similar to that of Chapter 2, using a simple and very general proof technique concerning maximal fixpoints of monotonic operators. Credit for this should go to Robin Milner, who suggested this technique at a point in time where I was stuck because I worked with minimal fixpoints.

Thanks to this technique we can present two results. Firstly, we can actually pinpoint the problem in so far as we can explain precisely why the naive extension of the polymorphic type discipline is unsound. Secondly, we can present a new solution to the problem and prove it correct. The remainder of the present chapter is devoted to presenting the first result; the type discipline is presented in Chapter 4 and its soundness proved in Chapter 5. Finally Chapter 6 contains a comparison with Damas' work.

3.1 A simple language

Let us use exactly the same syntax as before, that is we assume a set Var of variables, ranged over by x, and form the set Exp, of expressions, ranged over by e, by

$e ::=$	x	variable
$\mid$	$\lambda x.e_1$	lambda abstraction
$\mid$	$e_1\ e_2$	application
$\mid$	let x = e_1 in e_2	let expression

We assume values *ref*, *asg*, and *deref* bound to the variables `ref`, `:=`, and `!`, respectively. We use the infix form e_1:=e_2 to mean (:= e_1) e_2. We introduce a basic value *done* which is the value of expressions that are not meant to produce an ordinary value (an example is e_1:= e_2). The objects in the dynamic semantics are defined in Figure 3.1 and the rules appear in Figure 3.2. To reduce the number of rules, and hence the length of our inductive proofs, we have changed the dynamic semantics from Chapter 2 by removing *wrong* from the semantics.

[3]In the proof of Proposition 4, case INST, page 111, the requirements for using the induction hypothesis are not met; I do not see how to get around this problem.

$$
\begin{array}{rcl}
b & \in & \mathrm{BasVal} = \mathit{done}, \mathit{true}, \mathit{false}, 1, 2, \ldots \\
v & \in & \mathrm{Val} = \mathrm{BasVal} + \mathrm{Clos} + \{\mathit{asg}, \mathit{ref}, \mathit{deref}\} + \mathrm{Addr} \\
[x, e, E] & \in & \mathrm{Clos} = \mathrm{Var} \times \mathrm{Exp} \times \mathrm{Env} \\
s & \in & \mathrm{Store} = \mathrm{Addr} \xrightarrow{\mathrm{fin}} \mathrm{Val} \\
E & \in & \mathrm{Env} = \mathrm{Var} \xrightarrow{\mathrm{fin}} \mathrm{Val} \\
a & \in & \mathrm{Addr}
\end{array}
$$

Figure 3.1: Objects in the Dynamic Semantics

$$\frac{x \in \mathrm{Dom}\, E}{s, E \vdash x \longrightarrow E(x), s} \tag{3.1}$$

$$\frac{}{s, E \vdash \lambda x.e_1 \longrightarrow [x, e_1, E], s} \tag{3.2}$$

$$\frac{\begin{array}{l} s, E \vdash e_1 \longrightarrow [x_0, e_0, E_0], s_1 \\ s_1, E \vdash e_2 \longrightarrow v_2, s_2 \\ s_2, E_0 \pm \{x_0 \mapsto v_2\} \vdash e_0 \longrightarrow v, s' \end{array}}{s, E \vdash e_1\ e_2 \longrightarrow v, s'} \tag{3.3}$$

$$\frac{\begin{array}{l} s, E \vdash e_1 \longrightarrow \mathit{asg}, s_1 \\ s_1, E \vdash e_2 \longrightarrow a, s_2 \\ s_2, E \vdash e_3 \longrightarrow v_3, s_3 \end{array}}{s, E \vdash (e_1\ e_2)\ e_3 \longrightarrow \mathit{done}, s_3 \pm \{a \mapsto v_3\}} \tag{3.4}$$

$$\frac{s, E \vdash e_1 \longrightarrow \mathit{ref}, s_1 \quad s_1, E \vdash e_2 \longrightarrow v_2, s_2 \quad a \notin \mathrm{Dom}\, s_2}{s, E \vdash e_1\ e_2 \longrightarrow a, s_2 \pm \{a \mapsto v_2\}} \tag{3.5}$$

$$\frac{s, E \vdash e_1 \longrightarrow \mathit{deref}, s_1 \quad s_1, E \vdash e_2 \longrightarrow a, s' \quad s'(a) = v}{s, E \vdash e_1\ e_2 \longrightarrow v, s'} \tag{3.6}$$

$$\frac{s, E \vdash e_1 \longrightarrow v_1, s_1 \quad s_1, E \pm \{x \mapsto v_1\} \vdash e_2 \longrightarrow v, s'}{s, E \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \longrightarrow v, s'} \tag{3.7}$$

Figure 3.2: Dynamic Semantics

$$\frac{x \in \mathrm{Dom}\, TE \quad TE(x) > \tau}{TE \vdash x : \tau}$$

$$\frac{TE \pm \{x \mapsto \tau'\} \vdash e_1 : \tau}{TE \vdash \lambda x.e_1 : \tau' \to \tau}$$

$$\frac{TE \vdash e_1 : \tau' \to \tau \quad TE \vdash e_2 : \tau'}{TE \vdash e_1\ e_2 : \tau}$$

$$\frac{TE \vdash e_1 : \tau_1 \quad TE \pm \{x \mapsto \mathrm{Clos}_{TE}\tau_1\} \vdash e_2 : \tau}{TE \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

Figure 3.3: The Applicative Type Inference System (Repeated)

3.2 The Problem is Generalization

The naive generalization of the type discipline from Chapter 2 is to include types of the form

$$\tau \ ::= \ stm \ \mid \ \tau\ ref$$

among types and keep the inference system as it is (see Figure 3.3), assuming that the initial environments are

$$\begin{array}{rclrcl} TE_0(\texttt{ref}) & = & \forall t.t \to t\ ref & E_0(\texttt{ref}) & = & ref \\ TE_0(\texttt{:=}) & = & \forall t.t\ ref \to t \to stm & E_0(\texttt{:=}) & = & asg \\ TE_0(\texttt{!}) & = & \forall t.t\ ref \to t & E_0(\texttt{!}) & = & deref. \end{array}$$

However, the following example shows that with this system one can type nonsensical programs.

Example 3.1 Consider the following nonsensical program

```
let r = ref(λx.x)
in  (r:= λx.x+1; (!r)true)
```

where ; stands for sequential evaluation (the dynamic and static inference rules for ; will be given later). Also, `x+1` is infix notation for `(+x)1` and the type of `+` is $int \to int \to int$. This program can be typed; the expression `ref(λx.x)`

can get type $(\mathrm{t} \to \mathrm{t})\,\mathit{ref}$ and the body of the let expression is typable under the assumption

$$\{\mathrm{r} \mapsto \forall \mathrm{t}.((\mathrm{t} \to \mathrm{t})\,\mathit{ref})\} \tag{3.8}$$

using the instantiations

$$\forall \mathrm{t}.((\mathrm{t} \to \mathrm{t})\,\mathit{ref}) \;\; > \;\; (\mathit{int} \to \mathit{int})\,\mathit{ref}$$

and

$$\forall \mathrm{t}.((\mathrm{t} \to \mathrm{t})\,\mathit{ref}) \;\; > \;\; (\mathit{bool} \to \mathit{bool})\,\mathit{ref}$$

for the two occurrences of r. ■

Now let us formulate a consistency result and see the point at which the proof breaks down. Our starting point is the consistency result of Section 2.5, Theorem 2.13 which read[4]

Theorem 3.2 *If* $\models E : TE$ *and* $TE \vdash e : \tau$ *and* $E \vdash e \longrightarrow v$ *then* $\models v : \tau$.

In the presence of a store the addresses of which are values, the typing of values becomes dependent on the store. (Obviously, the type of address a must depend on what the store contains at address a). Thus the first step will be to replace the relations $\models v : \tau$ and $\models E : TE$ by $s \models v : \tau$ and $s \models E : TE$. Thus we arrive at the following conjecture

Conjecture 3.3 *If* $s \models E : TE$ *and* $TE \vdash e : \tau$ *and* $s, E \vdash e \longrightarrow v, s'$ *then* $s' \models v : \tau$.

However, it is not the case that the typing of values depends on just the dynamic store, it also depends on a particular typing of the store. To see this, consider the following example. Let

$$\begin{aligned} s &= \{\mathrm{a} \mapsto \mathtt{nil}\} \\ E &= \{\mathrm{x} \mapsto \mathrm{a},\ \mathrm{y} \mapsto \mathrm{a}\} \\ TE &= \{\mathrm{x} : (\mathit{int\ list})\,\mathit{ref},\quad \mathrm{y} : (\mathit{bool\ list})\,\mathit{ref}\} \\ e &= (\lambda \mathrm{z}.\,!\mathrm{y})(\mathrm{x}{:=}\ [7]) \end{aligned}$$

Notice that x and y are bound to the same address. At first it might look like we have $s \models E : TE$ — after all, x is bound to a and $s(\mathrm{a})$ has type *int list*

[4] As we are not noncerned with *wrong*, this is a slight simplification of the original theorem.

and, similarly, y is bound to a and $s(\mathtt{a})$ has type *bool list*. But if we admitted $s \models E : TE$, not even the above very fundamental conjecture would hold: we have $TE \vdash e : \mathit{bool\ list}$ and $s, E \vdash e \longrightarrow [7], s'$, but certainly not $s' \models [7] : \mathit{bool\ list}$.

The solution to inconsistent assumptions about the types of stored objects is to introduce a *store typing*, ST, to be a map from addresses to types, and replace the relations $s \models v : \tau$ and $s \models E : TE$ by $s : ST \models v : \tau$ and $s : ST \models E : TE$, respectively. Hence we arrive at the second conjecture.

Conjecture 3.4 *If $s : ST \models E : TE$ and $TE \vdash e : \tau$ and $s, E \vdash e \longrightarrow v, s'$ then there exists a store typing ST' such that $s' : ST' \models v : \tau$.*

The idea is that a stored object can have at most one type, namely the one given by the store typing; formally,

$$\text{if } s : ST \models a : \tau \text{ then } \tau = (ST(a))\,\mathit{ref} \text{ and } s : ST \models s(a) : ST(a) \tag{3.9}$$

With the current type inference system, conjecture 3.4 is in fact false. However one can "almost" prove it and the one point where the proof breaks down gives a hint how to improve the type inference system.

If we accept (3.9) and attempt a proof of Conjecture 3.4 then we see that store types must be able to contain free type variables. For instance, with $s = ST = \{\}$ and $e = \mathtt{ref}(\lambda\mathtt{x}.\mathtt{x})$ we have $TE_0 \vdash e : (\mathrm{t} \to \mathrm{t})\,\mathit{ref}$ and $E_0, \{\} \vdash e \longrightarrow a, \{a \mapsto [\mathtt{x}, \mathtt{x}, \{\}]\}$, for some a, so if we are to obtain the conclusion of the conjecture

$$\{a \mapsto [\mathtt{x}, \mathtt{x}, \{\}]\} : ST' \models a : (\mathrm{t} \to \mathrm{t})\,\mathit{ref}$$

then ST' must be $\{a \mapsto (\mathrm{t} \to \mathrm{t})\}$, c.f. (3.9).

These free type variables in the store typings are extremely important. In fact, they reveal what goes wrong in unsound inferences, as should soon become clear. Let us attempt a proof of Conjecture 3.4 by induction on the depth of inference of $s, E \vdash e \longrightarrow v, s'$. (It requires a definition of the relation $\models$, of course, but we shall soon see how that can be done). It turns out that all the cases go through, except one, namely the case concerning let expressions where the proof breaks down in the most illuminating way. So let us assume that we have defined $\models$ and dealt succesfully with all other cases; we then come to the dynamic inference rule for let expressions:

$$\frac{s, E \vdash e_1 \longrightarrow v_1, s_1 \quad s_1, E \pm \{x \mapsto v_1\} \vdash e_2 \longrightarrow v, s'}{s, E \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \longrightarrow v, s'} \tag{3.10}$$

The conclusion $TE \vdash e : \tau$ must have been by the rule

$$\frac{TE \vdash e_1 : \tau_1 \quad TE \pm \{x \mapsto \mathrm{Clos}_{TE}\tau_1\} \vdash e_2 : \tau}{TE \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau} \tag{3.11}$$

(Recall that $\mathrm{Clos}_{TE}\tau_1$ means $\forall\alpha_1 \ldots \alpha_n.\tau_1$, where $\{\alpha_1, \ldots, \alpha_n\}$ are the type variables in τ_1 that are *not* free in TE).

We now apply the induction hypothesis to the first premise of (3.10) together with the first premise of (3.11) and the given $s : ST \models E : TE$. Thus there exists a ST_1 such that

$$s_1 : ST_1 \models v_1 : \tau_1 \tag{3.12}$$

Before we can apply the induction hypothesis to the second premise of (3.10), we must establish

$$s_1 : ST_1 \models E \pm \{x \mapsto v_1\} : TE \pm \{x \mapsto \mathrm{Clos}_{TE}\tau_1\}$$

and to get this, we must strengthen (3.12) to

$$s_1 : ST_1 \models v_1 : \mathrm{Clos}_{TE}\tau_1. \tag{3.13}$$

It is precisely this step that goes wrong, if we by taking the closure generalize on type variables that occur free in ST_1. The snag is that when we have imperative features, there are really two places a type variable can occur free, namely (1) the type environment and (2) the store typing. In both cases, generalization on such a type variable is wrong. The naive extension of the polymorphic type discipline fails because it ignores the free type variables in the store typing.

As a counter–example to conjecture 3.4, we can revisit example 3.1.[5] Assuming $s = ST = \{\}$, the dynamic evaluation was

$$\{\}, E_0 \vdash \mathtt{ref}(\lambda\mathrm{x.x}) \longrightarrow a, \{a \mapsto [\mathrm{x}, \mathrm{x}, \{\}]\} \tag{3.14}$$

and the type inference $TE_0 \vdash \mathtt{ref}(\lambda\mathrm{x.x}) : (\mathrm{t} \to \mathrm{t})\ \mathit{ref}$. Thus, since
$\{\} : \{\} \models E_0 : TE_0$, the induction hypothesis yields an ST_1 such that

$$\{a \mapsto [\mathrm{x}, \mathrm{x}, \{\}]\} : ST_1 \models a : (\mathrm{t} \to \mathrm{t})\ \mathit{ref} \tag{3.15}$$

from which it follows that ST_1 must be $\{a \mapsto (\mathrm{t} \to \mathrm{t})\}$. The free occurrence of t in ST_1 expresses a dependence of the type of a on the store typing. therefore, we cannot strengthen (3.15) to

$$\{a \mapsto [\mathrm{x}, \mathrm{x}, \{\}]\} : \{a \mapsto (\mathrm{t} \to \mathrm{t})\ \mathit{ref}\} \models a : \forall \mathrm{t}.(\mathrm{t} \to \mathrm{t})\ \mathit{ref}\,.$$

[5] It is easy to extend the dynamic semantics with *wrong* and with basic functions to implement the arithmetic and other basic operations. Applying addition to *true* results in *wrong*.

The are various ways in which one can try to strengthen the theorem so that the induction goes through. One approach is to try to formulate a side condition on the let rule expressing on which type variables generalization is admissible. But we should not be surprised that a natural condition is hard to find – essentially it ought to involve the evolution of the store typings, but store typings do not occur in the static semantics at all. One way out of this is to give up having references as values and instead have updatable variables, because the store typing then essentially becomes a part of the type environment. This was what was done in the early version of ML used in Edinburgh LCF. Even though generalization of the types of updatable references is prevented, one still has to impose extra restrictions; see [15], page 49 rule (2)(i)(b) for details.

Another approach is to enrich the type inference system with information about the store typing. To include the store typing itself is pointless since we are not interested in the domain of the store. All that is of interest is the set of type variables that occur free in the store typing. One way of enriching the type inference system with information about this set is to partition the type variables into two syntactic classes, those that are assumed to occur in the store typing and those that are guaranteed not to occur in the store typing. Because type checking is purely structural, the set of type variables that are assumed to be in the store typing is in general a superset of the set of type variables that will actually have to be in the store typing (it is undecidable to determine given an arbitrary expression whether it generates a reference). This idea was first used by Damas in his thesis, and I also use it in the system I shall now present.

Chapter 4

The Type Discipline

We first present the type inference system. Then we give examples of its use and present a type checker.

4.1 The Inference system

The basis idea is to modify the language of types so that there is a visible difference between those types that occur in the implicit store typing and those that do not. This can be achieved by having two disjoint sets of type variables; ImpTyVar is the set of *imperative* type variables and AppTyVar is the set of *applicative* type variables:

$t \in \mathrm{AppTyVar} = \{\mathrm{t}, \mathrm{t}_1, \cdots\}$	applicative type variables
$u \in \mathrm{ImpTyVar} = \{\mathrm{u}, \mathrm{u}_1, \cdots\}$	imperative type variables
$\alpha \in \mathrm{TyVar} = \mathrm{AppTyVar} \cup \mathrm{ImpTyVar}$	type variables

The set Type of types, ranged over by τ and the set TypeScheme of type schemes, ranged over by σ, are defined by

$$\begin{array}{lll} \pi & \in & \mathrm{TyCon} = \{stm, int, bool, \ldots\} & \text{type constructors} \\ \tau & ::= & \pi \mid \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \; ref & \text{types} \\ \sigma & ::= & \tau \mid \forall \alpha . \sigma_1 & \text{type schemes} \end{array}$$

and type environments are defined by

$$TE \in \mathrm{TyEnv} = \mathrm{Var} \xrightarrow{\mathrm{fin}} \mathrm{TypeScheme}.$$

When T is a type, a type scheme, or a type environment then tyvars(T) means all the type variables that occur free in T. A type is *imperative* if it contains no applicative type variables:

$$\theta \in \mathrm{ImpType} = \{\tau \in \mathrm{Type} \mid \mathrm{apptyvars}\, \tau = \emptyset\}.$$

A *substitution* is now a map S : TyVar $\rightarrow$ Type that maps imperative type variables to imperative types. Hence the image of an imperative type variable cannot contain applicative type variables, but the image of an applicative type variable can contain imperative type variables.

The definition of instantiation, $\sigma > \tau$, is as before but now with the new meaning of substitution.

An expression is said to be *non-expansive* if it is a variable or a lambda abstraction. All other expressions, i.e., applications and let expressions, are said to be *expansive*. Although this distinction is purely syntactical it is supposed to suggest the dynamic behaviour; the dynamic evaluation of a non-expansive expression cannot expand the domain of the store, while the evaluation of an expansive expression might. Our syntactic classification is very crude as there are many expansive expressions that in fact will not expand the domain of the store. The classification is chosen so as to be very easy to remember; the proofs that follow do not rely heavily on this very crude classification.

As before, $\mathrm{Clos}_{TE}\tau$ means $\forall\alpha_1\ldots\alpha_n.\tau$ where

$$\{\alpha_1,\ldots,\alpha_n\} = \mathrm{tyvars}\,\tau \setminus \mathrm{tyvars}\,TE\,.$$

In addition we now define

$$\mathrm{AppClos}_{TE}\tau$$

to mean $\forall\alpha_1\ldots\alpha_n.\tau$ where $\{\alpha_1,\ldots,\alpha_n\} = \mathrm{apptyvars}\,\tau \setminus \mathrm{apptyvars}\,TE$ is the set of all *applicative* type variables in τ not free in TE.

The type inference rules appear in Figure 4.1 and they allow us to infer sequents of the form $TE \vdash e : \tau$. We see that the first three rules are as before but that the let rule has been split into two rules.

Notice that if TE contains no imperative type variables (free or bound) then every type inference that could be done in the original system can also be done in the new system. (Note that in rule 4.5 when τ_1 contains no imperative type variables then taking the applicative closure is the same as taking the ordinary closure). But in general TE will contain imperative type variables.

4.2 Examples of Type Inference

Let us try to type a couple of example programs to get a feel for the role of imperative type variables.

$$\boxed{TE \vdash e : \tau}$$

$$\frac{x \in \mathrm{Dom}\, TE \quad TE(x) > \tau}{TE \vdash x : \tau} \tag{4.1}$$

$$\frac{TE \pm \{x \mapsto \tau'\} \vdash e_1 : \tau}{TE \vdash \lambda x.e_1 : \tau' \to \tau} \tag{4.2}$$

$$\frac{TE \vdash e_1 : \tau' \to \tau \quad TE \vdash e_2 : \tau'}{TE \vdash e_1\ e_2 : \tau} \tag{4.3}$$

$$\frac{e_1 \text{ is non-expansive} \quad TE \vdash e_1 : \tau_1 \quad TE \pm \{x \mapsto \mathrm{Clos}_{TE}\tau_1\} \vdash e_2 : \tau}{TE \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau} \tag{4.4}$$

$$\frac{e_1 \text{ is expansive} \quad TE \vdash e_1 : \tau_1 \quad TE \pm \{x \mapsto \mathrm{AppClos}_{TE}\tau_1\} \vdash e_2 : \tau}{TE \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau} \tag{4.5}$$

Figure 4.1: The Type Inference Rules for Imperative Types

Throughout, we shall require

$$\begin{aligned} TE(\texttt{ref}) &= \forall \mathrm{u}.\mathrm{u} \rightarrow \mathrm{u}\, \mathit{ref} \\ TE(\texttt{:=}) &= \forall \mathrm{t}.\mathrm{t}\, \mathit{ref} \rightarrow \mathrm{t} \rightarrow \mathit{stm} \\ TE(\texttt{!}) &= \forall \mathrm{t}.\mathrm{t}\, \mathit{ref} \rightarrow \mathrm{t}. \end{aligned}$$

(Recall that u is imperative, while t is applicative; the reason that only the type of `ref` contains an imperative type variable should gradually become clear).

To give examples that have some bearing on real programming let us extend the types with lists

$$\tau ::= \ldots \mid \tau_1\, \mathit{list}$$

and with constructors

$$\begin{aligned} \texttt{nil} &: \forall \mathrm{t}.\mathrm{t}\, \mathit{list} \\ \texttt{cons} &: \forall \mathrm{t}.\mathrm{t} \rightarrow \mathrm{t}\, \mathit{list} \rightarrow \mathrm{t}\, \mathit{list} \end{aligned}$$

and functions

$$\begin{aligned} \texttt{hd} &: \forall \mathrm{t}.\mathrm{t}\, \mathit{list} \rightarrow \mathrm{t} \\ \texttt{tl} &: \forall \mathrm{t}.\mathrm{t}\, \mathit{list} \rightarrow \mathrm{t}\, \mathit{list} \end{aligned}$$

We shall write $e_1\texttt{::}e_2$ for $(\texttt{cons}\ e_1)e_2$.

Moreover, we add sequential evaluation and while loops to the language, see Figures 4.2 and 4.3.

Example 4.1 As long as an expression does not have free variables whose type contains imperative type variables, the type of the expression need not involve imperative type variables. As an example we have

$$TE \vdash \lambda \mathrm{x}.\texttt{!(!x)} : \mathrm{t}_1\, \mathit{ref}\ \mathit{ref} \rightarrow \mathrm{t}_1.$$

Notice that t_1 is applicative and that the typing of `!(!x)` involves two different instantiations of the type scheme for `!`. Thus, by use of the let rule for non-expansive expressions, rule 4.4, we get $TE \vdash e_2 : \mathit{stm}$, where

$e_2 =$ let `double-deref` = λ`x.!(!x)`
  in while `double-deref(ref(ref false))` do
    `double-deref(ref(ref 5))`

Syntax

e	$::=$	e_1 ; e_2	sequential evaluation
	$\mid$	**while** e_1 **do** e_2	while loop
	$\mid$	**begin** e_1 **end**	parenthesis
	$\mid$	(e_1)	parenthesis

Dynamic Semantics

$$\frac{s, E \vdash e_1 \longrightarrow done, s_1 \quad s_1, E \vdash e_2 \longrightarrow v, s'}{s, E \vdash e_1 \; ; \; e_2 \longrightarrow v, s'}$$

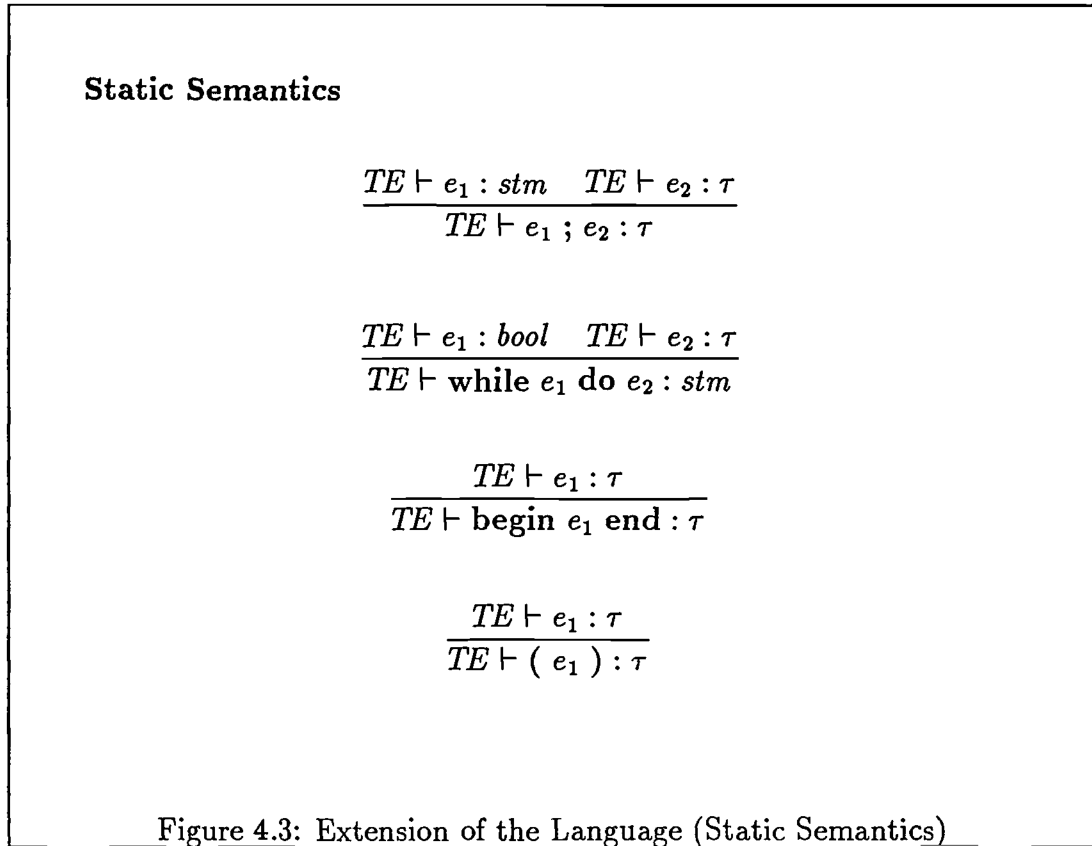
$$\frac{s, E \vdash e_1 \longrightarrow false, s'}{s, E \vdash \textbf{while } e_1 \textbf{ do } e_2 \longrightarrow done, s'}$$

$$\frac{\begin{array}{l} s, E \vdash e_1 \longrightarrow true, s_1 \\ s_1, E \vdash e_2 \longrightarrow v_2, s_2 \\ s_2, E \vdash \textbf{while } e_1 \textbf{ do } e_2 \longrightarrow done, s' \end{array}}{s, E \vdash \textbf{while } e_1 \textbf{ do } e_2 \longrightarrow done, s'}$$

$$\frac{s, E \vdash e_1 \longrightarrow v, s'}{s, E \vdash \textbf{begin } e_1 \textbf{ end} \longrightarrow v, s'}$$

$$\frac{s, E \vdash e_1 \longrightarrow v, s'}{s, E \vdash (\; e_1 \;) \longrightarrow v, s'}$$

Figure 4.2: Extension of the Language (Dynamic Semantics)



Static Semantics

$$\frac{TE \vdash e_1 : stm \quad TE \vdash e_2 : \tau}{TE \vdash e_1 \; ; \; e_2 : \tau}$$

$$\frac{TE \vdash e_1 : bool \quad TE \vdash e_2 : \tau}{TE \vdash \textbf{while } e_1 \textbf{ do } e_2 : stm}$$

$$\frac{TE \vdash e_1 : \tau}{TE \vdash \textbf{begin } e_1 \textbf{ end} : \tau}$$

$$\frac{TE \vdash e_1 : \tau}{TE \vdash (\; e_1 \;) : \tau}$$

Figure 4.3: Extension of the Language (Static Semantics)

Example 4.2 On the other hand, if the expression has a free variable whose type (scheme) contains an imperative type variable, in particular if `ref` occurs free, then the type of the expression may have to involve imperative type variables. Hence $TE \vdash e_1 : \mathrm{u} \to \mathrm{u}$ where

$$e_1 = \lambda \mathtt{x}.\,!\,(\mathtt{ref\ x})$$

but not $TE \vdash e_1 : \mathrm{t} \to \mathrm{t}$. Still, we can type

$$\textbf{let } \mathtt{f} = e_1 \textbf{ in } (\mathtt{f(7);\ f(true)})$$

using the let rule for non-expansive expressions which will allow a generalization from $\mathrm{u} \to \mathrm{u}$ to $\forall \mathrm{u}.\mathrm{u} \to \mathrm{u}$ for the type of `f`.

Example 4.3 We have $TE \vdash e_1 : (\mathrm{u} \to \mathrm{u})\ ref$, where

$$e_1 = \mathtt{ref}(\lambda \mathrm{x}.\mathrm{x})$$

but not $TE \vdash e_1 : (\mathrm{t} \to \mathrm{t})\ ref$. Consequently, in an expression of the form

$$\textbf{let } \mathtt{r}\texttt{=}\ \mathtt{ref}(\lambda \mathrm{x}.\mathrm{x}) \textbf{ in } e_2$$

the let rule for expansive expressions, rule 4.5, will prohibit generalization from $(u \to u)$ *ref* to $\forall u.((u \to u))$ *ref*). Thus the faulty expression

```
let r= ref(λx.x) in (r:= λx.x+1; (!r)true)
```

cannot be typed. Note that

```
let r= ref(λx.x) in (r:= λx.x+1; (!r)1)
```

will be typable using $TE \vdash$ `ref(λx.x)` : $(int \to int)$ *ref* and rule 4.5.

Example 4.4 Here is a function that reverses lists in one single scan using iteration and a minimum number of applications of the :: constructor:

```
e1 = λl. let data= ref l in
         let result= ref nil in
            begin
             while !data<>nil do
             begin result:= hd(!data)::!result;
                   data:= tl(!data)
             end;
             !result
            end
```

We have $TE \vdash e_1 : u\ list \to u\ list$ where the body of the second let expression will be typed under the assumptions

$$
\begin{aligned}
TE(\texttt{data}) &= u\ list\ ref \\
TE(\texttt{result}) &= u\ list\ ref
\end{aligned}
$$

Note that u remains free as u after "λl." occurs free in the type environment, namely in the type of `l`. Now

```
let fast_reverse= e1
in  begin fast_reverse [1, 9, 7, 5];
          fast_reverse [true, false, false]
    end
```

is typable using rule 4.4, which allows the generalization from $u\ list \to u\ list$ to $\forall u.u\ list \to u\ list$.

As one would expect, since `fast_reverse` has type $\forall u.u\ list \to u\ list$ while the applicative reverse function has type $\forall t.t\ list \to t\ list$, there are programs that are typable with the applicative version only. One example is

```
let reverse= ···
in let f= hd(reverse[λx.x])
   in begin f(7); f(true) end
```

which can be typed under the assumption the reverse has type $\forall$t.t *list* $\rightarrow$ t *list* but not under the assumption that reverse has the type $\forall$u.u *list* $\rightarrow$ u *list*.

Example 4.5 This example illustrates what I believe to be the only interesting limitation of the inference system. The fast_reverse function is a special case of folding a function f (e.g. cons) over a list l starting with initial result i (e.g. nil).

```
e1 = λf.λi.λl.
     let data= ref l in
     let result= ref i in
     begin
        while !data<>nil do
        begin result:= f(hd(!data))(!result);
              data:= tl(!data)
        end;
        !result
     end
```

We have $TE \vdash e_1 : (u_1 \rightarrow u_2 \rightarrow u_2) \rightarrow u_2 \rightarrow u_1\ \mathit{list} \rightarrow u_2$ and we can type

```
let fold= e1 in
begin fold cons nil [5,7,9];
      fold cons nil [true, true, false]
end
```

because the let rule for non-expansive let expressions allows us to generalize on u_1 and u_2 in the type of fold.

However, we will *not* be able to type the very similar

```
let fold= e1 in
let fast_reverse= fold cons nil in
begin
   fast_reverse [3,5,7];
   fast_reverse [true, true, false]
end
```

because `fold cons nil` will be deemed expansive so that `fast_reverse` cannot get the polymorphic type ∀u.u *list* → u *list* .

This illustrates that the syntactic classification into expansive and non-expansive expressions is quite crude. Fortunately, as we shall see when we study the soundness proof, soundness is not destroyed by taking a more sophisticated classification. Moreover, even with the simple classification we get a typable program by changing the definition of `fast_reverse` to

```
let fast_reverse= λl.fold cons nil l
```

so the limitation isn't really too bad.

Example 4.6 The final example is an expression which, if evaluated, would give a run-time error, and which therefore must be rejected by the typing rules. If you have some candidate for a clever type inference system, it might be worth making sure that the expression e below really is untypable in your system. The example illustrates the use of "own" variables, which form a dangerous threat to soundness!

When applied to an argument x, the function `mk_sham_id` produces a function, `sham_id`, which has an own variable with initial contents x. Each time `sham_id` is applied to an argument it returns the argument with which it was last called and stores the new argument.

```
e = let mk_sham_id= λx.
        let own=ref x
        in λy.(let temp=!own in (own:= y; temp))
    in let sham_id= mk_sham_id nil in
        begin sham_id [true];
              hd(sham_id[1])+1
        end
```

If we take the naive extension of Milner's polymorphic type discipline, see Chapter 3, then e becomes typable; first `mk_sham_id` gets type ∀t.t → t → t; then `sham_id` gets the type ∀t.t *list* → t *list* (hence its name) and that does it. The LCF rules were amended with a special syntactic constraint to exclude generalization of the type of expressions that use own variables.

With the imperative type variables, `mk_sham_id` gets type ∀u.u → u → u and `sham_id` gets type u *list* → u *list*, but then we are barred from generalizing on u

since `mk_sham_id nil` quite rightly is considered expansive. Therefore, at most one of the applications of `sham_id` can be typed. ■

Let us sum up the intuitions about inferences of the form

$$TE \vdash e : \tau.$$

Any imperative type variable *free* in TE stands for some fixed, but unknown type that occurs in the typing of the store prior to the dynamic evaluation of e.

Any imperative type variable *bound* in TE, in the type scheme ascribed to x, say, stands for a type that might have to be added to the present store typing to type new references that are created as a result of applying x.

Moreover, an imperative type variable that is free in τ but not free in TE ranges over types that may have to be added to the initial store typing to obtain the resulting store typing. The "may have to be" is because, in general, more type variables than strictly necessary may be imperative. For example we have $\vdash \lambda x.x : t \rightarrow t$ and also $\vdash \lambda x.x : u \rightarrow u$.

Finally let us discuss the let rule. The idea in the first of the two rules for let $x = e_1$ in e_2 is that if $TE \vdash e_1 : \tau_1$ and e_1 is non-expansive then the initial and the resulting store typings are identical and so none of the imperative type variables free in τ_1 but not free in TE will actually be added to the initial store typing. Therefore, these imperative type variables may be quantified.

On the other hand, if e_1 is expansive then we have to assume that the new imperative type variables in τ_1 will have to be added to the initial store typing and so the second let rule does not allow generalization on these.

In both cases, if τ_1 contains an applicative type variable t that is not free in TE then t will not be added to the initial store typing simply because store typings by definition contain no applicative type variables, and so generalization on t is safe.

4.3 A Type Checker

Figure 4.4 contains a type checker for the new type discipline. I call it W_1 because it is similar to Milner's type checker W, (Section 2.4). W_1 uses a modified

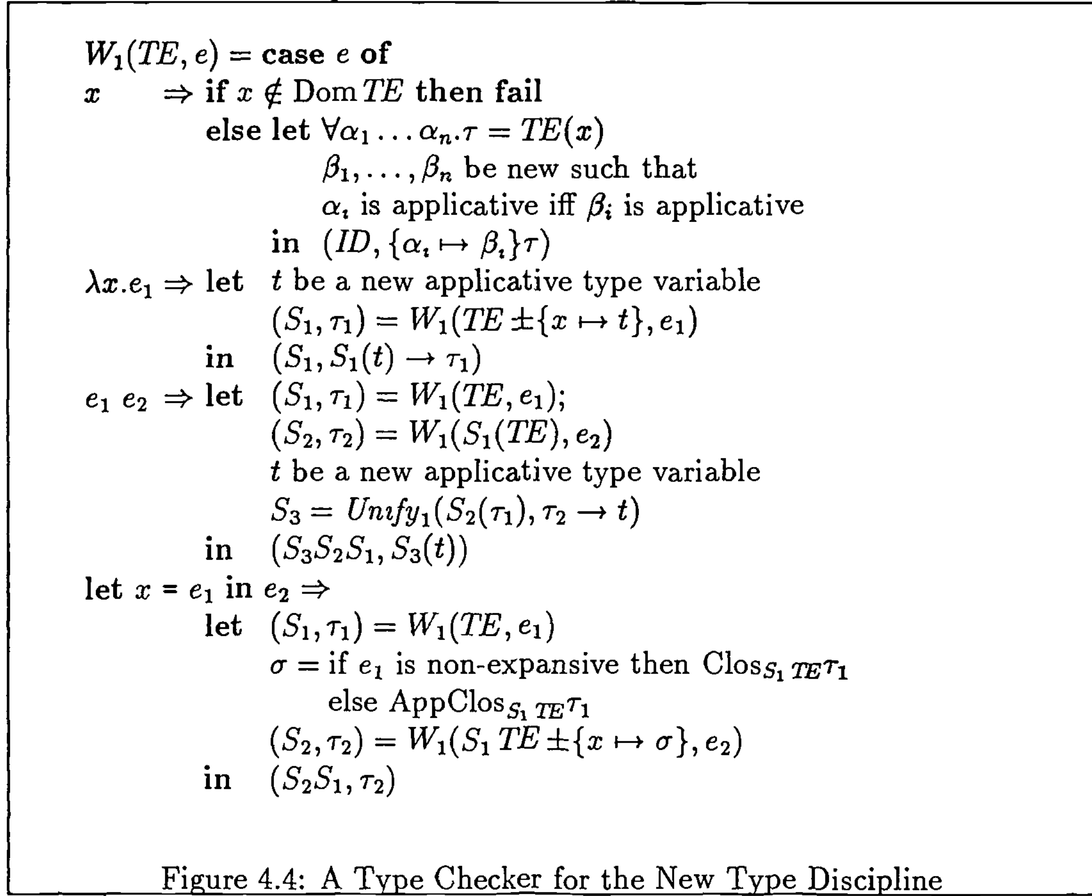


Figure 4.4: A Type Checker for the New Type Discipline

unification algorithm, $Unify_1$, which is like ordinary unification, except that

$$Unify_1(\alpha, \tau) = \begin{cases} \{\alpha \mapsto \tau\}, & \text{if } \alpha \text{ is applicative;} \\ \{\alpha \mapsto S(\tau)\} \cup S, & \text{if } \alpha \text{ is imperative} \end{cases}$$

provided α does not occur in τ, where $\{\alpha_1, \ldots, \alpha_n\}$ is apptyvars τ and $\{\mathrm{u}_1, \ldots, \mathrm{u}_n\}$ are new imperative type variables, and S is $\{\alpha_1 \mapsto \mathrm{u}_1, \ldots, \alpha_n \mapsto \mathrm{u}_n\}$.

That W_1 is sound in the sense that $(S, \tau) = W_1(TE, e)$ implies that for all TE' with $TE \xrightarrow{S} TE'$ we have $TE' \vdash e : \tau$ is easy to prove by structural induction on e given a lemma that type inference is preserved under substitutions (this lemma will be proved later, see lemma 5.2).

To prove that W_1 is complete in the sense that it finds principal types for all typable expressions requires more work. I have not done it simply because this is where I had to stop, but I feel confident of the completeness of W_1 because, intuitively, W_1 never has to make arbitrary choices. I believe that the proof will be similar in spirit and simpler in detail than the proof of the completeness of the signature checker in Part III.

Chapter 5

Proof of Soundness

We shall now prove the soundness of the type inference system. Substitutions are at the core of all we do, so we start by proving lemmas about substitutions and type inference. Then we shall define the quaternary relation $s : ST \models v : \tau$ (discussed in Section 3.2) as the maximal fixpoint of a monotonic operator. Finally we give the inductive consistency proof itself.

5.1 Lemmas about Substitutions

As defined earlier, a substitution is a map $S : \text{TyVar} \to \text{Type}$ such that $S(u) \in \text{ImpType}$ for all $u \in \text{ImpTyVar}$. Substitutions are extended to types and they can be composed. The restriction and region of a substitution is defined as before, see Section 2.3. Similarly, we can define substitutions on type schemes and type environments by ternary relations $\sigma_1 \xrightarrow{S} \sigma_2$ and $TE \xrightarrow{S} TE'$:

Definition 5.1 *Let $\sigma_1 = \forall\alpha_1 \ldots \alpha_n.\tau_1$ and $\sigma_2 = \forall\beta_1 \ldots \beta_m.\tau_2$ be type schemes and S be a substitution. We write $\sigma_1 \xrightarrow{S} \sigma_2$ if*

1. *$m = n$, and $\{\alpha_i \mapsto \beta_i \mid i \le i \le n\}$ is a bijection and α_i is imperative iff β_i is imperative, and no β_i is in $\text{Reg}(S_0)$, and*

2. *$(S_0 \,|\{\alpha_i \mapsto \beta_i\})\tau_1 = \tau_2$*

where $S_0 \stackrel{\text{def}}{=} S \downarrow \text{tyvars}\,\sigma_1$. Moreover, we write $TE \xrightarrow{S} TE'$ if $\text{Dom}\,TE = \text{Dom}\,TE'$ and for all $x \in \text{Dom}\,TE$, $TE(x) \xrightarrow{S} TE'(x)$.

The definition of instantiation ($\sigma > \tau$) is as before but now with the new meaning of type variables and substitutions. One can prove that if $\sigma > \tau$ and $\sigma \xrightarrow{S} \sigma'$ then $\sigma' > S\,\tau$. The proof is almost identical to that of Lemma 2.6.

The following lemma will be used again and again in what follows.

Lemma 5.2 *If* $TE \vdash e : \tau$ *and* $TE \xrightarrow{S} TE'$ *then* $TE' \vdash e : S\,\tau$.

Proof. By structural induction on e. The only interesting case is the one for e = **let** $x = e_1$ **in** e_2 which in turn is proved by case analysis. (The two cases are similar, but there are subtle differences and since this lemma is terribly important, we had better be careful here).

$\boxed{e_1 \text{ is non-expansive}}$ Here $TE \vdash e : \tau$ was inferred by

$$\frac{e_1 \text{ is non-expansive} \quad TE \vdash e_1 : \tau_1 \quad TE \pm \{x \mapsto \text{Clos}_{TE}\tau_1\} \vdash e_2 : \tau}{TE \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau}. \tag{5.1}$$

Let $\sigma_1 = \text{Clos}_{TE}\tau_1 = \forall\alpha_1 \ldots \alpha_n.\tau_1$, let $S_1 = S \downarrow \text{tyvars}\, TE$ and let $S_0 = S \downarrow \text{tyvars}\,\sigma_1$. Let $\beta_1 \ldots \beta_n$ be distinct type variables where β_i is imperative iff α_i is imperative and no β_i is in $\text{Reg}\, S_1$. Then no β_i is in $\text{Reg}\, S_0$, so by definition 5.1 we have

$$\begin{aligned} \forall\alpha_1 \ldots \alpha_n.\tau_1 &\xrightarrow{S} \forall\beta_1 \ldots \beta_n.(S_0 \,|\{\alpha_i \mapsto \beta_i\})\tau_1 \\ &= \forall\beta_1 \ldots \beta_n.(S_1 \,|\{\alpha_i \mapsto \beta_i\})\tau_1. \end{aligned}$$

Since $TE \xrightarrow{S} TE'$, no β_i is free in TE'. Moreover, any type variable that is not a β_i but occurs in $(S_1 \,|\{\alpha_i \mapsto \beta_i\})\tau_1$ must be free in TE'. (The reason for this is that every type variable free in σ is in TE and $TE \xrightarrow{S} TE'$). Therefore,

$$\forall\beta_1 \ldots \beta_n.(S_1 \,|\{\alpha_i \mapsto \beta_i\})\tau_1 = \text{Clos}_{TE'}(S_1 \,|\{\alpha_i \mapsto \beta_i\})\tau_1.$$

Let $S' = S_1 \,|\{\alpha_i \mapsto \beta_i\}$. Then we have

$$\text{Clos}_{TE}\tau_1 \xrightarrow{S} \text{Clos}_{TE'}\, S'\,\tau_1. \tag{5.2}$$

Since $TE \xrightarrow{S} TE'$ and no α_i is free in TE we have $TE \xrightarrow{S'} TE'$. Thus by induction on e_1, using the second premise of (5.1),

$$TE' \vdash e_1 : S'\,\tau_1 \tag{5.3}$$

By (5.2) we have

$$TE \pm \{x \mapsto \text{Clos}_{TE}\tau_1\} \xrightarrow{S} TE' \pm \{x \mapsto \text{Clos}_{TE'}\, S'\,\tau_1\}.$$

Therefore, by induction on e_2, using the third premise of (5.1),

$$TE' \pm \{x \mapsto \text{Clos}_{TE'}\, S'\,\tau_1\} \vdash e_2 : S\,\tau. \tag{5.4}$$

Thus by rule (4.4) on (5.3) and (5.4) we have $TE' \vdash e : S\,\tau$ as desired.

$\boxed{e_1 \text{ is expansive}}$ Here $TE \vdash e : \tau$ was inferred by

$$\frac{e_1 \text{ is expansive} \quad TE \vdash e_1 : \tau_1 \quad TE \pm \{x \mapsto \mathrm{AppClos}_{TE}\tau_1\} \vdash e_2 : \tau}{TE \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}. \tag{5.5}$$

Let $\sigma_1 = \mathrm{AppClos}_{TE}\tau_1 = \forall\alpha_1 \ldots \alpha_n.\tau_1$, let $S_1 = S \downarrow (\mathrm{tyvars}\, TE \cup \mathrm{imptyvars}\, \tau_1)$ and let $S_0 = S \downarrow \mathrm{tyvars}\, \sigma_1$. Every α_i is applicative. Let $\beta_1 \ldots \beta_n$ be distinct applicative type variables none of which is in $\mathrm{Reg}\, S_1$. Then no β_i is in $\mathrm{Reg}\, S_0$, so by definition 5.1 we have

$$\begin{aligned} \forall\alpha_1 \ldots \alpha_n.\tau_1 \quad &\xrightarrow{S} \quad \forall\beta_1 \ldots \beta_n.(S_0 \,|\, \{\alpha_i \mapsto \beta_i\})\tau_1 \\ &= \quad \forall\beta_1 \ldots \beta_n.(S_1 \,|\, \{\alpha_i \mapsto \beta_i\})\tau_1 \end{aligned}$$

Since $TE \xrightarrow{S} TE'$ no β_i is free in TE'. Also, any applicative type variable that is not a β_i but occurs in $(S_1 \,|\, \{\alpha_i \mapsto \beta_i\})\tau_1$ must be free in TE'. The reasons for this are

1. Any applicative α in τ_1 which is not an α_i is free in TE and $TE \xrightarrow{S} TE'$.

2. Any imperative α in τ_1 is mapped by S to an imperative type, i.e., a type with no applicative type variables.

Thus

$$\forall\beta_1 \ldots \beta_n.(S_1 \,|\, \{\alpha_i \mapsto \beta_i\})\tau_1 = \mathrm{AppClos}_{TE'}(S_1 \,|\, \{\alpha_i \mapsto \beta_i\})\tau_1.$$

Let $S' = S_1 \,|\, \{\alpha_i \mapsto \beta_i\}$. Then we have

$$\mathrm{AppClos}_{TE}\tau_1 \xrightarrow{S} \mathrm{AppClos}_{TE'}\, S'\, \tau_1 \tag{5.6}$$

Since $TE \xrightarrow{S} TE'$ and no α_i is free in TE we have $TE \xrightarrow{S'} TE'$.

Thus by induction on e_1, using the second premise of (5.5),

$$TE' \vdash e_1 : S'\, \tau_1. \tag{5.7}$$

We have

$$TE \pm \{x \mapsto \mathrm{AppClos}_{TE}\tau_1\} \xrightarrow{S} TE' \pm \{x \mapsto \mathrm{AppClos}_{TE'}\, S'\, \tau_1\}$$

by (5.6), so by induction on e_2, using the third premise of (5.5),

$$TE' \pm \{x \mapsto \mathrm{AppClos}_{TE'}\, S'\, \tau_1\} \vdash e_2 : S\, \tau$$

which with (5.7) gives the desired $TE \vdash e : S\, \tau$ by rule 4.5. ∎

The following lemma will be used in Section 5.2 to prove Lemma 5.11.

Lemma 5.3 *Assume* $\sigma \xrightarrow{S} \sigma'$ *and* $\sigma' > \tau_1'$ *and* A *is a set of type variables with* $\text{tyvars}\,\sigma \subseteq A$. *Then there exists a type* τ_1 *and a substitution* S_1 *such that* $\sigma > \tau_1$, $S_1\tau_1 = \tau_1'$, *and* $S \downarrow A = S_1 \downarrow A$.

