Operational Semantics and Polymorphic Type Inference

Mads Tofte

Ph. D.
University of Edinburgh
1988

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.

Acknowledgements. First and foremost thanks to Robin Milner. Not only did he give supervision on the highest technical level, he also did it in the most pleasant way. Thanks to Bob Harper and David MacQueen for good discussions about polymorphic references and modules. It is thanks to David MacQueen and his ideas about polymorphic references that I made another attempt to solve that problem after Kevin Mitchell had discovered that my first type inference system was unsound. Thanks also to Kevin Mitchell for giving constructive criticism on the draft of this thesis, and to Moira Norrie for lending me her office.

Thanks to my parents, John and Birgitte Tofte, for their continuous support, and to all friends and relatives in Denmark for letters and visits. Thanks also to everybody in Edinburgh who kindly distracted me from work: Helen, Barbara, Duncan, Jalid, Dipti, John, Ai Jun, Kim, Merete, Mads, Charlotte, Sid, Daniela, June, and Joan.

My stay in Edinburgh would have been impossible without the kandidat-stipendium from Datalogisk Institut, Københavns Universitet.

This is the revised version. The comments and corrections put forward by my examinors, Michael Gordon and Don Sannella, have been most helpful.

Declaration. I hereby declare that this thesis has been composed by myself, that the work reported has not been presented for any university degree before, and that ideas and results that I do not attribute to others are due to myself.

Mads Tofte

Contents

III A Module Language 79

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

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

can be read: " if you can prove that the expression e_1 has type $\tau' \to \tau$ 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

$$\texttt{let I = } \lambda \texttt{ x.x in (I 3, I true)} \tag{1.1}$$

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

$$\texttt{(}\lambda \texttt{ I. (I 3, I true))(}\lambda \texttt{ x.x)} \tag{1.2}$$

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 $\forall a.a \to a$ which in the two applications can be instantiated to $int \to int$ and $bool \to bool$, 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 $(int \to int) \cap (bool \to bool)$ 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
	$\lambda x.e_1$	lambda abstraction
	$e_1\ e_2$	application
	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

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

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

$$A \xrightarrow{\text{fin}} B.$$

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 $f \in A \xrightarrow{\text{fin}} B$ can be written in the form

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

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

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

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 mathmatical 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".

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

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

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

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

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

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

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

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

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

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*.

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

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

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

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:

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

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 $\text{tyvars}\,\sigma = \emptyset$ and $\text{tyvars}\,TE = \emptyset$ and τ is said to be a *monotype* if $\text{tyvars}(\tau) = \emptyset$.

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

where $S_0 \stackrel{\mathrm{def}}{=} S \downarrow \mathrm{tyvars}\,\sigma_1$. 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]

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

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

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

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

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

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

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

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

$$\frac{\dfrac{\dfrac{TE_2 \vdash \mathtt{y} : int \to int \quad TE_2 \vdash \mathtt{x} : int}{TE_2 \vdash \mathtt{y}\ \mathtt{x} : int}}{TE_1 \vdash \lambda \mathtt{y}.\mathtt{y}\ \mathtt{x} : (int \to int) \to int}}{TE_0 \vdash \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}\ \mathtt{x} : int \to ((int \to int) \to int)}$$

[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

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

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:

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

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

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

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

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 $TE_1 = TE_0 \pm \{\texttt{x} \mapsto (\mathrm{t}'\ \mathit{list})\ \mathit{list}\}$ and $TE_2 = TE_1 \pm \{\texttt{g} \mapsto \mathrm{t}' \to \mathrm{t}\}$. Thus t and t′ are free in TE_2. Now the inference for the same expression becomes

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

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

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

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

Note that if we substitute *int* for t′ and *int* → *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

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

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 $\forall \mathrm{t}_1\mathrm{t}_2.\mathrm{t}_1 \to \mathrm{t}_2 \to \mathrm{t}_1$)

let `f=` λ`g.g(hd(rev(hd x)))`
in

$\cdots$ (`f first`) `7`$\cdots$
$\cdots$ (`f first`) `true`$\cdots$
$\cdots$ (`f cons`) `nil`$\cdots$

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`:

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

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,

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

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

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

since $\mathrm{Reg}\, S_0 \cap \{\beta_1, \ldots, \beta_n\} = \emptyset$. 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.

