Type Assignment in Programming Languages

Luis Manuel Martins Damas

Doctor of Philosophy

University of Edinburgh

1984

Abstract

The purpose of this work is to present and study a family of polymorphic type disciplines for programming languages similar to the type discipline of ML, the metalanguage of the LCF system, which are based on the use of type inference systems to define the notion of well typed expressions and programs and on the use of type assignment algorithms to compute the type or types that can be inferred for those same expressions or programs.

Previous work on the theoretical foundations of the ML type discipline is reexamined and completed here. It is also extended in two directions, namely to handle overloading of identifiers and also to cope with a semantics involving references to a store as first class objects.

For each of the theories studied here we present proofs of the semantic soundness of type inference, i.e. that well typed expressions evaluate to objects of the correct type and that in particular they do not lead to run-time errors like trying to add an integer to a list.

Algorithms for computing the type or types which can be inferred for expressions are also presented together with proofs of the soundness and completeness of the algorithms, i.e. that the algorithms compute exactly the types which can be actually inferred for the expressions.

Acknowledgments

I would like to thank my supervisor, Robin Milner, for having introduced me to the subject of this work, for the helpful discussions we had and for the encouraging interest he showed during its development.

I am also grateful to Michael Gordon for many helpful discussions and sugestions on quantified types and on the aspects of type assignment related to references.

Finally I thank the Portuguese Instituto Nacional de Investigacao Cientifica for their financial support.

STATEMENT

The work described in this thesis is the author's own, except where otherwise indicated, and has not been submitted in whole or part for another degree at this or any other university.

Luis Manuel Martins Damas

TABLE OF CONTENTS

INTRODUCTION

Introduction

The advantages gained by imposing some kind of type discipline on programs (and, particularly, on programmers...) have been recognized since the earliest days of programming language design. As a matter of fact most of the more successful programming languages since then have included such a discipline, e.g. Fortran, Algol 60, PL/I, Algol 68, SIMULA, PASCAL, C, etc. . Nevertheless the kind of strong typechecking which is a common feature of all those languages becomes sometimes too restrictive, particularly after the introduction of constructs which allow programmers to define new types to model data structures. This kind of restriction is well illustrated by the following example from Algol 68. Consider the two following Algol 68 *mode* (the Algol 68 equivalent to *type* [Wijngaarden *et al* 75]) definitions

```
        struct IntList = (int hd, Ref IntList  tl);
        struct RealList = (Real hd, Ref RealList tl);
```

which are intended to model lists of integers and lists of reals. Now if, in a particular application, we need to reverse such lists, we would have to define two functions, one to act on lists of integers and the other to act on lists of reals, even though the text of the definitions of those functions would differ only by the mode declarations for the argument and result of the function. This restrictions may explain, in part, why those languages had a very small impact on areas such as Artificial Intelligence where programs usually involve a relatively large number of different, but in many cases similar, data types which tend to change often during program development.

Apart from the obvious solution of allowing programmers to defeat the type discipline, one way of overcoming the limitations exemplified above is to allow parametrization on types. This is in fact the approach followed in recent languages like Russel [Demers & Donahue 79], CLU [Liskov & Snyder 77] and Alphard [Wulf *et al* 76] which allow types to be passed to procedures as arguments. However it should be pointed out that this is done at the cost of increasing the complexity of both the language syntax and semantics and, even if the semantics of such languages is now on solid theoretical grounds due to the works of [Reynolds 74], [McCracken 79] and others, the fact remains that, when compared with type free languages, this sometimes leads to more complicated programs which require an extra programming effort. This is particularly true when type parameters are only used to satisfy the typechecking constraints and do not play an active semantical role, in contrast with those situations where they are dynamically tested to achieve a type dependent semantics like in, e.g., a function capable of printing objects of different types.