Proof. The following diagram may be helpful:

$$\begin{array}{ccccc} & \sigma & \xrightarrow{S} & \sigma' & \\ I & \vee & & \vee & I' \\ & \tau_1 & \xrightarrow{S_1} & \tau_1' & \end{array}$$

Write σ as $\forall\alpha_1 \ldots \alpha_n.\tau$ and σ' as $\forall\beta_1 \ldots \beta_m.\tau'$. Since $\sigma \xrightarrow{S} \sigma'$ we have $m = n$, α_i is imperative if and only if β_i is imperative, $\{\alpha_i \mapsto \beta_i\}$ is a bijection, no β_i is in $\text{Reg}\,S_0$ and $(S_0 \,|\{\alpha_i \mapsto \beta_i\})\tau = \tau'$, where $S_0 = S \downarrow \text{tyvars}\,\sigma$.

Since $\sigma' > \tau_1'$ there exists an

$$I' = \{\beta_i \mapsto \tau^{(i)} \mid 1 \leq i \leq n\}$$

such that $I'\tau' = \tau_1'$. Let $\{\gamma_1, \ldots, \gamma_k\}$ be the set of type variables occurring in the range of I' and let $\{\delta_1, \ldots, \delta_k\}$ be a set of type variables such that the renaming substitution $R = \{\gamma_i \mapsto \delta_i\}$ is a bijection, γ_i is imperative iff δ_i is imperative, and $\text{Rng}\,R \cap (A \cup \text{Reg}(S \downarrow A)) = \emptyset$.

Let I be $\{\alpha_i \mapsto R\,\tau^{(i)} \mid 1 \leq i \leq n\}$. Then I maps imperative type variables to imperative types. Let $\tau_1 = I\,\tau$. Then $\sigma > \tau_1$ as desired.

Let $R^{-1} = \{\delta_j \mapsto \gamma_j \mid 1 \leq j \leq k\}$ and let $S_1 = R^{-1} \circ (S \downarrow A)$. Then S_1 is a substitution and as required we have $S_1 \downarrow A = S \downarrow A$, since $\{\delta_1, \ldots, \delta_k\} \cap \text{Reg}(S \downarrow A) = \emptyset$.

Now

$$\begin{array}{rcll} S_1\tau_1 & = & S_1 I\,\tau & \text{by the definition of } \tau_1 \\ & = & R^{-1}(S \downarrow A)\,I\,\tau & \text{by the definition of } S_1 \\ & = & I'(S_0 \,|\{\alpha_i \mapsto \beta_i\})\tau & \text{see below} \\ & = & I'\tau' & \text{as } \sigma \xrightarrow{S} \sigma' \\ & = & \tau_1' & \text{by the definition of } I' \end{array}$$

as required. To see the above equality

$$R^{-1}(S \downarrow A)\,I\,\tau = I'(S_0 \,|\{\alpha_i \mapsto \beta_i\})\tau \tag{5.8}$$

take any α occurring in τ. There are two cases:

$\boxed{\alpha \text{ is free in } \sigma}$ Here

$$\begin{array}{rcll} R^{-1}(S\downarrow A)\,I\,\alpha & = & R^{-1}(S\downarrow A)\alpha & \text{as } \alpha \text{ is free} \\ & = & R^{-1}\,S_0\,\alpha & \text{as tyvars}\,\sigma \subseteq A \\ & = & S_0\,\alpha & (*) \\ & = & I'\,S_0\,\alpha & \text{as no } \beta_i \text{ is in Reg}\,S_0 \\ & = & I'(S_0\,|\{\alpha_i \mapsto \beta_i\})\alpha & \text{as } \alpha \text{ is free} \end{array}$$

where $(*)$ is because $\{\delta_1, \ldots, \delta_k\} \cap \text{Reg}(S \downarrow A) = \emptyset$. This shows 5.8 when α is free in σ.

$\boxed{\alpha \text{ is bound in } \sigma}$ Here $\alpha = \alpha_i$, say, and

$$\begin{array}{rcl} R^{-1}(S\downarrow A)\,I\,\alpha_i & = & R^{-1}(S\downarrow A)\,R\,\tau^{(i)} \\ & = & \tau^{(i)} \end{array}$$

as tyvars $\tau^{(i)} \subseteq \text{Dom}\,R$ and $\text{Rng}\,R \cap A = \emptyset$. But

$$\begin{array}{rcll} \tau^{(i)} & = & I'\,\beta_i & \text{by the definition of } I' \\ & = & I'(S_0\,|\{\alpha_i \mapsto \beta_i\})\alpha_i & \end{array}$$

showing 5.8 when α is bound in σ. ■

5.2 Typing of Values using Maximal Fixpoints

Recall from the discussion in Section 3.2 that the relation between dynamic values v and types τ depends not just on the store, but also on a store typing. We introduced the notion of imperative type to be able to recognize types that are types of objects in the store. Hence a store typing is a map from addresses to imperative types; a *typed store* is a store and a store typing with equal domains:

$ST \in \text{StoreTyping} = \text{Addr} \xrightarrow{\text{fin}} \text{ImpType}$
$s : ST \in \text{TypedStore} = \{(s, ST) \in \text{Store} \times \text{StoreTyping} \mid \text{Dom}\,s = \text{Dom}\,ST\}$

We shall now define three relations $s : ST \models v : \tau$, $s : ST \models v : \sigma$, and $s : ST \models E : TE$ that provide the link we need between the static and the dynamic semantics.

We start out from a relation $R_{\text{Bas}} \subseteq \text{BasVal} \times \text{TyCon}$ giving the typing of the basic values. Thus $(3, \mathit{int}) \in R_{\text{Bas}}$, $(\mathit{true}, \mathit{bool}) \in R_{\text{Bas}}$, etc. We shall assume that $(\mathit{done}, \mathit{stm}) \in R_{\text{Bas}}$ and that *done* is the only value of type *stm*. We are then looking for relations satisfying the following:

EX BIBL. UNIV. EDINBURGH

Property 5.4 (of $\models$) We have

$s : ST \models v : \tau \iff$
 if $v = b$ then $(v, \tau) \in R_{\text{Bas}}$;
 if $v = [x, e_1, E]$ then there exists a TE such that
  $s : ST \models E : TE$ and $TE \vdash \lambda x.e_1 : \tau$;
 if $v = asg$ then $\tau = \tau_1\ ref \to \tau_1 \to stm$ for some τ_1;
 if $v = ref$ then $\tau = \theta \to \theta\ ref$ for some imperative θ;
 if $v = deref$ then $\tau = \tau_1\ ref \to \tau_1$ for some τ_1;
 if $v = a$ then $\tau = (ST(a))\ ref$ and $s : ST \models s(a) : ST(a)$

$s : ST \models v : \sigma \iff \forall \tau < \sigma,\ s : ST \models v : \tau$

$s : ST \models E : TE \iff$
 $\mathrm{Dom}\, E = \mathrm{Dom}\, TE$ and $\forall x \in \mathrm{Dom}\, E,\ s : ST \models E(x) : TE(x)$.

The above property does not define a unique relation $\models$. However, it can be regarded as a fixpoint equation

$$\models \quad = \quad F(\models) \tag{5.9}$$

where F is a certain operator. More precisely, let $U = U_1 \times U_2 \times U_3$ where

$$\begin{aligned} U_1 &= \text{TypedStore} \times \text{Val} \times \text{Type} \\ U_2 &= \text{TypedStore} \times \text{Val} \times \text{TypeScheme} \\ U_3 &= \text{TypedStore} \times \text{Env} \times \text{TyEnv}. \end{aligned}$$

Whenever A is a set, we write $P(A)$ for the set of subsets of A. Then F is a map $F : P(U) \to P(U)$. For every $Q \subseteq U$, let $Q_i = \pi_i(Q)$ be the ith projection of Q, $i = 1, 2, 3$, and let $F(Q) = F_1(Q) \times F_2(Q) \times F_3(Q)$, where

$F_1(Q) = \{(s : ST, v, \tau) \mid$
 if $v = b$ then $(v, b) \in R_{\text{Bas}}$;
 if $v = [x, e_1, E]$ then there exists a TE such that
  $(s : ST, E, TE) \in Q_3$ and $TE \vdash \lambda x.e_1 : \tau$;
 if $v = asg$ then $\tau = \tau_1\ ref \to \tau_1 \to stm$ for some τ_1;
 if $v = ref$ then $\tau = \theta \to \theta\ ref$ for some imperative θ;
 if $v = deref$ then $\tau = \tau_1\ ref \to \tau_1$ for some τ_1;
 if $v = a$ then $\tau = (ST(a))\ ref$ and $(s : ST, s(a), ST(a)) \in Q_1\}$

$F_2(Q) = \{(s : ST, v, \sigma) \mid \forall \tau < \sigma,\ (s : ST, v, \tau) \in Q_1\}$

$F_3(Q) = \{(s : ST, E, TE) \mid$
$\quad \mathrm{Dom}\, E = \mathrm{Dom}\, TE \text{ and } \forall x \in \mathrm{Dom}\, E, (s : ST, E(x), TE(x)) \in Q_2\}.$

It is crucial that F is monotonic (i.e., that $Q \subseteq Q'$ implies $F(Q) \subseteq F(Q')$). This would not have been the case, had we taken the following, perhaps more natural, definition of F:

$F_1(Q) = \{(s : ST, v, \tau) \mid$
$\quad \ldots$
$\quad$ if $v = [x, e_1, E]$ then $\tau = \tau_1 \to \tau_2$ and
$\qquad$ for all v_1, v_2, s'
$\qquad$ if $(s : ST, v_1, \tau_1) \in Q_1$ and $s, E \pm \{x \mapsto v_1\} \vdash e_1 \longrightarrow v_2, s'$
$\qquad$ then $\exists\ ST' \supseteq ST$ such that $(s' : ST', v_2, \tau_2) \in Q_1$
$\quad \ldots\}$

However, the chosen F is monotonic, so it has a smallest and a greatest fixpoint in the complete lattice $(P(U), \subseteq)$, namely

$$R^{\min} = \bigcap \{Q \subseteq U \mid F(Q) \subseteq Q\}$$

and

$$R^{\max} = \bigcup \{Q \subseteq U \mid Q \subseteq F(Q)\}. \tag{5.10}$$

For our particular F, the minimal fixpoint $R^{\min}$ is strictly contained in the maximal fixpoint $R^{\max}$ and it turns out that it is the latter we want. This is due to the possibility of cycles in the store as illustrated by the following example.

Example 5.5 Consider the evaluation of

```
let r= ref(λx.x+1)
in let s= ref(λy.(!r)y+2)
   in r:= !s
```

in the empty store. At the point just before "r:= !s" is evaluated, the store looks as follows

$$\{\ a_1 \mapsto [\mathtt{x}, \mathtt{x+1}, E_0],\quad a_2 \mapsto [\mathtt{y}, (\mathtt{!r})\mathtt{y+2}, E_0 \pm \{\mathtt{r} \mapsto a_1\}]\ \}$$

where E_0 is the initial environment; after the assignment the store becomes cyclic:

$$\begin{array}{ll} s' = \{ & a_1 \mapsto [\mathtt{y}, (\mathtt{!r})\mathtt{y+2}, E_0 \pm \{\mathtt{r} \mapsto a_1\}], \\ & a_2 \mapsto [\mathtt{y}, (\mathtt{!r})\mathtt{y+2}, E_0 \pm \{\mathtt{r} \mapsto a_1\}] \\ \} & \end{array}$$

Now we would expect to have $s' : ST' \models a_1 : (int \to int)\, ref$, where

$$ST' = \{a_1 \mapsto int \to int,\ a_2 \mapsto int \to int\}.$$

Indeed, if we let $q = (s' : ST', a_1, (int \to int)\, ref)$ then we do have $q \in R^{\max}$. To prove this it will suffice to find a Q with $q \in Q$ and $Q \subseteq F(Q)$ since we have (5.10). One such $Q = Q_1 \times Q_2 \times Q_3$ is

$$\begin{array}{rcl} Q_1 & = & \{(s' : ST', a_1, (int \to int)\, ref), \\ & & (s' : ST', [\mathtt{y}, (\mathtt{!r})\mathtt{y+2}, \{\mathtt{r} \mapsto a_1\}],\ int \to int)\} \\ Q_2 & = & \{(s' : ST', a_1, (int \to int)\, ref)\} \\ Q_3 & = & \{(s' : ST', \{\mathtt{r} \mapsto a_1\}, \{\mathtt{r} \mapsto (int \to int)\, ref\})\} \end{array}$$

where for simplicity I have ignored E_0 and the initial type environment.

As we shall see below, one can think of this Q as the *smallest consistent* set of typings containing q.

On the other hand, q is not in $R^{\min}$. This can be seen as follows. There is an alternative characterization of $R^{\min}$, namely

$$R^{\min} = \bigcup_{\lambda} F^{\lambda} \tag{5.11}$$

where

$$F^{\lambda} = F(\cup_{\mu<\lambda} F^{\mu}) \tag{5.12}$$

where λ ranges over all ordinals (see [1] for an introduction to inductive definitions). In other words, one obtains $R^{\min}$ by starting from the empty set and then applying F iteratively. It is easy to show that because q is cyclic there is no least ordinal λ such that $q \in F^{\lambda}$. Therefore $q \notin R^{\min}$. ∎

Let us try to explain informally why maximal fixpoints are often useful in operational semantics. Let us first review the essential differences between minimal and maximal fixpoints. In what follows, let U be a set and let F be a function $F : P(U) \to P(U)$ which is monotonic with respect to set inclusion.

Think of the elements of U as witnesses in a trial the judge of which is F. Of course witnesses may refer to each other so the question of the acceptance of a witness, q, may depend on the acceptance of other witnesses. For instance, in the case of the particular F we are considering , if $q = (s : ST, v, \sigma)$ then q is acceptable to F given Q, where $Q = Q_1 \times Q_2 \times Q_3$, if and only if

$$\{(s' : ST', v', \tau) \in U \mid s' : ST' = s : ST \wedge v' = v \wedge \tau < \sigma\} \subseteq Q_1.$$

There are two completely different ways of interpreting the set Q.

The first is to consider Q to be a set of witnesses that have been positively proved to speak the truth. Then $F(Q)$ are the witnesses that are proved to speak the truth as a logical consequence of the fact that the witnesses in Q speak the truth. The monotonicity of F implies that the more witnesses that are known to speak the truth, the more witnesses F can clear. Starting from $Q = \emptyset$, the set $F(\emptyset)$ consists of those witnesses that can be proved to speak the truth without reference to any other witnesses. For example we have $(s : ST, 3, \mathit{int}) \in F(\emptyset)$ for the F we consider. Next, $F(F(\emptyset))$ are the witnesses that can be proved to speak the truth because $F(\emptyset)$ speak the truth and so on. Using that F is monotonic, it can be shown that the (perhaps transfinite) limit of the chain

$$\emptyset \subseteq F(\emptyset) \subseteq F(F(\emptyset)) \subseteq \ldots$$

is a fixpoint of F, in fact the least fixpoint of F. In other words, q is in the minimal fixpoint of F if and only if it can eventually be proved that q speaks the truth.

As an example, if $U =$

{A:"On the day of the crime I was in Paris with B",
B:"On the day of the crime I was in Paris with A"}

and if, in order to prove claim of the form "X:On the day of the crime I was in Paris with Y", F proceeds by investigating all claims of the form "Y: ... " then F is never going to be able to prove either of the two claims.

The other way to interpret Q is to think of it as being the set of witnesses that F has not proved wrong. Then $F(Q)$ are the witnesses that cannot be proved wrong, given that the witnesses in Q cannot be proved wrong. The monotonicity of F now implies that the more witnesses that have been weeded out, the more witnesses F can reject. Starting from $Q = U$, meaning that at the outset F

rejects nothing, $F(U)$ is the set of witnesses that cannot be rejected because they refer to witnesses in U. For example, we have $(s : \mathit{ST}, 3, \mathit{int}) \in F(U)$ but $(s : \mathit{ST}, 3, \mathit{bool}) \notin F(U)$.

Using that F is monotonic we have

$$U \supseteq F(U) \supset F(F(U)) \supseteq \dots$$

and it can be proved that the (perhaps transfinite) limit of this chain is a fixpoint of F, in fact the maximal fixpoint of F. Thus q is in the maximal fixpoint of F if and only if F can never prove that q is wrong. In the international example above, both claims will be in the maximal fixpoint because A and B consistently claim that they were in Paris.

To see why the use of the word "truth" in the interpretation of minimal fixpoints is replaced by the word "consistency" in the interpretation of maximal fixpoints, let us consider the alternative characterization of the maximal fixpoint:

$$R^{\max} = \bigcup\{Q \subseteq U \mid Q \subseteq F(Q)\}.$$

For a set Q to have the property $Q \subseteq F(Q)$ means that for each $q \in Q$, q is acceptable to F perhaps by virtue of other witnesses, but only witnesses within Q. Each of these witnesses, in turn, are acceptable to F in the same sense. Thus, no matter how many iterations F embarks on, no new witnesses need be called.

This motivates the following definition: Q is *(F-) consistent* if $Q \subseteq F(Q)$.

To return to example 5.5 the Q we defined was the smallest F-consistent set containing q and, as such, contained in the largest F-consistent set there is, namely $R^{\max}$.

To sum up,

$$\begin{aligned} &\text{the maximal fixpoint of } F : U \to U \\ &= \text{the largest } F\text{-consistent subset of } U \\ &= \{q \in U \mid F \text{ can never reject } q\}. \end{aligned}$$

It should now be obvious that maximal fixpoints are of interest whenever one considers consistency properties, in particular when one wants to define relations between objects that are inherently infinite, although they may have a finite representation. In CCS [28], for example, *observation equivalence* of processes is defined as the maximal fixpoint of a monotonic operator. As yet another example, let us mention that the technique can be used to extend the soundness result of

Part I to a language that admits recursive functions represented in the dynamic semantics by finite representations of infinite closures. We shall now proceed to see how the technique is used to prove the soundness of the imperative type discipline.

Take $\models$ to be $R^{\max}$:

Definition 5.6 *We write*

$s : ST \models v : \tau$ *for* $(s : ST, v, \tau) \in R_1^{\max}$,
$s : ST \models v : \sigma$ *for* $(s : ST, v, \sigma) \in R_2^{\max}$, *and*
$s : ST \models E : TE$ *for* $(s : ST, E, TE) \in R_3^{\max}$.

With this definition the consistency result we conjectured earlier (Conjecture 3.4) is true. However a proof by induction on the depth of inference of $s, E \vdash e \longrightarrow v, s'$ will not work as we can see by considering any inference rule with more than one premise, for example

$$\frac{s, E \vdash e_1 \longrightarrow ref, s_1 \quad s_1, E \vdash e_2 \longrightarrow v_2, s_2 \quad a \notin \mathrm{Dom}\, s_2}{s, E \vdash e_1\ e_2 \longrightarrow a, s_2 \pm \{a \mapsto v_2\}} \tag{5.13}$$

with the corresponding typing rule

$$\frac{TE \vdash e_1 : \tau' \rightarrow \tau \quad TE \vdash e_2 : \tau'}{TE \vdash e_1\ e_2 : \tau} \tag{5.14}$$

By assumption $s : ST \models E : TE$ and by (5.13) and (5.14) we have $s, E \vdash e_1 \longrightarrow ref, s_1$ and $TE \vdash e_1 : \tau' \rightarrow \tau$. Hence by induction there exists an ST_1 such that

$$s_1 : ST_1 \models ref : \tau' \rightarrow \tau.$$

To apply induction a second time, this time on the second premise of (5.13) and (5.14), we need to know that $s_1 : ST_1 \models E : TE$; however, we just know $s : ST \models E : TE$.

What we need is that every typing that holds in the initial typed store, $s : ST$, still holds in the resulting typed store, $s_1 : ST_1$.

This motivates the following definition.

Definition 5.7 *A typed store* $s' : ST'$ succeeds *a typed store* $s : ST$*, written*

$$s : ST \sqsubseteq s' : ST',$$

if

1. $ST \subseteq ST'$
2. $\forall v, \tau \quad s : ST \models v : \tau \Longrightarrow s' : ST' \models v : \tau.$

As usual, $ST \subseteq ST'$ is short for

$$\mathrm{Dom}\, ST \subseteq \mathrm{Dom}\, ST' \text{ and for all } x \in \mathrm{Dom}\, ST,\ ST(x) = ST'(x).$$

Notice that if $s : ST \sqsubseteq s' : ST'$ and $\mathrm{Dom}\, s = \mathrm{Dom}\, s'$ then $ST = ST'$, since $\mathrm{Dom}\, ST = \mathrm{Dom}\, s = \mathrm{Dom}\, s' = \mathrm{Dom}\, ST'$.

The relation $\sqsubseteq$ is obviously reflexive and transitive. It is not antisymmetric. From Property 5.4 we immediately get

Lemma 5.8 *Assume* $s : ST \sqsubseteq s' : ST'$*. Then for all* v*,* σ*,* E*, and* TE

$$s : ST \models v : \sigma \quad \Longrightarrow \quad s' : ST' \models v : \sigma$$

and

$$s : ST \models E : TE \quad \Longrightarrow \quad s' : ST' \models E : TE\,.$$

The following lemmas are all used in the proof of the consistency result. Moreover, they are proved using an important proof technique regarding maximal fixpoints.

The first lemma will be needed for the case concerning creation of a reference. For brevity, let us say that $s' : ST'$ *expands* $s : ST$, written $s : ST \subseteq s' : ST'$, if $s \subseteq s'$ and $ST \subseteq ST'$.

Lemma 5.9 (Store Expansion) *If* $s : ST \subseteq s' : ST'$ *then* $s : ST \sqsubseteq s' : ST'$*.*

Proof. It will suffice to prove

$$\forall v, \tau \qquad s : ST \models v : \tau \Longrightarrow s' : ST' \models v : \tau.$$

To this end we define $Q = Q_1 \times Q_2 \times Q_3$, where

$$\begin{aligned} Q_1 &= \{(s' : ST', v, \tau) \mid s : ST \models v : \tau\} \\ Q_2 &= \{(s' : ST', v, \sigma) \mid s : ST \models v : \sigma\} \\ Q_3 &= \{(s' : ST', E, TE) \mid s : ST \models E : TE\} \end{aligned}$$

where $s : ST$ and $s' : ST'$ both are given.

The point is that it will suffice to prove $Q \subseteq F(Q)$, as

$$\models \;=\; R^{\max} \;=\; \bigcup\{Q \subseteq U \mid Q \subseteq F(Q)\}$$

is the largest set contained in its image under F.

First, take $q_1 = (s' : ST', v, \tau) \in Q_1$; then

$$s : ST \models v : \tau. \tag{5.15}$$

If $v = b$ then $(v, \tau) \in R_{\text{Bas}}$ by Property 5.4 on (5.15). Thus $q_1 \in F_1(Q)$ as desired.

Similarly for $v = asg$, ref, and $deref$.

If $v = [x, e_1, E]$ then by the Property of $\models$ on (5.15) there exists a TE such that $s : ST \models E : TE$ and $TE \vdash \lambda x.e_1 : \tau$. Thus $(s' : ST', E, TE) \in Q_3$. Thus $q_1 \in F_1(Q)$.

If $v = a$ then $\tau = (ST(a))\, ref$ and $s : ST \models s(a) : ST(a)$ by Property 5.4 on (5.15). Since $s : ST \sqsubseteq s' : ST'$ this means $\tau = (ST'(a))\, ref$ and $s : ST \models s'(a) : ST'(a)$, i.e., $(s' : ST', s'(a), ST'(a)) \in Q_1$. Hence $q_1 \in F_1(Q)$.

Next, take $q_2 = (s' : ST', v, \sigma) \in Q_2$. Thus

$$s : ST \models v : \sigma. \tag{5.16}$$

Assume $\tau < \sigma$. then $s : ST \models v : \tau$ by Property 5.4 on (5.16). Thus $(s' : ST', v, \tau) \in Q_1$. Thus $q_2 \in F_2(Q)$.

Finally, take $q_3 = (s' : ST', E, TE) \in Q_3$. Then $s : ST \models E : TE$; thus $\operatorname{Dom} E = \operatorname{Dom} TE$. Moreover, for all $x \in \operatorname{Dom} E$, we have $s : ST \models E(x) : TE(x)$, i.e., $(s' : ST', E(x), TE(x)) \in Q_2$. Thus $q_3 \in F_3(Q)$. ∎

The following lemma is used in the case concerning assignment (of the value v_0 to the address a_0).

Lemma 5.10 (Assignment) *Assume $s : ST \models v_0 : ST(a_0)$; let $s' = s \pm \{a_0 \mapsto v_0\}$. Then $s : ST \sqsubseteq s' : ST$.*

Note that if $s : ST \models v_0 : ST(a_0)$ then $a_0 \in \mathrm{Dom}\, ST = \mathrm{Dom}\, s$, so $s' = s \pm \{a_0 \mapsto v_0\}$ is obtained by overwriting s on an already existing address.

Proof. Define

$$\begin{aligned} Q_1 &= \{(s' : ST, v, \tau) \mid s : ST \models v : \tau\} \\ Q_2 &= \{(s' : ST, v, \sigma) \mid s : ST \models v : \sigma\} \\ Q_3 &= \{(s' : ST, E, TE) \mid s : ST \models E : TE\} \end{aligned}$$

where a_0, v_0, s, s', and ST all are given and satisfy $s : ST \models v_0 : ST(a_0)$ and $s' = s \pm \{a_0 \mapsto v_0\}$.

It will suffice to prove $Q \subseteq F(Q)$.

First, take $q_1 = (s' : ST, v, \tau) \in Q_1$. Then

$$s : ST \models v : \tau. \tag{5.17}$$

If $v = b$, *asg*, *ref*, or *deref* we immediately get $q_1 \in F(Q)$ from Property 5.4 on (5.17).

If $v = [x, e_1, E]$ then by (5.17), there exists a TE such that $s : ST \models E : TE$ and $TE \vdash \lambda x.e_1 : \tau$. Thus $(s' : ST, E, TE) \in Q_3$. Thus $q_1 \in F(Q)$.

If $v = a$ then by (5.17) we have $\tau = (ST(a))\, ref$ and $s : ST \models s(a) : ST(a)$. Since $s : ST \models v_0 : ST(a_0)$ this gives $s : ST \models s'(a) : ST(a)$. Thus $(s' : ST, s'(a), ST(a)) \in Q_1$. Thus $(s' : ST, a, (ST(a))\, ref) \in F_1(Q)$ i.e., $q_1 \in F_1(Q)$.

Next, take $q_2 = (s' : ST, v, \sigma) \in Q_2$. Then $s : ST \models v : \sigma$. Assume $\tau < \sigma$. Then $s : ST \models v : \tau$, so $(s' : ST, v, \tau) \in Q_1$. Thus $q_2 \in F_2(Q)$.

Finally, take $q_3 = (s' : ST, E, TE) \in Q_3$. Then $s : ST \models E : TE$. Thus $\mathrm{Dom}\, E = \mathrm{Dom}\, TE$ and for all $x \in \mathrm{Dom}\, E$, $s : ST \models E(x)\ .\ TE(x)$, i.e., $(s' : ST, E(x), TE(x)) \in Q_2$. Thus $q_3 \in F_3(Q)$. ∎

Finally, this lemma is crucial in the case regarding the let rule.

Lemma 5.11 (Semantic Substitution)
If $s : ST \models v : \tau$ *then* $s : S(ST) \models v : S(\tau)$ *for all substitutions* S.

Proof. Define

$$\begin{aligned}
Q_1 &= \{(s : ST', v, \tau') \mid \exists S, \tau \text{ s.t.} \\
&\quad S(ST) = ST' \wedge S(\tau) = \tau' \wedge s : ST \models v : \tau\} \\
Q_2 &= \{(s : ST', v, \sigma') \mid \exists S, \sigma \text{ s.t.} \\
&\quad S(ST) = ST' \wedge \sigma \xrightarrow{S} \sigma' \wedge s : ST \models v : \sigma\} \\
Q_3 &= \{(s : ST', E, TE') \mid \exists S, TE \text{ s.t.} \\
&\quad S(ST) = ST' \wedge TE \xrightarrow{S} TE' \wedge s : ST \models E : TE\}
\end{aligned}$$

where s, ST, and ST' are given.

It will suffice to prove $Q \subseteq F(Q)$.

First, take $q_1 = (s : ST', v, \tau') \in Q_1$. Let S and τ be such that

$$S(ST) = ST' \text{ and } S(\tau) = \tau' \text{ and } s : ST \models v : \tau. \tag{5.18}$$

If $v = b$ then $(v, \tau) \in R_{\text{Bas}}$ by Property 5.4 on (5.18). Thus $\tau \in \text{TyCon}$, so $\tau' = \tau$. Thus $q_1 \in F_1(Q)$.

If $v = asg$, then by (5.18) we have $\tau = \tau_1\ ref \to (\tau_1 \to stm)$ for some τ_1. Thus $\tau' = (S\,\tau_1)\ ref \to (S\,\tau_1 \to stm)$ showing $q_1 \in F_1(Q)$. Similarly for $v = deref$.

If $v = ref$ then $\tau = \theta \to \theta\ ref$ for some imperative type θ. Since substitutions are required to map imperative type variables to imperative types, we have that $S(\theta)$ is an imperative type. Thus $\tau' = S\,\theta \to (S\,\theta)\ ref$ showing $q_1 \in F_1(Q)$.

If $v = [x, e_1, E]$ then by (5.18) there exists a TE such that $s : ST \models E : TE$ and $TE \vdash \lambda x.e_1 : \tau$. There exists a TE' such that $TE \xrightarrow{S} TE'$. Thus $(s : ST', E, TE') \in Q_3$. Moreover, by Lemma 5.2 we have $TE' \vdash \lambda x.e_1 : S(\tau)$ i.e., $TE' \vdash \lambda x.e_1 : \tau'$. Thus $q_1 \in F_1(Q)$.

If $v = a$ then by (5.18) we have $\tau = (ST(a))\ ref$ and $s : ST \models s(a) : ST(a)$. Thus $\tau' = (ST'(a))\ ref$. Also, $S(ST(a)) = ST'(a)$, so $(s : ST', s(a), ST'(a)) \in Q_1$. Thus $(s : ST', a, (ST'(a))\ ref) \in F_1(Q)$, i.e., $q_1 \in F_1(Q)$.

Next, take $q_2 = (s : ST', v, \sigma') \in Q_2$. Let S and σ be such that $S(ST) = ST'$ and $\sigma \xrightarrow{S} \sigma'$ and $s : ST \models v : \sigma$. Assume $\tau_1' < \sigma'$.

Let $A = \text{tyvars}(ST) \cup \text{tyvars}(\sigma)$. Then by lemma 5.3 there exists a τ_1 and and S_1 such that $\sigma > \tau_1$, $S_1\tau_1 = \tau_1'$ and $S_1 \downarrow A = S \downarrow A$. In particular, $S_1(ST) = ST'$.

Since $s : ST \models v : \sigma$ and $\sigma > \tau_1$ we have $s : ST \models v : \tau_1$ by Property 5.4. This, together with $S_1(ST) = ST'$ and $S_1\tau_1 = \tau_1'$ gives $(s : ST', v, \tau_1') \in Q_1$. Since we can do this for every $\tau_1' < \sigma'$, we have $q_2 = (s : ST', v, \sigma') \in F_2(Q)$ as desired.

Finally, take $q_3 = (s : ST', E, TE') \in Q_3$. Let S and TE be such that $TE \xrightarrow{S} TE'$ and $S(ST) = ST'$ and $s : ST \models E : TE$. Then $\text{Dom}\, E = \text{Dom}\, TE$ and $\forall x \in \text{Dom}\, E$, $s : ST \models E(x) : TE(x)$ and $TE(x) \xrightarrow{S} TE'(x)$. Thus $\forall x \in \text{Dom}\, E$, $(s : ST', E(x), TE'(x)) \in Q_2$ showing that $q_3 = (s : ST', E, TE') \in F_3(Q)$. ■

5.3 The Consistency Theorem

We can now state the main theorem.

Theorem 5.12 (Consistency of Static and Dynamic Semantics)
If $s : ST \models E : TE$ and $TE \vdash e : \tau$ and $s, E \vdash e \longrightarrow v, s'$ then there exists an ST' with $s : ST \sqsubseteq s' : ST'$ and $s' : ST' \models v : \tau$.

The special case that is of basic concern is when the initial store is empty (hence $s = ST = \{\}$), the only free variables in e are `ref`, `!`, and `:=`, and that E and TE are the initial environments

$$
\begin{array}{lll@{\qquad}lll}
E_0(\texttt{ref}) & = & \mathit{ref} & TE_0(\texttt{ref}) & = & \forall u.u \to u\ \mathit{ref} \\
E_0(\texttt{!}) & = & \mathit{deref} & TE_0(\texttt{!}) & = & \forall t.t\ \mathit{ref} \to t\ \mathit{ref} \\
E_0(\texttt{:=}) & = & \mathit{asg} & TE_0(\texttt{:=}) & = & \forall t.t\ \mathit{ref} \to t \to \mathit{stm}
\end{array}
$$

Since we have $\{\} : \{\} \models E_0 : TE_0$ we have

Corollary 5.13 (Basic Soundness) *If $TE_0 \vdash e : \tau$ and $\{\} : E_0 \vdash e \longrightarrow b, s'$ then $(b, \tau) \in R_{\text{Bas}}$.*

where R_{Bas} still is the relation between basic values and types (relating 3 to *int*, *false* to *bool* etc.).

Proof. The proof of 5.12 is by induction on the depth of the dynamic evaluation. There is one case for each rule. The first case where one really sees the induction hypothesis at work is the one for application of a closure. The crucial case is of course the one for let expressions. Given the lemmas in the previous section, neither assignment nor creation of a new reference offers great resistance.

Variable, rule 3.1 Here $e = x \in \mathrm{Dom}\, E = \mathrm{Dom}\, TE$ and $v = E(x)$, $\tau < TE(x)$, and $s = s'$. Let $ST' = ST$. Then $s : ST \sqsubseteq s' : ST'$. Since $s' : ST' \models E : TE$ we have $s' : ST' \models E(x) : TE(x)$, and since $TE(x) < \tau$, Property 5.4 gives $s' : ST' \models v : \tau$ as desired.

Lambda Abstraction, rule 3.2 Here $e = \lambda x.e_1$ and $TE \vdash \lambda x.e_1 : \tau$ and $v = [x, e_1, E]$ and $s' = s$. Let $ST' = ST$. Then $s : ST \sqsubseteq s' : ST'$. Moreover, $s' : ST' \models [x, e_1, E] : \tau$ by Property 5.4.

Application of a Closure, rule 3.3 Here the situation is

$$\frac{TE \vdash e_1 : \tau' \rightarrow \tau \quad TE \vdash e_2 : \tau'}{TE \vdash e_1\ e_2 : \tau} \tag{5.19}$$

and

$$\frac{\begin{array}{l} s, E \vdash e_1 \longrightarrow [x_0, e_0, E_0], s_1 \\ s_1, E \vdash e_2 \longrightarrow v_2, s_2 \\ s_2, E_0 \pm \{x_0 \mapsto v_2\} \vdash e_0 \longrightarrow v, s' \end{array}}{s, E \vdash e_1\ e_2 \longrightarrow v, s'} \tag{5.20}$$

By induction on the first premises of (5.19) and (5.20) there exists an ST_1 such that $s : ST \sqsubseteq s_1 : ST_1$ and

$$s_1 : ST_1 \models [x_0, e_0, E_0] : \tau' \rightarrow \tau. \tag{5.21}$$

Thus $s_1 : ST_1 \models E : TE$ by Lemma 5.8. Using this with the second premises of (5.19) and (5.20) we get (by induction) an ST_2 such that $s_1 : ST_1 \sqsubseteq s_2 : ST_2$ and

$$s_2 : ST_2 \models v_2 : \tau'. \tag{5.22}$$

Now (5.21) together with $s_1 : ST_1 \sqsubseteq s_2 : ST_2$ gives

$$s_2 : ST_2 \models [x_0, e_0, E_0] : \tau' \rightarrow \tau.$$

Thus by Property 5.4 there exists a TE_0 such that

$$s_2 : ST_2 \models E_0 : TE_0 \tag{5.23}$$

and

$$TE_0 \vdash \lambda x_0.e_0 : \tau' \rightarrow \tau. \tag{5.24}$$

But (5.24) must be due to

$$TE_0 \pm \{x_0 \mapsto \tau'\} \vdash e_0 : \tau. \tag{5.25}$$

From (5.23) and (5.22) we get

$$s_2 : ST_2 \models E_0 \pm \{x_0 \mapsto v_2\} : TE_0 \pm \{x_0 \mapsto \tau'\} \tag{5.26}$$

We now use induction on (5.26), (5.25), and the third premise of (5.20) to get the desired ST'.

Notice that we could not have done induction on the depth of the type inference as we do not know anything about the depth of (5.24). Also note that the present definition of what it is for a closure to have a type (which almost was forced upon us because we needed F to be monotonic) now most conveniently provides the TE_0 for (5.23).

Assignment, rule 3.4 Here $e = (e_1\ e_2)\ e_3$ so the inferences must have been

$$\frac{TE \vdash e_1 : \tau'' \to (\tau' \to \tau) \quad TE \vdash e_2 : \tau''}{TE \vdash e_1\ e_2 : \tau' \to \tau} \tag{5.27}$$

$$\frac{TE \vdash e_1\ e_2 : \tau' \to \tau \quad TE \vdash e_3 : \tau'}{TE \vdash (e_1\ e_2)\ e_3\ :\ \tau} \tag{5.28}$$

$$\frac{\begin{array}{c} s, E \vdash e_1 \longrightarrow asg, s_1 \\ s_1, E \vdash e_2 \longrightarrow a, s_2 \\ s_2, E \vdash e_3 \longrightarrow v_3, s_3 \end{array}}{s, E \vdash (e_1\ e_2)\ e_3 \longrightarrow done, s_3 \pm \{a \mapsto v_3\}} \tag{5.29}$$

where $s' = s_3 \pm \{a \mapsto v_3\}$.

By induction on the first premises of (5.27) and (5.29) there exists a ST_1 such that $s : ST \sqsubseteq s_1 : ST_1$ and

$$s_1 : ST_1 \models asg : \tau'' \to (\tau' \to \tau).$$

By Property 5.4 we must have $\tau'' = \tau'\,ref$ and $\tau = stm$.

Now $s_1 : ST_1 \models E : TE$. By induction on the second premises of (5.27) and (5.29) we therefore get a ST_2 such that $s_1 : ST_1 \sqsubseteq s_2 : ST_2$ and $s_2 : ST_2 \models a : \tau'\,ref$.

Thus $s_2 : ST_2 \models E : TE$. By induction on the second premise of (5.28) and the third premise of (5.29) there exists an ST' such that $s_2 : ST_2 \sqsubseteq s_3 : ST'$ and $s_3 : ST' \models v_3 : \tau'$. Thus we have $s_3 : ST' \models a : \tau'\,ref$.

Thus we must have $\tau' = ST'(a)$ so $s_3 : ST' \models v_3 : ST'(a)$. Thus by the assignment lemma, Lemma 5.10, we have $s_3 : ST' \sqsubseteq s' : ST'$. Since $(done, stm) \in R_{\mathrm{Bas}}$ we have $s' : ST' \models done : stm$, i.e., the desired $s' : ST' \models v : \tau$.

Creation of a Reference, rule 3.5 Here

$$\frac{TE \vdash e_1 : \tau' \to \tau \quad TE \vdash e_2 : \tau'}{TE \vdash e_1\ e_2 : \tau}$$

$$\frac{s, E \vdash e_1 \longrightarrow ref, s_1 \quad s_1, E \vdash e_2 \longrightarrow v_2, s_2 \quad a \notin \mathrm{Dom}\, s_2}{s, E \vdash e_1\ e_2 \longrightarrow a, s_2 \pm \{a \mapsto v_2\}}$$

where $s' = s_2 \pm \{a \mapsto v_2\}$. By induction on the first premises there exists a ST_1 such that $s : ST \sqsubseteq s_1 : ST_1$ and $s_1 : ST_1 \models ref : \tau' \to \tau$. Thus by Property 5.4 we have $\tau = \tau'\,ref$ and τ and τ' are imperative types.

Now $s_1 : ST_1 \models E : TE$. Thus induction on the second premises gives an ST_2 such that $s_1 : ST_1 \sqsubseteq s_2 : ST_2$ and $s_2 : ST_2 \models v_2 : \tau'$.

Let $ST' = ST_2 \pm \{a \mapsto \tau'\}$. This makes sense since τ' is an imperative type! Since $a \notin \mathrm{Dom}\, s_2$ and $\mathrm{Dom}\, s_2 = \mathrm{Dom}\, ST_2$, we have $a \notin \mathrm{Dom}\, ST_2$. Thus $\mathrm{Dom}\, ST' = \mathrm{Dom}\, s'$ so $s' : ST'$ is a typed store and it clearly expands $s_2 : ST_2$. Thus by the expansion lemma, Lemma 5.9, we get $s_2 : ST_2 \sqsubseteq s' : ST'$. Hence $s' : ST' \models v_2 : \tau'$ i.e., $s' : ST' \models s'(a) : ST'(a)$ so $s' : ST' \models a : \tau'\,ref$ i.e., $s' : ST' \models v : \tau$.

Dereferencing, rule 3.6 Here

$$\frac{TE \vdash e_1 : \tau' \to \tau \quad TE \vdash e_2 : \tau'}{TE \vdash e_1\ e_2 : \tau}$$

$$\frac{s, E \vdash e_1 \longrightarrow deref, s_1 \quad s_1, E \vdash e_2 \longrightarrow a, s' \quad s'(a) = v}{s, E \vdash e_1\ e_2 \longrightarrow v, s'}$$

By induction of the first premises there exists an ST_1 such that $s : ST \sqsubseteq s_1 : ST_1$ and $s_1 : ST_1 \models deref : \tau' \to \tau$. Thus $\tau' = \tau\, ref$.

Now $s_1 : ST_1 \models E : TE$. Thus by induction on the second premises there is an ST' such that $s_1 : ST_1 \sqsubseteq s' : ST'$ and $s' : ST' \models a : \tau\, ref$. Thus $s : ST \sqsubseteq s' : ST'$ and $s' : ST' \models s'(a) : \tau$ i.e., $s' : ST' \models v : \tau$.

Let Expressions, rule 3.7 The dynamic evaluation is

$$\frac{s, E \vdash e_1 \longrightarrow v_1, s_1 \quad s_1, E \pm \{x \mapsto v_1\} \vdash e_2 \longrightarrow v, s'}{s, E \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \longrightarrow v, s'} \tag{5.30}$$

Now there are two subcases:

$\boxed{e_1 \text{ is expansive}}$ Then $TE \vdash e : \tau$ must have been inferred by

$$TE \vdash e_1 : \tau_1 \tag{5.31}$$

and

$$TE \pm \{x \mapsto \text{AppClos}_{TE}\tau_1\} \vdash e_2 : \tau \tag{5.32}$$

for some τ_1, by rule 4.5. By induction on the first premise of (5.30) and (5.31) there exists an ST_1 such that $s : ST \sqsubseteq s_1 : ST_1$ and

$$s_1 : ST_1 \models v_1 : \tau_1 \tag{5.33}$$

Thus

$$s_1 : ST_1 \models E : TE \tag{5.34}$$

Bearing in mind that we have (5.32), we now want to strengthen (5.33) to

$$s_1 : ST_1 \models v_1 : \text{AppClos}_{TE}\tau_1 \tag{5.35}$$

So take any $\tau < \text{AppClos}_{TE}\tau_1$. Any bound variable in $\text{AppClos}_{TE}\tau_1$ is applicative, so it does not occur in ST_1, simply because store typings by definition cannot contain applicative type variables. Thus $\tau < \text{AppClos}_{TE}\tau_1$ ensures the existence of a substitution S such that $S(ST_1) = ST_1$ and $S(\tau_1) = \tau$. Thus, when we apply the semantic substitution lemma, Lemma 5.11, on (5.33) we get

$$s_1 : ST_1 \models v_1 : \tau \tag{5.36}$$

Since (5.36) holds for arbitrary $\tau < \text{AppClos}_{TE}\tau_1$ we have proved (5.35), c.f. Property 5.4.

Then (5.34) and (5.35) give

$$s_1 : ST_1 \models E \pm \{x \mapsto v_1\} : TE \pm \{x \mapsto \text{AppClos}_{TE}\tau_1\} \tag{5.37}$$

Applying induction on the second premise of (5.30) and (5.37) and (5.32), we get an ST' such that $(s : ST \sqsubseteq) s_1 : ST_1 \sqsubseteq s' : ST'$ and $s' : ST' \models v : \tau$ as desired.

$\boxed{e_1 \text{ is non-expansive}}$ Then $\vdash e : \tau$ must have been inferred from

$$TE \vdash e_1 : \tau_1 \tag{5.38}$$

$$TE \pm \{x \mapsto \mathrm{Clos}_{TE}\tau_1\} \vdash e_2 : \tau \tag{5.39}$$

for some τ_1 by application of rule 4.4.

Let $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{tyvars}\,\tau_1 \setminus \mathrm{tyvars}\,TE$. Then $\mathrm{Clos}_{TE}\tau_1 = \forall\alpha_1 \ldots \alpha_n.\tau_1$. Let $\{u_1, \ldots, u_m\}$ be the imperative type variables among $\{\alpha_1, \ldots, \alpha_n\}$. Let $\{u'_1, \ldots, u'_m\}$ be imperative type variables such that $R = \{u_i \mapsto u'_i \mid 1 \leq i \leq m\}$ is a bijection and

$$\mathrm{Rng}\,R \cap \mathrm{imptyvars}\,ST = \emptyset \tag{5.40}$$

$$\mathrm{Rng}\,R \cap \mathrm{tyvars}\,TE = \emptyset \tag{5.41}$$

Now $TE \xrightarrow{R} TE$ as no u_i is free in TE, so the substitution lemma, Lemma 5.2 applied to (5.38) gives

$$TE \vdash e_1 : R\,\tau_1 \tag{5.42}$$

Moreover, $\mathrm{Clos}_{TE}\tau_1 \underset{\alpha}{=} \mathrm{Clos}_{TE}(R\,\tau_1)$ by (5.41) so from (5.39) we get

$$TE \pm \{x \mapsto \mathrm{Clos}_{TE}(R\,\tau_1)\} \vdash e_2 : \tau \tag{5.43}$$

by using lemma 5.2 on the identity substitution. Applying induction to the first premises of (5.30) and (5.42) we get an ST_1 such that $s : ST \sqsubseteq s_1 : ST_1$ and

$$s_1 : ST_1 \models v_1 : R\,\tau_1 \tag{5.44}$$

Since e_1 is non-expansive, we have $\mathrm{Dom}\,s = \mathrm{Dom}\,s'$ — *and this is the crucial property of non-expansive expressions.* Since $s : ST \sqsubseteq s_1 : ST_1$ we have $ST_1 = ST$ (recall the definition of $\sqsubseteq$ and note that $\mathrm{Dom}\,ST = \mathrm{Dom}\,s = \mathrm{Dom}\,s' = \mathrm{Dom}\,ST'$). Thus

$$s_1 : ST \models E : TE \tag{5.45}$$

and, by (5.44)

$$s_1 : ST \models v_1 : R\,\tau_1. \tag{5.46}$$

Bearing (5.43) in mind we want to strengthen (5.46) to

$$s_1 : ST \models v_1 : \mathrm{Clos}_{TE}(R\,\tau_1). \tag{5.47}$$

So take any $\tau < \mathrm{Clos}_{TE}(R\,\tau_1)$. No variable α bound in $\mathrm{Clos}_{TE}(R\,\tau_1)$ can occur in ST, either because α is applicative or because of (5.40) — this is precisely why we do the renaming.

Hence $\tau < \mathrm{Clos}_{TE}(R\,\tau_1)$ implies the existence of a substitution S with $S(ST) = ST$ and $S(R\,\tau_1) = \tau$. We now apply the semantic substitution lemma, Lemma 5.11, to (5.46) to obtain

$$s_1 : ST \models v_1 : \tau \tag{5.48}$$

Since (5.48) holds for every $\tau < \mathrm{Clos}_{TE}(R\,\tau_1)$, we have proved (5.47), c.f. Property 5.4.

From (5.45) and (5.47) we then get

$$s_1 : ST \models E \pm \{x \mapsto v_1\} : TE \pm \{x \mapsto \mathrm{Clos}_{TE}(R\,\tau_1)\} \tag{5.49}$$

Finally we apply induction to (5.49), the second premise of (5.30), and to (5.43) to get the desired ST'. ■

From the proof case concerned with non-expansive expressions in let expressions we learn that the important property of a non-expansive expression is that it does not expand the domain of the store. Because of the very simple way we have defined what it is for an expression to be non-expansive, non-expansive expressions will in fact leave the store unchanged. The proof shows that this is not necessary; assignments are harmless, only creation of new references is critical.[1]

In larger languages — ML, say — the class of syntactically non–expansive expressions is quite large. The class is closed under the application of many primitive functions, and under many constructions. For instance, if e_1 and e_2 are non–expansive, so is let x = e_1 in e_2.

As a corollary of the Consistency Theorem we have that *monomorphic references* in the original purely applicative type inference system (Section 3.2) are sound. To be more precise, let TE'_0 be as TE_0 except that $TE'_0(\texttt{ref}) = \forall\alpha.\alpha \to \alpha\ ref$, where α is *applicative*. (The initial environments TE_0 and E_0 were defined in connection with corollary 5.13). Moreover, let P' be a proof of $TE'_0 \vdash e : \tau$ in

[1]In retrospect, this explains why the type scheme for `ref` has a bound imperative type variable, while the type schemes for `:=` and `!` are purely applicative.

the *applicative* system where P' is special in the sense that the type scheme for `ref` is always instantiated to a monotype (i.e., a type without any type variables at all). Then there is a proof P of $TE_0 \vdash e : \tau$ in the modified inference system. Hence corollary 5.13 gives the basic soundness of the special applicative type inference.