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

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

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

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

No β_i is free in TE' (since $TE \xrightarrow{S} TE'$ and $\beta_i \neq \mathrm{Reg}(S_0)$). Moreover, any type variable that occurs in $S'\,\tau_1$ and is not a β_i 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

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

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 and the first premise of (2.24) we have

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

Moreover, we have

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

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 λx.x can get type t → t, *int* → *int*, and *bool* → *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.

$$
\begin{array}{lll}
W(TE, e) = \textbf{case } e \textbf{ of} & & \\
x & \Rightarrow \textbf{if } x \notin \mathrm{Dom}\, TE \textbf{ then fail} & \\
& \quad \textbf{else let } \forall\alpha_1 \ldots \alpha_n.\tau = TE(x) & \\
& \qquad\qquad \beta_1, \ldots, \beta_n \text{ be new} & \\
& \qquad \textbf{in } (ID, \{\alpha_i \mapsto \beta_i\}\tau) & \\
\lambda x.e_1 & \Rightarrow \textbf{let } \alpha \text{ be a new type variable} & \\
& \qquad\quad (S_1, \tau_1) = W(TE \pm\{x \mapsto \alpha\}, e_1) & \\
& \quad \textbf{in } (S_1, S_1(\alpha) \to \tau_1) & \\
e_1\ e_2 & \Rightarrow \textbf{let } (S_1, \tau_1) = W(TE, e_1); & \\
& \qquad\quad (S_2, \tau_2) = W(S_1(TE), e_2) & \\
& \qquad\quad \alpha \text{ be a new type variable} & \\
& \qquad\quad S_3 = \mathit{Unify}(S_2(\tau_1), \tau_2 \to \alpha) & \\
& \quad \textbf{in } (S_3S_2S_1, S_3(\alpha)) & \\
\textbf{let } x = e_1 \textbf{ in } e_2 & \Rightarrow & \\
& \quad \textbf{let } (S_1, \tau_1) = W(TE, e_1) & \\
& \qquad\quad (S_2, \tau_2) = W(S_1\, TE \pm\{x \mapsto \mathrm{Clos}_{S_1\, TE}\tau_1\}, e_2) & \\
& \quad \textbf{in } (S_2S_1, \tau_2) &
\end{array}
$$

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* $\mu = \mu_1 \to \mu_2$, *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

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

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 $r_1 \neq$ *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

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

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 $\models r : \tau$.

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

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

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* $\rightarrow$ 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 functions 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
	$\lambda x.e_1$	lambda abstraction
	$e_1\ e_2$	application
	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.

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

Figure 3.1: Objects in the Dynamic Semantics

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

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

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

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

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

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

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

Figure 3.2: Dynamic Semantics

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

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

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

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

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

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

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

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

using the instantiations

$$\forall \mathrm{t}.((\mathrm{t} \to \mathrm{t})\ ref) \quad > \quad (int \to int)\ ref$$

and

$$\forall \mathrm{t}.((\mathrm{t} \to \mathrm{t})\ ref) \quad > \quad (bool \to bool)\ ref$$

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

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

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 $s(\mathtt{a})$ 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:

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

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

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

(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

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

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

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

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

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

[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:

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

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

and type environments are defined by

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

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:

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

A *substitution* is now a map $S : \mathrm{TyVar} \to \mathrm{Type}$ 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.

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

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

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

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

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

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

Figure 4.1: The Type Inference Rules for Imperative Types

Throughout, we shall require

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

(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

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

and functions

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

We shall write $e_1 :: e_2$ 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

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

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 = λx.!(!x)`
in **while** `double-deref(ref(ref false))` **do**
`double-deref(ref(ref 5))`

Syntax

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

Dynamic Semantics

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

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

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

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

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

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

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

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

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

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

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

the let rule for expansive expressions, rule 4.5, will prohibit generalization from $(\mathrm{u} \to \mathrm{u})\ ref$ to $\forall \mathrm{u}.((\mathrm{u} \to \mathrm{u}))\ ref)$. 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 $TE \vdash e_1 : \mathrm{u}\ list \to \mathrm{u}\ list$ where the body of the second let expression will be typed under the assumptions

$$TE(\texttt{data}) = \mathrm{u}\ list\ ref$$
$$TE(\texttt{result}) = \mathrm{u}\ list\ ref$$

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 $\mathrm{u}\ list \to \mathrm{u}\ list$ to $\forall \mathrm{u}.\mathrm{u}\ list \to \mathrm{u}\ list$.