A completely different aproach to the problem we have been discussing has been introduced with ML, the metalanguage of the LCF system [Gordon *et al* 79]. Rather than introducing types in the semantics, ML manages to combine the advantages of a strict type discipline with the conciseness of a type free language, by exploring the fact that the semantics of functions like the identity function

$$\lambda x.x$$

is independent (in a type free semantics) of the types of their arguments. Such objects are *type polymorphic* in the sense that they belong to or possess more than one type. The approach followed in ML essentially explores the fact that, since many of the primitives of a programming language are naturally polymorphic, then many of the functions one can define using those primitives will also be type polymorphic. In fact it is possible to derive the type(s) of a function from its definition. As a consequence of this ML does not require programmers to mention types at all although, for pragmatic reasons, it allows them to do so.

To illustrate how type inference is achieved consider a simple declaration like

$$\text{let } compose\ f\ g = \lambda x.f(g(x))$$

which defines function composition. It is obvious that if the expression

$$f(g(x))$$

is to be well typed then the following assumptions about the types of f, g and x

$$f:\alpha\rightarrow\beta$$

$g:\gamma\to\alpha$

$x:\gamma$

must hold for some types α, β and γ, in which case the type of the expression will be β. It follows then that, whatever the types α, β and γ, are, *compose* has type

$$(\alpha\to\beta)\to(\gamma\to\alpha)\to(\gamma\to\beta).$$

We will express this by saying that we can infer the *type scheme*

$$\forall\alpha\forall\beta\forall\gamma.(\alpha\to\beta)\to(\gamma\to\alpha)\to(\gamma\to\beta)$$

for *compose*.

To a great extent, that is while we do not consider the facilities provided by ML for introducing new types and new type operators, ML can be seen as an untyped language in which the type of an object is a property which can be derived from its definition. As a matter of fact what the ML interpreter does is to refuse to evaluate programs when it is unable to derive a type for them. Thus types play only a passive role in the semantics of the language in the sense that they are used, as a filter, to restrict the set of acceptable programs to those for which a type can be derived. The main advantage gained by this filtering arises from the fact that it can be proved that well typed programs do not lead to error failures like trying to apply a non-functional value to an argument or trying to add an integer to a list.

For the sake of the reader unfamiliar with ML we will present a few programming examples which show that polymorphism and type inference are not restricted to simple cases like the one above. First of all we note that types in ML are built from primitive types like bool and int, and type variables α, β ..., using type operators like → for functional types, + for disjoint sums(unions), × for cartesian product, and other operators like postfixed list for lists.

Using the following primitives on lists

$null : \forall\alpha.\alpha\ \mathrm{list} \to \mathrm{bool}$

$nil : \forall\alpha.\alpha\ \mathrm{list}$

$hd : \forall\alpha.\alpha\ \mathrm{list} \to \alpha$

$tl : \forall\alpha.\alpha\ \mathrm{list} \to \alpha\ \mathrm{list}$

$cons : \forall\alpha.\alpha \to \alpha\ \mathrm{list} \to \alpha\ \mathrm{list}$

with the obvious meanings, we can write the following definition of a function for concatenating two lists

```
letrec conc l l' =
   if null(l) then l'
   else cons (hd l) (conc (tl l) l')
```

(where letrec is used to introduce a recursive definition) from which the ML type assignment algorithm would infer the type scheme

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

for *conc*.

Similarly, the function *map* which maps a given function over a given list, i.e. such that

$$map\ f\ [x_1;\ldots;x_n] = [f(x_1);\ldots;f(x_n)]$$

can be defined in ML by

```
letrec map f l =
           if null l then nil
           else cons(f(hd l))(map f (tl l))
```

In this case the type scheme inferred for *map* by the ML type checker will be

$$\forall\alpha\forall\beta.((\alpha\to\beta) \to (\alpha\ \mathrm{list} \to \beta\ \mathrm{list})).$$

As a final example involving lists we will consider a polymorphic function for sorting a given list. This function will take as arguments a predicate *p* defining the order relation and a list *l*.

```
letrec sort p l =
 letrec insert x l =
    if null l then cons x nil
    if p (hd l) x then cons x l
    else cons (hd l) (insert x (tl l))
 in
   if null l then nil
   else insert (hd l) (sort p (tl l)).
```

As it could be expected the type scheme inferred for *sort* will be

$$\forall\alpha.(\alpha \to \alpha \to \mathrm{bool}) \to \alpha\ \mathrm{list} \to \alpha\ \mathrm{list}.$$