Chapter 6

Comparison with Damas' Inference System

The purpose of this chapter is to compare our system with Damas' system, presented in Chapter III of his Ph. D. thesis [13]. I shall first try to explain his inference system. Sadly, the inventor himself economized hard on motivations and explanations, so let the reader be warned that my interpretations might not be consistent with what he had in mind.

Having described his system, as I understand it, I shall give examples of its use. I informally conjecture that every expression that I can type, he can type; the converse is not true. Finally, I shall discuss the pragmatics of the two systems.

6.1 The Inference System

First, *types* are defined by

$$\tau \ ::= \ \iota \mid \alpha \mid \tau\ \mathit{ref} \mid \tau_1 \rightarrow \tau_2$$

where ι ranges over primitive types and α ranges over one universal set of type variables, $\{\mathrm{t}, \mathrm{t}_1, \ldots\}$.

Next, *type schemes*, η are defined by

$$\eta \ ::= \ \tau \mid \tau_1 \rightarrow \tau_2 * \Delta \mid \forall \alpha . \eta$$

where Δ ranges over finite sets of types. Thus

$$\eta = \forall \alpha_1 \ldots \alpha_n . \tau_1 \rightarrow \tau_2 * \Delta \tag{6.1}$$

is a type scheme. The quantification binds over both $\tau_1 \rightarrow \tau_2$ and Δ. The set Δ should be thought of as an addendum to the whole of $\tau_1 \rightarrow \tau_2$, not just to τ_2.

Roughly speaking, a function has the type (6.1) if it has the polymorphic type $\forall\alpha_1 \ldots \alpha_n.(\tau_1 \to \tau_2)$ and in addition Δ is an upper bound for the types of objects to which new references will be created as a side-effect of *applying* the function. For example,

$$TE_0(\texttt{ref}) = \forall \mathrm{t}.\mathrm{t} \to \mathrm{t}\ \mathit{ref} * \{\mathrm{t}\} \tag{6.2}$$

where TE_0 as usual means the initial type environment.

Type environments map variables to type schemes of the above kind. The type inference rules allow us to infer sequents of the form

$$\boxed{TE \vdash e : \eta * \Delta.} \tag{6.3}$$

Since η can be of the form $\forall\alpha_1 \ldots \alpha_n.\tau_1 \to \tau_2 * \Delta'$ one can infer sequents of the form

$$TE \vdash e : (\forall\alpha_1 \ldots \alpha_n.\tau_1 \to \tau_2 * \Delta') * \Delta. \tag{6.4}$$

This device of nested sets of types is extremely important and provides a generalization of my rough classification of "expansive" versus "non-expansive" expressions. The rough idea in (6.3) is that Δ contains the types of all objects to which new references may be created as a side-effect of evaluating e. In addition, in (6.4), the set Δ' contains the types of all objects to which new references will be created when e is *applied* to an argument.

As an example, we will have

$$TE_0 \vdash \texttt{ref} : (\forall \mathrm{t}.\mathrm{t} \to \mathrm{t}\ \mathit{ref} * \{\mathrm{t}\}) * \emptyset$$

Notice that $\Delta = \emptyset$ since simply *mentioning* `ref` does not create any references; the *application* of `ref`, however, will create a reference, so Δ' is non-empty.

This motivates the inference rule for variables:

$$\frac{TE(x) = \eta}{TE \vdash x : \eta * \emptyset} \tag{6.5}$$

The rule for lambda abstraction is

$$\frac{TE \pm \{x \mapsto \tau'\} \vdash e_1 : \tau * \Delta}{TE \vdash \lambda x.e_1 : (\tau' \to \tau * \Delta) * \emptyset} \tag{6.6}$$

Notice the outer $\emptyset$ in the conclusion; the evaluation of a lambda abstraction does not create any references. Also notice that in the premise the inferred type scheme must take the special form τ i.e., it cannot have a set of types associated with it. One could imagine a rule

$$\frac{TE + \{x \mapsto \tau'\} \vdash e_1 : (\tau * \Delta') * \Delta}{TE \vdash \lambda x.e_1 : (((\tau' \to \tau) * \Delta') * \Delta) * \emptyset}$$

and so on for arbitrary levels of nesting. The present rule, however, does not distinguish between whether it is the application of the function itself, or, in the case where the result of the application is a function, the application of the resulting function that creates the reference. For example, with the present rule we will have

$$TE \vdash \lambda\text{x.let y=ref x in } \lambda\text{z.z} : \eta * \emptyset$$

and also

$$TE \vdash \lambda\text{x.}\lambda\text{z.let y= ref x in z} : \eta * \emptyset$$

where $\eta = \forall \mathrm{t}_1, \mathrm{t}_2.\mathrm{t}_1 \rightarrow \mathrm{t}_2 \rightarrow \mathrm{t}_2 * \{\mathrm{t}_1\}$.

To suggest some terminology, we might call the types in the outer Δ-set for the *immediate store types* and call the types in the inner Δ-set the *future store types.*

The generalization rule is

$$\frac{TE \vdash e : \eta * \Delta \qquad \alpha \notin \text{tyvars}\, TE \cup \text{tyvars}\, \Delta}{TE \vdash e : (\forall \alpha.\eta) * \Delta} \tag{6.7}$$

Hence, if η has the form $\tau_1 \rightarrow \tau_2 * \Delta'$ then α may be free in Δ', the future store types, but it must not be free in Δ, the immediate store types. This makes sense following the discussion in Section 3.2 where it was demonstrated that it is the immediate extension of the present store typing that can make generalization unsound.

Next, the rule for application is

$$\frac{TE \vdash e_1 : (\tau' \rightarrow \tau) * \Delta \quad TE \vdash e_2 : \tau' * \Delta}{TE \vdash e_1\ e_2 : \tau * \Delta} \tag{6.8}$$

Note that the type inferred for e_1 has no future store types, only the immediate store types. The point is that store types associated with e_1 that have hitherto been considered as belonging to the future are forced to be regarded as immediate now that we apply e_1.

The rule for let expressions is

$$\frac{TE \vdash e_1 : \eta_1 * \Delta \quad TE \pm \{x \mapsto \eta_1\} \vdash e_2 : \eta * \Delta}{TE \vdash \text{let } x = e_1 \text{ in } e_2 : \eta * \Delta} \tag{6.9}$$

Since η_1 can have quantified type variables, the assignment of η to x allows polymorphic use of x within e_2. In contrast to our system no explicit closure operation is applied in the second premise. Instead, the generalization rule (6.7)

may be applied repeatedly to quantify all variables that are not in Δ and not free in TE.

Also notice that η_2 is a type *scheme* whereas in the purely applicative system the result was a type. This makes a difference in Damas' system because only type schemes can contain store types. Thus we can infer for instance

$$\frac{TE_0 \vdash 7 : int * \emptyset \qquad TE_0 + \{x \mapsto int\} \vdash \lambda \mathtt{y.!(ref\ y)} : (\mathtt{t} \to \mathtt{t} * \{\mathtt{t}\}) * \emptyset}{TE_0 \vdash \mathtt{let\ x=\ 7\ in} \lambda \mathtt{y.!(ref\ y)} : (\mathtt{t} \to \mathtt{t} * \{\mathtt{t}\}) * \emptyset} \tag{6.10}$$

where t subsequently can be bound because it is a future store type only.

The distinction between immediate and future store types is finer than the distinction between expansive and non-expansive expressions (see Section 4.1). That variables and lambda abstraction are non-expansive corresponds to the set of immediate store types being empty in rules (6.5) and (6.6). However, as we saw in (6.10), the property of having an empty set of immediate store types can be synthesized from subexpressions. This is true even when applications are involved; for example we have $TE_0 \vdash e : (\mathtt{t} \to \mathtt{t} * \{\mathtt{t}\}) * \emptyset$, where

$$e = \mathtt{let\ x=(\lambda x.x)1\ in\ \lambda y.!(ref\ y)}. \tag{6.11}$$

When doing proofs in Damas' system, one will seek to partition the store types in such a way that as many of them as possible belong to the future so that one can get maximal use of the generalization rule. However, in typing an application, future store types are being forced to be considered immediate. The inference rule that allows this transition is the instantiation rule,

$$\frac{TE \vdash e : \eta * \Delta \qquad \eta * \Delta > \eta' * \Delta'}{TE \vdash e : \eta' * \Delta'} \tag{6.12}$$

because η and η' may take the forms

$$\begin{aligned} \eta &= \forall \alpha_1 \ldots \alpha_n . \tau_1 \to \tau_2 * \Delta_0 \\ \eta' &= \forall \beta_1 \ldots \beta_m . \tau_1' \to \tau_2' \end{aligned}$$

The relation $\eta * \Delta > \eta' * \Delta'$ is defined in terms of type scheme instantiation, $\eta > \eta'$, which in turn is defined as follows[1] (here θ ranges over non-quantified type schemes, i.e., terms of the form τ or $\tau_1 \to \tau_2 * \Delta$):

[1]Taken from [13], Page 95

Definition. We will say that a type scheme $\eta' = \forall\beta_1 \ldots \beta_m.\theta'$ is a *generic instance* of another type scheme $\eta = \forall\alpha_1 \ldots \alpha_n.\theta$, and write $\eta > \eta'$, iff the β_j do not occur free in $\forall\alpha_1 \ldots \alpha_n.\theta$ and there are types $\tau_1, \ldots, \tau_n$ such that either

1. both θ and θ' are types and $\theta' = [\tau_i/\alpha_i]\theta$
2. θ is $\tau' \to \tau * \Delta$, θ' is $\nu' \to \nu * \Delta'$, $\nu' \to \nu = [\tau_i/\alpha_i](\tau' \to \tau)$ and $[\tau_i/\alpha_i]\Delta$ is a subset of Δ'.

Here (1) corresponds to type scheme instantiation in the purely applicative inference system. In (2), the condition $[\tau_i/\alpha_i]\Delta \subseteq \Delta'$ corresponds to my requirement that substitutions map imperative type variables to imperative types.

Then the relation $\eta * \Delta > \eta' * \Delta'$ is defined as follows:[2]

Definition. Given two terms $\eta * \Delta$ and $\eta' * \Delta'$ we will write $\eta * \Delta > \eta' * \Delta'$ iff Δ is a subset of Δ' and either

1. $\eta > \eta'$
2. η is $\forall\alpha_1 \ldots \alpha_n.\tau' \to \tau * \Delta''$, η' is $\forall\beta_1 \ldots \beta_m.\nu$ and, assuming the β_j do not occur free in η nor in Δ', there are types $\tau_1, \ldots, \tau_n$ such that $\nu = [\tau_i/\alpha_i](\tau' \to \tau)$ and $[\tau_i/\alpha_i]\Delta''$ is a subset of Δ'.

Note that the requirement $\Delta \subseteq \Delta'$ allows us to increase the set of immediate store types at any point in the proof. This will make generalization harder, of course, but it may be needed to get identical sets of immediate store types in the premises of rules (6.8) and (6.9).

It is condition (2) in the above definition that allows the transition from future store types to immediate store types. For example, we have

$$\begin{aligned}(\forall \mathrm{t}.\mathrm{t} \to \mathrm{t}\ \mathit{ref} * \{\mathrm{t}\}) * \emptyset &> (\forall \mathrm{t}.\mathrm{t} \to \mathrm{t}\ \mathit{ref}) * \{\mathrm{t}\} \\ &> (\mathrm{t}\ \mathit{list} \to \mathrm{t}\ \mathit{list}\ \mathit{ref}) * \{\mathrm{t}\ \mathit{list}\}\end{aligned}$$

and

$$(\forall \mathrm{t}.\mathrm{t}\ \mathit{list}) * \emptyset > \mathrm{t}\ \mathit{list} * \{\mathrm{t}\ \mathit{list}\}$$

Hence, having used the instantiation rule twice we can conclude

$$\frac{TE_0 \vdash \texttt{ref} : \mathrm{t}\ \mathit{list} \to \mathrm{t}\ \mathit{list}\ \mathit{ref} * \{\mathrm{t}\ \mathit{list}\} \quad TE_0 \vdash \texttt{nil} : \mathrm{t}\ \mathit{list} * \{\mathrm{t}\ \mathit{list}\}}{TE_0 \vdash \texttt{ref nil} : \mathrm{t}\ \mathit{list}\ \mathit{ref} * \{\mathrm{t}\ \mathit{list}\}}$$

in which the unsound generalization on t is prevented.

Damas' inference rules translated into our notation are repeated in Figure 6.1. At this point, I hope that the reader agrees that Damas' system looks plausible. At the same time it is subtle enough that the question of soundness is difficult.

[2] From [13], page 96

$$\frac{TE(x) = \eta}{TE \vdash x : \eta * \emptyset}$$

$$\frac{TE \vdash e : \eta * \Delta \quad \eta * \Delta > \eta' * \Delta'}{TE \vdash e : \eta' * \Delta'}$$

$$\frac{TE \vdash e : \eta * \Delta \qquad \alpha \notin \text{tyvars}\, TE \cup \text{tyvars}\, \Delta}{TE \vdash e : (\forall \alpha.\eta) * \Delta}$$

$$\frac{TE \vdash e_1 : (\tau' \to \tau) * \Delta \quad TE \vdash e_2 : \tau' * \Delta}{TE \vdash e_1\ e_2 : \tau * \Delta}$$

$$\frac{TE \pm \{x \mapsto \tau'\} \vdash e_1 : \tau * \Delta}{TE \vdash \lambda x.e_1 : (\tau' \to \tau * \Delta) * \emptyset}$$

$$\frac{TE \pm \{f \mapsto \tau' \to \tau, x \mapsto \tau'\} \vdash e_1 : \tau * \Delta}{TE \vdash \text{rec}\ f\ x.e_1 : (\tau' \to \tau * \Delta) * \emptyset}$$

$$\frac{TE \vdash e_1 : \eta_1 * \Delta \quad TE \pm \{x \mapsto \eta_1\} \vdash e_2 : \eta * \Delta}{TE \vdash \text{let}\ x = e_1\ \text{in}\ e_2 : \eta * \Delta}$$

Figure 6.1: Damas' Inference Rules

6.2 Comparison

There are expressions that Damas can type, but I cannot. An example is

$$\texttt{let f=}\ e\ \texttt{in(f 1 ;f true)}$$

where e is the expression from (6.11). In Damas' system we get

$$TE_0 \vdash e : (\forall \mathrm{t}.\mathrm{t} \to \mathrm{t} * \{\mathrm{t}\}) * \emptyset$$

so that `f` can be used polymorphically, while in my system

$$TE_0 \vdash e : u \to u$$

where u cannot be bound since e is considered to be expansive.

Moreover, I suggest that every expression typable in my system is typable in Damas' system. I have not made a serious attempt to construct the embedding. The naive idea would be that a type τ in my system corresponds to a term $\hat{\tau} * \Delta$ where $\mathrm{tyvars}\,\Delta \cap \mathrm{tyvars}\,\hat{\tau}$ corresponds to $\mathrm{imptyvars}\,\tau$. Unfortunately, this does not work because terms of the form $\tau * \Delta$ are not types according to Damas' definition. Only type schemes, indeed only function type schemes can contain store types. In fact, Damas claims that one cannot include the store types in the types:[3]

> "The inclusion of terms of the form
>
> $$\tau' \to \tau * \Delta$$
>
> among types, which would be the natural thing to do according to the discussion above, would preclude the extension of the type assignment algorithm of the previous chapter to this extended type inference system. This is so because the algorithm relies heavily on the properties of the unification algorithm and the latter cannot be extended to unify terms involving sets of types while preserving those properties."

However, the current ML implementation of Damas' ideas, due, I believe, to Luca Cardelli, in effect includes store types among types. Every type variable is either *weak* or *strong*. A weak type variable corresponds to a variable that occurs free in a store type; if it is free in a future store type only, then it is a *generic* weak type variable, otherwise it is a *non-generic* weak type variable. Since types

[3] See [13] page 90–91

can contain a mixture of strong and weak type variables, store types have in effect been included among types.

As for unification, if one unifies a weak α against a type τ in which α does not occur, then the unifying substitution is $\{\alpha \mapsto R\,\tau\} \cup R$ where R is a substitution mapping the strong type variables occurring in τ to new weak type variables. (In the imperative version of the type checker, each variable has a bit for weak/strong and the effect of applying R is achieved by overwriting this bit). It is true that when the unification algorithm has to choose new type variables, the unifier, S_0, it produces cannot be mediating for *every* unifier, S, because S may have a different idea of what should happen to the new type variables. Perhaps this is the difficulty Damas refers to. But if one wants to be precise, the proof of the completeness of the type checker must account for what it is for a type variable to be new and I claim that all that is needed of S_0 is that it is mediating for all other unifiers on the domain of *used* variables. The proof of the completeness of the signature checker in Part III should give an idea of how the proof would go.

Even though Damas' system admits programs that mine rejects, there are still sensible programs that cannot be typed by Damas. A good example is the `fold` function from Example 4.5. The problem is that Damas' system too is vulnerable to curried functions or, more generally, lambda abstractions within lambda abstractions.

One idea to overcome this problem, due to David MacQueen, is to associate a natural number,the rank, say, with each type variable. Moreover, type inference is relative to the current level of lambda nesting, n, say. Intuitively, immediate store type variables correspond to type variables of rank n; the type variables with rank $n+1$ range over types that will enter the store typing after one application of the current expression, and so on. An applicative type variable can then be represented as a variable with rank ∞. It would be interesting to see the rules written down and their soundness investigated using the method of the previous chapter.

6.3 Pragmatics

Obviously, there is a tension between the desire for very simple rules that we can all understand and very advanced rules that admit as many sensible programs as

possible. In the one end of this spectrum is the system I propose – it is very easy to explain what it is for an expression to be non-expansive and, judging from the examples given, the system allows a fairly wide range of programs to be typed.

Then comes Damas' system. I am not convinced that the extra complexity it offers is justified by the larger class of programs it admits. If one wants to be clever about lambda nesting, then one might as well take the full step and investigate MacQueen's idea further.

Part III

A Module Language

Chapter 7

Typed Modules

The type disciplines in the previous parts are suitable when programs are compiled in their entirety. A small program can simply be recompiled every time it is changed, but this becomes impractical for larger programs. Many languages provide facilities for breaking large programs into relatively independent *modules.* A module can then be compiled separately provided its *interface* to its surroundings is known.

Sometimes modules are defined not as the units of separate compilation but rather as a means of delimiting scope and protecting "private" data from the surroundings. In any case, a module is typically a collection of components, where a component can be a value, a variable, a function, a procedure or a data type. Other terms for "module" are "package" [41], "object" [2] and "structure" [25].

The notion of interface varies from language to language, but typically an interface lists names together with some specification involving these names. A module, M, *matches* an interface, I, if M has the components specified in I and if these components satisfy the specifications in I. Other terms for "interface" are "package specification" [41], "class" [2] and "signature"[25].

One often wants to write and compile a module, M_2, which depends on another module, M_1, before M_1 is written. In that case the meaning of M_2 is relative to some assumptions about the properties of M_1. Let us assume that M_2 can access M_1 as specified by the interface I_1. Then we want to be able to write an *abstraction* of the form $\lambda M_1 : I_1 \,.\, M_2$. Later, we can define an actual argument module, M, and perform the application

$$(\lambda M_1 : I_1 \,.\, M_2)(M) \tag{7.1}$$

to create a module in which those components of M_2 that stem from the formal

parameter M_1 are replaced by the corresponding components of M. The notion of parameterized module is found in Simula [2], Ada (generic package) [41] and CLU [23], and ML (functors) [25], among others.

Seeing things this way, it becomes obvious that interfaces play the role of types, that relative independence of modules can be achieved by abstraction on a typed formal parameter, and that "linking" corresponds to substitution.

An interface has now become a syntactic object. It can be used as a filter to hide implementation details of an existing module. Alternatively, one can write all the interfaces of a system before one writes the first real module.[1] When the "type" of module is easier to grasp than the module itself (and it had better be!) working with interfaces can ease design and communication between people.

Corresponding to the notion of type checking we get the notion of "interface checking". For instance, in (7.1) it is highly desirable that the machine can check that M really does match I_1. As a bonus, if M matches I_1, then we are sure that the linking from M_2 to M is meaningful.

There is an unpleasant ring to the distinction between "small" and "big" programs. "What's next?", one feels like asking. As I have tried to indicate above many of the terms that are used on the modules level are just other names for things we are familiar with at the level of individual modules. The work by Burstall and Lampson on the language Pebble [6] shows that one can collapse the two layers into one general language. In Pebble one can have records that can contain types as well as more traditional values. Such a record can be viewed as a module. The type of such a record is a *dependent type*. (A type component of a record can be significant for the type of other components). More complex "modules" can be built using abstraction and application. Dependent sum and dependent product suffice to give strong expressive power.

It is desirable that interface checking can be done at compile time rather than as a part of the execution so that one can make sure that modules at least fit together in a meaningful way before they are executed.

Large programs contain many modules and each module may be known under different names to other modules. Therefore it is necessary to be able to specify *sharing*. In fact, when we are concerned with statically typed languages, sharing

[1]A textbook on top–down design would no doubt advocate the latter, but when we consider the maintenance of a large system it seems quite reasonable that existing modules influence subsequent interfaces.

constraints may be necessary to assist the type checker. For example in

$$\lambda M_1 : I_1 \, . \, M_2$$

if the interface I_1 postulates two modules both containing a type called `stack` it might well be that the body of M_2 makes sense only if the two stack types are assumed to be identical. Thus one would find a sharing specification, perhaps something like

sharing `M.stack` $=$ `M'.stack`

in I_1.

If by removing the distinction between small and big programs we obtain that modules can be handled as other values it becomes impossible to tell statically whether two modules (or types) are identical.

This leads to the idea of a stratified language: at the bottom the *core language* in which we write small programs; on top a *module language*. The module language may allow abstraction and application, but the operations on modules are kept simple enough that module equality and other properties we want to check statically remain decidable.

So the main motivation for having a stratified language is that we want more information about modules than ordinary type checking can give us.

The languages in Part I and II are typical core languages. We shall now present a module language which has a type discipline that handles sharing and is very similar to the previous polymorphic type disciplines.

Chapter 8

The Language ModL

Our module language is called ModL. It is a skeletal language originally studied as part of the work on giving an operational semantics of David MacQueen's modules proposal for ML [25] to single out the difficult semantics issues of the full language. Yet, ModL in no way depends on the ML core language, in fact the type discipline we shall present is completely independent of which core language is chosen.

ModL has *structures*, *signatures*, and *functors*. A structure corresponds to what we in the previous chapter called a module. In ML, a structure is a collection of components, where a component can be a datatype, a type, a value, a function, an exception, or even another structure. In ModL, however, we remove all components except the structures, so that we can study the structure – substructure relationship.

A *signature* corresponds to what we in the previous chapter called an interface. In ML, a signature can specify a datatype with its value constructors, a type (without saying what the type is), the types of values, functions and exceptions and the signatures of substructures. In addition one can specify sharing between two structures and between two type constructors. In ModL, however, signatures just give signatures of substructures and allow sharing specifications.

A *functor* is a map from structures to structures. (There is no significant difference between ML and ModL here). Functors do not take functors as arguments nor can they produce functors as results.

ModL is almost identical to the language we studied in [19], and we shall prove the theorems conjectured there. This is not the place for a theoretical investigation of the full ML semantics [20]. However, I do hope that what I have to say about ModL can explain some of the decisions we made when writing the

ML semantics.

We shall first define the syntax of ModL and then give a static semantics which enables one to infer what structures are created as a result of evaluating a structure expression so that sharing specifications and functor applications can be checked statically. We shall then show theorems about the semantics, in particular that there exists an algorithm, the *signature checker*, that finds so-called principal signatures of all legitimate signature expressions.

8.1 Syntax

Assume the following sets of identifiers:

strid	$\in$	StrId	structure identifiers
sigid	$\in$	SigId	signature identifiers
funid	$\in$	FunId	functor identifiers.

Substructures are accessed via *long* structure identifiers:

$$\textit{longstrid} \text{ or } \textit{strid}_1 \,.\, \ldots \,.\, \textit{strid}_k (k \geq 1) \;\in \text{LongStrId} = \text{StrId}^+.$$

The syntax of ModL appears in Figure 8.1

For the sake of the following examples imagine that we can declare and specify types, values and functions as well as structures.

dec	$\in$	Dec	declarations
strexp	$\in$	StrExp	structure expressions
spec	$\in$	Spec	specifications
sigexp	$\in$	SigExp	signature expressions
prog	$\in$	Prog	programs

dec	::=		empty
	|	**structure** *strid* = *strexp*	structure
	|	dec_1 dec_2	sequential
strexp	::=	**struct** *dec* **end**	generative
	|	*longstrid*	
	|	*funid*(*strexp*)	functor application
spec	::=		empty
	|	**structure** *strid* : *sigexp*	structure
	|	$spec_1$ $spec_2$	sequential
	|	**sharing** $longstrid_1$ = $longstrid_2$	sharing
sigexp	::=	**sig** *spec* **end**	basic
	|	*sigid*	
prog	:=	*dec*	top level declaration
	|	**signature** *sigid* = *sigexp*	signature declaration
	|	**functor** *funid*(*strid* : *sigexp*)=*strexp*	functor declaration
	|	$prog_1$ $prog_2$	sequential program

Figure 8.1: The Syntax of ModL.

Example 8.1 Here is a structure, a signature, and a functor;

```
structure IntStack=
 struct
   type elt= int
   val stack= ref[ ]
   fun push(x: elt)= ...
   fun pop( ... )= ...
 end

signature StackSig=
 sig
   type elt
   val stack: elt list ref
   fun push: elt-> ...
   fun pop: ...
 end

functor MkStack(A: sig type elt end)=
 struct
   val stack= ref [ ]
   fun push(x: A.elt)= ...
   fun pop( ... )= ...
 end

structure IntStack= MkStack (struct type elt= int
                                    ...
                                    end)
```

The first IntStack is an example of a basic structure expression which is just a named collection of components. It matches the signature StackSig, because it has at least the components required by StackSig and the components match their specifications.

The functor MkStack allows the creation of a stack given a structure that contains the type of stack elements. Thus we could also have obtained an integer stack by the above functor application.

Example 8.2 Here is an example of how sharing can come about by construction.

```
struct structure SharedStack=
         struct
           val stack= ref [ ]
           fun push(x)= ...
           fun pop( ... )= ...
         end

       structure StackUser1=
         struct
           structure A1= SharedStack
           fun f(x)= ... A1.push( ... ) ...
         end

       structure StackUser2=
         struct
           structure A2= SharedStack
           fun g( ... )= ... A2.pop ...
         end
end
```

Here StackUser1 pushes SharedStack while StackUser2 pops it. The explicit dependency of both users upon the stack has been acknowledged by incorporating the shared stack as a substructure in each of the users, under the identifiers A1 and A2, respectively. The above structure with its three substructures matches the following signature:

```
signature Users=
 sig
  structure StackUser1:
   sig
    structure A1: sig end
     ...
   end
  structure StackUser2:
   sig
    structure A2: sig end
     ...
   end
  sharing StackUser1.A1 = StackUser2.A2
 end
```

Sharing is particularly important in functor declarations since the body of the functor may only make sense provided certain sharing constraints are met. If, for example two stack users are mentioned in a functor parameter it might be important that they use the same stack. Therefore one needs to be able to declare a functor like

```
functor F(A: Users)= ...
```

where the signature Users defined above specifies the required sharing. If we later apply F to a structure then it must be checked statically that the actual argument really has the required sharing.

8.2 Semantic Objects

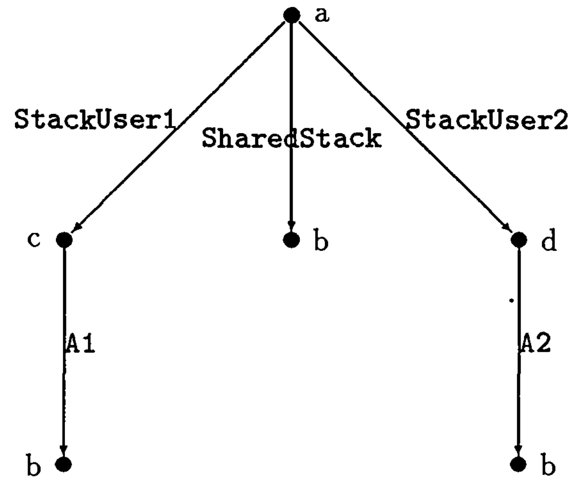
The semantic objects are defined in Figure 8.2.

From now on we shall maintain the terminological distinction between phrase classes and semantic objects. For instance a *structure expression* is a phrase (a member of StrExp) while a *structure* is a member of Str. Note that structures contain so-called *structure names*, whereas structure expressions do not. Structure names are used for representing sharing information; we use a, b, c, and other small letters (in roman font) for the individual names. Structures can be depicted as finitely branching trees of finite depth. For instance the structure expression from Example 8.2 elaborates to the structure

n	$\in$	StrName	structure names
N	$\in$	NameSet	(structure) name sets
E	$\in$	Env	environments
S or (n, E)	$\in$	Str	structures
Σ or $(N)S$	$\in$	Sig	signatures
I or $(S, (N')S')$	$\in$	FunInst	functor instances
Φ or $(N)(S, (N')S')$	$\in$	FunSig	functor signatures
F	$\in$	FunEnv	functor environments
G	$\in$	SigEnv	signature environments
B or M, F, G, E	$\in$	Basis	bases

$$
\begin{array}{lcl}
\text{NameSet} & = & \text{Fin}(\text{StrName}) \\
\text{Env} & = & \text{StrId} \xrightarrow{\text{fin}} \text{Str} \\
\text{Str} & = & \text{StrName} \times \text{Env} \\
\text{Sig} & = & \text{NameSet} \times \text{Str} \\
\text{FunInst} & = & \text{Str} \times \text{NameSet} \times \text{Str} \\
\text{FunSig} & = & \text{NameSet} \times \text{Str} \times \text{NameSet} \times \text{Str} \\
\text{FunEnv} & = & \text{FunId} \xrightarrow{\text{fin}} \text{FunSig} \\
\text{SigEnv} & = & \text{SigId} \xrightarrow{\text{fin}} \text{Sig} \\
\text{Basis} & = & \text{NameSet} \times \text{FunEnv} \times \text{SigEnv} \times \text{Env}
\end{array}
$$

Figure 8.2: Semantic Objects.

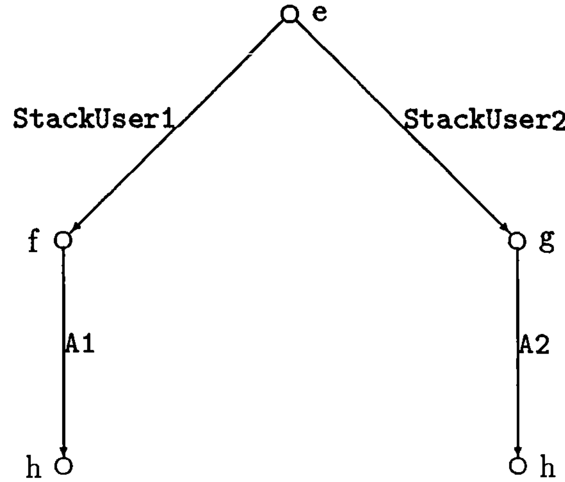


Here the difference between structure names and structure identifiers is obvious.

For any structure $S = (n, E)$ we call n the *structure name* of S; also, the *proper substructures* of S are the members of Rng E and their proper substructures. The *substructures* of S are S and its proper substructures.

A *signature* Σ takes the form $(N)S$ where the prefix (N) binds names in S. A structure is *bound* in $(N)S$ if it occurs in S and its name is in N. The set of bound structures of Σ is denoted boundstrs(Σ).

The idea is that bound names can be instantiated to other names during signature matching, whereas free names in the signature must be left unchanged. In the signature from Example 8.2 all names are bound since no sharing with structures outside the signature was specified. In diagrams, nodes with bound names are open circles, whereas free nodes are filled out. The signature from Example 8.2 can be drawn as follows:



We now see the first sign of a correspondence with the polymorphic type disciplines of Part I and II; a structure S corresponds to a type τ, while a signature $(N)S$ corresponds to a type scheme $\forall\alpha_1\ldots\alpha_n.\tau$.

There are two important aspects of signature matching. Given a signature $(N')S'$ and a structure S to match it, S is allowed to have *more* components than S', i.e., it may be necessary to add more components to S' in order to get S. Also, the bound names in the signature must be brought to agree with the names in S. Actually, it is convenient formally to factor matching into these two aspects, called *enrichment* and *instantiation*.

Definition 8.3 (Enrichment) *A structure* $S_2 = (n_2, E_2)$ enriches $S_1 = (n_1, E_1)$*, written* $S_1 \prec S_2$*, if* $n_1 = n_2$ *and* $\mathrm{Dom}\,E_1 \subseteq \mathrm{Dom}\,E_2$ *and for all strid* $\in \mathrm{Dom}\,E_1$*,* $E_2(strid)$ *enriches* $E_1(strid)$*.*

Then matching is the combination of getting agreement on names and adding components:

Definition 8.4 (Signature Matching) *A structure* S matches *a signature* $(N')S'$ *if there exists a structure* S_0 *such that* $(N')S' > S_0 \prec S$*.*

This leaves open the definition of signature instantiation $>$. One possibility is to define it so that it only changes names but preserves the number of components and then leave all the widening to $\prec$. The other extreme is to allow instantiation to do widening so that we strengthen the requirement $S_0 \prec S$ to $S_0 = S$. The two definitions give different semantics, but curious as it may seem, the same

inference rules can be used in both cases. We shall therefore defer the detailed discussion of the two alternatives till we have presented the rules. In both cases, the instantiation $\Sigma > S$ to structures extends to instantiation to signatures:

Definition 8.5 (Signature Instantiation) *Σ' is an* instance *of Σ, written $\Sigma \succeq \Sigma'$, if for all $S \in \mathrm{Str}$, $\Sigma' > S$ implies $\Sigma > S$.*

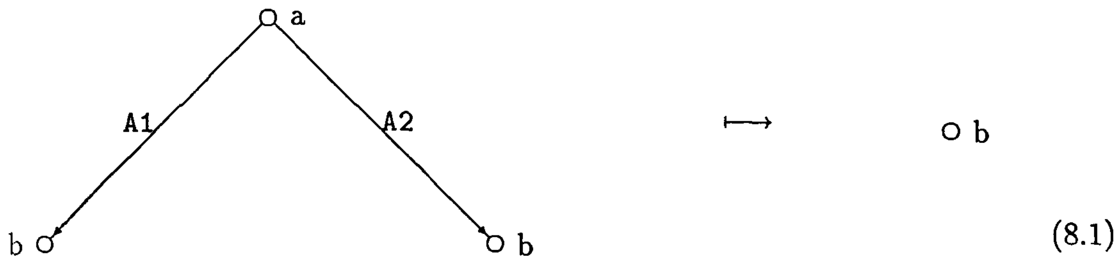
A *functor signature* Φ takes the form $(N)(S,(N')S')$. Here the prefix (N') binds names over S' and the prefix (N) binds over S and also over free names in $(N')S'$. Here $(N)S$ stems from the formal parameter signature of the functor declaration. Moreover, S' is the formal result structure of the functor; the prefix (N') binds those names of S' that must be generated afresh upon each application of the functor as will be explained in more detail below. Names that are free in $(N')S'$ and also occur in S signify sharing between the formal argument and the formal result. An example where this occurs is

```
functor F(A: sig structure A1: sig end
                 structure A2: sig end
                 sharing A1 = A2
             end
         ) = A.A1
```

This functor has the functor signature

$$\Big(\{\mathrm{a},\mathrm{b}\}\Big)\Big((\mathrm{a},\{\mathtt{A1}\mapsto(\mathrm{b},\{\}),\mathtt{A2}\mapsto(\mathrm{b},\{\})\})\ ,\ (\mathrm{b},\{\})\Big)$$

which can be drawn as follows:



(8.1)

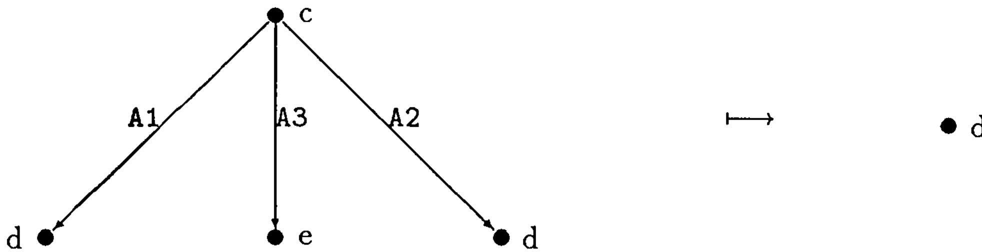
A functor with signature $(N_1)(S_1,(N_1')S_1')$ can be applied to an actual argument, S_2, that has more components than S_1. The result of the application will be an actual result $(N_2')S_2'$. Pairs of the form $(S_2,(N_2')S_2')$ are called *functor instances*. (The meaning of the prefix (N_2') will be explained later). Again we split matching into instantiation and enrichment:

Definition 8.6 *A functor instance* $(S_2,(N_2')S_2')$ matches *a functor signature* $(N_1)(S_1,(N_1')S_1')$ *if there exists an* S_0 *such that* $(N_1)(S_1,(N_1')S_1') > (S_0,(N_2')S_2')$ *and* $S_0 \prec S_2$.

This leaves open the definition of functor instantiation

$$(N_1)(S_1,(N_1')S_1') > (S_2,(N_2')S_2')$$

but again there are two natural definitions for the same inference rules, so we defer the detailed discussion of the alternatives till later. No matter which definition is chosen, the following functor instance matches the functor signature we drew above.



Note how sharing between the formal argument and result has been converted into sharing between the actual argument and result.

Finally, a *basis* $B = M, F, G, E$ is the semantic object in which all phrases are elaborated. The environments F, G and E bind semantic objects to identifiers that occur free in the phrase. Here F is a *functor environment* mapping functor identifiers to functor signatures, G is a *signature environment* mapping signature identifiers to signatures and E is a *(structure) environment* mapping structure identifiers to structures. Finally, M is a set of structure names. It serves two purposes.

Firstly, M records the names of all structures that have been generated so far. This use is relevant to the elaboration of declarations, structure expressions and programs since these are the phrase classes that can generate "new" structures. Whenever we need a "fresh" structure name, we simply choose it disjoint from all names in M. In general, a name can have been used although it is not currently accessible via the F, G, or E components of the basis. Therefore M is not just redundant information about the other components.

Secondly, M contains all names that are to be considered as "constants" in the current basis. This becomes important when we study the signature checker but it is worth having in mind already when reading the rules. The type checker

W discussed in Section 2.4 may perform substitutions on type variables that are free in the type environment, but it will never replace one type constant (*int*, say) with another (*bool*, say). Similarly, the signature checker may identify structure names that occur in $E \, \mathrm{of} \, B$ but are not in M, but it will not change a name which is in M. If two structures are specified to be shared and at least one of the structures has a name which is *not* in M then the signature checker will identify the two names when processing the sharing specification. However, if the names of the two structures that are specified to share are two different constants, then the signature checker has discovered that the sharing specification is not met. In the applicative discipline we simply had a syntactic distinction between type constants and type variables. That is not elegant in the modules semantics, because the set of "constants" is not constant; in a functor declaration, the components that have been specified in the formal parameter are to be seen as constants local to the functor body. This use of M pertains to the elaboration of signature expressions and specifications since it only during the checking of these phrases that the signature checker will try to unify structures.

8.3 Notation

Free structures and names: It is sometimes convenient to work with an arbitrary semantic object A, or assembly A of such objects. In general, strnames(A) denotes the set of structure names occurring free in A. A structure is said to be *free* if its name is free and strs(A) means the set of structures that occur free in A.

We often need to change bound names in semantic objects. For arbitrary A it is sometimes convenient to assume that *all* nameset prefixes N occurring in A are disjoint. In that case we say that we are *disjoining bound names* in A.

Projection: We often need to select components of tuples – for example the environment of a basis. In such cases we rely on variable names to indicate which component is selected. For instance "$E \, \mathrm{of} \, B$" means "the environment component of B" and "$n \, \mathrm{of} \, S$" means "the name of S."

Moreover, when a tuple contains a finite map we shall "apply" the tuple to an argument, relying on the syntactic class of the argument to determine the relevant function. For instance B(*funid*) means $(F \, \mathrm{of} \, B)$*funid*.

Finally, environments may be applied to long identifiers. For instance if

$longstrid = strid_1 \, . \, \ldots \, . \, strid_k$ then $E_0(longstrid)$ means

$$(E \,\mathrm{of} \cdots (E \,\mathrm{of}(E \,\mathrm{of}\, E_0) strid_1) strid_2 \cdots) strid_k.$$

Modification: The modification of one map f by another map g, written $f \pm g$ has already been mentioned. As before, a common use is environment modification $E_1 \pm E_2$. Often empty components will be left implicit in a modification. For set components, modification means union, so that for instance $B \pm N$ means

$$M \,\mathrm{of}\, B \cup N, F \,\mathrm{of}\, B, G \,\mathrm{of}\, B, E \,\mathrm{of}\, B\,.$$

Finally, we frequently need to modify a basis by an environment E, at the same time extending $M \mathrm{of} B$ to include the type names of E. We therefore define $B \oplus E$ to mean $B \pm (\mathrm{strnames}\, E, E)$.

8.4 Inference Rules

The rules allow us to infer sequents of the form

$$B \vdash phrase \Rightarrow A$$

where A is some semantic object. The relation is read "*phrase* **elaborates** to A in the basis B."

8.4.1 Declarations and Structure Expressions

These rules are "monomorphic" in the sense that given B and *phrase* they leave very little freedom as to what the result A can be. We shall later show that the only freedom is in the choice of new structure names.

It might be helpful to read the rules with the understanding that in every basis, the M component contains all the names that are free in the F, G, and E components. (We shall later show that the rules preserve that property).

Declarations

$$\boxed{B \vdash dec \Rightarrow E}$$

$$\frac{}{B \vdash \quad \Rightarrow \{\}} \tag{8.2}$$

$$\frac{B \vdash strexp \Rightarrow S}{B \vdash \textbf{structure}\ strid\ = strexp \Rightarrow \{strid \mapsto S\}} \tag{8.3}$$

$$\frac{B \vdash dec_1 \Rightarrow E_1 \quad B \oplus E_1 \vdash dec_2 \Rightarrow E_2}{B \vdash dec_1\ dec_2 \Rightarrow E_1 \pm E_2} \tag{8.4}$$

Comment:

(8.4) The use of $\oplus$, here and elsewhere, ensures that the names generated during the first sub-phrase are considered used during the elaboration of the second sub-phrase.

Structure Expressions $\boxed{B \vdash strexp \Rightarrow S}$

$$\frac{B \vdash dec \Rightarrow E \quad n \notin \text{strnames}(E) \cup (M \text{ of } B)}{B \vdash \textbf{struct}\ dec\ \textbf{end} \Rightarrow (n, E)} \tag{8.5}$$

$$\frac{B(longstrid) = S}{B \vdash longstrid \Rightarrow S} \tag{8.6}$$

$$\frac{\begin{array}{c} B(funid) > (S, (N')S') \quad B \vdash strexp \Rightarrow S_1 \quad S \prec S_1 \\ N' \cap (M \text{ of } B) = \emptyset \end{array}}{B \vdash funid(strexp) \Rightarrow S'} \tag{8.7}$$

Comments:

(8.5) The side condition ensures that the resulting structure receives a new name. If the expression occurs in a functor body the structure name will be bound by (N') in rule 8.18. This will ensure that for each application of the functor, by rule 8.7 (last premise), a new distinct name will be chosen for the structure generated.

(8.7) The interpretation of this rule depends of the definition of the functor instantiation relation as will be discussed in Section 9.1. The purpose of the last premise was explained above. It can always be satisfied by choosing the bound names in the functor instance $(S, (N')S')$ appropriately.

8.4.2 Specifications and Signature Expressions

These rules are polymorphic in the sense that given the basis and the phrase there may be many different results of the elaboration. This is essential for getting a simple rule for sharing. However, as with the applicative polymorphic type discipline, the richness of results that can be inferred is not bigger than can be captured by a notion of principality. In the modules semantics the notion is as follows:

Definition 8.7 (Principal Signatures) *A signature* Σ *is* principal *(for sigexp in B), if* $B \vdash sigexp \Rightarrow \Sigma$*, and for all* Σ' *satisfying* $B \vdash sigexp \Rightarrow \Sigma'$ *we have* $\Sigma \geq \Sigma'$.

Intuitively, Σ is principal if it has precisely the components and the sharing implied by *spec.*[1] The definition of principality is used in the inference rules concerning programs.

Specifications

$$\boxed{B \vdash spec \Rightarrow E}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\}} \tag{8.8}$$

$$\frac{B \vdash sigexp \Rightarrow (\emptyset)\, S}{B \vdash \textbf{structure}\ strid\ :\ sigexp \Rightarrow \{strid \mapsto S\}} \tag{8.9}$$

$$\frac{B \vdash spec_1 \Rightarrow E_1 \quad B \pm E_1 \vdash spec_2 \Rightarrow E_2}{B \vdash spec_1\ spec_2 \Rightarrow E_1 \pm E_2} \tag{8.10}$$

$$\frac{n \text{ of } B(longstrid_1) = n \text{ of } B(longstrid_2)}{B \vdash \textbf{sharing}\ longstrid_1 = longstrid_2 \Rightarrow \{\}} \tag{8.11}$$

Comments:

(8.10) Here $\pm$ is used instead of $\oplus$ because the elaboration of $spec_1$ cannot generate any new constant structures.

[1]For each of the two definitions of signature instantiation "implied" means a different thing.

(8.11) The premise is a simple test of identity of names. The liberty that rule 8.12 and 8.13 give to choose names can be used to achieve the sharing before it is tested.[2]

Signature Expressions

$$\boxed{B \vdash sigexp \Rightarrow \Sigma}$$

$$\frac{B \vdash spec \Rightarrow E}{B \vdash \mathbf{sig}\ spec\ \mathbf{end} \Rightarrow (\emptyset)(n, E)} \tag{8.12}$$

$$\frac{B(sigid) = \Sigma}{B \vdash sigid \Rightarrow \Sigma} \tag{8.13}$$

$$\frac{B \vdash sigexp \Rightarrow (N)S \quad n \notin \mathrm{strnames}\, B}{B \vdash sigexp \Rightarrow (N \cup \{n\})S} \tag{8.14}$$

$$\frac{B \vdash sigexp \Rightarrow \Sigma' \quad \Sigma' \geq \Sigma}{B \vdash sigexp \Rightarrow \Sigma} \tag{8.15}$$

Comments:

(8.12) In contrast to rule 8.5, n is not here required to be new. The name n may be chosen to achieve the sharing required in rule 8.11.

(8.14) This rule is called the *generalization rule.*

(8.15) This rule is called the *instantiation rule.* It's interpretation obviously depends on the definition of signature instantiation.[3] Regardless of which of the two definitions we take, the instance is not determined by the rule; the generalization and instantiation rules are often used to achieve sharing properties.

8.4.3 Programs

These rules are monomorphic in the same sense as the rules for declarations and structure expressions.

[2]While humans can make such predictive choices when doing proofs in this formal system, the signature checker will try to unify the two structures upon encountering the sharing specification

[3]Exactly how, will be discussed in Section 9 1.

Programs

$$\boxed{B \vdash prog \Rightarrow B'}$$

$$\frac{B \vdash dec \Rightarrow E}{B \vdash dec \Rightarrow (\text{strnames}\, E, \{\}, \{\}, E)} \tag{8.16}$$

$$\frac{B \vdash sigexp \Rightarrow \Sigma \quad \Sigma \text{ principal for } sigexp \text{ in } B}{B \vdash \textbf{signature}\ sigid = sigexp \Rightarrow (\text{strnames}\, \Sigma, \{\}, \{sigid \mapsto \Sigma\}, \{\})} \tag{8.17}$$

$$\frac{\begin{array}{c} B \vdash sigexp \Rightarrow (N)S \quad (N)S \text{ principal for } sigexp \text{ in } B \\ N \cap M \text{ of } B = \emptyset \\ B \oplus \{strid \mapsto S\} \vdash strexp \Rightarrow S' \\ N' = \text{strnames}(S') \setminus (N \cup (M \text{ of } B)) \quad \Phi = (N)(S, (N')S') \end{array}}{B \vdash \textbf{functor}\ funid(strid : sigexp) = strexp \Rightarrow (\text{strnames}\, \Phi, \{funid \mapsto \Phi\}, \{\}, \{\})} \tag{8.18}$$

$$\frac{B \vdash prog_1 \Rightarrow B_1 \quad B \pm B_1 \vdash prog_2 \Rightarrow B_2}{B \vdash prog_1\ prog_2 \Rightarrow B_1 \pm B_2} \tag{8.19}$$

Comments:

(8.17) The principality requirement ensures that the signature bound to *sigid* has exactly the components and sharing implied by *sigexp*.

(8.18) Here $(N)S$ is required to be principal so as to have exactly the components and the sharing implied by *sigexp*. The requirement $N \cap M \,\text{of}\, B = \emptyset$ ensures that no accidental sharing is assumed between S and B. Since $\oplus$ is used, any structure name n in S acts like a constant in the functor body; in particular, it ensures that further names generated during elaboration of the body are distinct from n. The set N' is the set of names that, as an addition to the names in the original basis and the names in S, have been generated by the elaboration of the functor body. Since it is the application of the functor that creates new names, not the declaration of it, these names are all bound in Φ.

Chapter 9

Foundations of the Semantics

The inference rules allow us given some assembly of semantic objects to elaborate programs hence producing a modified assembly of semantic objects. The semantic objects that make up an assembly must be consistent with each other. For example, sharing must be hereditary, so both of the following structures cannot be in the same assembly:

The static elaboration may change the underlying assembly of semantic objects but only as long as the consistency conditions are maintained. Therefore the rules are only part of the semantics. The foundation on which they are built is the definition of what an admissible assembly of semantic objects is.

In this chapter we shall describe two different definitions of admissibility. They give rise to different notions of signature instantiation and functor instantiation and two quite different interpretations of the inference rules.