As one would expect, since fast_reverse has type $\forall \mathrm{u}.\mathrm{u}\ list \to \mathrm{u}\ list$ while the applicative reverse function has type $\forall \mathrm{t}.\mathrm{t}\ list \to \mathrm{t}\ list$, 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`).

$e_1 =$ 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 $TE \vdash e_1 : (u_1 \rightarrow u_2 \rightarrow u_2) \rightarrow u_2 \rightarrow u_1$ *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 $\forall$u.u *list* $\rightarrow$ 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=` λ`l.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 $\forall$t.t $\rightarrow$ t $\rightarrow$ t; then `sham_id` gets the type $\forall$t.t *list* $\rightarrow$ 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 $\forall$u.u $\rightarrow$ u $\rightarrow$ u and `sham_id` gets type u *list* $\rightarrow$ 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 $\vdash \lambda\mathtt{x}.\mathtt{x} : \mathrm{t} \to \mathrm{t}$ and also $\vdash \lambda\mathtt{x}.\mathtt{x} : \mathrm{u} \to \mathrm{u}$.

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

$W_1(TE, e) =$ **case** e **of**
$x \quad \Rightarrow$ **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 such that
α_i is applicative iff β_i is applicative
in $(ID, \{\alpha_i \mapsto \beta_i\}\tau)$
$\lambda x.e_1 \Rightarrow$ **let** t be a new applicative type variable
$(S_1, \tau_1) = W_1(TE \pm \{x \mapsto t\}, e_1)$
in $(S_1, S_1(t) \to \tau_1)$
$e_1\ e_2 \Rightarrow$ **let** $(S_1, \tau_1) = W_1(TE, e_1)$;
$(S_2, \tau_2) = W_1(S_1(TE), e_2)$
t be a new applicative type variable
$S_3 = \mathit{Unify}_1(S_2(\tau_1), \tau_2 \to t)$
in $(S_3S_2S_1, S_3(t))$
let x = e_1 **in** $e_2 \Rightarrow$
let $(S_1, \tau_1) = W_1(TE, e_1)$
$\sigma =$ if e_1 is non-expansive then $\mathrm{Clos}_{S_1\,TE}\tau_1$
else $\mathrm{AppClos}_{S_1\,TE}\tau_1$
$(S_2, \tau_2) = W_1(S_1\,TE \pm \{x \mapsto \sigma\}, e_2)$
in (S_2S_1, τ_2)

Figure 4.4: A Type Checker for the New Type Discipline

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

$$\mathit{Unify}_1(\alpha, \tau) = \begin{cases} \{\alpha \mapsto \tau\}, & \text{if } \alpha \text{ is applicative;} \\ \{\alpha \mapsto S(\tau)\} \cup S, & \text{if } \alpha \text{ is imperative} \end{cases}$$

provided α does not occur in τ, where $\{\alpha_1, \ldots, \alpha_n\}$ is $\mathrm{apptyvars}\,\tau$ 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 $S : \mathrm{TyVar} \to \mathrm{Type}$ such that $S(u) \in \mathrm{ImpType}$ for all $u \in \mathrm{ImpTyVar}$. 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 $\{\alpha_i \mapsto \beta_i \mid i \leq i \leq n\}$ is a bijection and α_i is imperative iff β_i is imperative, and no β_i is in $\mathrm{Reg}(S_0)$, and*
2. *$(S_0 \,|\{\alpha_i \mapsto \beta_i\})\tau_1 = \tau_2$*

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

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

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

Let $\sigma_1 = \mathrm{Clos}_{TE}\tau_1 = \forall\alpha_1 \ldots \alpha_n.\tau_1$, let $S_1 = S \downarrow \mathrm{tyvars}\,TE$ and let $S_0 = S \downarrow \mathrm{tyvars}\,\sigma_1$. 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 $\mathrm{Reg}\,S_1$. 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'. 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,

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

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

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

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

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

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

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

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

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

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

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

Since $TE \xrightarrow{S} TE'$ no β_i is free in TE'. 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* $\mathrm{tyvars}\,\sigma \subseteq A$. *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 $\mathrm{Reg}\,S_0$ and $(S_0\,|\{\alpha_i \mapsto \beta_i\})\tau = \tau'$, where $S_0 = S \downarrow \mathrm{tyvars}\,\sigma$.