Finally, to conclude this digression into the ML type discipline, we will show how it provides for the introduction of new types and type operators in terms of older ones. This declaration is in fact accompanied by the introduction of a set of functions which operate on values of the new type and is such that the representation of those values is only accessible to those functions. As an example we will consider here the definition of binary trees whose tips are labelled by objects of an arbitrary type. This can be done in ML as follows

```
absrectype α bitree = α + (α bitree × α bitree)
```

with *sons* = *outr o repbitree*

and *tip* = *outl o repbitree*

and *maketree* = *absbitree o inr*

and *tiptree* = *absbitree o inl*

and *nulltree* = *isl o repbitree*

In the above definition *o* is a binary operator denoting function composition similar to the function *compose* defined above, *outl* and *outr* are projections from an union type into each of the summand types, and *inl* and *inr* are the associated injections. More importantly *repbitree* and *absbitree* are the functions which map a binary tree into its representation and vice versa and thus have type schemes

$$repbitree\colon \forall\alpha.\alpha \text{ bitree} \to (\alpha + \alpha \text{ bitree} \times \alpha \text{ bitree})$$

$$absbitree\colon \forall\alpha.(\alpha + \alpha \text{ bitree} \times \alpha \text{ bitree}) \to \alpha \text{ bitree}.$$

Note that although in current ML implementations the maps *absbitree* and *repbitree* behave, from a semantical point of view, just like the identity function they could be used to implement a more efficient representation, in terms of storage required, of the data type (e.g. by packing and unpacking). Furthermore they are only available when defining the basic operations on the data type but not in the scope of the data type definition thus providing a way of encapsulating the representation chosen for the data type, hence the designation of abstract data type.

Now, even the introduction of abstract data types does not change much the point of view mentioned above of ML being seen as an untyped language in which type assignment is used as a filter. This is so because, from a semantical point of view, we can, in a declaration like the one above, ignore the part concerning types and treat the declarations of *nulltree*, *tip* ..., as any other function definitions. Furthermore, even from a point of view of type assignment, if we ignore questions of the scope of the new type operator, we can handle the above declaration by assigning the types above to *absbitree* and *repbitree*, and then assigning types to *nulltree*, *tip* ..., in the usual way.

Having acquainted those readers unfamiliar with ML, with the main features of its type discipline we should also point out that ML, even if it provides assignable variables, is essentially a purely applicative language, and its lack of updatable structures like, e.g. arrays, does limit its acceptability as a general purpose programming language.

The theoretical basis for the ML type discipline was introduced in [Milner 78] where a type assignment algorithm was defined and type assignment was shown to be semantically sound.

The main aim of this work is to complete Milner's work and to extend it in two directions. On~~e~~ one hand we will present here the proofs of the results anounced in [Damas & Milner 82] stating the existence of principal type schemes and the completeness of the ML type assignment algorithm. On the other hand we will extend that theory to handle overloading of identifiers and to a semantics including references to an updatable store as first class objects.

We will now present the criteria that, in our opinion, a theory of type assignment for a programming language should satisfy if it is to be of any practical use.

First of all it is desirable from a pragmatic point of view that the type discipline should be stated in some simple form. In accordance with previous works on type assignment we will use inference systems to specify what expressions are well typed and what types can be inferred for them. For an algebraic approach to type assignment see [Shultis 82].

Secondly the type system should be semantically sound in the sense that if a type can be inferred for an expression than the result of evaluating the expression should be of that type. An example of an application where such strict view about soundness of the type discipline is essential, is provided by the LCF system itself where one of the primitive types is type "theorem" and one wants to ensure that objects of that type can only be produced using primitive functions modelling axioms and inference rules.

Thirdly, for any practical purpose, it is essential to have some form of algorithm to infer types for expressions. Moreover the type inferred by the algorithm should be as general as any other type which could be inferred for the expression because otherwise it could fail to come up with a type which the user was entitled to expect from the inference rules. This also means that the algorithm should only fail if no type could be inferred for the expression.