9.1 Coherence and Consistency

At least we must require that sharing is hereditary in the following sense:

Definition 9.1 (Consistency) *A semantic object A or assembly A of objects is said to be* consistent *if, after disjoining bound names, for all S_1 and S_2 in A and for every longstrid* if n of $S_1 = n$ of S_2 *and both S_1(longstrid) and S_2(longstrid) exist,* then n of S_1(*longstrid*) $= n$ of S_2(*longstrid*).

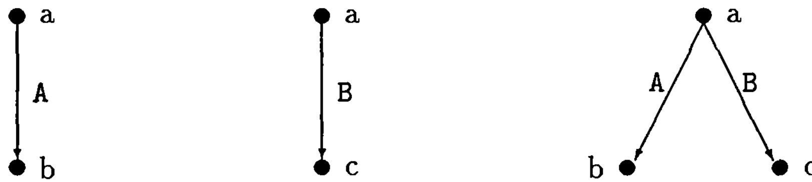
Given the definition of Str, the natural definition of structure equality is

Definition 9.2 (Structure Equality) *Two structures* (n, E) *and* (n', E') *are* equal, *written* $(n, E) = (n', E')$, *if* $n = n'$ *and* $E = E'$. *Two environments* E *and* E' *are* equal, *written* $E = E$, *if* $\mathrm{Dom}(E) = \mathrm{Dom}(E')$ *and for all strid* $\in \mathrm{Dom}(E)$ *one has* $E(strid) = E'(strid)$.

The definition of consistency should be compared carefully with the following definition.

Definition 9.3 (Coherence) *A semantic object A or assembly A of objects is said to be* coherent *if, after disjoining bound names, for all* S_1 *and* S_2 *in A* **if** n of $S_1 = n$ of S_2 **then** $S_1 = S_2$.

Then it will be obvious that coherence implies consistency, but not the other way around. For instance the assembly consisting of the three structures



is consistent, but not coherent.

Coherence is natural: to talk about a substructure being shared among several structures almost grammatically implies that a shared structure is one real "thing" with a certain number of components regardless of the structures of which it is a substructure.

This is certainly so for the semantic objects phrases evaluate to in the dynamic semantics. But in the static semantics we can think of two different, but consistent, structures as being two different, but consistent, *views* (or *approximations*) of an object in the dynamic semantics. Thus the three structures drawn above "hide" different information.

We shall now demonstrate that if we choose appropriate definitions of signature and functor instantiations based on coherence then the static semantics will predict the creation of structures without missing out any of the components that will be there at run-time. Later we shall see in more detail how consistency leads to a semantics with information hiding.

9.1.1 Coherence

First let us consider signature matching. Recalling Definition 8.4, if S_0 and S are coherent and $S_0 \prec S$ then $S_0 = S$. Therefore S matches $(N')S'$ precisely if $(N')S' > S$. Thus the instantiation will have to do both the change of bound names and the widening.

If two substructures S'_1 and S'_2 of S' have the same name then they must be matched by structures S_1 and S_2 with n of $S_1 = n$ of S_2. By coherence n of $S_1 = n$ of S_2 implies $S_1 = S_2$. Thus matching could be described by a map from names in S' to substructures of S. By coherence of S' this is equivalent to a map from substructures of S' to substructures of S. The change of names together with the widening is captured by the following definition

Definition 9.4 (Realisation) *A map* φ : Str $\rightarrow$ Str *is called a* realisation *if for all* $S \in$ Str *and all strid*, if $S(strid)$ *exists* then $(\varphi\, S)strid$ *exists and* $\varphi(S\, strid) = (\varphi\, S)strid$.

Notice that realisations preserve existing paths: if $S(longstrid)$ exists then $(\varphi\, S)longstrid$ exist. They also preserve sharing:
if $S(longstrid_1) = S(longstrid_2)$ then $(\varphi\, S)longstrid_1 = (\varphi\, S)longstrid_2$.

Realisations in the modules semantics correspond to substitutions in the other polymorphic type disciplines. The technology of substitutions carries over to realisations. A *finite* realisation is a realisation restricted to a finite set of structures. The domain and range of a finite realisation are denoted Dom φ and Rng φ. Moreover, when φ is finite, the *region* of φ, written Reg φ, is the set of names that occur in the range of φ. Thus Dom φ and Rng φ are sets of structures while Reg φ is a set of structure names.

Realisations can be applied to environments by pointwise application i.e., we define $\varphi\, E = \varphi \circ E$. In particular, we have that $\varphi\{\} = \{\}$.

Any map α : StrName $\rightarrow$ StrName naturally extends to a realisation map, written $\alpha^\sharp$, that changes names without adding components:

$$\alpha^\sharp(n, E) = (\alpha(n), \alpha^\sharp(E)).$$

Any finite renaming ρ of the form

$$\{n_i \mapsto n'_i \mid 1 \leq i \leq k\}$$

can first be extended to a total map on StrName by letting it be the identity outside $\{n_1, \cdots, n_k\}$ and this map then induces a realisation map, denoted $\rho^\sharp$.

A set A of structures is said to be *substructure closed* if whenever it contains a structure S then it also contains all the proper substructures of S. Whenever φ is a realisation map and A is substructure closed then we can extend the finite realisation $\varphi_0 = \varphi \downarrow A$ to a total realisation, written $\varphi_0^\sharp$, defined by

$$\varphi_0(n, E) = \begin{cases} (n, \varphi_0^\sharp E) & \text{if } (n, E) \notin A, \\ \varphi(n, E) & \text{if } (n, E) \in A. \end{cases}$$

We shall often omit the $^\sharp$ from $\alpha^\sharp$, $\rho^\sharp$, and $\varphi_0^\sharp$.

In general, it is not true that any finite map of structures to structures,

$$\{S_i \mapsto S_i' \mid 1 \leq i \leq k\}$$

determines a realisation map.[1]

The ordinary function composition of two realisation maps is a realisation map. The identity map, ID, from structures to structures is a realisation map and it is the identity for the composition of realisations. We shall often write $\varphi_2 \varphi_1 S$ to mean $(\varphi_2 \circ \varphi_1) S$, i.e., $\varphi_2(\varphi_1(S))$.

A realisation φ is said to *glide* on a structure $S = (n, E)$ if $\varphi S = (n, \varphi E)$. The *support* of φ, written $\mathrm{Supp}\,\varphi$, is the set of structures on which φ does *not* glide i.e., where φ changes the name or adds more components.

We can now define signature instantiation as follows:

Definition 9.5 (Signature Instantiation) *A structure S is an* instance *of a signature $(N')S'$ if there exists a realisation φ such that $\varphi(S') = S$ and* $\mathrm{Supp}(\varphi) \subseteq \mathrm{boundstrs}((N')S')$.

Again we notice a strong resemblance with the earlier polymorphic disciplines. Just as instantiation earlier was defined by substitution on bound variables then instantiation is now defined as realisation on bound structures.

A signature Σ is *well-formed* if $\mathrm{strs}\,\Sigma$ is substructure closed. This is the same as demanding that, writing Σ on the form $(N)S$, whenever (n, E) is a substructure of S and $n \notin N$, then $N \cap \mathrm{strnames}\,E = \emptyset$.

We shall now define a relation $\Sigma_1 \xrightarrow{\varphi} \Sigma_2$ which defines what it is to apply a realisation to a signature (Σ_1). The ideas is that φ is applied to the *free* structures of Σ_1 if necessary doing a simultaneous renaming of the bound names of Σ_1 to avoid capture of names. This only makes sense provided Σ_1 is well-formed (recall that

[1] However, as we shall see in Chapter 11, there is a natural sufficient condition.

this means that no free structure contains a bound structure). More generally, if ρ (for renaming) is a finite bijection from structure names to structure names, A is a substructure closed set of structures such that $\mathrm{strnames}(A) \cap \mathrm{Dom}(\rho) = \emptyset$, and φ_0 is a finite realisation whose domain is A then *the simultaneous composition* of φ_0 and ρ, written $\varphi_0 \,|_A \rho$ is the realisation defined as follows. For all $S = (n, E) \in \mathrm{Str}$,

$$(\varphi_0 \,|_A \rho)S = \begin{cases} \varphi_0(S) & \text{if } S \in A; \\ (\rho(n), (\varphi_0 \,|_A \rho)\, E) & \text{if } n \in \mathrm{Dom}\,\rho; \\ (n, (\varphi_0 \,|_A \rho)\, E) & \text{otherwise.} \end{cases}$$

It is easy to check that $\varphi_0 \,|_A \rho$ really is a realisation. We shall often omit the subscript A in $\varphi_0 \,|_A \rho$ since A has to be the domain of φ_0.

Definition 9.6 *Let $\Sigma_1 = (N_1)S_1$ and $\Sigma_2 = (N_2)S_2$ be signatures and φ be a realisation. We write $\Sigma_1 \xrightarrow{\varphi} \Sigma_2$ if Σ_1 is well-formed and*

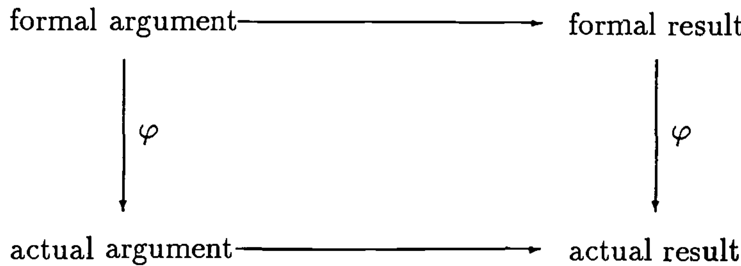
1. *$N_2 \cap \mathrm{Reg}(\varphi_0) = \emptyset$, and*

2. *there is a bijection $\rho : N_1 \to N_2$ such that $(\varphi_0 \,|_A \rho)S_1 = S_2$,*

where $A = \mathrm{strs}\,\Sigma_1$ and $\varphi_0 = \varphi \downarrow A$. Likewise, one can define the relation $A_1 \xrightarrow{\varphi} A_2$ for any semantic objects A_1, A_2 or assemblies A_1, A_2 of semantic objects.

We write $A_1 \underset{\alpha}{=} A_2$ as a shorthand for $A_1 \xrightarrow{ID} A_2$, where ID is the identity realisation. Note that this is the usual notion of α-conversion.

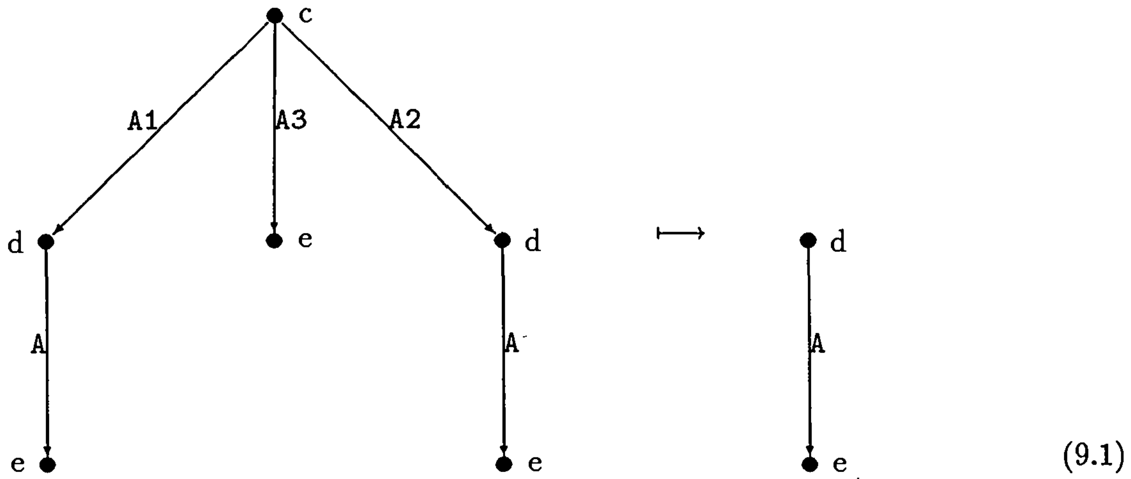
Now let us consider functor signature instantiation. As with signature instantiation the definition of functor matching (Definition 8.6) becomes simpler because of coherence: a pair $(S_2, (N_2')S_2')$ matches a functor signature $(N_1)(S_1, (N_1')S_1')$ if $(N_1)(S_1, (N_1')S_1') > (S_2, (N_2')S_2')$. The idea is that the matching of the actual argument against the formal parameter induces a realisation which is applied to the formal result to get the actual result:



Formally

Definition 9.7 (Functor Instantiation) *Given* $\Phi = (N_1)(S_1, (N_1')S_1')$, *a functor instance* $(S_2, (N_2')S_2')$ *is an* instance *of* Φ, *written* $\Phi > (S_2, (N_2')S_2')$ *if there exists a realisation* φ *such that* $(S_1, (N_1')S_1') \xrightarrow{\varphi} (S_2, (N_2')S_2')$ *and* $\mathrm{Supp}\,\varphi \subseteq \mathrm{boundstrs}((N_1)S_1)$.

Since φ performs widening as well as change of names the actual result may have more components that the formal result. For example the following functor instance



(9.1)

is an instance of the functor signature (8.1). Another example is the functor

functor F(A: **sig end**)= A

which has the functor signature $\{n\}((n, \{\}), (\emptyset)(n, \{\}))$. In any instance of this functor signature the actual argument and result will be identical, so F acts like the identity functor.

Now let us review the inference rules that in interesting ways depend on these definitions.

Rule 8.7 simplifies to to

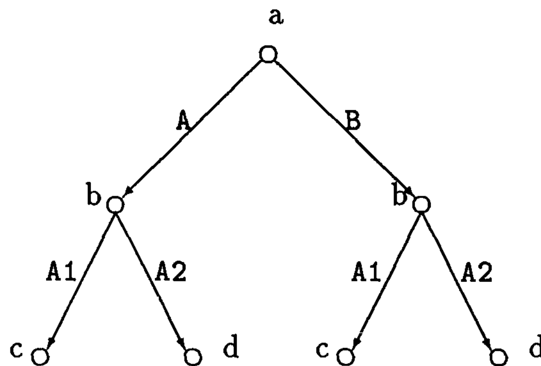
$$\frac{B(\mathit{funid}) > (S, (N')S') \quad B \vdash \mathit{strexp} \Rightarrow S \qquad N' \cap (M \text{ of } B) = \emptyset}{B \vdash \mathit{funid}(\mathit{strexp}) \Rightarrow S'} \tag{9.2}$$

and the thing to note is that the actual result, S', may have more components that the formal result, in fact no components of the actual result are lost as a result of the functor application.

Rule 8.11 concerning sharing is now effectively a test for structure identity as the premise by coherence implies $B(\textit{longstrid}_1) = B(\textit{longstrid}_2)$. Structure identity can be achieved by the generalization and instantiation rules, which now allow widening as well as name instantiation. An example of this is the signature expression

```
sig
  structure A:
    sig structure A1: sig end
    end
  structure B:
    sig structure A2: sig end
    end
  sharing A = B
end
```

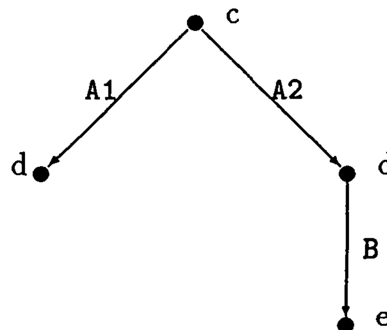
which has principal signature



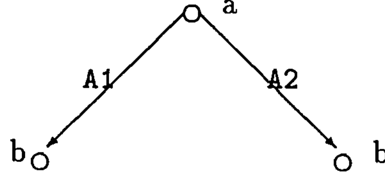
where A has an A2 component and B an A1 component.

9.1.2 Consistency

First, let us consider signature matching. We can now allow S to match Σ, where S is



and Σ is



but since here we have an example of one substructure of the signature being matched by two different substructures we can no longer describe the matching by a realisation map from structures to structures. Instead we let realisation simply be a map from names to names and leave the widening to the enrichment relation $\prec$.

Definition 9.8 (Realisation) *A* realisation *is a map* φ : StrName $\rightarrow$ StrName.

Now Supp φ means the set of n for which $\varphi n \neq n$ and the definition of signature instantiation becomes

Definition 9.9 (Signature Instantiation) *A structure S is an* instance *of a signature $(N')S'$ if there exists a realisation φ such that $\varphi(S') = S$ and* $\text{Supp}(\varphi) \subseteq N'$.

The definition of simulataneous composion $\varphi_0 |_A \rho$ is revised in the obvious way and definition 9.6 changes slightly to

Definition 9.10 *Let $\Sigma_1 = (N_1)S_1$ and $\Sigma_2 = (N_2)S_2$ be signatures and φ be a realisation. We write $\Sigma_1 \xrightarrow{\varphi} \Sigma_2$ if Σ_1 is well-formed and*

1. *$N_2 \cap \text{Reg}(\varphi_0) = \emptyset$, and*

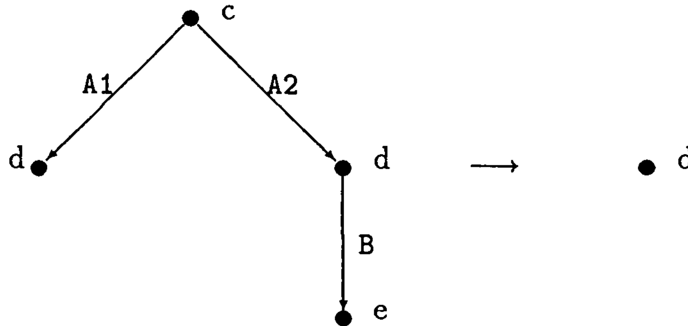
2. *there is a bijection $\rho : N_1 \rightarrow N_2$ such that $(\varphi_0 |_A \rho)S_1 = S_2$,*

where $A = strnames\Sigma_1$ and $\varphi_0 = \varphi \downarrow A$. Likewise, one can define the relation $A_1 \xrightarrow{\varphi} A_2$ for any semantic objects A_1, A_2 or assemblies A_1, A_2 of semantic objects.

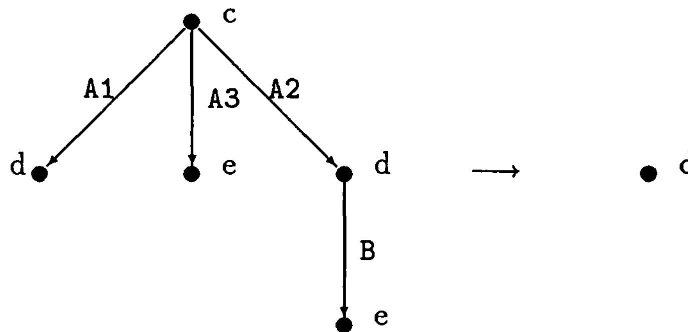
Now let us consider functor signature instantiation. Definition 9.7 is modified slightly to

Definition 9.11 (Functor Instantiation) *Given $\Phi = (N_1)(S_1, (N_1')S_1')$, a functor instance $(S_2, (N_2')S_2')$ is an* instance *of Φ, written $\Phi > (S_2, (N_2')S_2')$, if there exists a realisation φ such that $(S_1, (N_1')S_1') \xrightarrow{\varphi} (S_2, (N_2')S_2')$ and* Supp $\varphi \subseteq N_1$.

However, since realisation no longer performs widening, the actual result will always have exactly the same number of components as the formal result. So now the functor instance



is an instance of (8.1), which is now matched by



Notice that (9.1) no longer matches (8.1). More extremely,

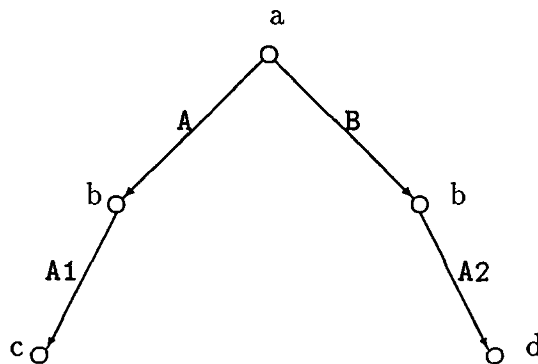
functor `F(A:` **sig end**`)= A`

now is the functor that cuts out all the components of its argument.

As to the inference rules, we have already noted that in the rule for functor application (8.7) the formal and actual results have the same number of components. The instantiation rule no longer admits widening, but that is no longer needed since the sharing rule only tests for name equality. Hence the signature expression

```
sig
  structure A:
    sig structure A1: sig end
    end
  structure B:
    sig structure A2: sig end
    end
  sharing A = B
end
```

now has the principal signature



Since fewer paths are present in principal signatures, fewer functor declarations (rule 8.18) will be admitted.

9.1.3 Coherence Only

For the rest of this thesis we shall examine the coherent semantics in detail. It would have been nice to investigate the consistent semantics in a similar way, but I'm afraid that this thesis is already getting a bit long. Historically, all the following work was done assuming coherence at a point in time where it was not clear that the essential choice one has to make is between coherence and consistency, because the rest of the definitions depend on this choice. In the ML semantics [20] we actually ended deciding on the consistent semantics because it was felt that explicit signature constraints

structure *strid* : *sigexp* = *strexp*

had to coerce the number of components of the structure to which *strexp* elaborates down to the number of components mentioned in *sigexp* without ruining

sharing information between the full and the cut down version of the structure. This cannot be done in the coherent semantics, but it can be done in the consistent semantics.

It is dangerous to claim that the results I prove in the coherent semantics carry over to the consistent semantics. However, I would be very surprised if it should turn out that the main results (the existence of principal unifiers and signatures) did not carry over.

I shall now turn to other restrictions that semantic objects must satisfy in order to be admissible.

9.2 Well-formedness and Admissibility

We have already defined what it is for a signature to be well-formed (Section 9.1.1). A functor signature $(N)(S,(N')S')$ is *well-formed* if $(N)S$, $(N')S'$ and $(N \cup N')S'$ are well-formed and $N \cap \mathrm{strnames}(N')S' \subseteq \mathrm{strnames}\, S$.

An object or assembly A is *well-formed* if every signature and functor signature occurring in A is well-formed.

An object or assembly A is *admissible* if it is coherent and well-formed. Beware the difference between, say, the statement "the structure S is admissible and the structure $\varphi(S)$ is admissible" and "the assembly $\{S, \varphi(S)\}$ is admissible." The latter is stronger than the former, as it requires that the set of all structures that occur in S or in $\varphi(S)$ be coherent.

For any $M \in \mathrm{NameSet}$, *the set of M-structures*, written M–Str, is

$$M\text{–Str} = \{S \in \mathrm{Str} \mid n \text{ of } S \in M\}.$$

Moreover, for any object or assembly, A, we define *the (free) M-structures of A*, written

$$M\text{–strs}\, A,$$

to mean $M\text{–Str} \cap \mathrm{strs}\, A$, i.e.,

$$M\text{–strs}\, A = \{S \in \mathrm{Str} \mid n \text{ of } S \in M \quad \text{and} \quad S \text{ occurs free in } A\}.$$

We do not impose admissibility constraints on the inference rules for declarations, structure expressions and programs, as these rules will be shown to preserve admissibility automatically. This is not true for the rules concerning specifications and signature expressions, so we impose the following constraints

Definition 9.12 (Global Constraints) *Let* $B = M, F, G, E$. *A conclusion* $B \vdash spec \Rightarrow E_1$ *is* admissible *if*

$$\{B, E_1\} \text{ is admissible} \tag{9.3}$$

$$\text{strnames}\, F \cup \text{strnames}\, G \subseteq M \tag{9.4}$$

$$M\text{–strs}\, E \text{ and } M\text{–strs}\, E_1 \text{ are substructure closed.} \tag{9.5}$$

Similarly, a conclusion $B \vdash sigexp \Rightarrow \Sigma$ *is* admissible *if*

$$\{B, \Sigma\} \text{ is admissible} \tag{9.6}$$

$$\text{strnames}\, F \cup \text{strnames}\, G \subseteq M \tag{9.7}$$

$$M\text{–strs}\, E \text{ and } M\text{–strs}\, \Sigma \text{ are substructure closed.} \tag{9.8}$$

A proof P *of* $B \vdash spec \Rightarrow E$ *or* $B \vdash sigexp \Rightarrow \Sigma$ *is* admissible *if every conclusion in* P *is admissible.*

In what follows we require that all proofs be admissible, so if we write for example "if $B \vdash sigexp \Rightarrow \Sigma$ then ...", it is always to be read "if there exists an admissible proof of $B \vdash sigexp \Rightarrow \Sigma$ then ...".

A basis $B = M, F, G, E$ is said to be *robust* if it is admissible, strnames $F \subseteq M$, strnames $G \subseteq M$, and M–strs E is substructure closed. B is said to be *strictly* robust if it is robust and strnames $E \subseteq M$.

For bases in which signature expressions and specifications are elaborated robustness is tacitly assumed – see definition 9.12. In a strictly robust basis, all free structures are considered as "constants".[2] In the theorems concerning the elaboration of structure expressions, declarations and programs bases will often be assumed strictly robust.

Realisation of bases is defined as follows. Assume $B = M, F, G, E$ and $B' = M', F', G', E'$. We define

$$B \xrightarrow{\varphi} B'$$

to mean $F \xrightarrow{\varphi} F'$, $G \xrightarrow{\varphi} G'$, and $\varphi(E) = E'$. We write $B \xrightarrow[\text{rb}]{\varphi} B'$, if B, and B' are robust and $B \xrightarrow{\varphi} B'$. Similarly, we write $B \xrightarrow[\text{srb}]{\varphi} B'$ if B and B' are strictly robust and $B \xrightarrow{\varphi} B'$.

If B is a robust basis some of the structures in E of B, namely those whose name is not in M of B, may be affected by realisation. Therefore, among all realisations, the following are of particular use:

[2]See the discussion in Section 8.4.

Definition 9.13 (M-realisation) *An* M-realisation *is a realisation* φ *with* $\mathrm{Supp}\,\varphi \cap M\text{–strs} = \emptyset$.

If $B = M, F, G, E$ is a robust basis and φ is an M-realisation then φ is the identity on all structures free in F and G, so $F \xrightarrow{\varphi} F$ and $G \xrightarrow{\varphi} G$. Thus it makes sense to define the *application* of φ to B, written $\varphi\,B$, to be $(M, F, G, \varphi\,E)$.

As in the other type disciplines the notion of *closure* is important.

Definition 9.14 *For any two objects or assemblies A, B, the B-*closure *of A, written* $\mathrm{Clos}_B A$, *is* $(N)A$, *where* $N = \mathrm{strnames}\,A \setminus \mathrm{strnames}\,B$. *We abbreviate* $\mathrm{Clos}_\emptyset A$ *to* $\mathrm{Clos}\,A$.

9.3 Two Lemmas about Instantiation

The following lemma gives a very useful characterization of signature instantiation.

Lemma 9.15 (Characterization of $\geq$) *We have*

$$(N)S \geq (N')S' \iff (N)S > S' \text{ and } N' \cap \mathrm{strnames}(N)S = \emptyset.$$

Proof. We first prove the implication from left to right. Assume $(N)S \geq (N')S'$. Clearly $(N')S' > S'$. Thus by definition 8.5 we have $(N)S > S'$.

We prove $N' \cap \mathrm{strnames}((N)S) = \emptyset$ indirectly. Assume $n' \in N' \cap \mathrm{strnames}((N)S)$. Let n be a name different from n'. Then $\{n' \mapsto n\}$ is a realisation and $\mathrm{Supp}(\{n' \mapsto n\}) \cap \mathrm{strs}\,S' \subseteq \mathrm{boundstrs}((N')S')$. Thus $(N')S' > \{n' \mapsto n\}\,S'$. By definition 8.5 this implies $(N)S > \{n' \mapsto n\}\,S'$. But that is impossible; n' occurs free in $(N)S$ and hence in any instance of it, but it obviously does not occur in $\{n' \mapsto n\}\,S'$. Thus $N' \cap \mathrm{strnames}((N)S) = \emptyset$.

To prove the implication from right to left, assume $(N)S > S'$ and $N' \cap \mathrm{strnames}((N)S) = \emptyset$. Assume $(N')S' > S_0$ and prove $(N)S > S_0$. By assumption there exists a realisation ψ such that $\psi\,S' = S_0$ and $\mathrm{Supp}\,\psi \cap \mathrm{strs}\,S' \subseteq \mathrm{boundstrs}((N')S')$, and there exists a realisation φ with $\varphi\,S = S'$ and $\mathrm{Supp}\,\varphi \cap \mathrm{strs}\,S \subseteq \mathrm{boundstrs}((N)S)$. Thus $\psi \circ \varphi$ is a realisation with $\psi \circ \varphi(S) = S_0$. Moreover, if $S_1 \in \mathrm{strs}((N)S)$ then φ glides on S_1 so n of $\varphi\,S_1 = n$ of S_1 which is not bound in $(N')S'$ — since by assumption $N' \cap \mathrm{strnames}((N)S) = \emptyset$.

Therefore ψ glides on $\varphi\, S_1$. Hence $\psi \circ \varphi$ glides on all free structures of $(N)S$, i.e., $\mathrm{Supp}(\psi \circ \varphi) \cap \mathrm{strs}\, S \subseteq \mathrm{boundstrs}((N)S)$. This shows $(N)S > S_0$ as required. ∎

Nothing in the proof required admissibility. However, as a corollary we immediately get

Corollary 9.16 *If* $\Sigma \geq \Sigma'$ *and* Σ *is well-formed then* $\mathrm{strs}\,\Sigma \subseteq \mathrm{strs}\,\Sigma'$.

The relation $\Sigma \geq \Sigma'$ is obviously reflexive and transitive. Thus we get an equivalence relation $\equiv$ defined by

$$\Sigma \equiv \Sigma' \iff \Sigma \geq \Sigma' \text{ and } \Sigma' \geq \Sigma.$$

The following lemma gives a characterization of this equivalence. Basically, two signatures are equivalent if and only if one can be obtained from the other by renaming of bound names together with deletion or insertion of redundant bound variables. More precisely, defining $\mathrm{boundnames}((N)S)$ to mean the set $N \cap \mathrm{strnames}\, S$, we have

Lemma 9.17 (Characterization of $\equiv$) *Assume that* $\{S, S'\}$ *is coherent. Then the following two statements are equivalent:*

1. $(N)S \equiv (N')S'$
2. a) $\mathrm{strnames}((N)S) = \mathrm{strnames}((N')S')$ *and*
   b) *There exists a bijection* $\alpha : \mathrm{boundnames}((N)S) \to \mathrm{boundnames}((N')S')$ *with* $\alpha^{\sharp}(S) = S'$.

Proof. First assume (2). We have $\alpha^\sharp(S) = S'$, $\mathrm{Supp}(\alpha^\sharp) \cap \mathrm{strs}\, S \subseteq \mathrm{boundstrs}((N)S)$ and $N' \cap \mathrm{strnames}((N)S) = N' \cap \mathrm{strnames}((N')S') = \emptyset$, so $(N)S \geq (N')S'$ by lemma 9.15.

To prove $(N')S' \geq (N)S$, we first prove that $(\alpha^{-1})^\sharp \circ \alpha^\sharp$ is the identity on all substructures of S. (In general, it is not the identity everywhere). Take $(n, E) \in \mathrm{strs}\, S$. If $n \in N$ then $\alpha^\sharp(n, E) = (\alpha n, \alpha^\sharp E)$ so $((\alpha^{-1})^\sharp \circ \alpha^\sharp)(n, E) = ((\alpha^{-1} \circ \alpha)n, ((\alpha^{-1})^\sharp \circ \alpha^\sharp)\, E) = (n, ((\alpha^{-1})^\sharp \circ \alpha^\sharp)\, E)$ i.e., $(\alpha^{-1})^\sharp \circ \alpha^\sharp$ glides on (n, E).

Otherwise $n \notin N$, so $\alpha^\sharp$ glides on (n, E). Now n is free in $(N')S'$ by (a) so $(\alpha^{-1})^\sharp$ glides on $\alpha^\sharp(n, E)$. Thus $(\alpha^{-1})^\sharp \circ \alpha^\sharp$ glides on (n, E) in this case as well. Thus $(\alpha^{-1})^\sharp \circ \alpha^\sharp$ is the identity on $\mathrm{strs}\, S$.

Now $(\alpha^{-1})^\sharp(S') = (\alpha^{-1})^\sharp \circ \alpha^\sharp(S) = S$; $\mathrm{Supp}(\alpha^{-1})^\sharp \cap \mathrm{strs}\, S' \subset \mathrm{boundstrs}((N')S')$; and $N \cap \mathrm{strnames}((N')S') = N \cap \mathrm{strnames}((N)S) = \emptyset$, so $(N')S' \geq (N)S$ by lemma 9.15. Hence $(N)S \equiv (N')S'$.

Conversely, assume

$$(N)S \geq (N')S' \tag{9.9}$$

$$(N')S' \geq (N)S. \tag{9.10}$$

Then by lemma 9.15 there exist realisations φ and ψ such that

$$\mathrm{Supp}(\varphi) \subseteq \mathrm{boundstrs}((N)S) \wedge \varphi(S) = S' \wedge N' \cap \mathrm{strnames}((N)S) = \emptyset \tag{9.11}$$

$$\mathrm{Supp}(\psi) \subseteq \mathrm{boundstrs}((N')S') \wedge \psi(S') = S \wedge N \cap \mathrm{strnames}((N')S') = \emptyset \tag{9.12}$$

Thus $\varphi \circ \psi$ and $\psi \circ \varphi$ are realisations and $\psi \circ \varphi(S) = S$ and $\varphi \circ \psi(S') = S'$. The identity is such a realisation so $\psi \circ \varphi$ must coincide with the identity on $\mathrm{strs}\, S$ and $\varphi \circ \psi$ must coincide with the identity on $\mathrm{strs}\, S'$:

$$\forall S_1 \in \mathrm{strs}\, S \quad \psi \circ \varphi(S_1) = S_1 \tag{9.13}$$

$$\forall S_1' \in \mathrm{strs}\, S' \quad \varphi \circ \psi(S_1') = S_1'. \tag{9.14}$$

Since φ maps substructures of S to substructures of S' and ψ maps substructures of S' to substructures of S these equations express that φ is a bijection from $\mathrm{strs}\, S$ to $\mathrm{strs}\, S'$ with inverse ψ. Moreover, φ and ψ may change structure names, but they cannot add components:

$$\forall (n, E) \in \mathrm{strs}\, S\, \exists n'. \quad \varphi(n, E) = (n', \varphi\, E) \tag{9.15}$$

$$\forall (n', E') \in \mathrm{strs}\, S'\, \exists n. \quad \psi(n', E') = (n, \psi\, E'). \tag{9.16}$$

(This is easy to prove using (9.13) and (9.14).)

Now φ maps free structures of $(N)S$ to free structures of $(N')S'$: if $(n, E) \in \text{strs}((N)S)$ then φ glides on (n, E) by (9.11) so n of $\varphi(n, E) = n$. But $N' \cap \text{strnames}((N)S) = \emptyset$ by (9.11) so $n \notin N'$. Thus $\varphi(n, E)$ is free in $(N')S'$.

Similarly ψ maps free structures of $(N')S'$ to free structures of $(N)S$. But that means that $\varphi : \text{strs}((N)S) \to \text{strs}((N')S')$ is a bijection with inverse $\psi : \text{strs}((N')S') \to \text{strs}((N)S)$. This in turn gives that $\varphi : \text{boundstrs}((N)S) \to \text{boundstrs}((N')S')$ makes sense and is a bijection with inverse $\psi : \text{boundstrs}((N')S') \to \text{boundstrs}((N)S)$.

Since φ is a bijection on the free structures and since φ glides on the free structures, we immediately have the desired (a).

To define the desired α, note that for each $n \in \text{boundnames}((N)S)$ there is exactly one E such that $(n, E) \in \text{strs}\, S$. This is where the coherence of S is used. Thus we can define

$$\alpha(n) = n \text{ of } \varphi(n, E).$$

Similarly, by the coherence of S' we define $\alpha' : \text{boundnames}((N')S') \to \text{boundnames}((N)S)$ by $\alpha'(n') = n \text{ of } \psi(n', E')$ where (n', E') is the unique structure in strs S' whose name is n'.

Since φ is a bijection on the bound structures with inverse ψ, α is a bijection from $\text{boundnames}((N)S)$ onto $\text{boundnames}((N)S')$ with inverse α'.

By (9.15), $\alpha^\sharp$ coincides with φ on $\text{strs}\, S$ so by (9.11) we get $\alpha^\sharp(S) = S'$, showing (b). ∎

Finally, here is a little lemma that says that functor instantiation is deterministic up to the choice of new bound names. The lemma follows easily from definition 9.7.

Lemma 9.18 *Given* $\Phi = (N_1)(S_1, (N_1')S_1')$ *and assume* $(N_1)S_1 > S_2$. *Then there exists an* $(N_2')S_2'$ *such that* $\Phi > (S_2, (N_2')S_2')$. *Moreover, if* $\Phi > (S_2, (N_2')S_2')$ *and* $\Phi > (S_2, (N_2'')S_2'')$ *then* $(N_2')S_2' \underset{\alpha}{=} (N_2'')S_2''$.

Chapter 10

Robustness Results

In this chapter we shall state a number of results relating realisation and elaboration. Just like one can get a better understanding of a program by stating and proving properties of it, one can get a better understanding of an inference system by stating and proving properties of it. So fundamental are the results we shall now present that if they did not hold, we would feel that the semantics would have to be changed. Moreover, we shall use most of the results in subsequent proofs.

To avoid unreasonable punishment of readers who do not want to study the details most of the proofs are deferred to Appendix A.

10.1 Realisation and Instantiation

We first prove that realisation respects signature instantiation. Then we prove that signature inference is preserved under realisation.

Lemma 10.1 *If* Σ *is admissible and* $\Sigma > S'$ *and* $\Sigma \xrightarrow{\psi} \Sigma_1$ *then* $\Sigma_1 > \psi\, S'$.

The result extends to signatures:

Lemma 10.2 *If* $\Sigma \xrightarrow{\varphi} \Sigma_1$ *and* $\Sigma' \xrightarrow{\varphi} \Sigma_1'$ *and* $\Sigma \geq \Sigma'$ *and* Σ *is admissible then* $\Sigma_1 \geq \Sigma_1'$.

In definition 9.6 we boldly claimed that we could define the relation $A_1 \xrightarrow{\varphi} A_2$ for any semantic objects. Functor signatures require a bit of care, though:

Definition 10.3 *Let* $\Phi_1 = (N_1)(S_1, (N_1')S_1')$ *and* $\Phi_2 = (N_2)(S_2, (N_2')S_2')$ *be well-formed functor signatures and* φ *be a realisation. We write* $\Phi_1 \xrightarrow{\varphi} \Phi_2$ *if there exists a bijection* $\rho_1 : N_1 \rightarrow N_2$ *and a bijection* $\rho : N_1 \cup N_1' \rightarrow N_2 \cup N_2'$ *extending* ρ_1 *such that*

(a) $\mathrm{Reg}\,\varphi_1 \cap N_2 = \emptyset$ *and* $(\varphi_1\,|\rho_1)\,S_1 = S_2$,
where $\varphi_1 = \varphi \downarrow \mathrm{strs}(N_1)S_1$.

(b) $\mathrm{Reg}\,\varphi_2 \cap (N_2 \cup N_2') = \emptyset$ *and* $(\varphi_2\,|\rho)\,S_1' = S_2'$,
where $\varphi_2 = \varphi \downarrow \mathrm{strs}(N_1 \cup N_1')\,S_1'$.

This is stronger than requiring $(N_1)\,S_1 \xrightarrow{\varphi} (N_2)\,S_2$ and $(N_1 \cup N_1')\,S_1' \xrightarrow{\varphi} (N_2 \cup N_2')\,S_2'$ because having one common renaming ensures that sharing between bound structures in $(N_1)S_1$ and free structures in $(N_1')S_1'$ is preserved. When $N_1 = \emptyset$ definition 10.3 simplifies to the obvious definition of

$$(S_1, (N_1')\,S_1') \xrightarrow{\varphi} (S_2, (N_2')\,S_2').$$

We can now prove that functor signature instantiation is preserved under realisation:

Lemma 10.4 *For all functor signatures* Φ_1, Φ_2, *and functor instances* I_3, I_4, *if* Φ_1 *is admissible and* $\Phi_1 > I_3$ *and* $\Phi_1 \xrightarrow{\varphi} \Phi_2$ *and* $I_3 \xrightarrow{\varphi} I_4$ *then* $\Phi_2 > I_4$.

We can now describe under which conditions admissible signature inferences are transformed into admissible signature inferences by realisation.

Theorem 10.5 *Let* $B = M, F, G, E$ *and* $B' = M', F', G', E'$ *be bases with* $B \xrightarrow[\text{rb}]{\varphi} B'$.
If

$$B \vdash spec \Rightarrow E_1 \tag{10.1}$$

$$M'\text{-strs}(\varphi\, E_1) \text{ is substructure closed} \tag{10.2}$$

$$\{B', \varphi\, E_1\} \text{ is coherent} \tag{10.3}$$

then

$$B' \vdash spec \Rightarrow \varphi\, E_1 .$$

Similarly, if

$$B \vdash sigexp \Rightarrow \Sigma \text{ and } \Sigma \xrightarrow{\varphi} \Sigma' \tag{10.4}$$

$$M'\text{-strs}\, \Sigma' \text{ is substructure closed} \tag{10.5}$$

$$\{B', \Sigma'\} \text{ is coherent} \tag{10.6}$$

then

$$B' \vdash sigexp \Rightarrow \Sigma'.$$

Recall that when we write $B' \vdash sigexp \Rightarrow \Sigma'$ we implicitly refer to admissible proofs only (c.f. definition 9.12). Some of the admissibility constraints listed in that definition follow from the assumptions (10.1) and (10.4), but others have to be ensured by (10.2),(10.3) and (10.5),(10.6).

We shall mostly use the above theorem in the situation where φ is an M-realisation:

Corollary 10.6 *Let* $B = M, F, G, E$ *be a robust basis and* φ *be an* M*-realisation such that* $\varphi\, B$ *is robust. If*

$$B \vdash spec \Rightarrow E_1$$

$$M\text{-strs}(\varphi\, E_1) \text{ is substructure closed}$$

$$\{\varphi\, B, \varphi\, E_1\} \text{ is coherent}$$

then

$$\varphi\, B \vdash spec \Rightarrow \varphi\, E_1 .$$

Similarly, if

$$B \vdash sigexp \Rightarrow \Sigma \text{ and } \Sigma \xrightarrow{\varphi} \Sigma'$$

$$M\text{-strs}(\Sigma') \text{ is substructure closed}$$

$$\{\varphi\, B, \Sigma'\} \text{ is coherent}$$

then

$$\varphi\, B \vdash sigexp \Rightarrow \Sigma'.$$

10.2 The Strict Rules Preserve Admissibility

The rules for signature expressions and specifications were forced to respect admissibility (c.f. definition 9.12). No such constraints are needed for the remaining rules because, as we shall now show, they "automatically" preserve admissibility.

Theorem 10.7 *Assume $B = M, F, G, E$ is strictly robust.*
If $B \vdash dec \Rightarrow E$ then $\{B, E\}$ is coherent,
if $B \vdash strexp \Rightarrow S$ then $\{B, S\}$ is coherent, and
if $B \vdash prog \Rightarrow B'$ then $\{B, B'\}$ is coherent.

This theorem is a consequence of a stronger theorem which will be used again and again:

Theorem 10.8 *Let $B = M, F, G, E$ be strictly robust.*
If $B \vdash dec \Rightarrow E$ then E is coherent and M-$\mathrm{strs}\, E \subseteq \mathrm{strs}\, B$,
if $B \vdash strexp \Rightarrow S$ then S is coherent and M-$\mathrm{strs}\, S \subseteq \mathrm{strs}\, B$, and
if $B \vdash prog \Rightarrow B'$ then B' is strictly robust and M-$\mathrm{strs}\, B' \subseteq \mathrm{strs}\, B$.

So if one of these phrases evaluates to a result, every structure in the result is either brand new i.e., its name is not in M, or the structure is simply inherited from the basis.

It is not hard to see that theorem 10.8 really implies theorem 10.7. For declarations, for instance, since B is coherent and $\mathrm{strs}\, B \subseteq M$-Str and E is coherent, and since every structure in E whose name is in M is free in B, we have that $\{B, E\}$ is coherent.

In the proof of theorem 10.8 we shall use the following lemma.

Lemma 10.9 *If Σ is principal for sigexp in B then* $\mathrm{strs}\, \Sigma \subseteq \mathrm{strs}\, B$.

Proof. By contradiction. We assume that Σ is principal for *sigexp* in B and that $\text{strs}\,\Sigma \setminus \text{strs}\,B \neq \emptyset$ and construct a Σ' with $B \vdash sigexp \Rightarrow \Sigma'$ but $\Sigma \not\geq \Sigma'$.

Since we have assumed that $\text{strs}\,\Sigma \setminus \text{strs}\,B \neq \emptyset$ there exists a structure $S_0 \in \text{strs}\,\Sigma \setminus \text{strs}\,B$ and we can choose S_0 so that it is not a proper substructure of any other structure free in Σ.

Write S_0 in the form (n_0, E_0), let n_ν be a name not in $\text{strnames}\{B, \Sigma\}$, and let φ be the realisation $\{n_0 \mapsto n_\nu\}$. There exists a Σ' such that $\Sigma \xrightarrow{\varphi} \Sigma'$. Note that n_0 is not free in Σ' so we cannot have $\Sigma \geq \Sigma'$. Thus it will suffice to prove that $B \vdash sigexp \Rightarrow \Sigma'$. This in turn follows from theorem 10.5, if we can show

$$B \xrightarrow[\text{rb}]{\varphi} B \tag{10.7}$$

$$M\text{–strs}\,\Sigma' \text{ is substructure closed} \tag{10.8}$$

$$\{B, \Sigma'\} \text{ is coherent.} \tag{10.9}$$

But (10.7) follows from $n_\nu \notin \text{strnames}(B)$ and the fact that any strictly robust basis is also robust. As to (10.8),

$$M\text{–strs}\,\Sigma' = \begin{cases} M\text{–strs}(\Sigma), & \text{if } n_0 \notin M \\ M\text{–strs}(\Sigma) \setminus S_0, & \text{if } n_0 \in M, \end{cases}$$

in either case a substructure closed set (in the latter case because we chose S_0 so that it is not a proper substructure of any other structure free in Σ).

Finally, (10.9) follows from the coherence of $\{B, \Sigma\}$ and the fact that $n_\nu \notin \text{strnames}\{B, \Sigma\}$. ∎

In the proof of theorem 10.8 (see Appendix A) one checks for each rule that the bases occurring on the left hand side of the $\vdash$ in the premises are strictly robust assuming that the basis on the left hand side of the $\vdash$ in the conclusion was strictly robust. Therefore we accidently prove:

Corollary 10.10 *If B is strictly robust, P is an elaboration of a phrase containing as an intermediate conclusion of the form $B \vdash dec \Rightarrow E$, $B \vdash strexp \Rightarrow S$, or $B \vdash prog \Rightarrow B'$ then B is strictly robust.*

10.3 Realisation and Structure Expressions

We shall now prove that the static elaboration of structure expressions and declarations is "essentially" deterministic, namely deterministic up to the choice of new names. This is a consequence of a theorem about how realisation affects structure expression elaboration: realisation can only affect the elaboration of structure expressions by acting on structures that stem from the basis; new structures may have to be renamed to avoid name capture, but the number of new structures will not be changed by realisation, not even when the structure expression contains functor applications.

We prove

Theorem 10.11 *Assume B is a strictly robust basis and that $B \vdash strexp \Rightarrow S$. Then for all $S' \in \mathrm{Str}$,*

$$B \vdash strexp \Rightarrow S' \quad \textit{iff} \quad \mathrm{Clos}_B\, S \underset{\alpha}{=} \mathrm{Clos}_B\, S' .$$

by proving the stronger

Theorem 10.12 *If $B \vdash strexp \Rightarrow S$ and $B \xrightarrow[\mathrm{srb}]{\varphi} B'$, then for all $S' \in \mathrm{Str}$,*

$$B' \vdash strexp \Rightarrow S' \quad \textit{iff} \quad \mathrm{Clos}_B\, S \xrightarrow{\varphi} \mathrm{Clos}_{B'}\, S' .$$

Similarly for declarations.

Note that this realisation is the dual of the realisation on signature inferences. In the latter case we are mostly interested in M-realisations (Corollary 10.10) that can be used to change structures that do not stem from the basis, whereas in the above theorem the realisation affects structures stemming from the basis but glides on new structures (up to the renaming of new names).

Most importantly, note that theorem 10.12 justifies the definition of functor signatures and functor signature instantiation. Suppose for the moment that we extend ModL programs to include

$$prog ::= \textbf{let}\ prog_1\ \textbf{in}\ prog_2$$

with the obvious inference rule. Moreover, assume that we allow structure dec-

larations to be decorated with signature expressions as follows:[1]

$$\frac{B \vdash \mathit{strexp} \Rightarrow S \quad B \vdash \mathit{sigexp} \Rightarrow (\emptyset)\, S}{B \vdash \textbf{structure}\ \mathit{strid}\ :\ \mathit{sigexp} = \mathit{strexp} \Rightarrow \{\mathit{strid} \mapsto S\}}$$

Then using theorem 10.12 it is easy to prove that if

$$B \vdash \ \textbf{let functor}\ \mathit{funid}(\mathit{strid}\ :\ \mathit{sigexp}) = \mathit{strexp}_1\ \textbf{in}\ \mathit{funid}(\mathit{strexp}_2) \Rightarrow S$$

then also

$$B \vdash \ \textbf{let structure}\ \mathit{strid}\ :\ \mathit{sigexp} = \mathit{strexp}_2\ \textbf{in}\ \mathit{strexp}_1 \Rightarrow S$$

provided *funid* does not occur free in strexp_2. The converse, that one can always make an abstraction, does not in general hold here since strexp_1 may refer to paths that are present in strexp_2 but not in *sigexp*.