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 $\mathrm{Rng}\,R \cap (A \cup \mathrm{Reg}(S \downarrow A)) = \emptyset$.

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 $\{\delta_1, \ldots, \delta_k\} \cap \mathrm{Reg}(S \downarrow A) = \emptyset$.

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

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

where $(*)$ is because $\{\delta_1, \ldots, \delta_k\} \cap \mathrm{Reg}(S \downarrow A) = \emptyset$. 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 $\mathrm{tyvars}\, \tau^{(i)} \subseteq \mathrm{Dom}\, R$ and $\mathrm{Rng}\, R \cap A = \emptyset$. 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:

$$\begin{array}{l} ST \quad \in \mathrm{StoreTyping} = \mathrm{Addr} \xrightarrow{\mathrm{fin}} \mathrm{ImpType} \\ s : ST \in \mathrm{TypedStore} = \{(s, ST) \in \mathrm{Store} \times \mathrm{StoreTyping} \mid \mathrm{Dom}\, s = \mathrm{Dom}\, ST\} \end{array}$$

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

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 $s : ST \models s(a) : ST(a)$

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

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

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 i^{th} 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 $(s : ST, s(a), ST(a)) \in Q_1\}$

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

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

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:

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

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

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

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

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

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

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

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 $(s : ST, 3, int) \in F(\emptyset)$ 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 $(s : ST, 3, int) \in F(U)$ but $(s : ST, 3, bool) \notin F(U)$.

Using that F is monotonic we have

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

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

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

with the corresponding typing rule

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

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

$$s_1 : ST_1 \models \mathit{ref} : \tau' \to \tau.$$

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

$$s : ST \models v : \sigma \implies s' : ST' \models v : \sigma$$

and

$$s : ST \models E : TE \implies s' : ST' \models E : TE\,.$$

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

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

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 $s : ST \subseteq s' : ST'$ 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 $\mathrm{Dom}\, E = \mathrm{Dom}\, TE$. Moreover, for all $x \in \mathrm{Dom}\, E$, 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

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

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

$E_0(\texttt{ref})$	=	*ref*	$TE_0(\texttt{ref})$	=	$\forall u.u \to u\ ref$
$E_0(\texttt{!})$	=	*deref*	$TE_0(\texttt{!})$	=	$\forall t.t\ ref \to t\ ref$
$E_0(\texttt{:=})$	=	*asg*	$TE_0(\texttt{:=})$	=	$\forall t.t\ ref \to t \to stm$

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

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

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

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

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

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

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

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

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 $(done, stm) \in R_{\text{Bas}}$ 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

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

Now there are two subcases:

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

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

and

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

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

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

So take any $\tau < \mathrm{AppClos}_{TE}\tau_1$. Any bound variable in $\mathrm{AppClos}_{TE}\tau_1$ is applicative, so it does not occur in ST_1, simply because store typings by definition cannot contain applicative type variables. Thus $\tau < \mathrm{AppClos}_{TE}\tau_1$ 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 $\tau < \mathrm{AppClos}_{TE}\tau_1$ we have proved (5.35), c.f. Property 5.4.

Then (5.34) and (5.35) give

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

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

$$\tau \ ::= \ \iota \mid \alpha \mid \tau \ ref \mid \tau_1 \to \tau_2$$

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

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

where Δ ranges over finite sets of types. Thus

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

is a type scheme. The quantification binds over both $\tau_1 \to \tau_2$ and Δ. The set Δ should be thought of as an addendum to the whole of $\tau_1 \to \tau_2$, 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

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

and also

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

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

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

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

Hence, if η has the form $\tau_1 \to \tau_2 * \Delta'$ 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

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

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

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

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

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

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 $TE_0 \vdash e : (\mathrm{t} \to \mathrm{t} * \{\mathrm{t}\}) * \emptyset$, where

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

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,

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

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

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

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

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

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

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

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

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

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

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

Figure 6.1: Damas' Inference Rules

6.2 Comparison

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

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

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 tyvars $\Delta \cap$ tyvars $\hat{\tau}$ corresponds to imptyvars τ. 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.

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