[1]Note that the explicit signature serves simply as a control that S matches the signature; the decoration in no way constrains S. This choice is forced upon us in the coherent semantics. In the consistent semantics the signature could constrain S

Chapter 11

Unification of Structures

A *unifier* for two structures S, S' is a realisation φ such that $\varphi S = \varphi S'$. Moreover, φ is said to be a *principal* unifier for S and S' if whenever ψ is a unifier for S and S' there exists a realisation ψ' such that $\psi = \psi' \circ \varphi$.

In this chapter we shall prove that if S and S' are coherent and have a unifier then they have a principal unifier. We prove it by giving an algorithm, *Unify*, which when given S and S' as parameters either fails or succeeds and succeeds with a principal unifier precisely if S and S' have a unifier.

The unification algorithm will be vital when we in the next chapter give an algorithm for checking signature expression with sharing constraints, but the unification we consider can also be seen as a generalisation of unification from terms to certain directed acyclic graphs regardless of its use in the modules semantics. The graphs we consider are those that correspond to so-called *simple* subsets of Str:

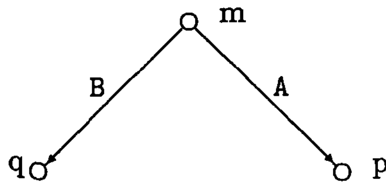
Definition 11.1 (Simple "worlds") *A set* $W \subseteq$ Str *is* simple (with respect to M) *if it is finite and coherent and if also* W *and* M–strs W *are substructure closed.*

Hence a simple W (W for "world") can be drawn as a directed acyclic graph in which structure names uniquely label nodes and in which all successors to M-nodes are M-nodes.

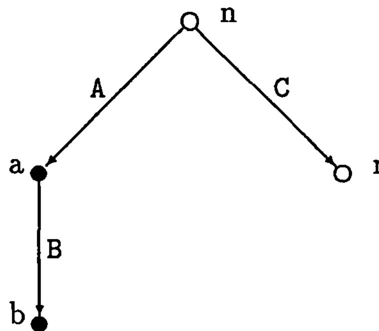
An M*-unifier* for S and S' is an M-realisation, φ, with $\varphi S = \varphi S'$. Moreover, φ is a *principal* M-unifier for S and S' if whenever ψ is an M-unifier for S and S' there exists an M-realisation ψ' such that $\psi = \psi' \circ \varphi$. For the sake of the semantics we are interested in M-unifiers. We shall solve the general problem of

determining when M-unifiers exist. The problem of deciding when unifiers exist is a special case, namely $M = \emptyset$.

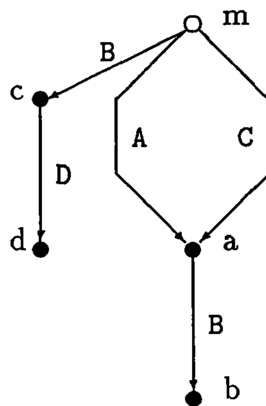
Example 11.2 In the following pictures, M-nodes are filled out while the others are little circles. There exists an M-unifier, φ, for S and S', where $S =$



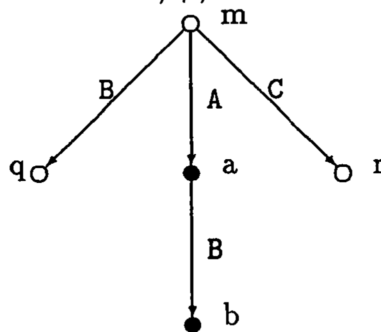
and $S' =$



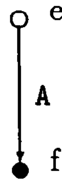
with $\varphi\, S = \varphi\, S' =$



and there exists a principal M-unifier, ψ, for S and S' with $\psi\, S = \psi\, S' =$



However, there is no M-unifier for S' and S'', where $S'' =$



or S' and S''', where $S''' =$

(The last example because S''' is a proper substructure of S'.)

The plain theorem, then, is

Theorem 11.3 (Principal Unifiers) *Assume that* $\{S, S'\}$ *is coherent and that* M–strs$\{S, S'\}$ *is substructure closed. If* S *and* S' *have an* M*-unifier then they have a principal* M*-unifier.*

This is nice to know from a theoretical point of view. But from a practical point of view it is also necessary to know how one finds out whether two structures are unifiable. Therefore we prove theorem 11.3 constructively by giving an algorithm, *Unify*, which finds unifiers, when possible. More precisely, let us say that an M-realisation φ is a *locally principal* M-unifier for S and S' if $\varphi\, S = \varphi\, S'$ and moreover, whenever ψ is an M-unifier for S and S' then there exists an M-realisation, ψ', such that

$$\psi \downarrow \mathrm{strs}\{S, S'\} = (\psi' \circ \varphi) \downarrow \mathrm{strs}\{S, S'\}.$$

Unify then has the following properties:

Theorem 11.4 (Unification) *Assume that* $\{S, S'\}$ *is coherent and that* M–strs$\{S, S'\}$ *is substructure closed. Either* $Unify(M, S, S')$ *fails or it succeeds with a locally principal* M*-unifier for* S *and* S'*. Moreover, Unify succeeds if and only if there exists an* M*-unifier for* S *and* S'*.*

This theorem really does imply theorem 11.3: if φ is a locally principal M-unifier for S and S', one obtains a principal M-unifier, φ_1, for S and S' as follows

$$\varphi_1(n, E) = \begin{cases} \varphi(n, E), & \text{if } (n, E) \text{ occurs in } S \text{ or in } S', \\ (n, \varphi_1\, E) & \text{otherwise.} \end{cases}$$

Here is *Unify*:

$Unify(M, S, S') =$
let $(n, E) = S;\ (n', E') = S'$ **in**
  if $n \in M$ **and** $n' \in M$ **then if** $n = n'$ **then** ID **else fail**
  if $n \in M$ **and** $n' \notin M$ **then**
    if $\mathrm{Dom}\, E' \subseteq \mathrm{Dom}\, E$ **then**
      let $\varphi_1 = Unify(M, E, E');\ \ \epsilon = \{(n', \varphi_1 E') \mapsto (n, E)\}$
      in $\epsilon \circ \varphi_1$
    else fail

  if $n \notin M$ **and** $n' \in M$ **then** ... (symmetric) ...

  if $n \notin M$ **and** $n' \notin M$ **then**
      if $S \notin \mathrm{strs}\, E'$ **and** $S' \notin \mathrm{strs}\, E$ **then**
      let $\varphi_1 = Unify(M, E, E');$
        $\epsilon = \{(n, \varphi_1 E) \mapsto (n, \varphi_1 E \cup \varphi_1 E'), (n', \varphi_1 E') \mapsto (n, \varphi_1 E \cup \varphi_1 E')\}$
      in $\epsilon \circ \varphi_1$
      else fail

$Unify(M, E, E') =$
  if $\mathrm{Dom}\, E \cap \mathrm{Dom}\, E' = \emptyset$ **then** ID
  else let $strid \in \mathrm{Dom}\, E \cap \mathrm{Dom}\, E'$
      $\{strid \mapsto S_1\} \cup E_1 = E$ (disjoint union)
      $\{strid \mapsto S'_1\} \cup E'_1 = E'$ (disjoint union)
      $\varphi_1 = Unify(M, S_1, S'_1)$
      $\varphi_2 = Unify(M, \varphi_1 E_1, \varphi_1 E'_1)$
    in $\varphi_2 \circ \varphi_1$.

Comments on *Unify*. The algorithm seeks to build a realisation by composition of realisations, starting from the identity realisation, ID. When at least one of n and n' is not in M, a recursive call is performed and, provided it succeeds, its result is composed with an elementary realisation, ϵ.

The notation $\{S_1 \mapsto S'_1, \ldots, S_k \mapsto S'_k\}$, $(k \geq 1)$, means the unique realisation that maps each S_i to S'_i and glides outsides $\cup_i \mathrm{strs}(S_i)$, when that realisation exists! In general, the realisation exists if and only if for all i, j in $\{1, \ldots, k\}$ and

for all *longstrid* $_1$, *longstrid* $_2$,

> if S_i(*longstrid* $_1$) is equal to S_j(*longstrid* $_2$) then S'_i *longstrid* $_1$ and S'_j *longstrid* $_2$ exist and are equal.

For $k = 1$ and $k = 2$ there are particularly simple conditions that ensure that $\{S_1 \mapsto S'_1, \ldots, S_k \mapsto S'_k\}$ denotes a realisation.

For $k = 1$ a sufficient condition is $E \,\mathrm{of}\, S_1 \subseteq E \,\mathrm{of}\, S'_1$.[1] In other words, the operation of changing a given structure by perhaps changing its name and perhaps widening it at the top is a realisation. The first ϵ in *Unify* is of this kind.

For $k = 2$ a sufficient condition is

$$S'_1 = S'_2,\ \ E \,\mathrm{of}\, S_1 \subseteq E \,\mathrm{of}\, S'_1,\ \ E \,\mathrm{of}\, S_2 \subseteq E \,\mathrm{of}\, S'_2,$$

$$S_1 \notin \mathrm{strs}(E \,\mathrm{of}\, S_2), \quad \text{and} \quad S_2 \notin \mathrm{strs}(E \,\mathrm{of}\, S_1).$$

The latter ϵ in *Unify* is of this kind.

The conditions $S \notin \mathrm{strs}\, E'$ **and** $S' \notin \mathrm{strs}\, E$ is what in ordinary term unification is known as the "occur check". From the definition of realisation (definition 9.4) it is easy to see that if S and S' have a unifier, then $S \notin \mathrm{strs}(E \,\mathrm{of}\, S')$ and $S' \notin \mathrm{strs}(E \,\mathrm{of}\, S)$.

In the case $n \in M$ **and** $n' \notin M$ we do not need an occur check "**if** $\mathrm{Dom}\, E' \subseteq \mathrm{Dom}\, E$ **and** $S' \notin \mathrm{strs}\, E$" because M–strs S is assumed substructure closed.

In general, the algorithm is expressed so as to make a proof of the previous thorems fairly easy. In an implementation, where one is more concerned with efficiency, one can represent coherent structure by directed acyclic graphs. A structure name is a pointer, then, and unification works by "swinging pointers" in the store instead of producing realisation maps.

With these explanations, I hope that it is at least plausible that *Unify* has the properties stated in theorem 11.4. *Proving* it is the aim of the next two sections. Having done that we shall compare our unification with ordinary first order unification and with a similar theory developed by Hassan Aït–Kaci.

[1] For arbitrary finite maps, f and g, $f \subseteq g$ means $\mathrm{Dom}\, f \subseteq \mathrm{Dom}\, g$ and $f(x) = g(x)$ for all $x \in \mathrm{Dom}\, f$.

11.1 Soundness of *Unify*

We first prove that *Unify* is *sound*, i.e., that when it succeeds then it succeeds with a unifier. In the next section we show "completeness" i.e., that *Unify* always terminates and that it succeeds with a locally principal unifier when there is any unifier.

Below, the *names involved in* φ, written $\mathrm{Inv}\,\varphi$, means

$$\bigcup\{\mathrm{strnames}\,S \cup \mathrm{strnames}(\varphi\, S) \mid S \in \mathrm{Supp}\,\varphi\}.$$

Moreover, when W is a set of structures and φ a realisation, $\varphi\, W$ means $\{\varphi\, w \mid w \in W\}$.

Theorem 11.5 (Soundness of *Unify*) *Assume W is simple with respect to M. If $S, S' \in W$ and $\varphi = Unify(M, S, S')$ succeeds then*

$$\varphi\, S = \varphi\, S' \tag{11.1}$$

$$\mathrm{Supp}\,\varphi \cap W \subseteq \mathrm{strs}\{S, S'\} \tag{11.2}$$

$$\mathrm{Inv}\,\varphi \subseteq \mathrm{strnames}\,S \cup \mathrm{strnames}\,S' \tag{11.3}$$

$$\varphi \textit{ is an } M\textit{-realisation} \tag{11.4}$$

$$\varphi\, W \textit{ is simple with respect to } M \tag{11.5}$$

$$\textit{If } \varphi \neq ID \textit{ then } |\varphi\, W| < |W| \tag{11.6}$$

$$\varphi \textit{ does not change the name of any structure whose name is in } \mathrm{strnames}(\varphi\, S). \tag{11.7}$$

Similarly, if $\mathrm{strs}\,E \cup \mathrm{strs}\,E' \subseteq W$ *and* $\varphi = Unify(M, E, E')$ *succeeds then*

$$\forall strid \in \mathrm{Dom}\,E \cap \mathrm{Dom}\,E',\ \varphi\, E\, strid = \varphi\, E'\, strid \tag{11.8}$$

$$\mathrm{Supp}\,\varphi \cap W \subseteq \mathrm{strs}\{E, E'\} \tag{11.9}$$

$$\mathrm{Inv}\,\varphi \subseteq \mathrm{strnames}\,E \cup \mathrm{strnames}\,E' \tag{11.10}$$

$$\varphi \textit{ is an } M\textit{-realisation} \tag{11.11}$$

$$\varphi\, W \textit{ is simple with respect to } M \tag{11.12}$$

$$\textit{If } \varphi \neq ID \textit{ then } |\varphi\, W| < |W| \tag{11.13}$$

$$\varphi \textit{ does not change the name of any structure whose name is in } \mathrm{strnames}\{\varphi\, E, \varphi\, E'\}. \tag{11.14}$$

Despair not! Among these properties (11.1) and (11.4) are the ones we primarily want, whereas (11.2), (11.3), (11.5) and (11.7) are used partly to get the induction to work and partly in the proof of the soundness of the signature checker. Property (11.6) will give a simple termination argument in the next section.

Note that (11.2) does not say "$\mathrm{Supp}\,\varphi \subseteq \mathrm{strs}\{S, S'\}$" for the very good reason that this inclusion is not true because φ accumulates all operations that have been performed during the unification. This is why φ is a locally principal M-unifier, not in general a principal unifier.

Proof. [of theorem 11.5] The proof is by induction on the length of the computation counted as the number, r, of recursive calls.

Base Case, $r = 0$. By the definition of *Unify* there are two cases, one for structures and one for environments. The latter trivially gives (11.8)–(11.14) and the former gives (11.1)–(11.7) using the coherence of W.

Inductive Step, $r > 0$. There are a number of cases corresponding to the definition of the algorithm. We start by considering structures $S = (n, E)$ and $S' = (n', E')$.

$n \in M$ and $n' \notin M$ where also $\mathrm{Dom}\,E' \subseteq \mathrm{Dom}\,E$, $\varphi_1 = \mathit{Unify}(M, E, E')$, $\epsilon = \{(n', \varphi_1 E') \mapsto (n, E)\}$ and $\varphi = \epsilon \circ \varphi_1$.

By induction,

$$\forall \mathit{strid} \in \mathrm{Dom}\,E',\ \varphi_1 E\,\mathit{strid} = \varphi_1 E'\,\mathit{strid} \tag{11.15}$$

$$\mathrm{Supp}\,\varphi_1 \cap W \subseteq \mathrm{strs}\{E, E'\} \tag{11.16}$$

$$\mathrm{Inv}\,\varphi_1 \subseteq \mathrm{strnames}\,E \cup \mathrm{strnames}\,E' \tag{11.17}$$

$$\varphi_1 \text{ is an } M\text{-realisation} \tag{11.18}$$

$$\varphi_1 W \text{ is simple with respect to } M \tag{11.19}$$

$$\text{If } \varphi_1 \neq \mathit{ID} \text{ then } |\varphi_1 W| < |W| \tag{11.20}$$

$$\begin{array}{l}\varphi_1 \text{ does not change the name of any structure whose}\\ \text{name is in strnames}\{\varphi_1 E, \varphi_1 E'\}.\end{array} \tag{11.21}$$

To see that ϵ is a realisation map, note that $\varphi E = E$ by (11.18) and the substructure closedness of M–strs W. Therefore, $\varphi_1 E' \subseteq E$ by (11.15). It follows from the remarks in the previous section that ϵ is a realisation. It is clearly an M-realisation. Thus $\varphi = \epsilon \circ \varphi_1$ is an M-realisation, showing (11.4).

Proof of (11.1): $\varphi(n', E') = \epsilon \circ \varphi_1(n', E') = \epsilon(n', \varphi_1 E')$ since (11.17) and $n' \notin \text{strnames}\{E, E'\}$. And $\epsilon(n', \varphi_1 E') = (n, E) = \varphi(n, E)$ since φ is an M-realisation.

Proof of (11.2): $\text{Supp}\,\varphi \cap W \subseteq \text{Supp}\,\varphi_1 \cap W \cup \text{Supp}\,\epsilon \cap W \subseteq \text{strs}\,E \cup \text{strs}\,E' \cup (\{(n', \varphi_1 E')\} \cap W)$. By the coherence of W, $\{(n', \varphi_1 E')\} \cap W \subseteq \{(n', E')\}$. Thus $\text{Supp}\,\varphi \cap W \subseteq \text{strs}\{S, S'\}$.

Proof of (11.3): $\text{Inv}\,\varphi \subseteq \text{Inv}\,\varphi_1 \cup \text{Inv}\,\epsilon \subseteq \text{strnames}\{E, E'\} \cup \text{strnames}(n', \varphi_1 E') \cup \text{strnames}(n, E) \subseteq \text{strnames}\{S, S'\}$ by (11.17).

Proof of (11.5): $\varphi\,W$ is obtained from $\varphi_1\,W$ by replacing all occurrences of $(n', \varphi_1 E')$ by (n, E). But (n, E) is already in $\varphi_1\,W$. It follows that $\varphi\,W$ is finite, coherent, and that $\varphi\,W$ and M–strs W are substructure closed. Hence $\varphi\,W$ is simple.

Proof of (11.6): φ is not *ID*. If $\varphi_1 = ID$ then $|\varphi\,W| = |\epsilon\,W| = |W| - 1$ as we saw in the proof of (11.5). Otherwise, $|\varphi\,W| = |\epsilon\varphi_1\,W| = |\varphi_1\,W| - 1 < |W| - 1 < |W|$ by (11.20).

Proof of (11.7): Take a (n_0, E_0) with $n_0 \in \text{strnames}(\varphi\,S)$. Since $\varphi\,S = S \in M$–Str we have $n_0 \in M$. Since φ is an M-realisation, φ glides on (n_0, E_0).

$\boxed{n \notin M \text{ and } n' \in M.}$ Symmetric.

$\boxed{n \notin M \text{ and } n' \notin M.}$ where also $S \notin \text{strs}\,E'$ and $S' \notin \text{strs}\,E$ and $\varphi_1 = \mathit{Unify}(M, E, E')$,

$$\epsilon = \{(n, \varphi_1 E) \mapsto (n, \varphi_1 E \cup \varphi_1 E'), (n', \varphi_1 E') \mapsto (n, \varphi_1 E \cup \varphi_1 E')\}$$

and $\varphi = \epsilon \circ \varphi_1$.

By induction,

$$\forall \mathit{strid} \in \text{Dom}\,E \cap \text{Dom}\,E',\ \varphi_1\,E\,\mathit{strid} = \varphi_1\,E'\,\mathit{strid} \tag{11.22}$$

$$\text{Supp}\,\varphi_1 \cap W \subseteq \text{strs}\{E, E'\} \tag{11.23}$$

$$\text{Inv}\,\varphi_1 \subseteq \text{strnames}\,E \cup \text{strnames}\,E' \tag{11.24}$$

$$\varphi_1 \text{ is an } M\text{-realisation} \tag{11.25}$$

$$\varphi_1\,W \text{ is simple with respect to } M \tag{11.26}$$

$$\text{If } \varphi_1 \neq ID \text{ then } |\varphi_1\,W| < |W| \tag{11.27}$$

φ_1 does not change the name of any structure whose name is in $\text{strnames}\{\varphi_1\,E, \varphi_1\,E'\}$. (11.28)

Here (11.22) ensures that $E_1 = \varphi_1 E \cup \varphi_1 E'$ really makes sense. Now n does not occur in $\varphi_1 E'$ — since $S \notin \text{strs}\, E'$ and (11.24). Therefore $S \notin \text{strs}\, \varphi_1 E'$ and similarly, $S' \notin \text{strs}\, \varphi_1 E$. It follows from the considerations in the previous section that ϵ is a realisation. It is clearly an M-realisation, so φ is an M-realisation showing (11.4).

Proof of (11.1):

$$\begin{aligned}
\varphi S &= \epsilon \varphi_1(n, E) \\
&= \epsilon(n, \varphi_1 E) \quad \text{as (11.23) and } S \notin \text{strs}\{E, E'\} \\
&= \epsilon(n', \varphi_1 E') \\
&= \epsilon \varphi_1(n', E') \quad \text{as (11.23) and } S' \notin \text{strs}\{E, E'\} \\
&= \varphi S'.
\end{aligned}$$

Proof of (11.2): $\text{Supp}\,\varphi \cap W \subseteq \text{Supp}\,\varphi_1 \cap W \cup \text{Supp}\,\epsilon \cap W \subseteq$ $\text{strs}\{E, E'\} \cup (\{(n, \varphi_1 E), (n', \varphi_1 E')\} \cap W) \subseteq \text{strs}\{E, E'\} \cup \{S, S'\}$ by the coherence of W, so $\text{Supp}\,\varphi \cap W \subseteq \text{strs}\{S, S'\}$.

Proof of (11.3): $\text{Inv}\,\varphi \subseteq \text{Inv}\,\varphi_1 \cup \text{Inv}\,\epsilon \subseteq \text{Inv}\,\varphi_1 \cup \text{strnames}\, S \cup \text{strnames}\, S'$ $\subseteq \text{strnames}\{S, S'\}$.

Proof of (11.5): φW is obtained from $\varphi_1 W$ by replacing all occurrences of $(n, \varphi_1 E)$ and $(n', \varphi_1 E')$ by $(n, \varphi_1 E \cup \varphi_1 E')$. $\varphi_1 W$ is simple by induction and every structure in $\text{strs}\{\varphi_1 E, \varphi_1 E'\}$ is already in $\varphi_1 W$. Therefore φW is simple.

Proof of (11.6): if $\varphi \neq ID$ then *either* $\varphi_1 = ID$ and $\epsilon \neq ID$ *or* $\varphi_1 \neq ID$. In the former case $n \neq n'$ so $|\varphi W| = |\epsilon W| = |W| - 1$ as we saw in the proof of (11.5). In the latter case,

$$\begin{aligned}
|\varphi W| = |\epsilon \varphi_1 W| &\leq |\varphi_1 W| \quad \text{see the proof of (11.5)} \\
&< |W| \quad \text{by (11.27)}
\end{aligned}$$

as required.

Proof of (11.7): n' does not occur in $\varphi_1 E$ or in $\varphi_1 E'$ by (11.24).

Let S_0 be a structure whose name, n_0, is in φS i.e., in $(n, \varphi_1 E \cup \varphi_1 E')$. If $n_0 = n$ then φ_1 does not change the name of S_0 since (11.24) and since ϵ does not change the name of any structure with name n_0 we have that φ does not change the name of S_0.

If $n_0 \neq n$ then n_0 occurs is $\varphi_1 E$ or in $\varphi_1 E'$. Then by (11.28) φ_1 does not change the name of S_0. Since n' does not occur in $\varphi_1 E$ or in $\varphi_1 E'$ we have $n_0 \neq n'$. Therefore ϵ does not change the name of $\varphi_1 S_0$. Thus φ does not change the name of S_0.

Now to environments.

$\boxed{\mathrm{Dom}\, E \cap \mathrm{Dom}\, E' \neq \emptyset}$ Here $\{strid_0 \mapsto S_1\} \cup E_1 = E$, $\{strid_0 \mapsto S_1'\} \cup E_1' = E'$, $\varphi_1 = \mathit{Unify}(M, S_1, S_1')$, $\varphi_2 = \mathit{Unify}(M, \varphi_1 E_1, \varphi_1 E_1')$, and $\varphi = \varphi_2 \circ \varphi_1$.

By induction on the first call of *Unify*,

$$\varphi_1 S_1 = \varphi_1 S_1' \tag{11.29}$$

$$\mathrm{Supp}\, \varphi_1 \cap W \subseteq \mathrm{strs}\{S_1, S_1'\} \tag{11.30}$$

$$\mathrm{Inv}\, \varphi_1 \subseteq \mathrm{strnames}\, S_1 \cup \mathrm{strnames}\, S_1' \tag{11.31}$$

$$\varphi_1 \text{ is an } M\text{-realisation} \tag{11.32}$$

$$\varphi_1 W \text{ is simple with respect to } M \tag{11.33}$$

$$\text{If } \varphi_1 \neq \mathit{ID} \text{ then } |\varphi_1 W| < |W| \tag{11.34}$$

$$\begin{array}{l}\varphi_1 \text{ does not change the name of any structure whose} \\ \text{name is in strnames}\, \varphi_1 S_1.\end{array} \tag{11.35}$$

Now $\mathrm{strs}\{E_1, E_1'\} \subseteq W$ so $\mathrm{strs}\{\varphi_1 E_1, \varphi_1 E_1'\} \subseteq \varphi_1 W$ which is simple with respect to M by (11.33).

Thus by induction on the second call,

$$\forall strid \in \mathrm{Dom}\, \varphi_1 E_1 \cap \mathrm{Dom}\, \varphi_1 E_1',\ \varphi_2 \varphi_1 E_1\, strid = \varphi_2 \varphi_1 E_1'\, strid \tag{11.36}$$

$$\mathrm{Supp}\, \varphi_2 \cap \varphi_1 W \subseteq \mathrm{strs}\{\varphi_1 E_1, \varphi_1 E_1'\} \tag{11.37}$$

$$\mathrm{Inv}\, \varphi_2 \subseteq \mathrm{strnames}(\varphi_1 E_1) \cup \mathrm{strnames}(\varphi_1 E_1') \tag{11.38}$$

$$\varphi_2 \text{ is an } M\text{-realisation} \tag{11.39}$$

$$\varphi_2(\varphi_1 W) \text{ is simple with respect to } M \tag{11.40}$$

$$\text{If } \varphi_2 \neq \mathit{ID} \text{ then } |\varphi_2 \varphi_1 W| < |\varphi_1 W| \tag{11.41}$$

$$\begin{array}{l}\varphi_2 \text{ does not change the name of any structure whose} \\ \text{name is in strnames}\{\varphi_2 \varphi_1 E_1, \varphi_2 \varphi_1 E_1'\}.\end{array} \tag{11.42}$$

Proof of (11.8): If $strid = strid_0$ then use (11.29), else use (11.36).

Proof of (11.9): Suffice to show that φ glides on every $S'' = (n'', E'') \in W \setminus \mathrm{strs}\{E, E'\}$. But φ_1 glides on S'' by (11.31) i.e., $\varphi_1 S'' = (n'', \varphi_1 E'')$. Now n'' does not occur in $\varphi_1 E_1$ or in $\varphi_1 E_1'$ (use (11.31)). Therefore, by (11.38), φ_2 glides on $(n'', \varphi_1 E'')$. Thus $\varphi_2 \circ \varphi_1$ glides on (n'', E'').

Proof of (11.10): $\mathrm{Inv}\, \varphi \subseteq \mathrm{Inv}\, \varphi_2 \cup \mathrm{Inv}\, \varphi_1 \subseteq \mathrm{Inv}\, \varphi_1 \cup \mathrm{strnames}\, E_1$ $\cup\, \mathrm{strnames}\, E_1'$ (by (11.38)) $\subseteq \mathrm{strnames}\, E \cup \mathrm{strnames}\, E'$ by (11.31).

Next, φ is an M-realisation by (11.32) and (11.39) and φW is simple with respect to M by (11.40). Moreover, (11.13) follows from (11.34) and (11.41).

Proof of (11.14): Take an $S_0 = (n_0, E_0)$ with $n_0 \in \text{strnames}\{\varphi\, E, \varphi\, E'\}$. By (11.31) used on (11.35) we get

$$\varphi_1 \text{ does not change the name of any structure whose name is in strnames}\{\varphi_1\, E, \varphi_1\, E'\}. \tag{11.43}$$

By (11.38) this gives

$$\varphi_1 \text{ does not change the name of any structure whose name is in strnames}\{\varphi_2\,\varphi_1\, E, \varphi_2\,\varphi_1\, E'\}. \tag{11.44}$$

Using (11.38) on (11.42) we get

$$\varphi_2 \text{ does not change the name of any structure whose name is in strnames}\{\varphi_2\,\varphi_1\, E, \varphi_2\,\varphi_1\, E'\}. \tag{11.45}$$

This together with (11.44) gives the desired result. ∎

As a corollary we have that unification of S and S' does not affect the M-structures of W.

Corollary 11.6 *Assume $S, S' \in W$, W is simple with respect to M and $\varphi = Unify(M, S, S')$ succeeds. Then M–strs $W = M$–strs$(\varphi\, W)$.*

Proof. On the one hand M–strs $W \subseteq M$–strs$(\varphi\, W)$ since φ is an M-realisation and M–strs W is substructure closed. Moreover, $\varphi\, W$ is coherent and we have just shown that it contains M–strs W. Therefore, any $(m, E) \in M$–strs$(\varphi\, W) \setminus M$–strs W would have to satisfy $m \notin \text{strnames}\, W$. But that is impossible since

$$\begin{aligned}
\text{strnames}(\varphi\, W) &\subseteq \text{Inv}\,\varphi \cup \text{strnames}\, W \\
&\subseteq \text{strnames}\{S, S'\} \cup \text{strnames}\, W \\
&= \text{strnames}\, W.
\end{aligned}$$

∎

11.2 Completeness of *Unify*

We shall now show that *Unify* is *complete* i.e., that it succeeds with a locally principal unifier if a unifier exists and that it fails otherwise.

First, let us say that an M-realisation is an *M-unifier for E and E'* if for all *strid* $\in \text{Dom}\, E \cap \text{Dom}\, E'$ we have $\varphi\, E\; strid = \varphi\, E'\; strid$. We then have

Theorem 11.7 (Completeness of *Unify*) *Assume W is simple with respect to M and that $S, S' \in W$. For every M-unifier, ψ, for S and S', the call $\varphi = Unify(M, S, S')$ succeeds and there exists an M-realisation, ψ', such that*

$$\psi \downarrow W = (\psi' \circ \varphi) \downarrow W. \tag{11.46}$$

If S and S' do not have an M-unifier, then $Unify(M, S, S')$ fails.

Similarly, assume $\mathrm{strs}\{E, E'\} \subseteq W$. *For every M-unifier, ψ, for E and E', the call $\varphi = Unify(M, E, E')$ succeeds and there exists an M-realisation such that (11.46). If E and E' do not have an M-unifier, then $Unify(M, E, E')$ fails.*

Proof. The proof relies on theorem 11.5. In particular (11.6) and (11.13) give a very painless proof of termination. The trick is to have an outer induction on $|W|$ with an inner structural induction on the structures in W.

Outer Base Case, $|W| = 0$ Thus $W = \emptyset$. The part of the statement regarding structures is vacuously true. For the second part, $\mathrm{strs}\{E, E'\} \subseteq W$ implies $E = E' = \{\}$. For every M-unifier, ψ, for $\{\}$ and $\{\}$ (i.e., for every M-realisation) $\varphi = Unify(M, \{\}, \{\})$ succeeds and

$$\psi \downarrow W = (\psi \circ ID) \downarrow W = (\psi \circ \varphi) \downarrow W$$

as desired.

Outer Inductive Step, $|W| \geq 1$ We assume that the theorem holds for all simple W' with $|W'| < |W|$ and prove that it holds for W by structural induction on the structures in W.

Inner Base Case, $E = E' = \{\}$ Handled as above.

Inner Inductive Step Let us first consider structures $S = (n, E)$ and $S' = (n', E')$. There are four, and only four, possibilities.

$n \in M$ and $n' \in M$ Let ψ be an M-unifier for S and S'. Then $S = \psi S = \psi S' = S'$, $\varphi = Unify(M, S, S')$ returns $\varphi = ID$ and $\psi \downarrow W = (\psi \circ \varphi) \downarrow W$. If S and S' do not have an M-unifier we must have $S \neq S'$ (as otherwise ID would be an M-unifier), so $n \neq n'$ by the coherence of W so $Unify$ fails.

$\boxed{n \in M \text{ and } n' \notin M}$ Let ψ be an M-unifier for S and S'. Since $(n, E) = \psi(n, E) = \psi(n', E')$ we must have $\mathrm{Dom}\, E' \subseteq \mathrm{Dom}\, E$ and $\psi\, E'\, strid = E\, strid = \psi\, E\, strid$ for all $strid \in \mathrm{Dom}\, E'$. Thus by the inner induction, $\varphi_1 = Unify(M, E, E')$ succeeds and there exists an M-realisation, ψ_1 with

$$\psi \downarrow W = (\psi_1 \circ \varphi_1) \downarrow W. \tag{11.47}$$

As we saw in the soundness proof, $\{(n', \varphi_1\, E') \mapsto (n, E)\}$ really does denote a realisation map so it makes sense to say that *Unify* succeeds with the composition $\varphi = \epsilon \circ \varphi_1$.

Moreover, $\varphi_1\, W$ is coherent and $\varphi\, W$ is coherent and the latter is obtained from the former by replacing all occurrences of $(n', \varphi_1\, E')$ by (n, E). Consider an $S_1 = (n_1, E_1) \in \varphi\, W$. If $n_1 \neq n$ there exists exactly one structure, call it $\epsilon^{-1}(S_1)$, in $\varphi_1\, W$ such that

$$\epsilon(\epsilon^{-1}\, S_1) = S_1\, .$$

Thus we can define for each $S_1 \in \varphi\, W$,

$$\psi'\, S_1 = \begin{cases} \psi_1(\epsilon^{-1}\, S_1) & \text{if } n \text{ of } S_1 \neq n, \\ \psi\, S & \text{if } n \text{ of } S_1 = n. \end{cases}$$

Extend ψ' to a total map on Str by letting it glide outside $\varphi\, W$. Now let us prove that ψ' is an M-realisation with the desired properites.

First, to prove that ψ' is a realisation we shall prove that for all $S_1 \in \varphi\, W$, and for all *strid*, if $S_1\, strid$ exists then $(\psi'\, S_1) strid$ exists and $(\psi'\, S_1) strid = \psi'(S_1\, strid)$.

When n of $S_1 \neq n$ and n of $(S_1\, strid) \neq n$ this follows from the definition of ψ' and the fact that ψ_1 is a realisation.

The two remaining cases are

Case 1: n of $S_1 = n$ and n of $(S_1\, strid) \neq n$, where

$$\begin{array}{lll}
\psi'(S_1\, strid) & = & \psi'(S\, strid) \\
& = & \psi'\, E\, strid \\
& = & \psi_1\, \epsilon^{-1}(E\, strid) \quad \text{as } n \text{ of } (E\, strid) \neq n \\
& = & \psi_1(E\, strid) \quad \text{as strnames } E \subseteq M \\
& = & \psi\, E\, strid \quad \text{by (11.47)} \\
& = & (\psi\, S) strid \quad \text{as } \psi \text{ is a realisation} \\
& = & (\psi'\, S_1) strid \quad \text{by the definition of } \psi'.
\end{array}$$

Case 2: n of $S_1 \neq n$ and n of $(S_1\, strid) = n$. Here ϵ glides on $\epsilon^{-1}(S_1)$, so $(\epsilon^{-1}(S_1)) strid$ must exist and

$$\begin{aligned}
\epsilon((\epsilon^{-1} S_1) strid) &= (\epsilon(\epsilon^{-1} S_1)) strid \quad \text{as } \epsilon \text{ is a realisation} \\
&= S_1\, strid \\
&= (n, E),
\end{aligned}$$

so $(\epsilon^{-1}(S_1)) strid = (n, E)$ or $(\epsilon^{-1}(S_1)) strid = (n', \varphi_1 E')$. In either case, by (11.47), we get $\psi_1((\epsilon^{-1} S_1) strid) = \psi\, S$. Thus, since ψ_1 is a realisation,

$$(\psi_1(\epsilon^{-1} S_1) strid = \psi\, S$$

i.e.,

$$(\psi' S_1) strid = \psi'(S_1\, strid)$$

as desired.

This proves that ψ' is a realisation. That it is an M-realisation follows from the definition of ϵ^{-1} and the fact that ψ_1 is an M-realisation.

To prove (11.46) take a $w \in W$. If $w = S$ then $\psi w = S = (\psi' \circ \varphi) S$ as ψ and $\psi' \circ \varphi$ are M-realisations. Otherwise, n of $w \neq n$. Since $\text{Inv}\, \varphi_1 \subseteq$ strnames $E \cup$ strnames E' and n does not occur in E or in E' we have $n\text{of}(\varphi_1 w) \neq n$. If $n\text{of}(\varphi_1 w) \neq n$ then $n\text{of}(\epsilon(\varphi_1 w)) \neq n$ so $\psi'(\varphi w) = \psi'(\epsilon(\varphi_1 w)) = \psi_1(\varphi_1 w) = \psi w$ by (11.47). Otherwise n of $(\varphi_1 w) = n$, so

$$\begin{aligned}
\psi' \varphi w &= \psi'(\epsilon \varphi_1 w) \\
&= \psi'(\epsilon(n', \varphi_1 E')) \\
&= \psi'(n, E) && \text{by the definition of } \epsilon \\
&= \psi\, S && \text{by the definition of } \psi' \\
&= \psi\, S' && \text{as } \psi \text{ is a unifier} \\
&= \psi_1 \varphi_1 S' && \text{by (11.47)} \\
&= \psi_1(n', \varphi_1 E') && \text{as } n' \notin \text{Inv}\, \varphi_1 \\
&= \psi_1(\varphi_1 w) \\
&= \psi w && \text{by (11.47).}
\end{aligned}$$

This proves (11.46).

Conversely, assume that S and S' have no M-unifier. If $\text{Dom}\, E' \not\subseteq \text{Dom}\, E$ then *Unify* fails, but if $\text{Dom}\, E' \subseteq \text{Dom}\, E$, there cannot be an M-unifier for E and E' (since if φ_1 were one, $\epsilon \circ \varphi_1$ would unify S and S' with ϵ as in the program). Therefore, by the inner induction, *Unify*(M, E, E') will fail so that also *Unify*(M, S, S') fails.

$n \notin M$ and $n' \in M$ Symmetric.

$\boxed{n \notin M \text{ and } n' \notin M}$ Similar to the two previous cases. Here are the details.

Assume ψ is an M-unifier for S and S'. Then $S \notin \text{strs}\, E'$ and $S' \notin \text{strs}\, E$ so *Unify* passes the test and calls $\varphi_1 = \mathit{Unify}(M, E, E')$. Since ψ is also an M-unifier for E and E' we get by the inner induction that $\varphi_1 = \mathit{Unify}(M, E, E')$ succeeds and that there exists an M-realisation ψ_1 with

$$\psi \downarrow W = (\psi_1 \circ \varphi_1) \downarrow W. \tag{11.48}$$

As we saw in the proof of theorem 11.5,

$$\epsilon = \{(n, \varphi_1 E) \mapsto (n, \varphi_1 E \cup \varphi_1 E'), (n', \varphi_1 E') \mapsto (n, \varphi_1 E \cup \varphi_1 E')\}$$

really is a realisation map, so it makes sense to say that *Unify* succeeds with result $\varphi = \epsilon \circ \varphi_1$.

We also saw that $\varphi_1 W$ is coherent, that φW is coherent, and that the latter is obtained from the former by replacing all occurrences of $(n, \varphi_1 E)$ and $(n', \varphi_1 E')$ by $(n, \varphi_1 E \cup \varphi_1 E')$.

Consider a $S_1 = (n_1, E_1) \in \varphi W$. If $n_1 \neq n$ then there exists exactly one structure in $\varphi_1 W$, call it $\epsilon^{-1}(S_1)$, that satisfies

$$\epsilon(\epsilon^{-1}(S_1)) = S_1 .$$

(If $n_1 = n$, S_1 is the value of ϵ on two structures, if $n \neq n'$, and on one structure, if $n = n'$). Thus it makes sense to define for all $S_1 \in \varphi W$,

$$\psi'(S_1) = \begin{cases} \psi_1(\epsilon^{-1}(S_1)) & \text{if } n \text{ of } S_1 \neq n, \\ \psi S & \text{if } n \text{ of } S_1 = n. \end{cases}$$

Extend ψ' to a total map by letting it glide outside φW. To prove that ψ' is a realisation we prove that for all $S_1 \in \varphi W$ and for all *strid*, if $S_1\, \mathit{strid}$ exists then $(\psi' S_1)\mathit{strid}$ exists and $(\psi' S_1)\mathit{strid} = \psi'(S_1\, \mathit{strid})$.

Again this is clear when n of $S_1 \neq n$ and n of $(S_1\, \mathit{strid}) \neq n$. The remaining cases are:

Case 1: n of $S_1 = n$ and n of $(S_1\, \mathit{strid}) \neq n$.

Here $S_1\, \mathit{strid} = (n, \varphi_1 E \cup \varphi_1 E')\mathit{strid}$ by the coherence of φW.

If $\mathit{strid} \in \text{Dom}\, E$ we have

$$\begin{aligned} \psi'(S_1\, \mathit{strid}) &= \psi'(\varphi_1 E\, \mathit{strid}) \\ &= \psi_1 \epsilon^{-1}(\varphi_1 E\, \mathit{strid}) && \text{as } n \text{ of } (\varphi_1 E\, \mathit{strid}) \neq n \\ &= \psi_1 \varphi_1 E\, \mathit{strid} && \text{as } n \notin \text{strnames}(\varphi_1 E\, \mathit{strid}) \\ &= \psi E\, \mathit{strid} && \text{by (11.48)} \end{aligned}$$

$$\begin{aligned} &= (\psi\, S)\mathit{strid} && \text{as } \psi \text{ is a realisation} \\ &= (\psi'\, S_1)\mathit{strid} && \text{by the definition of } \psi'. \end{aligned}$$

Otherwise $\mathit{strid} \in \mathrm{Dom}\, E'$ and we get

$$\begin{aligned} \psi'(S_1\, \mathit{strid}) &= (\psi\, S')\mathit{strid} && \text{as above} \\ &= (\psi\, S)\mathit{strid} && \text{since } \psi \text{ unifies } S \text{ and } S' \\ &= (\psi'\, S_1)\mathit{strid} \end{aligned}$$

as desired.

Case 2: n of $S_1 \neq n$ and n of $(S_1\, \mathit{strid}) = n$. Since $S_1\, \mathit{strid}$ exists and n of $S_1 \neq n$, $(\epsilon^{-1}\, S_1)\mathit{strid}$ must exist. Now ϵ is a realisation map (see the proof of theorem 11.5) so $(\epsilon(\epsilon^{-1}\, S_1))\mathit{strid}$ exists and

$$(\epsilon(\epsilon^{-1}\, S_1))\mathit{strid} = S_1\, \mathit{strid} = \epsilon((\epsilon^{-1}\, S_1)\mathit{strid}).$$

Since $S_1\, \mathit{strid} = (n, \varphi_1\, E \cup \varphi_1\, E')$ we must have

$$(\epsilon^{-1}\, S_1)\mathit{strid} = (n, \varphi_1\, E)$$

or

$$(\epsilon^{-1}\, S_1)\mathit{strid} = (n', \varphi_1\, E').$$

In either case, (11.48) gives $\psi_1((\epsilon^{-1}\, S_1)\mathit{strid}) = \psi\, S$. Since ψ_1 is a realisation, $(\psi_1(\epsilon^{-1}\, S_1))\mathit{strid}$ exists and

$$(\psi_1(\epsilon^{-1}\, S_1))\mathit{strid} = \psi\, S$$

i.e.,

$$(\psi'\, S_1)\mathit{strid} = \psi'(S_1\, \mathit{strid})$$

as desired.

This proves that ψ' is a realisation. That it is an M-realisation is obvious.

To show (11.46) take $w \in W$. If $w = S$ or $w = S'$ then $(\psi' \circ \varphi)W = (\psi' \circ \epsilon \circ \varphi_1)w = \psi\, S = \psi\, w$.

Otherwise $n \mathrm{of} w \neq n$ and $n \mathrm{of} w \neq n'$ by the coherence of W. By theorem 11.5, $\mathrm{Inv}\, \varphi_1 \subseteq \mathrm{strnames}\, E \cup \mathrm{strnames}\, E'$ and since

$$\{n, n'\} \cap (\mathrm{strnames}\, E \cup \mathrm{strnames}\, E') = \emptyset$$

we have $\{n, n'\} \cap \mathrm{Inv}\, \varphi_1 = \emptyset$. Therefore, n of $(\varphi_1\, w) \notin \{n, n'\}$. Thus by the definition of ψ', $\psi'(\varphi'\, w) = \psi'(\epsilon(\varphi_1\, w)) = \psi_1(\varphi_1\, w) = \psi\, w$ by (11.48). This shows (11.46).

Conversely assume that S and S' have no M-unifier. If $S \in \operatorname{strs} E'$ or $S' \in \operatorname{strs} E$ then *Unify* fails. But if $S \notin \operatorname{strs} E'$ and $S' \notin E'$ there cannot be any M-unifier for E and E' (if φ_1 were one, $\epsilon \circ \varphi_1$ would be an M-unifier for S and S', where ϵ is as in the algorithm). Therefore by the inner induction, $Unify(M, E, E')$ fails so that also $Unify(M, S, S')$ fails.

Continuing the case analysis in the inner inductive step, we now turn to environments. So assume that not both of E and E' are the empty map. There are two possibilities:

$\boxed{\operatorname{Dom} E \cap \operatorname{Dom} E' = \emptyset}$ Trivial

$\boxed{\operatorname{Dom} E \cap \operatorname{Dom} E' \neq \emptyset}$ It is possible to split E and E' in disjoint unions $E = \{strid \mapsto S_1\} \cup E_1$ and $E' = \{strid \mapsto S'_1\} \cup E'_1$, where $strid \in \operatorname{Dom} E \cap \operatorname{Dom} E'$. Assume ψ is an M-unifier for E and E'. Then ψ is also an M-unifier for S_1 and S'_1 so by the *inner* induction, $\varphi_1 = Unify(M, S_1, S'_1)$ succeeds and there exists and M-realisation ψ_1 such that

$$\psi \downarrow W = (\psi_1 \circ \varphi_1) \downarrow W. \tag{11.49}$$

Since $\operatorname{strs}\{E_1, E'_1\} \subseteq W$ we have $\operatorname{strs}\{\varphi_1 E_1, \varphi_1 E'_1\} \subseteq \varphi_1 W$. Moreover, ψ is an M-unifier for E_1 and E'_1 so by (11.49) ψ_1 is an M-unifier for $\varphi_1 E$ and $\varphi_1 E'_1$.

Now if $\varphi_1 = ID$ we have $\varphi_1 E_1 = E_1$ and $\varphi_1 E'_1 = E'_1$ and $\varphi_1 W = W$ and we simply apply the *inner* induction to conclude that $\varphi_1 = Unify(M, \varphi_1 E_1, \varphi_1 E'_1)$ succeeds and that there exists an M-realisation ψ' such that

$$\psi_1 \downarrow (\varphi_1 W) = (\psi' \circ \varphi_2) \downarrow \varphi_1 W$$

observing that *Unify* succeeds with $\varphi = \varphi_2 \circ \varphi_1$ and thus $\psi \downarrow W = (\psi' \circ \varphi) \downarrow W$.

On the other hand, if φ_1 is *not ID*, we use (11.6) of theorem 11.5 to infer $|\varphi_1 W| < |W|$. Then we use the *outer* induction to infer the same conclusions.

Conversely, assume that E and E' have no M-unifier. Then

$$\operatorname{Dom} E \cap \operatorname{Dom} E' \neq \emptyset.$$

If S_1 and S'_1 have no M-unifier then $\varphi_1 = Unify(M, S_1, S'_1)$ fails by the inner induction, so $Unify(M, S_1, S'_1)$ fails. If S_1 and S'_1 have a unifier, ψ say, then by the inner induction, $\varphi_1 = Unify(M, S_1, S'_1)$ succeeds. But then $\varphi_1 E_1$ and $\varphi_1 E'_1$ have no unifier (if ψ were one, then $\psi \circ \varphi_1$ would be a unifier for E and E'). Therefore the second call fails — by the inner induction, if $\varphi_1 = ID$, and by the outer induction otherwise.

End of inner inductive step.

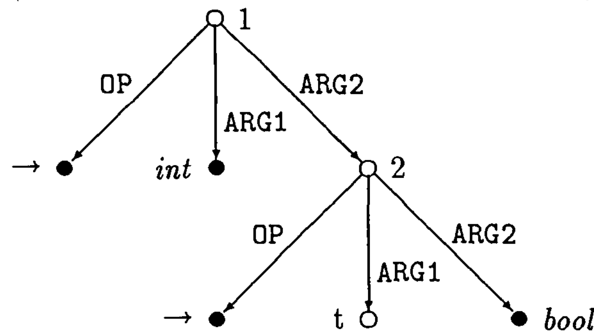
End of outer inductive step.

■

We now see that our initial theorem, theorem 11.4, follows from the soundness and completeness theorems (theorem 11.5 and theorem 11.7). Assume that $\{S, S'\}$ is coherent and that M–strs$\{S, S'\}$ is substructure closed. Then W = strs$\{S, S'\}$ is simple with respect to M. Either there exists an M-unifier for S and S' or there does not. In the former case, $\varphi = \mathit{Unify}(M, S, S')$ succeeds (by the completeness theorem), φ is an M-unifier (by the soundness theorem) which in fact is locally principal (by the completeness theorem). In the latter case, $\mathit{Unify}(M, S, S')$ fails by the completeness theorem.

11.3 Comparison With Term Unification

First order term unification [37] can be seen as a special case of structure unification. It is easy to encode terms as coherent structures. For instance, the term $t = \mathit{int} \rightarrow (\mathrm{t} \rightarrow \mathit{bool})$ can be represented by the structure rep(t) =



To be a bit more precise, take M to be the function symbols and constants (since unification must not change these). Take StrName to be the disjoint union of M, TyVar, and the natural numbers. Take StrId to be the (infinite) set $\{\mathtt{OP}, \mathtt{ARG1}, \mathtt{ARG2}, \ldots\}$. All internal nodes in rep(t) are named with different natural numbers to obtain coherence. Clearly the representations have substructure closed intersections with M–Str. Thus we can apply our algorithm and our results.

For every two terms t_1 and t_2 and every unifier for t_1 and t_2 there exists an M-unifier for rep(t_1) and rep(t_2). Conversely, if each function symbol has a fixed arity, every M-unifier for rep(t_1) and rep(t_2) gives a unifier for t_1 and t_2.

Still under the assumption that each function symbol has a fixed arity, a principal M-unifier for rep(t_1) and rep(t_2) gives a maximal unifier for t_1 and t_2. Therefore, the structure unification algorithm is also a decision procedure for the existence of term unifiers, and when it succeeds, it produces an M-unifier from which a maximal term unifier can be derived.[2]

11.4 Comparison with Aït–Kaci's Type Discipline

Another theory that generalizes first order unification has been developed by Hassan Aït–Kaci [3]. The purpose of this section is to point out the similarities and differences between his work and ours. The two theories were developed independently of each other.

I shall start with a review of some of the definitions and results from [3]. I take the liberty to reformulate some definitions slightly to ease the comparison.

There is given a set, L, of *labels.* A label corresponds to a structure identifier in our terminology. A *term domain* Δ on L is a subset of L^* which is prefix closed and finitely branching i.e.,

$$\forall u, v \in L^* \quad \text{if } u.v \in \Delta \text{ then } u \in \Delta \tag{11.50}$$

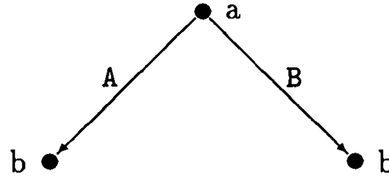
$$\text{if } u \in \Delta \text{ then } \{u.a \in \Delta \mid a \in L\} \text{ is finite.} \tag{11.51}$$

Next, we have a set, Σ, of *symbols.* A symbol corresponds to a structure name. Aït–Kaci considers the situation that Σ is any partially ordered set and in particular he studies the case where Σ is a lattice. This special case is general enough for our comparison, so from now on let us assume that Σ is a lattice. A map ψ maps paths in Δ to symbols in Σ hence decorating the nodes of the tree. Finally, to represent sharing, a *coreference relation* κ is an equivalence relation on Δ. A *term* is a triple $t = (\Delta, \psi, \kappa)$. The subterm of t at address w is written $t \setminus w$. Then a *well-formed* term is a term $t = (\Delta, \psi, \kappa)$ satisfying

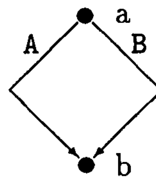
$$\forall u, v \in \Delta, \quad \text{if } u \,\kappa\, v \text{ then } t \setminus u = t \setminus v. \tag{11.52}$$

[2] The method by which we proved termination of the unification algorithm can also be used to give a simple termination argument for ordinary term unification. Use an outer induction on the size of the set W of terms *and their subterms*, with an inner structural induction on the terms of W, the point being that if a substitution "expands" a term it decreases the total number of terms and subterms under consideration.

Apart from the generality that lies in having σ be a lattice, these definitions give a richer variety of objects than our structures. Firstly, Δ need not be finite, so in particular terms can be cyclic. Secondly, one has the ability to distinguish between for example



and



(The first of these has three coreference classes, the second only two). In the modules semantics based on coherence sharing is the same as structure identity, so it is the latter interpretation we want. Any structure $S = (n, E)$ can be drawn as a tree or, equivalently, it can be regarded as a pair $S = (\Delta, \psi)$, where Δ is a finite term domain and ψ a map from addresses in Δ (i.e., nodes) to symbols (i.e., structure names). We then turn a "structure" $S = (\Delta, \psi)$ into a well-formed term $\text{rep}(S) = (\Delta, \psi, \kappa)$ by taking $\kappa = \kappa^{\text{max}}$, where

$$\forall u, v \in \Delta, \quad u\, \kappa^{\text{max}}\, v \quad \text{iff} \quad S(u) = S(v). \tag{11.53}$$

In other words, we take the largest possible coreference relation — compare with (11.52).

Then Aït–Kaci defines a relation $\preceq$ on terms as follows. Let $t_i = (\Delta_i, \psi_i, \kappa_i)$ for $i = 1, 2$. Then $t_1 \succeq t_2$ if

$$\Delta_1 \subseteq \Delta_2 \tag{11.54}$$

$$\kappa_1 \subseteq \kappa_2 \tag{11.55}$$

$$\forall w \in \Delta_1,\ \psi_1(w) \geq \psi_2(w). \tag{11.56}$$

This looks a bit like realisation maps (since realisations preserve paths (11.54) and sharing (11.55)) but the symbol ordering poses a problem. The most natural thing would be to take Σ to be the set of structure names with every name related to any other name but this fails because the total relation is not an ordering, let

alone a lattice. Instead, we shall take the flat lattice obtained by adding a gratest element, $\top$, and a smallest element, $\perp$, to the unordered set of structure names. Then we can think of a structure as a pair $S = (\Delta, \psi)$, where $\text{Rng}\,\psi \cap \{\top, \perp\} = \emptyset$.

Let us say that a realisation map φ is *name preserving* if $n \text{ of } S = n \text{ of } (\varphi\, S)$ for all S. Then

Observation 11.8 *There exists a name preserving realisation map, φ, with $\varphi\, S_1 = S_2$ if and only if* $\text{rep}(S_1) \succeq \text{rep}(S_2)$.

Proof. Assume $\varphi\, S_1 = S_2$, where φ is a name preserving realisation. Clearly $\Delta_1 \subseteq \Delta_2$. Since φ is a map from structures to structures, we have $\kappa_1^{\max} \subseteq \kappa_2^{\max}$ and since φ is name preserving we have $\psi_1(w) = \psi_2(w)$ showing (11.56).

Conversely, if $\text{rep}(S_1) \succeq \text{rep}(S_2)$ then $\Delta_1 \subseteq \Delta_2$ so for all $w \in \Delta_1$ we can define

$$\psi(S_1(w)) = S_2(w).$$

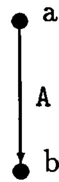
Since $\kappa_1^{\max} \subseteq \kappa_2^{\max}$ and these are maximal, we have that this equation actually defines a map on S_1 and its substructures. Extend this to a total map, also called ψ, letting ψ glide outside S_1. By definition ψ is a realisation. Moreover, $\text{rep}(S_1)$ does not contain $\top$ and $\text{rep}(S_2)$ does not contain $\perp$, so by (11.56), ψ preserves names. ■

Unfortunately, name preserving realisations are not terribly interesting in the modules semantics. Indeed, if we are not allowed to change the name of a given structure it is because the structure is "constant" (Section 8.4) and so we should not be allowed to add more components either.

This is the crucial difference between the two approaches. In Aït–Kaci's approach, the adding of components is independent from the symbol ordering. In our approach, the two are linked in that widening is allowed for structures with bound names only. (Names not in M can be represented by $\top$ so that they can be instantiated to either $\top$ or to an ordinary structure name). For instance in Aït–Kaci's approach one can unify

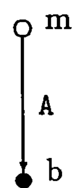
and

to obtain



whereas for us, the corresponding unification of

and



must fail.

Aït–Kaci has an elegant proof that if Σ is a lattice then the well-formed terms over Σ form a lattice with the ordering $\preceq$. The construction of the greatest lower bound corresponds to unification and is construed as a closure operation on the coreference relations. This operation is very similar to the algorithm *Unify* up to the difference described above.

Chapter 12

Principal Signatures

In this chapter we shall prove that if a signature expression has a signature then it has a principal signature, as defined in definition 8.7. We do so by giving an algorithm, a so-called *signature checker*, which either fails or succeeds and succeeds with a principal signature when any legal signature exists.

The algorithm is called SC, which can be read "signature checker" or "specification checker" depending on the phrase class, and uses the unification algorithm defined in Chapter 11. Before disclosing the fascinating inner workings of SC, let us summarize its properties.

In what follows, U ranges over finite subsets of StrName. Intuitively, U means the set of "used" names.

Theorem 12.1 (Simple Soundness of SC) *Assume* $B = M, F, G, E$ *is robust and that* $\mathrm{strnames}\, B \subseteq U$. *If*

$$(\varphi, S, U') = SC(B, sigexp, U)$$

succeeds then φ *is an* M*-realisation and*

$$\varphi B \vdash sigexp \Rightarrow (\emptyset)\, S\,.$$[1]

Similarly, if

$$(\varphi, E', U') = SC(B, spec, U)$$

succeeds then φ *is an* M*-realisation and*

$$\varphi B \vdash spec \Rightarrow E'\,.$$

[1] Because of the global constraints (definition 9.12), this says rather a lot about $\varphi(B)$ and S.

Intuitively, U contains all names "used" in B and U' is the set of new names used in the call i.e., $U \cap U' = \emptyset$.

Recall from Section 9.2 that $\varphi\, B$, the application of φ to B, is defined by

$$\varphi\, B = \varphi(M, F, G, E) = (M, F, G, \varphi\, E).$$

Here E may contain variable structures that are affected by unification that takes place during the call of SC. At the level of signature declarations, B will be strongly robust, so that $\varphi\, E = E$, but if we are inside a specification or signature expression then E may contain variable structures.

Moreover, SC has the property that it chooses new names in such a way that

Theorem 12.2 *Assume B is robust and that* strnames $B \subseteq U$. *If*

$$(\varphi, S, U') = SC(B, sigexp, U)$$

succeeds then for every substructure $S_0 = (n_0, E_0)$ of S,
either $n_0 \in \text{strnames}(\varphi\, B)$ *and* $S_0 \in \text{strs}(\varphi\, B)$,
or $n_0 \in U' \setminus \text{strnames}(\varphi\, B)$.

The principal signature is obtained as the closure

$$\Sigma = \text{Clos}_{\varphi B}\, S\,.$$

(Recall from definition 9.14 that $\text{Clos}_{\varphi B}\, S$ means $(N)S$, where
$N = \text{strnames}\, S \setminus \text{strnames}(M, F, G, \varphi\, E)$.) Note that theorem 12.2 implies that Σ is well-formed. (That Σ is coherent, in fact that $\{B, \Sigma\}$ is coherent, follows from theorem 12.1).

As for completeness, still assuming $B = (M, F, G, E)$,

Theorem 12.3 (Simple Completeness of SC) *Assume that B is robust and* strnames $B \subseteq U$ *and that ψ is an M-realisation and Σ a signature such that*

$$\psi\, B \vdash sigexp \Rightarrow \Sigma.$$

Then $(\varphi, S, U') = SC(B, sigexp, U)$ succeeds and there exists an M-realisation, ψ', such that whenever $\text{Clos}_{\varphi B} S \xrightarrow{\psi'} \Sigma'$ then

$$(\psi' \circ \varphi)\, E = \psi\, E \quad \textit{and} \quad \Sigma' \geq \Sigma.$$

Moreover, if no such ψ, Σ exist then SC fails.

Similarly for specifications.

In other words, if *sigexp* becomes legal by adding components and sharing to the variable structures in B, then SC succeeds and the sharing and extra components it introduces is as little as necessary.

Theorem 12.4 (Principal Signatures) *If sigexp has a signature in B then it has a principal signature in B.*

Proof. The proof uses theorem 12.1, theorem 12.2 and theorem 12.3 all of which we shall soon have the opportunity to prove. Moreover, we use corollary 10.6.

Assume $B \vdash sigexp \Rightarrow \Sigma$. Then by the global constraints (definition 9.12), B is robust. Let $U = \text{strnames}\, B$ and let $\psi = ID$. Thus $\psi\, B \vdash sigexp \Rightarrow \Sigma$, so by theorem 12.3,

$$(\varphi, S, U') = SC(B, sigexp, U)$$

succeeds and there exists an M-realisation, ψ', such that whenever $\text{Clos}_{\varphi B} S \xrightarrow{\psi'} \Sigma'$ then

$$(\psi' \circ \varphi)\, E = E \quad \text{and} \quad \Sigma' \geq \Sigma.$$

By theorem 12.1,

$$\varphi\, B \vdash sigexp \Rightarrow (\emptyset)\, S\,. \tag{12.1}$$

Moreover, by theorem 12.2, we can strengthen (12.1) to

$$\varphi\, B \vdash sigexp \Rightarrow \text{Clos}_{\varphi B}\, S \tag{12.2}$$

by use of the generalization rule without violating the global constraints. Let Σ' be such that $\text{Clos}_{\varphi B} S \xrightarrow{\psi'} \Sigma'$. Then $\psi'(\varphi\, B) = B$ and $\Sigma' \geq \Sigma$. Thus it will suffice to prove that

$$B \vdash sigexp \Rightarrow \Sigma'. \tag{12.3}$$

This we wish to prove by applying corollary 10.6 on (12.2). Thus we must prove that M–strs Σ' is substructure closed and that $\{B, \Sigma'\}$ is coherent. But since every structure free in $\text{Clos}_{\varphi B}\, S$ occurs free in $\varphi\, B$, every structure free in Σ' occurs free in $\psi'(\varphi\, B) = B$. Therefore M–strs Σ' is substructure closed and $\{B, \Sigma'\}$ is coherent. Thus the desired (12.3) follows from (12.2). ■

The reader will probably already have noticed the similarity with the soundness and completeness results that hold for the purely applicative type discipline (see Part I, Section 2.4).

12.1 The Signature Checker, SC

The algorithm SC appears below.

$SC(B, spec, U) =$
let $(M, F, G, E) = B$ **in**
case *spec* **of**

$\langle \quad \rangle : \quad (ID, \{\}, \emptyset)$

$\langle$**structure** *strid* : *sigexp*$\rangle$:

let $(\varphi_1, S_1, U_1) = SC(B, sigexp, U)$
in $(\varphi_1, \{strid \mapsto S_1\}, U_1)$

$\langle$**sharing** $longstrid_1 = longstrid_2\rangle$:

let $strid_1.longstrid'_1 = longstrid_1$
fail if $strid_1 \notin \mathrm{Dom}\, E$
$(R_1, U_1) = Fresh(longstrid'_1, U)$
$\varphi_1 = Unify(M, R_1, E(strid_1))$
$strid_2.longstrid'_2 = longstrid_2$
fail if $strid_2 \notin \mathrm{Dom}\, E$
$(R_2, U_2) = Fresh(longstrid'_2, U \cup U_1)$
$\varphi_2 = Unify(M, R_2, (\varphi_1\, E)strid_2)$
$\varphi_3 = Unify(M, (\varphi_2 \varphi_1\, E)longstrid_1, (\varphi_2 \varphi_1\, E)longstrid_2)$
in $(\varphi_3 \circ \varphi_2 \circ \varphi_1, \{\}, U_1 \cup U_2)$

$\langle spec_1\ spec_2\rangle$:

let $(\varphi_1, E_1, U_1) = SC(B, spec_1, U)$
$(\varphi_2, E_2, U_2) = SC(\varphi_1\, B \pm E_1, spec_2, U \cup U_1)$
in $(\varphi_2 \circ \varphi_1, \varphi_2\, E_1 \pm E_2, U_1 \cup U_2)$
end

$SC(B, sigexp, U) =$
let $(M, F, G, E) = B$ **in**
case *sigexp* **of**

$\langle$**sig** *spec* **end**$\rangle$:

let $(\varphi_1, E_1, U_1) = SC(B, spec, U)$
$n_1 \in \text{StrName} \setminus (U \cup U_1)$
in $(\varphi_1, (n_1, E_1), U_1 \cup \{n_1\})$

$\langle sigid \rangle$:

let $(N)S = G(sigid)$
$\{n_1, \ldots, n_k\} = N$
$\{n'_1, \ldots, n'_k\}$ be disjoint with U
in $(ID, \{n_i \mapsto n'_i\}\, S, \{n'_1, \ldots, n'_k\})$
end

In the algorithm ModL phrases are enclosed in angle brackets to avoid confusion of reserved words. The case

$$spec = \langle \textbf{sharing}\ longstrid_1 = longstrid_2 \rangle$$

deserves some comments. The inference rules admit specifications like

```
structure A : sig  end
structure B : sig  end
sharing A.C = B.C
```

because the generalisation and instantiation rules can be used to widen structures that are not constants. To capture this, *SC* calls a function *Fresh*, which has the following property:

$$(S, U') = Fresh(strid_1.\cdots.strid_k, U), \quad (k \geq 0)$$

always succeeds, S has the form

and $U' = \{n_0, \ldots, n_k\}$, $U \cap U' = \emptyset$ and all the n_i are distinct.

In the case *sigexp* = ⟨*sigid*⟩ the bound names are replaced by fresh names. This is similar to taking a fresh generic instance in the type checker for the applicative type discipline (Part I).

12.2 Soundness of SC

For brevity we shall use the following

Definition 12.5 (Cover) *Let B be robust basis. A set $W \subseteq$ Str covers B if W is simple with respect to M of B and $\mathrm{strs}(B) \subseteq W$.*

Theorem 12.6 (Soundness of SC) *Assume W covers $B = M, F, G, E$ and $M \cup \mathrm{strnames}\, W \subseteq U$. If*

$$(\varphi, S, U') = SC(B, sigexp, U)$$

succeeds, then

$$\varphi \text{ is an } M\text{-realisation and} \tag{12.4}$$

$$\varphi B \vdash sigexp \Rightarrow (\emptyset)\, S\,. \tag{12.5}$$

Moreover, letting $W' = \mathrm{strs}\{\varphi(W), S\}$,

$$W' \text{ covers } \varphi B, \tag{12.6}$$

$$M\text{–strs}\, W' = M\text{–strs}\, W \tag{12.7}$$

$$\mathrm{Inv}\, \varphi \cup \mathrm{strnames}\, S \subseteq \mathrm{strnames}\{G, E\} \cup U' \tag{12.8}$$

$$\varphi \text{ does not change the name of any structure whose name occurs in } \varphi E \text{ or in } S \tag{12.9}$$

$$U \cap U' = \emptyset. \tag{12.10}$$

Similarly for specifications.

Before proving this theorem, let us see some of its implications.

Observation 12.7 *Although in general* $\mathrm{strs}\, B \subset W$*, we always have*

$$\mathrm{Supp}\,\varphi \cap W \subseteq \mathrm{strs}\, E\,. \tag{12.11}$$

Proof. We have $\mathrm{Inv}\,\varphi \subseteq \mathrm{strnames}\, G \cup \mathrm{strnames}\, E \cup U'$. Since $\mathrm{strnames}\, W \subseteq U$ and $U \cap U' = \emptyset$, we have $\mathrm{strnames}(\mathrm{Supp}\,\varphi \cap W) \subseteq \mathrm{strnames}\, G \cup \mathrm{strnames}\, E$ and since φ is an M-realisation this can be strengthened to $\mathrm{strnames}(\mathrm{Supp}\,\varphi \cap W) \subseteq \mathrm{strnames}\, E$. Then the coherence of W ensures (12.11).

In general, it will not be the case that $\mathrm{Supp}\,\varphi \subseteq \mathrm{strs}\, E$ since SC is such that all the changes it performs to intermediate structures are accumulated in φ.

Observation 12.8 *For every substructure* $S_0 = (n_0, E_0)$ *of* S*,* either $n_0 \in \mathrm{strnames}(\varphi\, B)$ *and* $S_0 \in \mathrm{strs}\{G, \varphi\, E\}$ or $n_0 \in U' \setminus \mathrm{strnames}(\varphi\, W)$.

Proof. Take $S_0 = (n_0, E_0) \in \mathrm{strs}\, S$.

We first assume that $n_0 \in \mathrm{strnames}(\varphi\, W)$ and prove that $n_0 \in \mathrm{strnames}(\varphi\, B)$ and $S_0 \in \mathrm{strs}\{G, \varphi\, E\}$. By (12.8) we have $n_0 \in \mathrm{strnames}\, G$, $n_0 \in \mathrm{strnames}\, E$, or $n_0 \in U'$. We treat each of these three cases in turn.

Assume $n_0 \in \mathrm{strnames}\, G$. Now $\mathrm{strs}\, G \subseteq \mathrm{strs}(\varphi\, W) \subseteq W'$ and W' is coherent, so we must have $S_0 \in \mathrm{strs}\, G$.

Assume $n_0 \in \mathrm{strnames}\, E$. Since W is coherent no structure in $W \setminus \mathrm{strs}\, E$ has name n_0. By observation 12.7, φ glides on all structures in $W \setminus \mathrm{strs}\, E$. Since $n_0 \in \mathrm{strnames}(\varphi\, W)$, we must therefore have $S_0 \in \mathrm{strs}(\varphi\, E)$.

Assume $n_0 \in U'$. Then $n_0 \notin U$. Hence there must exist a structure in $\mathrm{Supp}\,\varphi \cap W$ the image of which contains n_0. Then, by observation 12.7 and the coherence of $\mathrm{strs}(\varphi\, W)$, we have $S_0 \in \mathrm{strs}(\varphi\, E)$.

Secondly, we assume that $n_0 \notin \mathrm{strnames}(\varphi\, W)$ and prove that $n_0 \in U'$. (This is where we shall need property (12 9)). By (12.8) we have $n_0 \in \mathrm{strnames}\, G$, $n_0 \in \mathrm{strnames}\, E$, or $n \in \mathrm{strnames}\, U'$. It will suffice to prove that $n_0 \notin \mathrm{strnames}\{G, E\}$. Since $n_0 \notin \mathrm{strnames}(\varphi\, W)$ and φ is an M-realisation we have $n_0 \notin \mathrm{strnames}\, G$. Moreover, if n_0 did occur in E then becuase n_0 occurs in S and (12 9) we would have that n_0 occurred in $\varphi\, E$ — but that is absurd since $n_0 \notin \mathrm{strnames}(\varphi\, W)$. Therefore n_0 does not occur in E, so it must occur in U'.

Observation 12.9 Theorem 12.6 implies theorem 12.1 and theorem 12.2 as follows. Assume that B is robust and that $\mathrm{strnames}\, B \subseteq U$. Let $W = \mathrm{strs}\, B$. Then W covers B and $M \cup \mathrm{strnames}\, W \subseteq U$. Assume that $(\varphi, S, U') = SC(B, \mathit{sigexp}, U)$ succeeds. Then, by theorem 12.6, φ in an M-realisation and $\varphi\, B \vdash \mathit{sigexp} \Rightarrow (\emptyset)\, S$

as required in theorem 12.1. Moreover, the conclusion of theorem 12.2 follows from observation 12.8 and the fact that $M \cap U' = \emptyset$.

Observation 12.10 To ease the proof of (12.6), we shall now show that if we can prove (12.4), (12.7), (12.8), (12.10) and that W' is coherent then it follows that W' covers $\varphi\, B$.

Firstly, that $\varphi\, B$ is robust is seen as follows. We have

$$\varphi\, B = \varphi(M, F, G, E) = (M, F, G, \varphi\, E).$$

Since B is well-formed, $\varphi\, B$ is well-formed. Moreover, $\text{strs}(\varphi\, B)$ is a subset of W', which is assumed coherent, so $\text{strs}(\varphi\, B)$ is coherent. Thus $\varphi\, B$ is admissible. Moreover, $\text{strnames}\, F \subseteq M$ and $\text{strnames}\, G \subseteq M$ since B is robust. We must also prove that $M\text{–strs}(\varphi\, E)$ is substructure closed. For this it will suffice to prove that $M\text{–strs}(\varphi\, B) = M\text{–strs}\, B$. But $M\text{–strs}\, B \subseteq M\text{–strs}(\varphi\, B)$ since φ is an M-realisation. Conversely, if $S = (m, E) \in M\text{–strs}(\varphi\, B)$ we have $S \in \text{strs}\{F, G, E\}$ or $m \in \text{Inv}\,\varphi$. In the latter case we have $m \in \text{strnames}\{G, E\}$, by (12.8), and since $S \in M\text{–strs}\, W' = M\text{–strs}\, W$ and W is coherent we have $S \in \text{strs}\{G, E\}$. So in either case $S \in M\text{–strs}\{F, G, E\}$, showing $M\text{–strs}(\varphi\, B) \subseteq M\text{–strs}(B)$.

Secondly, that W' is simple is seen as follows. It is finite since W is finite; it is coherent by assumption; it is substructure closed by definition; and $M\text{–strs}\, W'$ is substructure closed since $M\text{–strs}\, W' = M\text{–strs}\, W$ and $M\text{–strs}\, W$ is substructure closed.

Thirdly, $\text{strs}(\varphi\, B) \subseteq W'$ by the definition of W'. Thus W' covers $\varphi\, B$.

Observation 12.11 Under the same assumptions as in observation 12.10 we do not have to worry about the global constraints on (12.5). As we saw there, $\varphi\, B$ is robust, $(\emptyset)\, S$ is well-formed, $\text{strs}\{\varphi\, B, (\emptyset)\, S\}$ is coherent, since W' is assumed coherent, and $M\text{–strs}\, S$ is substructure closed by (12.7).

Proof. [of theorem 12.6] By induction on the number, r of recursive calls of SC. The proof uses the soundness theorem for *Unify* (theorem 11.5), its corollary (corollary 11.6), and the realisation corollary, corollary 10.6.

Base Case, $r = 0$ Either *spec* is the empty specification or *sigexp* = *sigid* or

$$spec = \textbf{sharing}\ longstrid_1 = longstrid_2.$$

The first of these is trivial, the second easy (use the instantiation rule once) so let us go straight to the third one, sharing.

By the definition of SC we must have $longstrid_1 = strid_1.longstrid'_1$ and also $strid_1 \in \mathrm{Dom}\,E$. Also, $(R_1, U_1) = Fresh(longstrid'_1, U)$. Now let

$$W_1 = W \cup \mathrm{strs}\,R_1.$$

It is easy to check that W_1 is simple with respect to M and that $R_1 \in W_1$ and $E(strid_1) \in W_1$.

Thus by the soundness theorem for *Unify* (theorem 11.5) and its corollary (corollary 11.6) we have

$\varphi_1\,R_1 = \varphi_1\,E\,strid_1$
$\mathrm{Inv}\,\varphi_1 \subseteq \mathrm{strnames}\{R_1, E\,strid_1\}$
φ_1 does not change the name of any structure whose name
is in $\mathrm{strnames}(\varphi_1\,R_1)$
φ_1 is an M-realisation
$\varphi_1\,W_1$ is simple with respect to M
M–$\mathrm{strs}(\varphi_1\,W_1) = M$–$\mathrm{strs}\,W_1$.

Next, let

$$W_2 = \varphi_1\,W_1 \cup \mathrm{strs}\,R_2.$$

Given that $\varphi_1\,W_1$ is simple with respect to M it is easy to check that W_2 is simple with respect to M and that R_2 and $\varphi_1\,E\,strid_2$ are members of W_2. Thus using theorem 11.5 and its corollary again, we get

$\varphi_2\,R_2 = \varphi_2(\varphi_1\,E\,strid_2)$
$\mathrm{Inv}\,\varphi_2 \subseteq \mathrm{strnames}\{R_2, \varphi_1\,E\,strid_2\}$
φ_2 does not change the name of any structure whose name
is in $\mathrm{strnames}(\varphi_2\,R_2)$
φ_2 is an M-realisation
$\varphi_2\,W_2$ is simple with respect to M
M–$\mathrm{strs}(\varphi_2\,W_2) = M$–$\mathrm{strs}\,W_2$.

Hence, letting

$$W_3 = \varphi_2 W_2,$$

W_3 is simple with respect to M and given the fact that φ_1 and φ_2 are unifiers it is easy to see that $(\varphi_2 \varphi_1 E)longstrid_1$ and $(\varphi_2 \varphi_1 E)longstrid_2$ both are members of W_3.

Therefore, applying the theorem and corollary a final time,

$\varphi_3((\varphi_2 \varphi_1 E)longstrid_1) = \varphi_3((\varphi_2 \varphi_1 E)longstrid_2)$
$\mathrm{Inv}\, \varphi_3 \subseteq \mathrm{strnames}\{(\varphi_2 \varphi_1 E)longstrid_1, (\varphi_2 \varphi_1 E)longstrid_2\}$
φ_3 does not change the name of any structure whose name is in $\mathrm{strnames}(\varphi_3((\varphi_2 \varphi_1 E)longstrid_1))$
φ_3 is an M-realisation
$\varphi_3 W_3$ is simple with respect to M
$M\text{–strs}(\varphi_3 W_3) = M\text{–strs}\, W_3$.

Now the result of SC is

$$(\varphi, E', U') = (\varphi_3 \circ \varphi_2 \circ \varphi_1, \{\}, U_1 \cup U_2).$$

Clearly φ is an M-realisation and we saw that $(\varphi E)longstrid_1 = (\varphi E)longstrid_2$ so we have (12.4) and (12.5).

Proof of (12.6). Let $W' = \mathrm{strs}\{\varphi W, E'\} = \mathrm{strs}(\varphi W)$. Then

$$\begin{aligned} W' &= \mathrm{strs}(\varphi_3 \varphi_2 \varphi_1 W) \\ &\subseteq \mathrm{strs}(\varphi_3 \varphi_2 \varphi_1 W_1) \\ &\subseteq \mathrm{strs}(\varphi_3 \varphi_2 W_2) \\ &= \varphi_3 W_3 \end{aligned}$$

and $\varphi_3 W_3$ is simple with respect to M so W' is coherent. Thus by observation 12.10 we have (12.6).

Proof of (12.7). We have

$$\begin{aligned} M\text{–strs}\, W' &\subseteq M\text{–strs}(\varphi_3 W_3) = M\text{–strs}\, W_3 = M\text{–strs}(\varphi_2 W_2) \\ &= M\text{–strs}\, W_2 = M\text{–strs}\{\varphi_1 W_1, R_2\} = M\text{–strs}(\varphi_1 W_1) \\ &= M\text{–strs}\, W_1 = M\text{–strs}\{W, R_1\} = M\text{–strs}\, W. \end{aligned}$$

The converse is obvious.

As to (12.8) we have

$$\begin{aligned} \mathrm{Inv}\, \varphi \cup \mathrm{strnames}\, E' &= \mathrm{Inv}\, \varphi \\ &\subseteq \mathrm{Inv}\, \varphi_3 \cup \mathrm{Inv}\, \varphi_2 \cup \mathrm{Inv}\, \varphi_1 \\ &= \mathrm{Inv}\, \varphi_2 \cup \mathrm{Inv}\, \varphi_1 \cup \mathrm{strnames}\, E \end{aligned}$$

$$\begin{aligned} &= U_2 \cup \mathrm{Inv}\,\varphi_1 \cup \mathrm{strnames}\,E \\ &= U_2 \cup U_1 \cup \mathrm{strnames}\,E \\ &\subseteq U' \cup \mathrm{strnames}\{E, G\}. \end{aligned}$$

Proof of (12.9). Since φ_1 does not change the name of any structure whose name is in strnames($\varphi_1\, R_1$) and since $\mathrm{Inv}\,\varphi_1 \subseteq \mathrm{strnames}\{R_1, E\; \mathit{strid}_{\,1}\}$, we have that φ_1 does not change the name of any structure whose name is in $\varphi_1\, E$.

φ_1 does not change the name of any structure whose name is in R_2 and since $\mathrm{Inv}\,\varphi_2 \subseteq \mathrm{strnames}\{R_2, \varphi_1\, E\; \mathit{strid}_{\,2}\}$, we have that φ_1 does not change the name of any structure whose name occurs is $\varphi_2\,\varphi_1\, E$. Since $\mathrm{Inv}\,\varphi_3 \subseteq \mathrm{strnames}\,\varphi_2\,\varphi_1\, E$, we have that φ_1 does not change the name of any structure whose name is in $\varphi_3\,\varphi_2\,\varphi_1\, E$ i.e., in $\varphi\, E$.

Next, since φ_2 does not change the name of any structure whose name is in strnames($\varphi_2\, R_2$) and $\mathrm{Inv}\,\varphi_2 \subseteq \mathrm{strnames}\{R_2, \varphi_1\, E\; \mathit{strid}_{\,2}\}$ we have that φ_2 does not change the name of any structure whose name is in $\varphi_2\,\varphi_1\, E$. Since $\mathrm{Inv}\,\varphi_3 \subseteq \mathrm{strnames}\{\varphi_2\,\varphi_1\, E\}$ we have that φ_2 does not change the name of any structure whose name occurs in $\varphi_3\,\varphi_2\,\varphi_1\, E$ i.e., $\varphi\, E$.

Finally, since φ_3 does not change the name of any structure whose name occurs in $\varphi_3((\varphi_2\,\varphi_1\, E)\mathit{longstrid}_{\,1})$ and

$$\mathrm{Inv}\,\varphi_3 \subseteq \mathrm{strnames}\{(\varphi_2\,\varphi_1\, E)\mathit{longstrid}_{\,1}, (\varphi_2\,\varphi_1\, E)\mathit{longstrid}_{\,2}\},$$

we have that φ_3 does not change the name of any structure the name of which is in $\varphi\, E$.

Since nither φ_1 nor φ_2 nor φ_3 change the name of any structure whose name is in $\varphi\, E$ we have that φ has that property too. This proves (12.9).

Finally $U \cap U' = U \cap (U_1 \cup U_2) = \{\}$, by the properties of ***Fresh***, showing (12.10).

Inductive Step, $r > 0$ There are the following cases

spec = **structure** *strid* : *sigexp* Use rule 8.9 once.

$\boxed{spec = spec_1\ spec_2}$ The call

$$(\varphi_1, E_1, U_1) = SC(B, spec_1, U)$$

succeeded so by induction, letting

$$W_1 = \text{strs}\{\varphi_1(W), E_1\} \tag{12.12}$$

we have

$$\varphi_1 \text{ is an } M\text{-realisation} \tag{12.13}$$

$$\varphi_1 B \vdash spec_1 \Rightarrow E_1 \tag{12.14}$$

$$W_1 \text{ covers } \varphi_1 B \tag{12.15}$$

$$M\text{-strs}\, W_1 = M\text{-strs}\, W \tag{12.16}$$

$$\text{Inv}\,\varphi_1 \cup \text{strnames}\, E_1 \subseteq \text{strnames}\{G, E\} \cup U_1 \tag{12.17}$$

$$\varphi_1 \text{ does not change the name of any structure whose name is in } \varphi_1 E \text{ or in } E_1 \tag{12.18}$$

$$U \cap U_1 = \emptyset. \tag{12.19}$$

We now wish to use the induction hypothesis once again with W_1 for W and $\varphi_1 B \pm E_1$ for B and $U \cup U_1$ for U. Given that W_1 covers $\varphi_1 B$ and contains all structures in E_1 we have that W_1 covers $\varphi_1 B \pm E_1$, if just $\varphi_1 B \pm E_1$ is robust. But $\varphi_1 B \pm E_1$ is admissible by (12.14), and $\text{strnames}\, F \subseteq M$ and $\text{strnames}\, G \subseteq M$ since φB is robust, and $M\text{-strs}(E \pm E_1)$ is substructure closed by (12.14). Thus $\varphi_1 B \pm E_1$ is robust, so W_1 covers $\varphi_1 B \pm E$.

Before we can apply the induction we must also check that

$$M \text{ of } (\varphi_1 B \pm E_1) \cup \text{strnames}\, W_1 \subseteq U \cup U_1,$$

which is seen as follows: $M \text{ of } (\varphi_1 B \pm E_1) \cup \text{strnames}\, W_1 = M \cup \text{strnames}\, W_1 \subseteq M \cup \text{strnames}\, W \cup U_1 \subseteq U \cup U_1$, using (12.17).

Thus we can apply the induction hypothesis to get

$$\varphi_2 \text{ is an } M\text{-realisation} \tag{12.20}$$

$$\varphi_2(\varphi_1 B \pm E_1) \vdash spec_2 \Rightarrow E_2 \tag{12.21}$$

$$W_2 \text{ covers } \varphi_2(\varphi_1 B \pm E_1) \tag{12.22}$$

$$M\text{-strs}\, W_2 = M\text{-strs}\, W_1 \tag{12.23}$$

$$\text{Inv}\,\varphi_2 \cup \text{strnames}\, E_2 \subseteq \text{strnames}\{G, \varphi_1 E \pm E_1\} \cup U_2 \tag{12.24}$$

φ_2 does not change the name of any structure whose name is in $\varphi_2(\varphi_1 E \pm E_1)$ or in E_2 (12.25)

$$U_2 \cap (U \cup U_1) = \emptyset, \tag{12.26}$$

where

$$W_2 = \text{strs}\{\varphi_2(W_1), E_2\}.$$

Now SC succeeds with

$$(\varphi, E', U') = (\varphi_2 \circ \varphi_1, \varphi_2 E_1 \pm E_2, U_1 \cup U_2).$$

Letting $W' = \text{strs}\{\varphi(W), E'\}$ we have

$$\begin{aligned} W' &= \text{strs}\{\varphi_2 \varphi_1 W, \varphi_2 E_1 \pm E_2\} \\ &\subseteq \text{strs}\{\varphi_2 \varphi_1 W, \varphi_2 E_1, E_2\} \\ &= \text{strs}\{\varphi_2(\text{strs}\{\varphi_1 W, E_1\}), E_2\} \\ &= \text{strs}\{\varphi_2(W_1), E_2\} \\ &= W_2 \end{aligned}$$

i.e.,

$$W' \subseteq W_2. \tag{12.27}$$

Since W_2 is coherent also W' is coherent and that will do for (12.6), c f observation 12.10. Moreover, φ is clearly an M-realisation i.e., we have (12.4). To see (12.7), note that M–strs $W' \subseteq M$–strs W by (12.27), (12.23) and (12.16). Conversely,

$$M\text{–strs}\, W \subseteq M\text{–strs}\{\varphi W, E'\} = M\text{–strs}\, W'$$

since φ is an M-realisation and M–strs W is substructure closed.

From (12.14) we may deduce

$$\varphi_2(\varphi_1 B) \vdash spec_1 \Rightarrow \varphi_2 E_1 \tag{12.28}$$

if (and now I refer to corollary 10.6)

$$\varphi_2(\varphi_1 B) \text{ is robust} \tag{12.29}$$

$$M\text{–strs}(\varphi_2 E_1) \text{ is substructure closed} \tag{12.30}$$

$$\{\varphi_2(\varphi_1 B), \varphi_2 E_1\} \text{ is coherent.} \tag{12.31}$$

As to (12.31), $\text{strs}\{\varphi_2(\varphi_1 B), \varphi_2 E_1\} = \text{strs}\{F, G, \varphi_2 \varphi_1 E, \varphi_2 E_1\} \subseteq \text{strs}(\varphi_2 W_1) \subseteq W_2$, which is coherent. Hence (12.31). As for (12.30) we have M–strs$(\varphi_2 E_1) \subseteq$

M–strs$(\varphi_2 W_1) \subseteq M$–strs$\{\varphi_2 W_1, E_2\} = M$–strs $W_2 = M$–strs W, which is substructure closed. To show (12.29) given that

$$\varphi_2 \varphi_1 B = (M, F, G, \varphi_2 \varphi_1 E)$$

and the coherence of $\varphi_2 \varphi_1 B$, it will suffice to prove that M–strs$(\varphi_2 \varphi_1 E)$ is substructure closed. But M–strs$(\varphi_2 \varphi_1 E) \subseteq M$–strs$(\varphi_2 W_1) \subseteq M$–strs W as in the proof of (12.30).

Thus we have (12.28). But this, combined with (12.21) gives the desired

$$\varphi B \vdash spec_1 \; spec_2 \Rightarrow \varphi_2 E_1 \pm E_2 \,.$$

As for (12.8) we have

$$\begin{array}{rcll} \mathrm{Inv}(\varphi) \cup \mathrm{strnames}\, E' & \subseteq & \mathrm{Inv}\,\varphi_1 \cup \mathrm{Inv}\,\varphi_2 \cup \mathrm{strnames}\{E_1, E_2\} & \\ & \subseteq & \mathrm{strnames}\{G, E, E_1\} \cup U_2 \cup \mathrm{Inv}\,\varphi_1 & \text{by (12.24)} \\ & \subseteq & \mathrm{strnames}\{G, E\} \cup U_1 \cup U_2 & \text{by (12.17)} \\ & \subseteq & \mathrm{strnames}\{G, E\} \cup U. & \end{array}$$

Proof of (12.9). By (12.24) and the fact that φ_1 glides on every structure whose name is in U_2 we get from (12.18) that φ_1 does not change the name of any structure whose name is is in $\varphi_2 \varphi_1 E$ or in $\varphi_2 E_1$ or in E_2.

Thus φ_1 does not change the name of any structure whose name is in φE or in $E' = \varphi_2 E_1 \pm E_2$. From (12.25) and (12.24) we get that φ_2 does not change the name of any structure whose name is in $\varphi_2 \varphi_1 E$. Thus by (12.25), φ_2 does not change the name of any structure whose name is in φE or in $E' = \varphi_2 E_1 \pm E_2$. Thus $\varphi = \varphi_2 \circ \varphi_1$ does not change the name of any structure whose name occurs in φE or in E'.

This leaves the trivial (12.10).

sigexp = **sig** *spec* **end**	Easy inductive step using one application of rule 8.12.

∎

12.3 Completeness of SC

We shall prove the following strengthened version of the completeness result.

Theorem 12.12 **(Completeness of** *SC***)** *Assume W covers $B = M, F, G, E$ and $M \cup \text{strnames}\, W \subseteq U$. For all M-realisations, ψ_0, and signatures $\Sigma_0 = (N_0)\, S_0$, if*

$$\psi_0\, B \vdash sigexp \Rightarrow \Sigma_0 \tag{12.32}$$

then $(\varphi, S, U') = SC(B, sigexp, U)$ succeeds and there exists an M-realisation, ψ_1, and a signature, Σ_1, such that

$$\psi_0 \downarrow W = (\psi_1 \circ \varphi) \downarrow W, \tag{12.33}$$

$$\Sigma \xrightarrow{\psi_1} \Sigma_1 \text{ and } \Sigma_1 \geq \Sigma_0, \tag{12.34}$$

where $\Sigma = \text{Clos}_{\varphi W}\, S$. Moreover, if SC does not succeed, it stops with fail.

Similarly for specifications.

From observation 12.8 we know that $\text{strs}\, \Sigma \subseteq \text{strs}(\varphi\, W)$. Therefore, if there exists a φ_1 as described above then there exists one that in addition satisfies $\text{Supp}\, \psi_1 \subseteq \text{strs}(\varphi\, W)$.

Moreover, to prove (12.34) without having to mention renamings explicitly, notice the following. Let $\Sigma = (N)S$ and $\Sigma_0 = (N_0)\, S_0$ be well-formed signatures, let φ be a realisation such that $N_0 \cap \text{Reg}(\varphi \downarrow \text{strs}\, \Sigma) = \emptyset$ and $\varphi(S) = S_0$. Then there exists a signature Σ_1 such that

$$\Sigma \xrightarrow{\varphi} \Sigma_1 \text{ and } \Sigma_1 \geq \Sigma_0. \tag{12.35}$$

More generally, let $\Sigma = (N)S$ and $\Sigma_0 = (N_0)\, S_0$ be well-formed signatures and φ be a realisation. Then there exists a set $W \subseteq \text{Str}$ with $\text{strs}\, \Sigma \subseteq W$ and $\text{boundstrs}(\Sigma) \cap W = \emptyset$ and $N_0 \cap \text{Reg}(\varphi \downarrow \text{strs}\, \Sigma) = \emptyset$ and $\varphi\, S = S_0$ if and only if there exists a Σ_1 such that (12.35).

Thus the conclusion of theorem 12.12 is equivalent to the existence of an M-realisation, ψ_1, with

$$\psi_0 \downarrow W = (\psi_1 \circ \varphi) \downarrow W,$$

$$N_0 \cap \text{Reg}(\psi_1 \downarrow \text{strs}\, \Sigma) = \emptyset \text{ and } \psi_1\, S = S_0.$$

Proof. [of theorem 12.12] The proof uses the soundness and completeness results about *Unify* (theorem 11.5 and theorem 11.7), the soundness of SC (theorem 12.6) and observation 12.8.

The two parts of the statement (regarding success and failure, respectively) are proved separately. The first part is proved by induction on the depth of inference of $\psi_0 B \vdash sigexp \Rightarrow \Sigma_0$ and $\psi_0 B \vdash spec \Rightarrow E_0$. There is one case for each inference rule.

The case for empty specifications is trivial — take $\psi_1 = \psi_0$. In the case for structure specifications one takes the ψ_1 that is supplied by induction. Let us therefore go straight to the first non-trivial case.

Sequential Specification, rule 8.10 Consider a proof ending

$$\frac{\psi_0 B \vdash spec_1 \Rightarrow E_1' \quad \psi_0 B \pm E_1' \vdash spec_2 \Rightarrow E_2'}{\psi_0 \vdash spec_1\ spec_2 \Rightarrow E_1' \pm E_2'}. \tag{12.36}$$

By induction $(\varphi_1, E_1, U_1) = SC(B, spec_1, U)$ succeeds and there exists an M-realisation ψ_1 with

$$\psi_0 \downarrow W = (\psi_1 \circ \varphi_1) \downarrow W \tag{12.37}$$

and

$$\psi_1 E_1 = E_1'. \tag{12.38}$$

Now we wish to use induction again, this time on the second premise of (12.36). To see that this basis is of the required kind, we compute

$$\begin{aligned} \psi_0 B \pm E_1' &= \psi_1(\varphi_1 B) \pm \psi_1 E_1 \quad \text{by (12.37) and (12.38)} \\ &= \psi_1(\varphi_1 B \pm E_1) \\ &= \psi_1 B_1, \end{aligned}$$

where

$$B_1 = \varphi_1 B \pm E_1.$$

Thus, by (12.36) we have $\psi_1 B_1 \vdash spec_2 \Rightarrow E_2'$ in strictly fewer steps.

Let $W_1 = \text{strs}\{\varphi_1(W), E_1\}$. As we saw in the proof of theorem 12.6, W_1 covers B_1 and $M \cup \text{strnames}\, W_1 \subseteq U \cup U_1$. Thus by induction $(\varphi_2, E_2, U_2) = SC(B_1, spec_2, U \cup U_1)$ succeeds and there exists an M-realisation ψ_2 with

$$\psi_1 \downarrow W_1 = (\psi_2 \circ \varphi_2) \downarrow W_1 \tag{12.39}$$

and

$$\psi_2 E_2 = E_2'. \tag{12.40}$$

Thus SC succeeds with $(\varphi, E', U') = (\varphi_2 \circ \varphi_1, \varphi_2 E_1 \pm E_2, U_1 \cup U_2)$. Now $\psi_0 \downarrow W = (\psi_2 \circ \varphi) \downarrow W$ since for each $w \in W$,

$$\begin{aligned} \psi_0 w &= \psi_1(\varphi_1 w) && \text{by (12.37)} \\ &= \psi_2(\varphi_2(\varphi_1 w)) && \text{by (12.39)} \\ &= \psi_2(\varphi w). \end{aligned}$$

as desired. Moreover,

$$\begin{aligned} \psi_2 E' &= \psi_2(\varphi_2 E_1 \pm E_2) \\ &= \psi_2 \varphi_2 E_1 \pm E_2' && \text{by (12.40)} \\ &= \psi_1 E_1 \pm E_2' && \text{by (12.39)} \\ &= E_1' \pm E_2', \end{aligned}$$

as desired.

Sharing, rule 8.11 Consider a proof of the form

$$\frac{n \text{ of } ((\psi_0 B) longstrid_1) = n \text{ of } ((\psi_0 B) longstrid_2)}{\psi_0 B \vdash \textbf{sharing } longstrid_1 = longstrid_2 \Rightarrow \{\}}. \tag{12.41}$$

Write $longstrid_i$ as $strid_i.longstrid'_i$, for $i = 1, 2$. By (12.41) we have $strid_1 \in \operatorname{Dom} E$ so SC does not fail at this point. As in SC, let

$$(R_1, U_1) = Fresh(longstrid'_1, U).$$

We now wish to use the completeness result about *Unify* (theorem 11.7) a first time. To this end let

$$W_1 = W \cup \operatorname{strs} R_1.$$

W_1 is simple with respect to M and it contains R_1 and $E(strid_1)$. Moreover, $(\psi_0(E\, strid_1))longstrid'_1$ exists by (12.41). Since $\operatorname{strs} R_1 \cap W = \emptyset$ there exists an M-realisation, ψ_1, such that $\psi \downarrow W = \psi_1 \downarrow W$ and $\psi_1(R_1) = \psi_1(E\, strid_1)$. Thus by theorem 11.7, $\varphi_1 = Unify(M, R_1, E\, strid_1)$ succeeds and there exists an M-realisation, ψ_2, such that

$$\psi_1 \downarrow W_1 = (\psi_2 \circ \varphi_1) \downarrow W_1. \tag{12.42}$$

Next, $strid_2 \in \operatorname{Dom} E$ by (12.41) so SC does not fail that test. Let $(R_2, U_2) = Fresh(longstrid_2, U \cup U_1)$. We now wish to apply the completeness result for *Unify* a second time.

To this end, let $W_2 = \varphi_1 W_1 \cup \operatorname{strs} R_2$. It follows from theorem 11.5 that W_2 is simple with respect to M. Now R_2 and $\varphi_1 E\, strid_2$ are in W_2 and they have an M-unifier. More precisely,

$$\begin{aligned}
(\psi_0 E)\mathit{longstrid}_2 &= ((\psi_0 E)\mathit{strid}_2)\mathit{longstrid}'_2 \\
&= (\psi_0(E\,\mathit{strid}_2))\mathit{longstrid}'_2 \\
&= (\psi_1 E\,\mathit{strid}_2)\mathit{longstrid}'_2 && \text{as } E\,\mathit{strid}_2 \in W \\
&= (\psi_2(\varphi_1 E\,\mathit{strid}_2))\mathit{longstrid}'_2 && \text{by (12.42)}
\end{aligned}$$

Thus $(\psi_2(\varphi_1 E\,\mathit{strid}_2))\mathit{longstrid}'_2$ exists and since $\mathrm{strs}(R_2) \cap \mathrm{strs}(\varphi_1 W) = \emptyset$ there exists an M-realisation, ψ_3 with $\psi_3 \downarrow \mathrm{strs}(\varphi_1 W_1) = \psi_2 \downarrow \mathrm{strs}(\varphi_1 W_1)$ which unifies $\varphi_1 E\,\mathit{strid}_2$ and R_2. Thus by theorem 11.7,

$$\varphi_2 = \mathit{Unify}(M, R_2, \varphi_1 E\,\mathit{strid}_2)$$

succeeds and there exists an M-realisation, ψ_4, such that

$$\psi_3 \downarrow W_2 = (\psi_4 \circ \varphi_2) \downarrow W_2. \tag{12.43}$$

We now want to apply the completeness result for *Unify* a third time. To this end, let

$$W_3 = \varphi_2 W_2.$$

Theorem 11.5 ensures that W_3 is simple with respect to M. Both $(\varphi_2 \varphi_1 E)\mathit{longstrid}_1$ and $(\varphi_2 \varphi_1 E)\mathit{longstrid}_2$ exist and are in W_3. Moreover, ψ_4 unifies them since

$$\begin{aligned}
\psi_4((\varphi_2 \varphi_1 E)\mathit{longstrid}_\iota) &= (\psi_4 \varphi_2 \varphi_1 E)\mathit{longstrid}_\iota \\
&= (\psi_3 \varphi_1 E)\mathit{longstrid}_\iota && \text{by (12.43)} \\
&= (\psi_2 \varphi_1 E)\mathit{longstrid}_\iota \\
&= (\psi_1 E)\mathit{longstrid}_\iota && \text{by (12.42)} \\
&= (\psi_0 E)\mathit{longstrid}_\iota,
\end{aligned}$$

so by the premise of (12.41) and the coherence of $\psi_0 B$, ψ_4 is an M-unifier for $(\varphi_2 \varphi_1 E)\mathit{longstrid}_1$ and $(\varphi_2 \varphi_1 E)\mathit{longstrid}_2$. Thus by theorem 11.7 the last call succeeds and there exists an M-realisation, ψ_5, such that

$$\psi_4 \downarrow W_3 = (\psi_5 \circ \varphi_3) \downarrow W_3. \tag{12.44}$$

Thus *SC* returns $(\varphi, E', U') = (\varphi_3 \circ \varphi_2 \circ \varphi_1, \{\}, U_1 \cup U_2)$. We have $\psi_0 \downarrow W = (\psi_5 \circ \varphi) \downarrow W$ since for all $w \in W$,

$$\begin{aligned}
\psi_0(w) &= \psi_1(w) \\
&= \psi_2 \varphi_1 w && \text{by (12.42)} \\
&= \psi_3 \varphi_1 w \\
&= \psi_4 \varphi_2 \varphi_1 w && \text{by (12.43)} \\
&= \psi_5 \varphi_3 \varphi_2 \varphi_1 w && \text{by (12.44)}
\end{aligned}$$

$= \quad \psi_5 \varphi w.$

Finally, $\psi_5 E' = \psi_5\{\} = \{\} = E_0$ as desired.

Basic Signature Expression, rule 8.12 Consider a proof ending

$$\frac{\psi_0 B \vdash spec \Rightarrow E_0}{\psi_0 B \vdash \mathbf{sig}\ spec\ \mathbf{end} \Rightarrow (\emptyset)(n, E_0)}. \tag{12.45}$$

By induction $(\varphi_1, E_1, U_1) = SC(B, spec, U)$ succeeds and there exists an M-realisation, ψ_1, such that

$$\psi_0 \downarrow W = (\psi_1 \circ \varphi_1) \downarrow W \tag{12.46}$$

and

$$\psi_1 E_1 = E_0. \tag{12.47}$$

Thus SC succeeds with $(\varphi, S, U') = (\varphi_1, (n_1, E_1), U_1 \cup \{n_1\})$, where $n_1 \notin U \cup U_1$. The desired conclusions follow from (12.46) and (12.47) provided n_1 really is bound in $\mathrm{Clos}_{\varphi W}(n_1, E_1)$. But that follows from theorem 12.6 which ensures that $\mathrm{strnames}(\varphi_1 W) \subseteq U \cup U_1$.

Signature Identifier, rule 8.13 Consider a proof of the form

$$\frac{(\psi_0 B) sigid = \Sigma_0}{\psi_0 B \vdash sigid \Rightarrow \Sigma_0}.$$

Since ψ_0 is an M-realisation and B is robust we have $G(sigid) = \Sigma_0$. Thus SC returns

$$(\varphi, S, U') = (ID, \{n_i \mapsto n_i'\}\, S_0, \{n_1', \ldots, n_k'\}),$$

where $\Sigma_0 = (N_0)\, S_0$, $N_0 = \{n_1, \ldots, n_k\}$ and $\{n_1', \ldots, n_k'\} \cap U = \emptyset$. Hence

$$\Sigma = \mathrm{Clos}_{\varphi W}\, S = (\{n_i'\})(\{n_i \mapsto n_i'\}\, S_0).$$

Let $\psi_1 = \psi_0$. Then (12.33) is obvious. As for (12.34) we have $\Sigma \xrightarrow{\psi_1} \Sigma$ since ψ_1 is an M-realisation and $\mathrm{strnames}\, \Sigma \subseteq M$. Moreover, $\Sigma \geq \Sigma_0$, in fact $\Sigma \equiv \Sigma_0$, since $\{n_1', \ldots, n_k'\} \cap U = \emptyset$ ensures that no n_i' is free in Σ.

The Generalization Rule, rule 8.14 Consider a proof ending

$$\frac{\psi_0 B \vdash sigexp \Rightarrow (N_0')\,S_0 \quad n_0 \notin \mathrm{strnames}(\psi_0 B)}{\psi_0 B \vdash sigexp \Rightarrow (N_0' \cup \{n_0\})\,S_0}.$$

By induction $(\varphi, S, U') = SC(B, sigexp, U)$ succeeds and there exists an M-realisation, ψ_1, such that

$$\psi_0 \downarrow W = (\psi_1 \circ \varphi) \downarrow W \tag{12.48}$$

as desired and also, letting $\Sigma = \mathrm{Clos}_{\varphi W}\, S$,

$$N_0' \cap \mathrm{Reg}(\psi_1 \downarrow \mathrm{strs}\,\Sigma) = \emptyset \text{ and } \psi_1 S = S_0. \tag{12.49}$$

To strengthen (12.49) to the desired

$$N_0 \cap \mathrm{Reg}(\psi_1 \downarrow \mathrm{strs}\,\Sigma) = \emptyset \text{ and } \psi_1 S = S_0,$$

we must show that $n_0 \notin \mathrm{Reg}(\psi_1 \downarrow \mathrm{strs}\,\Sigma)$. This is seen as follows. By observation 12.8 we have $\mathrm{strs}\,\Sigma \subseteq \mathrm{strs}(\varphi\, B)$. Thus

$$\begin{aligned} \mathrm{Rng}(\psi_1 \downarrow \mathrm{strs}\,\Sigma) &\subseteq \mathrm{strs}(\psi_1(\varphi\, B)) \\ &= \mathrm{strs}(\psi_0 B) \qquad \text{by (12.48)} \end{aligned}$$

so $\mathrm{Reg}(\psi_1 \downarrow \mathrm{strs}\,\Sigma) \subseteq \mathrm{strnames}(\psi_0 B)$. Since $n_0 \notin \mathrm{strnames}(\psi_0 B)$ we have $n_0 \notin \mathrm{Reg}(\psi \downarrow \mathrm{strs}\,\Sigma)$.

The Instantiation Rule, rule 8.15 Use induction once, take the ψ_1 provided by induction and use the transitivity of $\geq$.

This concludes the first half of the proof (regarding the success of *SC*). The second part is shown by structural induction on *sigexp* and *spec*. We show the two non–trivial cases:

Sharing, *spec* = **sharing** *longstrid* $_1$ = *longstrid* $_2$ So we have assumed that W covers B and $M \cup \mathrm{strnames}\, W \subseteq U$ and $SC(B, spec, U)$ does not succeed and we want to show that it terminates with **fail**. Looking at SC, this is clear if $strid_1 \notin \mathrm{Dom}\, E$, so assume $strid_1 \in \mathrm{Dom}\, E$. Next, $(R_1, U_1) = Fresh(longstrid'_1, U)$ succeeds. Let $W_1 = W \cup \mathrm{strs}(R_1)$. As was seen above, W_1 is simple with respect to M and contains R_1 and $E\, strid_1$. Then by theorem 11.7, the first call of unify either succeeds or stops with **fail**. In the latter case we are done, so assume the former. The argument is repeated for the second call using $W_2 = \varphi_1(W_1) \cup \mathrm{strs}(R_2)$ and finally, if the second call succeeded, we have that $W_3 = \varphi_2 W_2$ is simple with respect to M and contains the arguments of the final unification which, since it cannot succeed, must stop with **fail** by theorem 11.7.

Sequential Specification, *spec* = *spec*$_1$ *spec*$_2$ Assume $SC(B, spec, U)$ does not succeed. Then, by induction, if the first call of SC does not succeed it stops with **fail** (and we are done); so assume it succeeds. Then, as we saw above, $W_1 = \mathrm{strs}(\varphi_1 W) \cup \mathrm{strs}(E_1)$ covers $B_1 = \varphi_1 B \pm E_1$ and $M \cup \mathrm{strnames}\, W_1 \subseteq U \cup U_1$ and the second call of SC cannot have succeeded so by induction it stopped with **fail**. ■

Having proved theorem 12.12, we see that it really does imply theorem 12.3: Assume B robust and $\mathrm{strnames}\, B \subseteq U$ and ψ is an M-realisation and Σ a signature such that $\psi B \vdash sigexp \Rightarrow \Sigma$. Let $W = \mathrm{strs}\, B$. Then W covers B and $M \cup \mathrm{strnames}\, W \subseteq U$, so by theorem 12.12 $(\varphi, S, U') = SC(B, sigexp, U)$ succeeds and there exists an M-realisation, ψ', and a signature Σ_1 such that $\psi \downarrow W = (\psi' \circ \varphi) \downarrow W$, $\mathrm{Clos}_{\varphi W} S \xrightarrow{\psi'} \Sigma_1$, and $\Sigma_1 \geq \Sigma$. Thus $(\psi' \circ \varphi) E = \psi E$. Moreover, (and this is the only interesting point in this argument) $\mathrm{Clos}_{\varphi B} S = \mathrm{Clos}_{\varphi W} S$ by observation 12.8. Thus, if $\mathrm{Clos}_{\varphi B} S \xrightarrow{\psi'} \Sigma'$, then $\mathrm{Clos}_{\varphi W} S \xrightarrow{\psi'} \Sigma'$ and since also $\mathrm{Clos}_{\varphi W} S \xrightarrow{\psi'} \Sigma_1$ we must have $\Sigma' \underset{\alpha}{=} \Sigma_1$. Thus $\Sigma' \equiv \Sigma_1$ by lemma 9.17 so $\Sigma' \geq \Sigma_1 \geq \Sigma$ as desired.

Chapter 13

Conclusion

We first summarize what has been done and then comment on the role of operational semantics in language design.

13.1 Summary

We have investigated three type disciplines that accomodate different, and yet similar kinds of polymorphism.

The first was Milner's polymorphic type discipline. We stated its consistency with a dynamic operational semantics by defining a relation between values and types. In particular, the definition of what it is for a closure to have a type is similar to the definition of what it is for a function in a domain to have that type. We then proved the soundness by structural induction.

The second is a new type discipline for polymorphic references. It is less powerful but simpler than Damas' system and I believe the new system is powerful enough to be of practical use. It is easy to extend the distinction between expansive and non-expansive expressions to a larger language. Besides the type inference system itself, the technical novelty is the definition of the relation between values and types as the maximal fixpoint of a monotonic operator.

The third is a type discipline for program modules. We saw how one set of inference rules gives rise to two different semantics, one based on coherence and one based on consistency. The former was investigated in detail. A signature checker was presented and proved sound and complete with respect to the semantics. The signature checker uses a generalization of the ordinary first order term unification algorithm.

Milner's polymorphic type discipline strongly influenced the two others. The

connections are summarized in this table.

The applicative language	The imperative language	The module language
type variable, t	type variable, t or u.	structure name, n
type, τ	type, τ	structure, S
type scheme, σ	type scheme, σ	signature, Σ
substitution, S	substitution, S	realisation, φ
instantiation, $\sigma > \tau$	instantiation, $\sigma > \tau$	instantiation, $\Sigma > S$

As a consequence of the similarity the three type checkers are all very similar.

13.2 How the ML modules evolved.

The work I have done on the modules semantics has been just a small part of the efforts to define a modules concept for ML. This section gives a brief summary of how the ML modules evolved.

It is not possible to decide the origin of every bit of progress, since the design of ML was a collective effort involving a large number of people, but the following should give a rough idea about who did what.

The development of the core language and the development of the modules part up till May 85 is reported in the introduction of [18]. Since then, then main language design challenge has been to gain complete control over the semantics of modules.

Luca Cardelli's ML implementation [7] had a notion of separate compilation, but the basic scheme with structures, signatures and functors was proposed by David MacQueen in 1983 [24]. His proposal was discussed at a design meeting at Edinburgh held in Edinburgh in June 84 and finally approved on a second design meeting held at Edinburgh in May 85. Thus, the proposal gradually changed status and became a definition [18].

Don Sannella was the first to write a modules semantics. He produced several versions of a denotational semantics for modules [39] early in 85. From this effort it became clear that significant semantic issues had not been clarified by the informal definition. In projects where several people try to work together on arriving at a standard, a very delicate problem is who has the right to decide what the language is, and a major reason that the denotational semantics was never finished was that a consensus was hard to achieve. However, this first semantics contained the notion of names and unification quite explicitly.

The work on an operational semantics of ML dates back at least to 1984, when Milner wrote a dynamic semantics for the core language. In summer 85, Harper and I wrote a static operational semantics. I was responsible for the core language. Robert Harper did the modules language semantics concurrently with doing the first modules implementation. The notion of structure names is present both in the semantics and the implementation. The implementation did unification of structures. In the semantics, however, sharing equations did not lead to identification of names in signatures. Instead, sharing equations were held as an attribute of signatures that was checked as a part of matching.

Milner's "webs" [29] represented sharing using congruences rather than names. Eventually, the scheme with names was preferred. Milner presented the ideas of names, realisation and unification and gave inference rules in several notes [30, 31, 32, 33]. He invented the semantic objects that with very minor changes have been used in this thesis and in the full ML semantics [20]. He also gave a signature checker that is similar to Harper's implementation and which served as a model for the ones I present herein.

In June 86, Harper wrote yet another modules semantics with a function GENSTR producing what was subsequently called principal signatures.

I joined the work on the static semantics of modules in April 86. In June 86, I defined a slightly different notion of realisation, argued that realisations correspond to substitutions and generic instance corresponds to signature matching, removed unification from the rules by allowing instantiation and generalisation rules and started talking about principal signatures [40]. It took me from summer 86 to March 87 to prove the existence of principal unifiers and principal signatures based on a coherent semantics.

In March 87 David MacQueen visited Edinburgh and advocated that the semantics should be based on consistency rather than coherence. His ML compiler is based on consistency. (The informal language definition did not clarify whether coherence or consistency should underlie the semantics, so when people wrote a semantics or a compiler they made different choices). Milner realised that the inference rules we had arrived at by then with minor changes could express the consistent semantics. Thus, in March 87, Harper, Milner, MacQueen and I settled for the consistent semantics, because we felt that explicit signature constraints on structure declarations had to be coercive.

Finally, in August 87, we were able to print an operational semantics of

ML [20].

13.3 The Experience of Using Operational Semantics

The ease with which we could state the consistency results in Part I and II highlights the naturalness of operational semantics and basic set theory. We did not have to use domain theory, in fact the symbolic nature of operational semantics is essential for making the typing relation in Part II monotonic hence avoiding the difficult domain construction to which Damas had to resort.

From the outset it was known that the applicative type discipline was unsound in the imperative setting. However, even given a counter–example it is not obvious just why the system is unsound.

Knowing that the system was unsound, I pretended that I wanted to formulate and prove its soundness using operational semantics, and when I found that the proof broke down at one, and only one, point I felt satisfied that the problem is the quantification of type variables that occur free in the store typing. Store typings take part neither in the dynamic nor in the static semantics but without them the soundness theorem was obviously false (see Section 3.2). Concentrating on the proof method lead to the discovery of what the problem was.

The reader may argue that this is not too surprising now that we know that the problem is a very specific technicality. Indeed, the applicability of operational semantics to a problem of this character does not say much about operational semantics used on a large scale. So let us now turn to the experience gained from the work on one big project, the writing of the operational semantics of ML.

As described in the previous section, there were several "competing" semantic approaches. So in case anybody was in doubt, operational semantics does not automatically lead to a straight line development of program language definitions. (The same would be true of any semantic school, I'm sure). However, and this is essential, it *did* provide an excellent focus for technical dicourse. Different semantic alternatives could often be localized to a few changes in the rules and this made the alternatives easier to understand. Moreover, and this must be a criterion for any theoretical framework, most of the problems one is faced with when writing the semantics are genuine semantical problems rather than problems that arise because one has chosen this particular framework.

But how were the semantic issues isolated? Some came as a result of just trying to write down the rules. But there is a long way from writing down a set of rules to understanding how they interact and how they depend upon assumptions about well-formedness constraints on the semantic objects. These things were often understood only as a result of trying to prove some theorems about the skeletal language.

We did not first define a semantics and then prove theorems about it. The definition of the semantics and the properties of it developed side by side influencing each other. The process was akin to that of developing a program and proving it correct at the same time. Thus the proof activity is at the very heart of operational semantics and deserves special attention, not least because finding and proving theorems is a complex intellectual activity, costly in time and energy. What are the complications that arise if we try to scale up the work from a skeletal language to a realistic programming language? To what extend can we envisage the use of computers to assist the proof finding?

The proofs I have done regarding ModL are fairly long, I think, and I want to base my answer to the above two questions on the experience of doing these proofs by hand. In retrospect, it seems that there were two factors that complicated the work.

The first factor is that although we use relatively standard mathematical reasoning, we often find that we are not really interested in reasoning about natural numbers, polynomials, or other objects that we are already familiar with from mathematics. On the contrary, we have to define our own semantic objects, such as structures, signatures, realisations. As these objects are new in our experience we have to do a lot of basic reasoning. The technology of substitutions is a refined one that has taken time to mature. When we generalize to realisations, we have once again to think about renaming, free and bound names and so on, even though the definition of realisation maps itself is simple.

The second factor is that the constuctions we want to prove theorems about are bigger than normal in mathematics. A semantics with 15 inference rules is a very small semantics compared with a real language semantics but it feels big once you start a case analysis with 15 cases. I have not experienced that most of the cases are trivial, indeed the norm has been that many cases require good ideas and that it takes many failed attempts before one finds an induction hypothesis that survives all 15 cases.

The two factors are active at the same time so that the proof activity involves high-level considerations (will this induction hypothesis work?) and, at the same time, low-level considerations (what do I need to know about realisations to rewrite $\varphi\{n \mapsto n_\nu\}\, S$ to $\{n' \mapsto n\}\,\varphi\{n \mapsto n'\}\, S$?). The basic concepts are not given from the outset but emerge as a result of looking for theorems. When one is stuck in a proof, at least while the theory is still young, it is sometimes hard to decide whether to try to change the theorem or the basic definitions that underlie it.

Both factors contribute to giving long proofs, indeed the proofs I have done are long more because of these two factors than because I'm trying to prove something very surprising.

To return to the original questions, the larger the language, the more new semantics objects we have to define and get used to. There will be more connections to understand at the basic theoretical level. The larger the language, the more cases to consider and the more times we have to start all over again. Different individuals can cope with different levels of complexity, of course, but will not everybody feel that at some point they will start forgetting dependencies, getting the same good ideas many times, changing indecisively from alternative to alternative?

That machines should be able to take over the proof finding at the point where we have to give up seems absurd considering the things we can successfully program machines to do. A much more sensible goal is to use the machine to help us organize our proof finding.

A handwritten proof is a sequence of sentences. Some of these serve to explain the main ideas of the proof; some are used to introduce new notation and to bind variables. Others are judgements. A sequence of judgements will normally have to speak for itself. We do not consistently refer to a set of inference rules and lemmas to chain together the individual judgements. After all, we do not want to be more pedantic than we have to in order to feel confident that what we write is correct. For each judgement there is an implicit unproved lemma, namely that this judgement follows from the previous judgements. Whenever we choose to be explicit about how one judgement is obtained from its predecessors, there is the possibility of letting the machine "perform" the inference as a computation so that we don't have to worry about this proof step, but only the rule applied, if the proof is to be done again later on.

Writing a program using just a screen editor or pen and paper is like programming in an untyped language. One has complete freedom but little control of the inferences one makes. The more explicit one makes the structure of a proof, the more work one has to set up the proof, but the pay–off is that one (or a machine) can perform a kind of type checking of the proof. If, for instance you have to check 5 things before you apply an induction hypothesis, you should not get away with forgetting to check one of them. Indeed, the paradigm in much current work on formalizing logics is that proof checking is type checking (see for instance the Edinburgh Logical Framework, [17]). A tool that allows you to be explicit when you want to and does not force you to be explicit when you don't want to could be of tremendous help in the proof finding process.

Incidently, the problem of polymorphic references seems very suitable for machine assisted proof. It has deceived people again and again. Everybody I know who has tried to solve the problem share the experience that firstly, it is hard to say what the problem really is and secondly, that when something seems to work, it is hard to give a good explanation of why it works. With the proof method I have presented I think there has been sufficient progress that it would be a realistic task to try to prove (or disprove) the consistency theorem with the assistance of a machine. Firstly, it would be wonderful to have a strict control of proofs, and secondly, it would be interesting to find out whether the work would result in new knowledge about the problem itself.

13.4 Future Work

Regarding the polymorphic references, the conjecture that there exists principal types needs proving. Moreover we need practical experience to decide whether the inference system is good to use in practice. When "sensible" programs are rejected by the type checker, will the simple device of wrapping up expressions in closures suffice? Also, we need to see how the definitions could be incorporated in the present ML semantics (it should not be very hard).

Regarding the modules semantics an obvious task is to try to establish results for the consistent semantics that resemble those for the coherent semantics. At first sight one might think that the theory for the consistent semantics is simpler, because the notion of realisation is simpler (every realisation in the consistent semantics induces a realisation in the coherent semantics). However, coherence of a set of structures is a stronger condition than consistency, and not every-

thing that works for coherence works for consistency without modifications. For instance, coherence is important for the unification algorithm, I have presented. When relaxing coherence to consistency, special care has to be taken in the unification algorithm to avoid the creation of cyclic structures. One cannot unify the (sub)structures $(\mathtt{n}, \{\})$ and $(\mathtt{m}, \{\})$, say, if somewhere else there is a structure $(\mathtt{n}, \{\mathtt{A} \mapsto (\mathtt{m}, \{\})\})$.

The initial constaint that functors cannot take functors as arguments nor produce functors as result was introduced simply because it was felt that one should not try to do too many difficult things at the same time. Now that we understand the semantics of the first order functors, it does not seem frightening at all to start considering higher order functors. We have already had to introduce functor signatures and functor instances as semantic objects even before there is any program notation for writing down and binding functor signatures.

Appendix A

Robustness Proofs

This appendix contains the proofs of the robustness results stated in Chapter 10.

Lemma 10.1 *If* Σ *is admissible and* $\Sigma > S'$ *and* $\Sigma \xrightarrow{\psi} \Sigma_1$ *then* $\Sigma_1 > \psi\, S'$.

Proof. Write Σ as $(N)S$ and Σ_1 as $(N_1)S_1$. Let $\psi_0 = \psi \downarrow \mathrm{strs}\,\Sigma$. Since $\Sigma \xrightarrow{\psi} \Sigma_1$ there exists a bijection $\rho : N \rightarrow N_1$ such that $N_1 \cap \mathrm{Reg}\,\psi_0 = \emptyset$ and

$$(\psi_0\,|\rho)\, S = S_1\,. \tag{A.1}$$

By the coherence of S, for every structure, T, that occurs *bound* in Σ_1, there is a unique structure, let us call it $\psi^{-1}\,T$, bound in Σ, such that

$$(\psi_0\,|\rho)(\psi^{-1}\,T) = T.$$

Since $\Sigma > S'$ there exists a realisation φ with $\mathrm{Supp}\,\varphi \subseteq \mathrm{boundstrs}\,\Sigma$ and $\varphi\, S = S'$.

For any substructure T of S_1 let us define

$$\varphi'(T) = \begin{cases} \psi(\varphi(\psi^{-1}\,T)), & \text{if } n \text{ of } T \in N_1 \\ T, & \text{otherwise} \end{cases} \tag{A.2}$$

and extend it to a total map by letting it glide on all structures that do not occur in S_1. To prove $\Sigma_1 > \psi\, S'$ it will suffice to prove

$$\varphi' \text{ is a realisation} \tag{A.3}$$

$$\varphi'\, S_1 = \psi\, S' \tag{A.4}$$

$$\text{Supp}\,\varphi' \subseteq \text{boundstrs}\,\Sigma_1 \tag{A.5}$$

As for (A.3), take any T occurring in S_1 and any *strid* $\in \text{Dom}\,T$. We have to prove that *strid* $\in \text{Dom}\,\varphi' T$ and $(\varphi' T)\mathit{strid} = \varphi'(T\mathit{strid})$ c.f. definition 9.4. Looking at (A.2), this is obvious when T is free in Σ_1.

So assume T is bound in Σ_1. Recalling (A.1), we see that $\text{Dom}\,T = \text{Dom}(\psi^{-1}T)$. Thus *strid* $\in \text{Dom}(\psi^{-1}T)$. We proceed by case analysis.

$(\psi^{-1}T)\mathit{strid}$ is free in Σ Then

$$\begin{array}{lll}
& \varphi'(T\mathit{strid}) & \\
= & \varphi'(((\psi_0\,|\rho)(\psi^{-1}T))\mathit{strid}) & \text{as } T \text{ is bound} \\
= & \varphi'((\psi_0\,|\rho)((\psi^{-1}T)\mathit{strid})) & \text{as } \mathit{strid} \in \text{Dom}\,\psi^{-1}T \text{ and } \psi_0\,|\rho \text{ is a realisation} \\
= & \varphi'(\psi((\psi^{-1}T)\mathit{strid})) & \text{as } \Sigma \text{ is well-formed and } (\psi^{-1}T)\mathit{strid} \text{ is free in } \Sigma \\
= & \psi((\psi^{-1}T)\mathit{strid}) & \text{as this is free in } \Sigma_1 \\
= & \psi(\varphi((\psi^{-1}T)\mathit{strid})) & \text{as } \varphi \text{ is the identity on free structures of } \Sigma \text{ since } \Sigma \text{ is well-formed} \\
= & (\psi\,\varphi(\psi^{-1}T))\mathit{strid} & \text{as } \psi \text{ and } \varphi \text{ are realisations} \\
= & (\varphi'(T))\mathit{strid} & \text{by (A.2)}
\end{array}$$

as desired.

$(\psi^{-1}T)\mathit{strid}$ is bound in Σ Then

$$\begin{array}{lll}
T(\mathit{strid}) & = & ((\psi_0\,|\rho)(\psi^{-1}T))\mathit{strid} \quad \text{as } T \text{ is bound} \\
& = & (\psi_0\,|\rho)((\psi^{-1}T)\mathit{strid})
\end{array} \tag{A.6}$$

as *strid* $\in \text{Dom}(\psi^{-1}T)$. Thus, since $(\psi^{-1}T)\mathit{strid}$ is bound in Σ, we have that $T(\mathit{strid})$ is bound in Σ_1. Thus by the definition of φ',

$$\varphi'(T(\mathit{strid})) = \psi(\varphi(\psi^{-1}(T(\mathit{strid})))). \tag{A.7}$$

By (A.6) we have

$$\psi^{-1}(T(\mathit{strid})) = (\psi^{-1}T)\mathit{strid}$$

so continuing (A.7) we have

$$\begin{array}{lll}
\cdots & = & \psi(\varphi((\psi^{-1}T)\mathit{strid})) \\
& = & (\psi\,\varphi(\psi^{-1}T))\mathit{strid} \quad \text{as } \varphi \text{ and } \psi \text{ are realisations} \\
& = & (\varphi' T)\mathit{strid}
\end{array}$$

as desired.

This proves (A.3). Next, we prove (A.4) by a (much simpler) case analysis:

$\boxed{S \text{ is bound in } \Sigma}$ Then $(\psi_0 \,|\rho)(S)$ is bound in Σ_1. Then

$$\begin{aligned}
\varphi'(S_1) &= \varphi'((\psi_0 \,|\rho)\, S) \quad \text{by (A.1)} \\
&= \psi(\varphi(S)) \quad \text{by (A.2)} \\
&= \psi\, S'.
\end{aligned}$$

$\boxed{S \text{ is free in } \Sigma}$ Then boundstrs $\Sigma = \emptyset$, so $S = \varphi\, S = S'$. Also, $(\psi_0 \,|\rho)\, S = \psi\, S$ is free in Σ_1, so $\varphi'(S_1) = \varphi'(\psi\, S) = \psi\, S = \psi\, S'$ as desired.

This proves (A.4). Finally, (A.5) follows immediately from (A.2). ∎

Lemma 10.2 *If* $\Sigma \xrightarrow{\varphi} \Sigma_1$ *and* $\Sigma' \xrightarrow{\varphi} \Sigma'_1$ *and* $\Sigma \geq \Sigma'$ *and* Σ *is admissible then* $\Sigma_1 \geq \Sigma'_1$.

Proof. From $\Sigma \geq \Sigma'$ and Σ admissible we get

$$\operatorname{strs}\Sigma \subseteq \operatorname{strs}\Sigma' \tag{A.8}$$

by corollary 9.16. Write Σ' on the form $(N')S'$. Then since $\Sigma' \xrightarrow{\varphi} \Sigma'_1$ we have

$$\Sigma'_1 = (N'_1)S'_1 = (N'_1)(\varphi'_0 \,|\rho')\, S',$$

where $\rho' : N' \rightarrow N'_1$ is a bijection and $N'_1 \cap \operatorname{Reg}\varphi'_0 = \emptyset$, where $\varphi'_0 = \varphi \downarrow \operatorname{strs}\Sigma'$. Since (A.8) and $\Sigma \xrightarrow{\varphi} \Sigma_1$ we have

$$\Sigma \xrightarrow{\varphi'_0 \,|\rho'} \Sigma_1. \tag{A.9}$$

Surely $\Sigma > S$, so from (A.9) and lemma 10.1 we get

$$\Sigma_1 > (\varphi'_0 \,|\rho')\, S\,. \tag{A.10}$$

Now since $\Sigma \xrightarrow{\varphi} \Sigma_1$ and Σ is well-formed, we have

$$\operatorname{strnames}\Sigma_1 = \operatorname{Reg}(\varphi \downarrow \operatorname{strs}\Sigma). \tag{A.11}$$

Since $\operatorname{Reg}\varphi'_0 \cap N'_1 = \emptyset$ and (A.8) we have $\operatorname{Reg}(\varphi \downarrow \operatorname{strs}\Sigma) \cap N'_1 = \emptyset$. Thus by (A.11) we have $\operatorname{strnames}\Sigma_1 \cap N'_1 = \emptyset$. Therefore we can use lemma 9.15 to strengthen (A.10) to the desired

$$\begin{aligned} \Sigma_1 &\geq (N'_1)(\varphi'_0 \,|\rho')\, S \\ &= \Sigma'_1. \end{aligned}$$

■

Lemma 10.4 *For all functor signatures Φ_1, Φ_2, and functor instances I_3, I_4, if Φ_1 is admissible and $\Phi_1 > I_3$ and $\Phi_1 \xrightarrow{\varphi} \Phi_2$ and $I_3 \xrightarrow{\varphi} I_4$ then $\Phi_2 > I_4$.*

Proof. The following diagram may be used for reference:

$$\begin{array}{cccccc} & \Phi_1 & \xrightarrow{\varphi} & \Phi_2 & \\ \psi_1 & \vee & & \vee & \psi_2 \\ & I_3 & \xrightarrow{\varphi} & I_4 & \end{array}$$

Write

$$\begin{array}{ll} \Phi_1 = (N_1)(S_1, (N_1')S_1') & \Phi_2 = (N_2)(S_2, (N_2')S_2') \\ I_3 = (S_3, (N_3')S_3') & I_4 = (S_4, (N_4')S_4'). \end{array}$$

By the assumptions all of these are well-formed.

Let $\varphi_1 = \varphi \downarrow \text{strs}(N_1)S_1$ and let $\varphi_1' = \varphi \downarrow \text{strs}(N_1 \cup N_1')\, S_1'$. Since $\Phi_1 \xrightarrow{\varphi} \Phi_2$ there exists a bijection

$$\rho : N_1 \cup N_1' \to N_2 \cup N_2'$$

such that

$$\rho_1 : N_1 \to N_2 = \rho \downarrow N_1$$

and

$$\rho_1' : N_1' \to N_2' = \rho \downarrow N_1'$$

are bijections and

$$N_2 \cap \text{Reg}\,\varphi_1 = \emptyset \quad \text{and} \quad (\varphi_1\,|\rho_1)\,S_1 = S_2 \tag{A.12}$$

and

$$(N_2 \cup N_2') \cap \text{Reg}\,\varphi_1' = \emptyset \quad \text{and} \quad (\varphi_1'\,|\rho)\,S_1' = S_2'. \tag{A.13}$$

As in the proof of lemma 10.1 for any structure T *bound* in $(N_2)S_2$ there is a unique structure, call it $\varphi^{-1}T$, bound in $(N_1)S_1$ such that

$$(\varphi_1\,|\rho_1)(\varphi^{-1}T) = T \tag{A.14}$$

(this follows from (A.12) and the coherence of S_1).

Moreover, any structure T free in $(N_2')S_2'$ with n of $T \in N_2$ must occur bound in$(N_2)S_2$ (since Φ_2 is well-formed); thus $\varphi^{-1}(T)$ exists and occurs free in $(N_1')S_1'$ and n of $(\varphi^{-1}T) \in N_1$.

Since $\Phi_1 > I_3$ there exists a realisation ψ_1 with

$$(S_1, (N_1')S_1') \xrightarrow{\psi_1} (S_3, (N_3')S_3') \tag{A.15}$$

and

$$\operatorname{Supp}\psi_1 \subseteq \operatorname{boundstrs}(N_1)S_1. \tag{A.16}$$

Let ψ_1' be $\psi_1 \downarrow \operatorname{strs}(N_1')S_1'$. By (A.15) there exists a bijection $\rho_{13}' : N_1' \to N_3'$ such that

$$N_3' \cap \operatorname{Reg}\psi_1' = \emptyset \quad \text{and} \quad (\psi_1' \,|\rho_{13}')\, S_1' = S_3'. \tag{A.17}$$

By the final assumption, $I_3 \xrightarrow{\varphi} I_4$, we have

$$\varphi\, S_3 = S_4 \quad \text{and} \quad (N_3')\, S_3' \xrightarrow{\varphi} (N_4')\, S_4'. \tag{A.18}$$

Let $\varphi_3' = \varphi \downarrow \operatorname{strs}(N_3')S_3'$. Then there exists a bijection $\rho_3' : N_3' \to N_4'$ such that

$$N_4' \cap \operatorname{Reg}\varphi_3' = \emptyset \quad \text{and} \quad (\varphi_3' \,|\rho_3')\, S_3' = S_4'. \tag{A.19}$$

We can now define the realisation that instantiaties Φ_2 to I_4. For every $S = (n, E) \in \operatorname{Str}$, we define

$$\psi_2(S) = \begin{cases} \varphi(\psi_1(\varphi^{-1} S)), & \text{if } S \in \operatorname{boundstrs}(N_2)S_2 \\ S, & \text{if } S \in \operatorname{strs}(N_2)S_2 \\ (n, \psi_2\, E), & \text{otherwise.} \end{cases} \tag{A.20}$$

As in the proof of lemma 10.1 it is proved that ψ_2 is a realisation with $\operatorname{Supp}\psi_2 \subseteq \operatorname{boundstrs}(N_2)S_2$ and $\psi_2\, S_2 = S_4$. To prove $\Phi_2 > I_4$ we are therefore left with proving

$$(N_2')S_2' \xrightarrow{\psi_2} (N_4')S_4'.$$

Let $\psi_2' = \psi_2 \downarrow \operatorname{strs}(N_2')S_2'$ and let

$$\rho_2' = \rho_3' \circ \rho_{13}' \circ (\rho_1')^{-1}.$$

Notice that ρ_2' is a bijection from N_2' to N_4'. It suffices to prove

$$N_4' \cap \operatorname{Reg}\psi_2' = \emptyset \tag{A.21}$$

and

$$(\psi_2' \,|\rho_2')\, S_2' = S_4'. \tag{A.22}$$

To prove (A.21), observe that

$$\operatorname{strs}(N_2')S_2' = \operatorname{strs}\{(\varphi_1' \,|\rho_1)\, S \mid S \in \operatorname{strs}(N_1')S_1'\}. \tag{A.23}$$

Take any $S \in \operatorname{strs}(N_1')S_1'$. Two cases:

$\boxed{S \in \mathrm{strs}(N_1 \cup N_1')\, S_1'}$ Since $(N_1 \cup N_1')S_1'$ is well-formed we have that $(\varphi_1' \,|\rho_1)\, S = \varphi\, S$ is free in $(N_2 \cup N_2')\, S_2'$. Thus $\psi_2((\varphi_1' \,|\rho_1)\, S) = \varphi\, S$. Since $\mathrm{Supp}\, \psi_1 \subseteq N_1$–Str, we have that S is free in $(N_3')S_3'$. Thus $\varphi\, S$ is free in $(N_4')S_4'$ i.e., $\psi_2((\varphi_1' \,|\rho_1)\, S) \cap N_4\text{–Str} = \emptyset$ as desired.

$\boxed{S \in \mathrm{strs}(N_1')S_1' \text{ and } n \text{ of } S \in N_1}$ Then $(\varphi_1' \,|\rho_1)\, S$ is free in $(N_2')S_2'$ and n of $(\varphi_1' \,|\rho_1)\, S \in N_2$. Thus

$$\begin{aligned} \psi_2((\varphi_1' \,|\rho_1)\, S) &= \psi_2((\varphi_1 \,|\rho_1)\, S) \quad \text{as } \Phi_1 \text{ is well-formed} \\ &= \varphi(\psi_1\, S) \quad \text{by (A.20).} \end{aligned}$$

Now $\psi_1\, S$ is free in $(N_3')S_3'$ so $\varphi(\psi_1\, S)$ is free in $(N_4')\, S_4'$ as desired.

This proved (A.21). To prove (A.22) we calculate that

$$(\psi_2' \,|\rho_2')\, S_2' = (\psi_2' \,|\rho_2')(\varphi_1' \,|\rho)\, S_1'$$

using (A.13), and

$$\begin{aligned} S_4' &= (\varphi_3' \,|\rho_3')\, S_3' \\ &= (\varphi_3' \,|\rho_3')(\psi_1' \,|\rho_{13}')\, S_1' \end{aligned}$$

by (A.19) and (A.17), respectively. Thus it will suffice to prove

$$\forall\, S \in \mathrm{strs}\, S_1', \quad (\psi_2' \,|\rho_2')(\varphi_1' \,|\rho)\, S = (\varphi_3' \,|\rho_3')(\psi_1' \,|\rho_{13}')\, S\,. \tag{A.24}$$

This we prove by structural induction on S. Let $S = (n, E)$ and assume that (A.24) holds for all structures in E. We prove that it holds for S by case analysis:

$\boxed{S \text{ is free in } (N_1 \cup N_1')\, S_1'}$ Here

$$\begin{aligned} (\psi_2' \,|\rho_2')(\varphi_1' \,|\rho)\, S &= (\psi_2' \,|\rho_2')\, \varphi\, S && \text{as } (N_1 \cup N_1')S_1' \text{ is well-formed} \\ &= \varphi\, S && \text{as } \varphi\, S \text{ is free in } (N_2 \cup N_2')S_2' \\ &= (\varphi_3' \,|\rho_3')\, S && \text{as } S \text{ is free in } (N_3 \cup N_3')S_3' \\ &= (\varphi_3' \,|\rho_3')\, \psi_1\, S && \text{as } \mathrm{Supp}\, \psi_1 \subseteq N_1\text{–Str} \\ &= (\varphi_3' \,|\rho_3')(\psi_1' \,|\rho_{13}')\, S && \text{as } (N_1 \cup N_1')S_1' \text{ is well-formed.} \end{aligned}$$

$\boxed{S \text{ is free in } (N_1')\,S_1' \text{ and } n \text{ of } S \in N_1}$ Thus

$$
\begin{aligned}
(\psi_2' \,|\rho_2')(\varphi_1' \,|\rho)\, S &= (\psi_2' \,|\rho_2')(\varphi_1 \,|\rho_1)\, S && \text{as } S \in \text{boundstrs}(N_1)S_1 \text{ and } \Phi_1 \text{ is well-formed} \\
&= \psi_2(\varphi_1 \,|\rho_1)\, S && \text{as } (\varphi_1 \,|\rho_1)\, S \text{ is free in } (N_2')S_2' \\
&= \varphi(\psi_1\, S) && \text{by the definition of } \psi_2 \\
&= (\varphi_3' \,|\rho_3')(\psi_1\, S) && \text{as } \psi_1\, S \text{ is free in } (N_3')S_3' \\
&= (\varphi_3' \,|\rho_3')(\psi_1' \,|\rho_{13}')\, S && \text{as } S \text{ is free in } (N_1')S_1'.
\end{aligned}
$$

$\boxed{S \text{ is bound in } (N_1')S_1'}$ Thus

$$
\begin{aligned}
(\psi_2' \,|\rho_2')(\varphi_1' \,|\rho)\, S &= (\psi_2' \,|\rho_2')(\varphi_1' \,|\rho)(n, E) \\
&= (\psi_2' \,|\rho_2')(\rho(n), (\varphi_1' \,|\rho)\, E) \\
&= ((\rho_2' \circ \rho)n, (\psi_2' \,|\rho_2')(\varphi_1' \,|\rho)\, E) \\
&= ((\rho_3' \circ \rho_{13}')n, (\psi_2' \,|\rho_2')(\varphi_1' \,|\rho)\, E) \\
&= ((\rho_3' \circ \rho_{13}')n, (\varphi_3' \,|\rho_3')(\psi_1' \,|\rho_{13}')\, E) && \text{by induction} \\
&= (\varphi_3' \,|\rho_3')(\rho_{13}'(n), (\psi_1' \,|\rho_{13}')\, E) \\
&= (\varphi_3' \,|\rho_3')(\psi_1' \,|\rho_{13}')(n, E) \\
&= (\varphi_3' \,|\rho_3')(\psi_1' \,|\rho_{13}')\, S
\end{aligned}
$$

as required. ∎

Theorem 10.5 *Let* $B = M, F, G, E$ *and* $B' = M', F', G', E'$ *be bases with* $B \xrightarrow[\text{rb}]{\varphi} B'$.
If

$$B \vdash spec \Rightarrow E_1 \tag{10.1}$$

$$M'\text{-strs}(\varphi\, E_1) \text{ is substructure closed} \tag{10.2}$$

$$\{B', \varphi\, E_1\} \text{ is coherent} \tag{10.3}$$

then

$$B' \vdash spec \Rightarrow \varphi\, E_1 .$$

Similarly, if

$$B \vdash sigexp \Rightarrow \Sigma \text{ and } \Sigma \xrightarrow{\varphi} \Sigma' \tag{10.4}$$

$$M'\text{-strs}\, \Sigma' \text{ is substructure closed} \tag{10.5}$$

$$\{B', \Sigma'\} \text{ is coherent} \tag{10.6}$$

then

$$B' \vdash sigexp \Rightarrow \Sigma'.$$

Proof. By induction on the depth of inference of (10.1) and (10.4). One case for each inference rule.

| Empty specification, rule 8.8 | Trivial.

| Structure specification, rule 8.9 |

$$\frac{B \vdash sigexp \Rightarrow (\emptyset)\, S}{B \vdash \textbf{structure } strid\ :\ sigexp \Rightarrow \{strid\ \mapsto\ S\}}$$

So $E_1 = \{strid \mapsto S\}$. Assume (10.2) and (10.3). Now $M'\text{-strs}(\varphi\, E_1) = M'\text{-strs}(\varphi\, S) = M'\text{-strs}\,\Sigma'$, for $\Sigma' = (\emptyset)(\varphi\, S)$. Therefore we have (10.4)–(10.5), so by induction we have $B' \vdash sigexp \Rightarrow (\emptyset)(\varphi\, S)$. Thus by rule 8.9 we have $B' \vdash \textbf{structure } strid\ :\ sigexp \Rightarrow \varphi\, E_1$ as desired.

Sequential specification, rule 8.10

$$\frac{B \vdash spec_1 \Rightarrow E_1 \quad B \pm E_1 \vdash spec_2 \Rightarrow E_2}{B \vdash spec_1\ spec_2 \Rightarrow E_1 \pm E_2} \tag{A.25}$$

Assume

$$M'\text{–strs}(\varphi(E_1 \pm E_2)) \text{ is substructure closed} \tag{A.26}$$

$$\{B', \varphi(E_1 \pm E_2)\} \text{ is coherent.} \tag{A.27}$$

Unfortunately, this is not enough to allow us to use induction directly on the first premise (we need not have that M'–strs$(\varphi\, E_1)$ is substructure closed and that $\{B', \varphi\, E_1\}$ is coherent). Therefore we shall apply induction using a new realisation φ' derived from φ as follows:

$$\varphi'(n_0, E_0) = \begin{cases} \varphi(n_0, E_0), & \text{if } (n_0, E_0) \in \text{strs}\{B, E_1 \pm E_2\} \\ (\imath(n_0, E_0), \varphi' E_0), & \text{if } (n_0, E_0) \in \text{strs}\, E_1 \setminus \text{strs}\{B, E_1 \pm E_2\} \\ (n_0, \varphi' E_0) & \text{otherwise,} \end{cases} \tag{A.28}$$

where $\imath$ is any injective map from strs $E_1 \setminus \text{strs}\{B, E_1 \pm E_2\}$ to StrName $\setminus (M' \cup N')$, where $N' = \text{strnames}\{E', \varphi(E_1 \pm E_2)\}$. Clearly, φ' is a realisation.

In order to use induction we wish to establish that

$$B \xrightarrow[\text{rb}]{\varphi'} B' \tag{A.29}$$

$$M'\text{–strs}(\varphi' E_1) \text{ is substructure closed} \tag{A.30}$$

$$\{B', \varphi' E_1\} \text{ is coherent.} \tag{A.31}$$

But (A.29) is obvious; and (A.30) follows from (A.26) and the definition of φ'; and (A.31) follows from (A.27) and the properties of $\imath$.

Thus by induction

$$B' \vdash spec_1 \Rightarrow \varphi' E_1\,. \tag{A.32}$$

To use induction again, we check that

$$B \pm E_1 \xrightarrow[\text{rb}]{\varphi'} B' \pm \varphi' E_1 \tag{A.33}$$

$$M'\text{–strs}(\varphi' E_2) \text{ is substructure closed} \tag{A.34}$$

$$\{B' \pm \varphi' E_1, \varphi' E_2\} \text{ is coherent.} \tag{A.35}$$

As to (A.33), $B \pm E_1$ is robust by the first premise of (A.25), while $B' \pm \varphi' E_1$ is robust by (A.32) and

$$\varphi'(E \text{ of}(B \pm E_1)) = \varphi'(E \text{ of}\, B) \pm \varphi' E_1 = E \text{ of}(B' \pm \varphi' E_1).$$

Next, (A.34) follows from (A.26) and (A.28). Also, (A.35) follows from (A.27) and the definition of φ'. Hence by induction on the second premise of (A.25),

$$B' \pm \varphi' E_1 \vdash spec_2 \Rightarrow \varphi' E_2,$$

which with (A.32) gives

$$B' \vdash spec_1 \; spec_2 \Rightarrow \varphi' E_1 \pm \varphi' E_2 \qquad \text{(A.36)}$$

by rule (8.10). To check that this conclusion is admissible (c.f. definition 9.12) we must check that

$$\{B', \varphi'(E_1 \pm E_2)\} \text{ is admissible} \qquad \text{(A.37)}$$

$$\text{strnames}\, F' \cup \text{strnames}\, G' \subseteq M' \qquad \text{(A.38)}$$

$$M'\text{–strs}\, E' \text{ and } M'\text{–strs}(\varphi'(E_1 \pm E_2)) \text{ are substructure closed.} \qquad \text{(A.39)}$$

The well-formedness of $\{B', \varphi'(E_1 \pm E_2)\}$ comes from the robustness of B'; the coherence from (A.27) and the definition of φ'. Thus (A.37) holds.

Next, (A.38) and the first half of (A.39) follow from the robustness of B', and the second half of (A.39) follows from (A.26) and (A.28). From (A.36) and (A.28) we get

$$B' \vdash spec_1 \; spec_2 \Rightarrow \varphi(E_1 \pm E_2)$$

as required.

Sharing specification, rule 8.11

$$\frac{n \text{ of } B(longstrid_1) = n \text{ of } B(longstrid_2)}{B \vdash \textbf{sharing } longstrid_1 = longstrid_2 \Rightarrow \{\}} \qquad \text{(A.40)}$$

Since B is coherent the premise implies that $B(longstrid_1)$ equals $B(longstrid_2)$. Thus we have

$$\begin{aligned} B'(longstrid_1) &= E' \, longstrid_1 \\ &= (\varphi \, E) longstrid_1 \\ &= \varphi(E \, longstrid_1) \quad \text{as } \varphi \text{ is a realisation} \\ &= \varphi(E \, longstrid_2) \quad \text{by (A.40)} \\ &= (\varphi \, E) longstrid_2 \\ &= B'(longstrid_2). \end{aligned}$$

In particular, n of $B'(longstrid_1) = n$ of $B'(longstrid_2)$, so by rule (8.11) we have

$$B' \vdash \textbf{sharing } longstrid_1 = longstrid_2 \Rightarrow \{\}$$

which is the desired conclusion . The conclusion is admissible.

Basic signature expression, rule 8.12

$$\frac{B \vdash spec \Rightarrow E_1}{B \vdash \textbf{sig } spec \textbf{ end} \Rightarrow (\emptyset)(n, E_1)} \tag{A.41}$$

Thus $\Sigma = (\emptyset)(n, E_1)$. Assume $\Sigma \xrightarrow{\varphi} \Sigma'$. Then Σ' must be $(\emptyset)\,\varphi(n, E_1)$. Thus the assumptions (10.5) and (10.6) read

$$M'\text{–strs}(\varphi(n, E_1)) \text{ is substructure closed} \tag{A.42}$$

$$\{B', \varphi(n, E_1)\} \text{ is coherent.} \tag{A.43}$$

Since φ is a realisation this implies

$$M'\text{–strs}(\varphi\, E_1) \text{ is substructure closed} \tag{A.44}$$

$$\{B', \varphi\, E_1\} \text{ is coherent.} \tag{A.45}$$

Thus by induction on the premise of (A.41) we have

$$B' \vdash spec \Rightarrow \varphi\, E_1\,. \tag{A.46}$$

Using rule 8.12 on (A.46) we get

$$B' \vdash \textbf{sig } spec \textbf{ end} \Rightarrow (\emptyset)(n', \varphi\, E_1) \tag{A.47}$$

where we choose n' disjoint from strnames$\{B'\,\varphi\, E_1\}$ in order to make sure that (A.47) is an admissible conclusion.

Then by the generalization rule, rule 8.14, we have

$$B' \vdash \textbf{sig } spec \textbf{ end} \Rightarrow (\{n'\})(n'\,\varphi\, E_1), \tag{A.48}$$

and this is an admissible conclusion.

Now $(\{n'\})(n', \varphi\, E_1) > \varphi(n, E_1)$ even though φ may "widen" (n, E_1). Thus by lemma 9.15 we have $(\{n'\})(n', \varphi\, E_1) \geq (\emptyset)(\varphi(n, E_1))$. Thus by the instantiation rule, rule 8.15, on (A.48) we get

$$B' \vdash \textbf{sig } spec \textbf{ end} \Rightarrow (\emptyset)(\varphi(n, E_1))$$

i.e.,

$$B' \vdash \textbf{sig } spec \textbf{ end} \Rightarrow \Sigma'$$

as desired. (It is easy to check that this conclusion is admissible by using (A.42) and (A.43)).

The generalization rule, rule 8.14

$$\frac{B \vdash \mathit{sigexp} \Rightarrow (N_1)\, S_1 \quad n_1 \notin \text{strnames}\, B}{B \vdash \mathit{sigexp} \Rightarrow (N_1 \cup \{n_1\})\, S_1}.$$

Thus $\Sigma = (N_1 \cup \{n_1\})\, S_1$. Assume $\Sigma \xrightarrow{\varphi} \Sigma'$ and

$$M'\text{–strs}\, \Sigma' \text{ is substructure closed} \tag{A.49}$$

$$\{B', \Sigma'\} \text{ is coherent.} \tag{A.50}$$

Since $\Sigma \xrightarrow{\varphi} \Sigma'$, Σ' has the form $(N')\, S_1'$ where there is a bijection

$$\rho : N_1 \cup \{n_1\} \to N'$$

such that $\text{Reg}\, \varphi_0 \cap N' = \emptyset$ and $(\varphi_0 | \rho)\, S_1 = S_1'$, where $\varphi_0 = \varphi \downarrow \text{strs}\, \Sigma$. The bijection ρ can be partitioned into two bijections $R_1 : N_1 \to N_1'$ and $\rho_1 : \{n_1\} \to \{n_1'\}$, where $N_1' \cup \{n_1'\} = N'$.

First, let us consider the case where n_1 is redundant i.e., $n_1 \notin \text{strnames}(N_1)S_1$. We then have

$$(N_1)S_1 \xrightarrow{\varphi} (N_1')S_1'$$

since $\text{strs}(N_1)\, S_1 = \text{strs}(N \cup \{n_1\})\, S_1$. Since also $\text{strs}(N_1')\, S_1' = \text{strs}\, \Sigma'$ we have that $M'\text{–strs}(N_1')\, S_1'$ is substructure closed and that $\{B, (N_1')\, S_1'\}$ is coherent using (A.49) and (A.50). Thus by induction we have

$$B' \vdash \mathit{sigexp} \Rightarrow (N_1')\, S_1'$$

which is an admissible conclusion. Take any structure name n_ν which is neither in N_1' nor in $\text{strnames}\{B', S_1'\}$. Then by the generalization rule

$$B' \vdash \mathit{sigexp} \Rightarrow (N_1' \cup \{n_\nu\})\, S_1'. \tag{A.51}$$

If follows from lemma 9.17 that

$$(N_1' \cup \{n_1'\})S_1' \equiv (N_1' \cup \{n_\nu\})\, S_1'$$

so in particular

$$\begin{aligned}(N_1' \cup \{n_\nu\})\, S_1' &\geq (N_1' \cup \{n_1'\})\, S_1' \\ &= \Sigma'.\end{aligned}$$

Thus, using the instantiation rule on (A.51) we get the desired

$$B' \vdash sigexp \Rightarrow \Sigma'$$

which is an admissible conclusion.

Secondly, let us consider the case that n_1 is free in $\Sigma_1 = (N_1)\, S_1$. Since $(N_1)\, S_1$ and $(N_1 \cup \{n_1\})\, S_1$ are both well-formed there exists precisely one structure, S_0, free in $(N_1)\, S_1$ such that $n \text{ of } S_0 = n_1$. Every proper substructure of S_0 is free in $(N_1 \cup \{n_1\})\, S_1$.

Choose an n_ν not in $\text{strnames}\{B', \Sigma'\} \cup N_1'$. Since $n_1 \notin N_1$ we can define

$$\rho_\nu = R_1 \cup \{n_1 \mapsto n_\nu\}$$

which is a bijection

$$\rho_\nu : N_1 \cup \{n_1\} \rightarrow N_1' \cup \{n_\nu\}.$$

Define a map $\psi : \text{Str} \rightarrow \text{Str}$ by

$$\psi(n, E) = \begin{cases} (n_\nu, \varphi\, E), & \text{if } n = n_1 \\ \varphi(n, E), & \text{if } (n, E) \in \text{strs}\{B, (N_1 \cup \{n_1\})\, S_1\} \\ (n, \psi\, E), & \text{otherwise.} \end{cases}$$

Because φ is a realisation and $(N_1)\, S_1$ and $(N_1 \cup \{n_1\})\, S_1$ are well-formed and $n_1 \notin \text{strnames}\, B$, ψ is a realisation. We intend to use induction on ψ. Clearly we have $B \xrightarrow[\text{rb}]{\psi} B'$, and we also have that $\Sigma_1 \xrightarrow{\psi} \Sigma_1'$, where $\Sigma_1' = (N_1')\{n_1' \mapsto n_\nu\}\, S_1'$. Because of the way we chose n_ν, we get that M'–$\text{strs}\, \Sigma_1'$ is substructure closed and that $\{B', \Sigma_1'\}$ is coherent using (A.49) and (A.50).

Thus by induction we have

$$B' \vdash sigexp \Rightarrow (N_1')\{n_1' \mapsto n_\nu\}\, S_1' \tag{A.52}$$

which is an admissible conclusion. Since $n_\nu \notin \text{strs}\, B'$, and this is why we did the renaming, we can use the generalization rule to conclude

$$B' \vdash sigexp \Rightarrow (N_1' \cup \{n_\nu\})\{n_1' \mapsto n_\nu\}\, S_1'. \tag{A.53}$$

It follows from lemma 9.17 that

$$(N_1' \cup \{n_1'\})\, S_1' \equiv (N_1' \cup \{n_\nu\})\{n_1' \mapsto n_\nu\}\, S_1'$$

so in particular

$$\begin{aligned} (N_1' \cup \{n_\nu\})\{n_1' \mapsto n_\nu\}\, S_1' &\geq (N_1' \cup \{n_1'\})\, S_1' \\ &= \Sigma'. \end{aligned}$$

Thus, using the instantiation rule on (A.53) we get the desired

$$B' \vdash sigexp \Rightarrow \Sigma'$$

which is an admissible conclusion.

The instantiation rule, rule 8.15

$$\frac{B \vdash sigexp \Rightarrow \Sigma_1 \quad \Sigma_1 \geq \Sigma}{B \vdash sigexp \Rightarrow \Sigma} \tag{A.54}$$

Assume $\Sigma \xrightarrow{\varphi} \Sigma'$ and

$$M'\text{–strs}\,\Sigma' \text{ is substructure closed} \tag{A.55}$$

$$\{B', \Sigma'\} \text{ is coherent.} \tag{A.56}$$

Σ_1 is admissible, since the conclusion $B \vdash sigexp \Rightarrow \Sigma_1$ must be admissible. There exists a Σ_1' such that $\Sigma_1 \xrightarrow{\varphi} \Sigma_1'$. By lemma 10.2 we have $\Sigma_1' \geq \Sigma'$. Since Σ_1' is well-formed we then have that $\text{strs}\,\Sigma_1' \subseteq \text{strs}\,\Sigma'$ by corollary 9.16. Thus (A.55) and (A.56) imply

$$M'\text{–strs}\,\Sigma_1' \text{ is substructure closed}$$

$$\{B', \Sigma_1'\} \text{ is coherent.}$$

Using induction on these and the first premise of (A.54) we get $B' \vdash sigexp \Rightarrow \Sigma_1'$ and since $\Sigma_1' \geq \Sigma'$ we use the instantiation rule to get the desired conclusion $B' \vdash sigexp \Rightarrow \Sigma'$. The conclusion is admissible. ∎

Theorem 10.8 *Let $B = M, F, G, E$ be strictly robust.*
If $B \vdash dec \Rightarrow E$ then E is coherent and M–strs $E \subseteq$ strs B,
if $B \vdash strexp \Rightarrow S$ then S is coherent and M–strs $S \subseteq$ strs B, and
if $B \vdash prog \Rightarrow B'$ then B' is strictly robust and M–strs $B' \subseteq$ strs B.

Proof. By induction on the depth of inference. We start by declarations, leaving out the trivial cases where the declaration is an empty declaration or a structure declaration. The next case is

Sequential declaration, rule 8.4

$$\frac{B \vdash dec_1 \Rightarrow E_1 \quad B \oplus E_1 \vdash dec_2 \Rightarrow E_2}{B \vdash dec_1\ dec_2 \Rightarrow E_1 \pm E_2}$$

By induction

$$E_1 \text{ is coherent} \tag{A.57}$$

$$M\text{–strs}\, E_1 \subseteq \text{strs}\, B\,. \tag{A.58}$$

Thus $\{B, E_1\}$ is coherent. Therefore, letting $B_1 = B \oplus E_1$, B_1 is coherent. B_1 is strictly robust since B is strictly robust and because we use $\oplus$.

Thus by induction on the second premise,

$$E_2 \text{ is coherent} \tag{A.59}$$

$$M_1\text{–strs}\, E_2 \subseteq \text{strs}\, B_1, \tag{A.60}$$

where $M_1 = M$ of $B_1 = M \cup \text{strnames}\, E_1$.

To show that $E_1 \pm E_2$ is coherent, let us consider any $S_1 \in \text{strs}\, E_1$ and any $S_2 \in \text{strs}\, E_2$ with n of $S_1 = n$ of $S_2 = n$, say, and let us show $S_1 = S_2$ — this is the only case of interest due to (A.57) and (A.59). By (A.60) we have $S_2 \in \text{strs}\, B_1 = \text{strs}(B \oplus E_1)$ so $S_1 = S_2$ by the coherence of B_1.

To show that $M\text{–strs}(E_1 \pm E_2) \subseteq \text{strs}\, B$ it suffices to consider an $S_2 \in (\text{strs}\, E_2 \setminus \text{strs}\, E_1) \cap M\text{–Str}$ — the other structures in E_1 are accounted for by (A.58). Now by (A.60) we have $S_2 \in \text{strs}(B \oplus E_1)$ and since $S_2 \notin \text{strs}\, E_1$ we must have $S_2 \in \text{strs}\, B$ as desired.

Generative structure expression, rule 8.5

$$\frac{B \vdash dec \Rightarrow E \quad n \notin \text{strnames}(E) \cup (M \text{ of } B)}{B \vdash \textbf{struct } dec \textbf{ end} \Rightarrow (n, E)}$$

By induction E is coherent and M–strs $E \subseteq$ strs B. Since $n \notin$ strnames E we have that S is coherent. Since $n \notin M$ we have M–strs $S = M$–strs $E \subseteq$ strs B as desired.

Long structure identifier, rule 8.6

$$\frac{B(longstrid\,) = S}{B \vdash longstrid \;\Rightarrow S}$$

Trivial.

Functor application, rule 9.2

$$\frac{\begin{array}{c} B(funid) > (S, (N')S') \quad B \vdash strexp \Rightarrow S \\ N' \cap (M \text{ of } B) = \emptyset \end{array}}{B \vdash funid(strexp) \Rightarrow S'}$$

By induction S is coherent and M–strs $S \subseteq$ strs B. If follows that

$$\{B, S\} \text{ is coherent.}$$

Let $\Phi = (N_0)(S_0, (N_0')S_0')$ be $B(funid)$. Since $\Phi > (S, (N')\,S')$ there exist a realisation φ with Supp $\varphi \subseteq$ boundstrs$(N_0)S_0$, $\varphi\, S_0 = S$ and $(N_0')S_0' \xrightarrow{\varphi} (N')S'$. Now φ is the identity on all structures free in $(N_0 \cup N_0')S_0'$. All structures in N_0–strs$(N_0')S_0'$ occur in S_0 since Φ is admissible. It then follows from $(N_0')S_0' \xrightarrow{\varphi} (N')S'$ that

$$\text{strs}(N')S' \subseteq \text{strs}\, S \cup \text{strs}(N_0 \cup N_0')\, S_0' \subseteq \text{strs}\{B, S\}. \tag{A.61}$$

As we saw above, $\{B, S\}$ is coherent so strs$(N')S'$ must be coherent. Moreover, $(N_0')S_0'$ is admissible and $(N_0')S_0' \xrightarrow{\varphi} (N')S'$, so (A.61) can be strengthened to the desired coherence of S'.

Next,

$$\begin{array}{rcll} M\text{–strs}\, S' & = & M\text{–strs}(N')S' & \text{as } M \cap N' = \emptyset \\ & \subseteq & M\text{–strs}\{B, S\} & \text{by (A.61)} \\ & \subseteq & \text{strs}\, B & \text{since } M\text{–strs}\, S \subseteq \text{strs}\, B. \end{array}$$

Top level declaration, rule 8.16

$$\frac{B \vdash dec \Rightarrow E}{B \vdash dec \Rightarrow (\text{strnames}\, E, \{\}, \{\}, E)}$$

Trivial inductive case.

Signature declaration, rule 8.17

$$\frac{B \vdash sigexp \Rightarrow \Sigma \quad \Sigma \text{ principal for } sigexp \text{ in } B}{B \vdash \textbf{signature}\ sigid = sigexp \Rightarrow (\text{strnames}\, \Sigma, \{\}, \{sigid \mapsto \Sigma\}, \{\})}$$

Since Σ is admissible, $B' = (\text{strnames}\, \Sigma, \{\}, \{sigid \mapsto \Sigma\}, \{\})$ is strictly robust. By lemma 10.9 we have $\text{strs}\, \Sigma \subseteq \text{strs}\, B$. Thus M–$\text{strs}\, B' \subseteq \text{strs}\, B$.

Functor declaration, rule 8.18

$$\frac{\begin{array}{c} B \vdash sigexp \Rightarrow (N)S \quad (N)S \text{ principal for } sigexp \text{ in } B \\ N \cap M \text{ of } B = \emptyset \\ B \oplus \{strid \mapsto S\} \vdash strexp \Rightarrow S' \\ N' = \text{strnames}(S') \setminus (N \cup (M \text{ of } B)) \quad \Phi = (N)(S, (N')S') \end{array}}{B \vdash \textbf{functor}\ funid(strid : sigexp) = strexp \Rightarrow (\text{strnames}\, \Phi, \{funid \mapsto \Phi\}, \{\}, \{\})}$$

By lemma 10.9 we have

$$\text{strs}(N)S \subseteq \text{strs}\, B\,. \tag{A.62}$$

Since $(N)S$ is admissible, S is coherent. Let

$$B_1 = B \oplus \{strid \mapsto S\}.$$

Then B_1 is strictly robust — the coherence because S is coherent and $N \cap M = \emptyset$.

Thus by induction, S' is coherent and

$$M_1\text{–strs}\, S' \subseteq \text{strs}\, B_1, \tag{A.63}$$

where now $M_1 = M \text{ of } B_1 = (M \text{ of } B) \cup N$.

Now let us show that $\Phi = (N)(S, (N')S')$ is admissible. We already know that $(N)S$ is admissible and that S' is coherent. It remains to be shown that

$$(N')S' \text{ is well-formed} \tag{A.64}$$

$$(N \cup N')\, S' \text{ is well-formed} \tag{A.65}$$

$$N\text{–strs}(N')S' \subseteq \text{strs}\, S\,. \tag{A.66}$$

Here (A.64) follows from (A.63). As to (A.65) assume $S_0 \in \text{strs}(N \cup N')\, S'$. Since n of $S_0 \notin N'$ we must have n of $S_0 \in M \cup N$ (by the definition of N') and since n of $S_0 \notin N$ we must have n of $S_0 \in M$. Thus by (A.63), $S_0 \in \text{strs}\, B_1$ and since n of $S_0 \notin N$ and (A.62) we must have $S_0 \in \text{strs}\, B$. Since B is strictly robust, $\text{strs}\, S_0 \subseteq \text{strs}\, B \subseteq M\text{–strs}$. But $M\text{–strs} \cap (N \cup N')\text{–Str} = \emptyset$, so $\text{strs}\, S_0 \subseteq \text{strs}(N \cup N')\, S'$ as desired.

As to (A.66), we have (A.63) which implies $\text{strs}(N')S' \subseteq \text{strs}\, B_1$. Thus $N\text{–strs}(N')S' \subseteq N\text{–strs}\, B_1 \subseteq \text{strs}\, S$, since $N \cap (M \text{ of } B) = \emptyset$.

Thus Φ is admissible.

We have $\text{strs}\, \Phi \subseteq \text{strs}\, B$ by (A.62) and the above argument that $\text{strs}(N \cup N')S' \subseteq \text{strs}\, B$. Thus B' is strictly robust, and $M\text{–strs}\, B' = \text{strs}\, \Phi \subseteq \text{strs}\, B$ as required.

Sequential program, rule 8.19

$$\frac{B \vdash \mathit{prog}_1 \Rightarrow B_1 \quad B \pm B_1 \vdash \mathit{prog}_2 \Rightarrow B_2}{B \vdash \mathit{prog}_1\ \mathit{prog}_2 \Rightarrow B_1 \pm B_2}$$

By induction

$$B_1 \text{ is strictly robust} \tag{A.67}$$

$$M\text{–strs}\, B_1 \subseteq \text{strs}\, B\,. \tag{A.68}$$

Thus $B \pm B_1$ is strictly robust — the coherence because B and B' are coherent, $\text{strnames}\, B \subseteq M \text{ of } B$, and (A.68).

Thus by induction,

$$B_2 \text{ is strictly robust} \tag{A.69}$$

$$M_1\text{–strs}\, B_2 \subseteq \text{strs}(B \pm B_1), \tag{A.70}$$

where $M_1 = M \text{ of } (B \pm B_1)$. Now by (A.67) and (A.69), $B_1 \pm B_2$ is well-formed and the M component of $(B_1 \pm B_2)$ includes all names free in the other components. Moreover, $B_1 \pm B_2$ is coherent by (A 67), (A.69), and (A.70). Thus $B_1 \pm B_2$ is strictly robust. Finally,

$$\begin{array}{rcll} M\text{–strs}(B_1 \pm B_2) & \subseteq & M\text{–strs}\, B_1 \cup M\text{–strs}\, B_2 & \\ & \subseteq & \text{strs}\, B \cup M\text{–strs}(B \pm B_1) & \text{by (A.68) and (A.70)} \\ & \subseteq & \text{strs}\, B & \text{by (A.68)} \end{array}$$

as required. ∎

Theorem 10.12 *If $B \vdash strexp \Rightarrow S$ and $B \xrightarrow[\text{srb}]{\varphi} B'$, then for all $S' \in \text{Str}$,*

$$B' \vdash strexp \Rightarrow S' \quad \textit{iff} \quad \text{Clos}_B\, S \xrightarrow{\varphi} \text{Clos}_{B'}\, S'.$$

Similarly for declarations.

Proof. [of 10.12] By induction on the depth of proof of $B \vdash strexp \Rightarrow S$ and $B \vdash dec \Rightarrow E$. One case for each rule.

Empty declaration, rule 8.2

$$\overline{B \vdash \quad\quad \Rightarrow \{\}}$$

Here $B' \vdash \quad\quad \Rightarrow E'$ iff $E' = \{\}$ iff $\text{Clos}_B\{\} \xrightarrow{\varphi} \text{Clos}_{B'}\, E'$.

Structure declaration, rule 8.3

$$\frac{B \vdash strexp \Rightarrow S}{B \vdash \textbf{structure}\ strid\ =\ strexp \Rightarrow \{strid \mapsto S\}}$$

Here

$$\begin{array}{ll} & B' \vdash \textbf{structure}\ strid\ =\ strexp \Rightarrow E' \\ \text{iff} & \exists\, S' \quad E' = \{strid \mapsto S'\} \quad \text{and} \quad B' \vdash strexp \Rightarrow S' \\ \text{iff} & \exists\, S' \quad E' = \{strid \mapsto S'\} \quad \text{and} \quad \text{Clos}_B\, S \xrightarrow{\varphi} \text{Clos}_{B'}\, S' \quad \text{by induction} \\ \text{iff} & \text{Clos}_B\{strid \mapsto S\} \xrightarrow{\varphi} \text{Clos}_{B'}\, E'. \end{array}$$

Sequential declaration, rule 8.4

$$\frac{B \vdash dec_1 \Rightarrow E_1 \quad B \oplus E_1 \vdash dec_2 \Rightarrow E_2}{B \vdash dec_1\ dec_2 \Rightarrow E_1 \pm E_2}$$

First assume that $B' \vdash dec_1\ dec_2 \Rightarrow E'_{12}$. Then $E'_{12} = E'_1 \pm E'_2$ for some E'_1, E'_2 satisfying

$$\frac{B' \vdash dec_1 \Rightarrow E'_1 \quad B' \oplus E'_1 \vdash dec_2 \Rightarrow E'_2}{B' \vdash dec_1\ dec_2 \Rightarrow E'_1 \pm E'_2}.$$

Write $B = M, F, G, E$ and $B' = M', F', G', E'$. Note that for any semantic object A we have $\text{Clos}_B A = \text{Clos}_M A$ and $\text{Clos}_{B'} A = \text{Clos}_{M'} A$, since B and B' are strictly robust.

By induction

$$\text{Clos}_B\, E_1 \xrightarrow{\varphi} \text{Clos}_{B'}\, E'_1\,. \tag{A.71}$$

Let $N_1 = \text{strnames}\, E_1 \setminus M$ and $N'_1 = \text{strnames}\, E'_1 \setminus M'$. Then by (A.71) there exists a bijection $\alpha_1 : N_1 \to N'_1$ such that

$$\text{Reg}\, \varphi^\circ_1 \cap N'_1 = \emptyset \text{ and } (\varphi^\circ_1 \,|\alpha_1)\, E_1 = E'_1, \tag{A.72}$$

where $\varphi^\circ_1 = \varphi \downarrow M\text{–strs}\, E_1$.

Let $B_1 = B \oplus E_1$ and $B'_1 = B' \oplus E'_1$ and let $M_1 = M \text{ of } B_1 = M \cup \text{strnames}\, E_1$ and $M'_1 = M \text{ of } B'_1 = M' \cup \text{strnames}\, E'_1$.

Let $\varphi^\circ = \varphi \downarrow \text{strs}\, B$ and let $\varphi_1 = \varphi^\circ\,|\alpha_1$. We have

$$\begin{aligned}
\varphi_1(E \pm E_1) &= \varphi_1\, E \pm \varphi_1\, E_1 \\
&= \varphi\, E \pm \varphi_1\, E_1 && \text{as } N_1 \cap \text{strs}\, E = \emptyset \\
&= E' \pm (\varphi^\circ\,|\alpha_1)\, E_1 \\
&= E' \pm (\varphi^\circ_1\,|\alpha_1)\, E_1 && \text{by theorem 10.8} \\
&= E' \pm E'_1 && \text{by (A.72).}
\end{aligned}$$

B_1 and B'_1 are strictly robust by corollary 10.10. Thus we have $B_1 \xrightarrow[\text{srb}]{\varphi_1} B'_1$, so by induction,

$$\text{Clos}_{B_1}\, E_2 \xrightarrow{\varphi_1} \text{Clos}_{B'_1}\, E'_2\,. \tag{A.73}$$

Let $N_2 = \text{strnames}\, E_2 \setminus M_1$ and $N'_2 = \text{strnames}\, E'_2 \setminus M'_1$. Then by (A.73) there exists a bijection $\alpha_2 : N_2 \to N'_2$ such that

$$\text{Reg}\, \varphi^\circ_2 \cap N'_2 = \emptyset \text{ and } (\varphi^\circ_2 \,|\alpha_2)\, E_2 = E'_2, \tag{A.74}$$

where $\varphi^\circ_2 = \varphi_1 \downarrow M_1\text{–strs}\, E_2$.

Now $N_1 \cap N_2 = N_1' \cap N_2' = \emptyset$, so

$$\alpha_1 \cup \alpha_2 : N_1 \cup N_2 \to N_1' \cup N_2'$$

is a bijection.

Let $N = \mathrm{strnames}(E_1 \pm E_2) \setminus M$ and $N' = \mathrm{strnames}(E_1' \pm E_2') \setminus M'$. Clearly, $N \subseteq N_1 \cup N_2$ and $N' \subseteq N_1' \cup N_2'$ but in general $N = N_1 \cup N_2$ and $N' = N_1' \cup N_2'$ need not hold.

We wish to prove

$$(N)(E_1 \pm E_2) \xrightarrow{\varphi} (N')(E_1' \pm E_2'). \tag{A.75}$$

To this end we shall first prove that

$$(\varphi^\circ \,|(\alpha_1 \cup \alpha_2))(E_1 \pm E_2) = E_1' \pm E_2'. \tag{A.76}$$

We have

$$\begin{aligned} (\varphi^\circ \,|\alpha_1 \cup \alpha_2)\, E_1 &= (\varphi^\circ \,|\alpha_1)\, E_1 && \text{as } N_2\text{–strs}\, E_1 = \emptyset \\ &= (\varphi_1^\circ \,|\alpha_1)\, E_1 && \text{by theorem 10.8} \\ &= E_1' && \text{by (A.72).} \end{aligned}$$

Moreover, by theorem 10.8, for every structure S_2 in E_2 either S_2 is free in B_1 or n of $S_2 \in N_2$. In the former case,

$$(\varphi^\circ \,|\alpha_1 \cup \alpha_2)\, S_2 = \varphi_1\, S_2 = \varphi_2^\circ\, S_2 = (\varphi_2^\circ \,|\alpha_2)\, S_2\,.$$

In the latter case, writing $S_2 = (n_2, E_2'')$,

$$\begin{aligned} (\varphi^\circ \,|\alpha_1 \cup \alpha_2)\, S_2 &= (\alpha_2 n_2, (\varphi^\circ \,|\alpha_1 \cup \alpha_2)\, E_2'') \\ &= (\alpha_2 n_2, (\varphi_2^\circ \,|\alpha_2)\, E_2'') \\ &= (\varphi_2^\circ \,|\alpha_2)(n_2, E_2'') \\ &= (\varphi_2^\circ \,|\alpha_2)\, S_2 \end{aligned}$$

by an easy structural induction.

Thus

$$\begin{aligned} (\varphi^\circ \,|\alpha_1 \cup \alpha_2)\, E_2 &= (\varphi_2^\circ |\alpha_2)\, E_2 \\ &= E_2' \quad \text{by (A.74)} \end{aligned}$$

which with the above equation, $(\varphi^\circ \,|\alpha_1 \cup \alpha_2)\, E_1 = E_1'$, gives (A.76).

Now let $\alpha = (\alpha_1 \cup \alpha_2) \downarrow N$. Let us show that the co-domain of α is N'.

Assume $n \in \text{strnames}(E_1 \pm E_2) \cap N$. Then by (A.76), $\alpha(n)$ occurs in $E'_1 \pm E'_2$ and since $M' \cap (N'_1 \cup N'_2) = \emptyset$ we have $\alpha(n) \in N'$.

Conversely, if $n' \in \text{strnames}(E'_1 \pm E'_2) \setminus M'$ then there must be an $n \in \text{strnames}(E_1 \pm E_2) \setminus M$ with $\alpha(n) = n'$, the reason being that we have (A.76) and that if a structure, S, is free in $(N)(E_1 \pm E_2)$ then it is free in B (by theorem 10.8) and thus $\varphi_0 S$ is free in $(N')(E'_1 \pm E'_2)$ — as $B \xrightarrow[\text{srb}]{\varphi} B'$.

Thus the co-domain of α is N'.

Let $\varphi^\circ_{12} = \varphi \downarrow \text{strs}(N)(E_1 \pm E_2)$. Since by theorem 10.8 every structure in $E_1 \pm E_2$ is either free in B or is an N structure we have

$$\begin{aligned} (\varphi^\circ_{12} \,|\alpha)(E_1 \pm E_2) &= (\varphi^\circ \,|\alpha_1 \cup \alpha_2)(E_1 \pm E_2) \\ &= E'_1 \pm E'_2 \qquad \text{by (A.76)} \end{aligned}$$

showing (A.75).

Conversely, assume

$$\text{Clos}_B(E_1 \pm E_2) \xrightarrow{\varphi} \text{Clos}_{B'} E'_{12}\,.$$

Again, write $B = M, F, G, E$ and $B' = M', F', G', E'$.

Let $N_{12} = \text{strnames}(E_1 \pm E_2) \setminus M$ and $N'_{12} = \text{strnames}\, E'_{12} \setminus M'$. Thus there exists a bijection $\alpha_{12} : N_{12} \to N'_{12}$ such that

$$\text{Reg}\, \varphi^\circ_{12} \cap N'_{12} = \emptyset \text{ and } (\varphi^\circ_{12} \,|\alpha_{12})(E_1 \pm E_2) = E'_{12}. \tag{A.77}$$

where $\varphi^\circ_{12} = \varphi \downarrow M\text{–strs}(E_1 \pm E_2)$.

Now extend α_{12} to a bijection, $\alpha_{1\cup 2}$, whose domain is

$$\text{Dom}\, \alpha \cup \text{strnames}\, E_1 \setminus (M \cup \text{strnames}(E_1 \pm E_2))$$

i.e., all names that are new in E_1 or in E_2, and whose co-domain does not intersect $M' \cup N'_{12}$.

Let $\varphi^\circ = \varphi \downarrow \text{strs}\, B$ and let us define

$$E'_1 = (\varphi^\circ \,|\alpha_{1\cup 2})\, E_1 \tag{A.78}$$

$$E'_2 = (\varphi^\circ \,|\alpha_{1\cup 2})\, E_2\,. \tag{A.79}$$

Now

$$E'_{12} = (\varphi^\circ_{12} \,|\alpha_{12})(E_1 \pm E_2) \qquad \text{by (A.77)}$$

$$\begin{aligned} &= (\varphi^\circ \,|\alpha_{1\cup 2})(E_1 \pm E_2) && \text{by theorem 10.8} \\ &= (\varphi^\circ \,|\alpha_{1\cup 2})\, E_1 \pm (\varphi^\circ \,|\alpha_{1\cup 2})\, E_2 \\ &= E_1' \pm E_2'. \end{aligned}$$

To use induction a first time we wish to prove

$$(N_1)E_1 \xrightarrow{\varphi} (N_1')E_1', \tag{A.80}$$

where $N_1 = \text{strnames}\, E_1 \setminus M$ and $N_1' = \text{strnames}\, E_1' \setminus M'$.

Let $\alpha_1 = \alpha_{1\cup 2} \downarrow \text{strnames}\, E_1$. Then α_1 is a bijection of N_1 on N_1' by (A.78). Moreover, letting $\varphi_1^\circ = \varphi \downarrow \text{strs}(N_1)\, E_1$ we have that φ_1° is a restriction of φ° by theorem 10.8 and thus $\text{Reg}\, \varphi_1^\circ \cap N_1' = \emptyset$. Surely, $(\varphi_1^\circ \,|\alpha_1)\, E_1 = (\varphi^\circ \,|\alpha_{1\cup 2})\, E_1 = E_1'$, by (A.78), showing (A.80).

Thus by induction we have

$$B' \vdash dec_1 \Rightarrow E_1'. \tag{A.81}$$

Let $B_1 = B \oplus E_1$ and $B_1' = B' \oplus E_1'$ and $M_1 = M \text{ of } B_1$ and $M_1' = M \text{ of } B_1'$. By theorem 10.8, B_1 and B_1' are strictly robust.

Let $\varphi_1 = \varphi^\circ \,|\alpha_1$. Then we have $B_1 \xrightarrow[\text{srb}]{\varphi_1} B_1'$ by (A.78).

To use induction a second time we wish to prove

$$(N_2)\, E_2 \xrightarrow{\varphi_1} (N_2')\, E_2', \tag{A.82}$$

where $N_2 = \text{strnames}\, E_2 \setminus M_1$ and $N_2' = \text{strnames}\, E_2' \setminus M_1'$.

Let $\alpha_2 = \alpha_{1\cup 2} \downarrow \text{strnames}\, E_2$. Then α_2 is a bijection of N_2 on N_2' by (A.79). Moreover, letting $\varphi_2^\circ = \varphi_1 \downarrow \text{strs}(N_2)\, E_2$ we have that φ_2° is a restriction of $\varphi_1 \downarrow \text{strs}\, B_1$ and thus, since $B_1 \xrightarrow{\varphi_1} B_1'$, $\text{Rng}\, \varphi_2^\circ \cap N_2' = \emptyset$. Also,

$$\begin{aligned} (\varphi_2^\circ \,|\alpha_2)\, E_2 &= ((\varphi_1 \downarrow \text{strs}(N_2)\, E_2) \,|\, \alpha_2)\, E_2 \\ &= (\varphi^\circ \,|\alpha_1 \cup \alpha_2)\, E_2 && \text{since every structure free in } (N_2)\, E_2 \text{ is free in } B_1 \\ &= (\varphi^\circ \,|\alpha_{1\cup 2})\, E_2 \\ &= E_2' && \text{by (A.79)} \end{aligned}$$

showing (A.82).

Thus by induction we have $B_1' \vdash dec_2 \Rightarrow E_2'$ which with (A.81) and the fact that $E_{12}' = E_1' \pm E_2'$ gives the desired $B' \vdash dec_1\ dec_2 \Rightarrow E_{12}'$.

Generative structure expression, rule 8.5

$$\frac{B \vdash dec \Rightarrow E \quad n \notin \text{strnames}(E) \cup (M \text{ of } B)}{B \vdash \textbf{struct } dec \textbf{ end} \Rightarrow (n, E)}$$

Assume $B \vdash dec \Rightarrow E$ and $n \notin \text{strnames}\, E \cup M$. Then

$$B' \vdash \textbf{struct } dec \textbf{ end} \Rightarrow S'$$

iff

there exists n', E', such that $S' = (n', E')$ and $B' \vdash dec \Rightarrow E'$ and $n' \notin \text{strnames}\, E' \cup M'$

iff (by induction)

the exists n', E', such that $S' = (n', E')$ and $\text{Clos}_B\, E \xrightarrow{\varphi} \text{Clos}_{B'}\, E'$ and $n' \notin \text{strnames}\, E' \cup M'$

iff

$$\text{Clos}_B(n, E) \xrightarrow{\varphi} \text{Clos}_{B'}\, S'.$$

Long structure identifier, rule 8.6

$$\frac{B(longstrid) = S}{B \vdash longstrid \Rightarrow S}$$

Let $E = E \text{ of } B$. Then

$$B' \vdash longstrid \Rightarrow S'$$

iff

$$(E \text{ of } B')longstrid = S'$$

iff

$$(\varphi\, E)longstrid = S'$$

iff

$$\varphi(E\, longstrid) = S' \quad \text{(given that } E\, longstrid \text{ exists)}$$

iff

$$\varphi\, S = S'$$

iff

$$\text{Clos}_B\, S \xrightarrow{\varphi} \text{Clos}_{B'}\, S'$$

as B and B' strictly robust implies $\text{Clos}_B\, S = (\emptyset)\, S$ and $\text{Clos}_{B'}\, S' = (\emptyset)\, S'$.

Functor application, rule 9.2

$$\frac{B(\mathit{funid}) > (S_0, (N)S) \quad B \vdash \mathit{strexp} \Rightarrow S_0 \quad N \cap (M \,\mathrm{of}\, B) = \emptyset}{B \vdash \mathit{funid}(\mathit{strexp}) \Rightarrow S}$$

As usual, write $B = M, F, G, E$ and $B' = M', F', G', E'$.

Assume $B' \vdash \mathit{funid}(\mathit{strexp}) \Rightarrow S'$. Then for some N', S'_0,

$$\frac{B'(\mathit{funid}) > (S'_0, (N')S') \quad B' \vdash \mathit{strexp} \Rightarrow S'_0 \quad N' \cap M' = \emptyset}{B \vdash \mathit{funid}(\mathit{strexp}) \Rightarrow S'}$$

By induction $\mathrm{Clos}_B\, S_0 \xrightarrow{\varphi} \mathrm{Clos}_{B'}\, S'_0$, so letting $N_0 = \mathrm{strnames}\, S_0 \setminus M$ and $N'_0 = \mathrm{strnames}\, S'_0 \setminus M'$ there exists a bijection $\alpha : N_0 \to N'_0$ such that

$$\mathrm{Reg}\, \psi_0 \cap N'_0 = \emptyset \text{ and } (\psi_0 \,|\alpha)\, S_0 = S'_0, \tag{A.83}$$

where $\psi_0 = \varphi \downarrow M\text{–strs}\, S_0$.

Let $\varphi_0 = \varphi \downarrow \mathrm{strs}\, B$ and let $\varphi_1 = \varphi_0 \,|\alpha$. We have $B \xrightarrow[\mathrm{srb}]{\varphi_1} B'$. There exists an $(N'')S''$ such that

$$(S_0, (N)S) \xrightarrow{\varphi_1} (S'_0, (N'')S'') \tag{A.84}$$

since $\varphi_1\, S_0 = S'_0$. Since $B \xrightarrow[\mathrm{srb}]{\varphi_1} B'$ we have

$$B(\mathit{funid}) \xrightarrow{\varphi_1} B'(\mathit{funid}). \tag{A.85}$$

Using lemma 10.4 on (A.84), (A.85) and $B(\mathit{funid}) > (S_0, (N)S)$ gives

$$B'(\mathit{funid}) > (S'_0, (N'')S'').$$

Since we also have $B'(\mathit{funid}) > (S'_0, (N')S')$ we must have $(N'')S'' \underset{\alpha}{=} (N')S'$ by lemma 9.18. Thus by (A.84),

$$(N)S \xrightarrow{\varphi_1} (N')S'. \tag{A.86}$$

We wish to prove

$$(N_1)S \xrightarrow{\varphi} (N'_1)S', \tag{A.87}$$

where $N_1 = \mathrm{strnames}\, S \setminus M$ and $N'_1 = \mathrm{strnames}\, S' \setminus M'$. Since $N \cap M = N' \cap M' = \emptyset$, we have $N \subseteq N_1$ and $N' \subseteq N'_1$ but the equalities need not hold as $(N)S$ may contain free structures that are in S_0 but not free in B. More precisely, since $F(\mathit{funid})$ is well-formed and coherent and $F(\mathit{funid}) > (S_0, (N)S)$, every

structure free in $(N)S$ *either* is free in B *or* it occurs free in S_0 and is not an M structure. By (A.86) and (A.83), structures of the latter kind are renamed by α conflicting neither with N' nor (by the coherence of $(N')S'$) with names in $\mathrm{Reg}(\varphi \downarrow \mathrm{strs}(N_1)\,S)$. Hence from (A.86) we get (A.87) as required.

Conversely, assume $\mathrm{Clos}_B\, S \xrightarrow{\varphi} \mathrm{Clos}_{B'}\, S'$.

Let $N_1 = \mathrm{strnames}\, S \setminus M$, $N_1' = \mathrm{strnames}\, S' \setminus M'$, and $\varphi_1^\circ = \varphi \downarrow M\text{–strs}\, S$. Then there exists a bijection $\alpha_1 : N_1 \to N_1'$ such that

$$\mathrm{Reg}\,\varphi_1^\circ \cap N_1' = \emptyset \text{ and } (\varphi_1^\circ \,|\alpha_1)\, S = S'. \qquad \text{(A.88)}$$

Let $N_0 = \mathrm{strnames}\, S_0 \setminus M$ and let us refer to the N_0 structures of S_0 as the *new* structure in S_0. Not all new structures in S_0 need be free in $(N)S$ but those which are free, are bound in $(N_1)S$ and thus susceptible to renaming by α_1. Extend the bijection $\alpha_1 \downarrow (\mathrm{strnames}(N)S \cap N_0)$ to a bijection

$$\alpha_0 : N_0 \to N_0',$$

making sure that $\alpha_0(n_0) \notin N_1' \cup M'$ for all $n_0 \in N_0 \setminus \mathrm{strnames}(N)S$. Let $\varphi_0^\circ = \varphi \downarrow M\text{–strs}\, S_0$. Then $\mathrm{Reg}\,\varphi_0^\circ \cap N_0' = \emptyset$ (because of the way we chose α_0). Let $S_0' = (\varphi_0^\circ \,|\alpha_0)\, S_0$. Then $(N_0)\, S_0 \xrightarrow{\varphi} (N_0')\, S_0'$. Thus by induction $B' \vdash \mathit{strexp} \Rightarrow S_0'$.

Let $\varphi^\circ = \varphi \downarrow \mathrm{strs}\, B$ and let $\varphi_1 = \varphi^\circ \,|\alpha_0$. Then $\varphi_1\, S_0 = S_0'$ and $(N)S \xrightarrow{\varphi_1} (N')S'$ for $N' = \alpha_1(N_1)$. Thus $(S_0, (N)S) \xrightarrow{\varphi_1} (S_0', (N')S')$.

Since also $B(\mathit{funid}) \xrightarrow{\varphi_1} B'(\mathit{funid})$ and $B(\mathit{funid}) > (S_0, (N)S)$ we have $B'(\mathit{funid}) > (S_0', (N')S')$ by lemma 10.4. Also $N' \cap M' \subseteq N_1' \cap M' = \emptyset$. Therefore, by the functor application rule we have the desired $B' \vdash \mathit{funid}(\mathit{strexp}) \Rightarrow S'$. ∎

Bibliography

[1] P. Aczel, *An Introduction to Inductive Definitions*, in J. Barwise (ed.) *Handbook of Mathematical Logic*, North–Holland Publishing Company, 1977

[2] G. M. Birtwistle, O. Dahl, B. Myhrhaug, K. Nygaard, *SIMULA BEGIN*, Studentlitteratur, Lund, Sweden, 1976

[3] H. Aït–Kaci, *An Algebraic Semantics Approach to the Effective Resolution of Type Equations*, Theoretical Computer Science 45 (3) (1986) 293–351

[4] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Nauer (ed.), A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden and M. Woodger, *Revised report on the algorithmic language ALGOL 60*, Comm. ACM 6, 1 (1963, January), 1–17; Computer Journal 5, 349–367; Num. Math. 2, 106–136.

[5] H. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam 1981.

[6] R. Burstall and B. Lampson, *A kernel language for abstract data types and module*, in Symposium on Semantics of Data Types, Sophia Antipolis, Springer, Lecture Notes in Computer Science 173, 1984, 1–50.

[7] L. Cardelli. *ML under Unix*. Polymorphism, Vol. 1,Number 3, December, 1983.

[8] L. Cardelli, *A Semantics of Multiple Inheritance*, Semantics of Datatypes, International Symposium, Lecture Notes in Computer Science 173, (1984), 51–68

[9] D. Clément, J. Despeyroux, T. Despeyroux, L. Hascoet, G. Kahn *Natural Semantics in the Computer*, Technical report RR 416, INRIA, Sophia–Antipolis, France, June 1985

[10] M. Coppo, M. Dezani-Ciancaglini and P. Sallé, *Functional characterization of some semantic equalities inside λ-calculus*, E. Maurer, ed., in: *Automata, Languages and Programming*, Lecture Notes in Computer Science 71 (Springer, Berlin, 1979) pp. 133–146.

[11] M. Coppo, M. Dezani-Ciancaglini and B. Venneri, *Principal type schemes and λ-calculus semantics*, in J.P. Seldin and J.R. Hindley, eds., *To H.B. Curry. Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, London, 1980 pp. 535–560

[12] M. Coppo, M. Dezani-Ciancaglini an B. Venneri, *Functional characters of solvable terms*, Zeit. Math. Logik Grund. 27 (1981) 45–58.

[13] L. Damas, *Type Assignment in Programming Languages*, Ph. D. Thesis, University of Edinburgh, Department of Computer Science, CST-33-85, 1985.

[14] L. Damas and R. Milner, *Principal type schemes for functional programs*, in Proceedings of the 9th ACM Symposium on the Principles of Programming Languages, pp. 207–212, 1982

[15] M. Gordon, R. Milner and C. Wadsworth, *Edinburgh LCF*, Springer, Lecture Notes in Computer Science 78, Berlin – Heidelberg – New York, 1979

[16] R. Harper, *Introduction to Standard ML*, LFCS Report Series, ECS-LFCS-86-14, Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, Scotland.

[17] R. Harper, F. Honsell, G. Plotkin, *A Framework for Defining Logics*, Proceedings of the Symposion on Logic in Computer Science, Ithaca, New York, June 1987, 194–204

[18] R. Harper, David MacQueen and R Milner, *Standard ML*, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, ECS–LFCS–86–2, 1986

[19] R. Harper, R. Milner and M. Tofte, *A type discipline for program modules*, Proc. TAPSOFT '87, Springer, Lecture Notes in Computer Science 250, Berlin – Heidelberg – New York, 1987, 308–319.

[20] R. Harper, R. Milner and M. Tofte, *The semantics of Standard ML, Version 1*, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, ECS–LFCS–87–36, 1987

[21] R Harper and M. Tofte. *The Static Semantics of Standard ML, DRAFT*. Unpublished. 1 November, 1985.

[22] R. Hindley, *The pincipal type scheme of an object in combinatory logic*, Trans. Amer. Math. Soc. 146, 1969, 29–60.

[23] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*, The MIT Press, Cambridge, Mass., 1986

[24] D. MacQueen *Modules for Standard ML (Draft)*, Polymorphism, Vol 1, Number 3, December 83.

[25] D. B. MacQueen *Modules for Standard ML*, in [18]

[26] J. McCarthy *et al. LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, Mass. 1965

[27] R. Milner, *A theory of type polymorphism in programming languages*, Journal of Computer and System Sciences, 17 : 348–375 (1978)

[28] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, ed., G. Goos and J. Hartmanis, Springer, Berlin, 1980.

[29] R. Milner: *Webs.* (Working note). 1 September 85.

[30] R. Milner: *Static Semantics of Modules.* (Working note). May 86.

[31] R. Milner: *Static Semantics of Modules, Mark 2.* (Working note). May 24, 1986.

[32] R. Milner: *Static Semantics of Modules, Mark 3.* (Working note). June 22, 1986

[33] R. Milner: *Static Semantics of Modules, Mark 4.* (Working note). July 20, 1986

[34] G. Plotkin, *A Structural Approach to Operational Semantics*, Technical Report DAIMI–FN–19, Computer Science Department, Aarhus University, Denmark, 1981

[35] J.C. Reynolds, *Towards a Theory of Type Structure*, Proc. Colloque sur la Programmation, Lecture Notes in Computer Science 19, Springer, New York, 1974, pp. 408–425.

[36] J.C. Reynolds, *Three Approaches to Type Scructure*, Mathmatical Foundations of Software Development, ed., H. Ehrig, Proceedings, TAPSOFT, Lecture Notes in Computer Science 185, Springer, New York, 1985, pp. 97–138

[37] J. Robinson *A machine-oriented logic based on the resolution principle*, Journal of the ACM 12, 1, (1965),23–41.

[38] S. Ronchi Della Rocca and B. Venneri, *Principal type schemes for an extended type theory*, Theoretical Computer Science 28, North-Holland (1984) 151–169.

[39] D. Sannella, *A Denotational Semantics for ML modules.* Edinburgh University. (Unpublished). Jan 1985.

[40] M. Tofte: *A Theory of Realisation Maps, Mark 1.* (Working note). 31 June, 86.

[41] *Reference Manual for the Ada Programming Language*, (Proposed Standard Document), United States Department of Defence, July 1980

[42] M. Wand, *Complete Type Inference for Simple Objects*, Symposium on Logic in Computer Science, Ithaca, New York, Proceedings, 1987, 37–44.

[43] Å. Wikström, *Functional Programming Using Standard ML*, Prentice Hall, 1987

[44] N. Wirth, *The programming language PASCAL*, Acta Informatica 1 (1971), 35–63