Finally the type discipline achieved should be powerful enough to overcome the limitations of strict type disciplines exemplified at the very beginning of this introduction.

So far we have not made any special reference to the programming language to which apply such theories. For the sake of conciseness we will study, in this work, type assignment for a simple extension of the λ-calculus obtained by the introduction of a declarative construct of the form

let x=e in e'

denoting the result of evaluating e' whith x denoting the value of e.

Since the above construct is essentially equivalent (although not necessarily identical) to the β-redex

$(\lambda x.e')\,e$

thus not playing any new semantical role, it is worthwhile to try to justify its inclusion.

To that end it is convenient to mention other previous works on type assignment outside computer science, namely, Functionality Theory in Combinatory Logic [Curry&Feys·58] and in particular the works of [Curry 69] and [Hindley 69] on the existence of principal type schemes of an object of combinatory logic, and the subsequent extension of this result by [Yelles 79] to terms of the λ-calculus.

A common feature of all the works mentioned above is that they all study the set of types which can be inferred for the terms of some simple formal language using some system of inference rules. Amongst the properties of such systems one usually finds some sort of semantic soundness of type inference, the existence of most general or principal types amongst the types that can be inferred for a term and, in certain cases, the existence of algorithms to compute them.

However none of the works mentioned above, except for the one by Milner, provides a type assignment algorithm powerful enough to overcome the limitations imposed by strict type disciplines mentioned at the very beginning of this introduction.

The reason for the success of Milner's work can be found on the fact that, instead of restricting himself to the λ-calculus, he studies type assignment for an extension of the λ-calculus including a declarative contruct of the form described above, and to the fact that his type inference system is such that that construct might have a type even when the β-redex

$$(\lambda x.e')\ e$$

(which from a semantical point of view is essentially equivalent) has not.

An explanation for the above asymmetry can be traced to the fact that, when x is used polymorphically in e', the abstraction $\lambda x.e'$ might fail to have a type at all.

Although one could formulate Milner's theory using only the λ-calculus by typing β-redexes the way the let-construct is typed, it seems natural, when dealing with programming languages, specially when a non purely applicative semantics is involved, to include some form of declarative construct.

In relation with the above posed problem of typing β-redexes it is worthwhile to refer to two different approaches that were studied for the λ-calculus. The first one consists in postulating the invariance of type inference under β-reduction, i.e. to say that the types which can be inferred for the β-redex

$$(\lambda x.e')\ e$$

are all those which can be inferred for

$$[e/x]e'.$$

The system thus obtained has been studied extensively in logic and it has been proved [Yelles 79] that no type assignment algorithm can be found for it as the problem of deciding whether a type can be inferred for a term is, for the system in question, only semidecidable.

A different approach, which also achieves invariance of type inference under β-reduction, was studied in [Coppo *et al* 80] and is based on extending the notion of functional type to allow $\lambda x.e$ to be well typed even when x is used polymorphically in e. In that system, functional types of the form

$$[\tau_1,\dots,\tau_n] \rightarrow \tau$$

are introduced to model the type of a function which returns an object of type τ when given an argument of, simultaneously, types τ_1, ..., τ_n. However the authors of that work have also shown that what was said above about the undecidability of type assignmemt also applies to this system which detracts from its usefulness in practical applications on computer science.

To conclude this discussion we note on passing that we will show in this work that if we define a let-reduction, analogous to β-reduction, for Milner's extension of the λ-calculus, then type inference is preserved by let-reduction.

We can now outline the remainder of this work in which we present three theories which meet the criteria mentioned above.

In chapter I we study an inference system for inferring types for expressions. This system overcomes the limitation of at most one assumption for each variable of the system presented in [Damas & Milner 82] while preserving the existence of principal types and of a type assignment algorithm. From a practical point of view the importance of the results of that chapter is that they provide a basis for handling overloading of identifiers such as arithmetic operators or the equality operator =.

In chapter II we study the type scheme inference system of [Damas & Milner 82] and, besides presenting the proofs of the results stated in that work, we also study its relation with the inference system of chapter I.

In chapter III we extend the theory of chapter II to the case where the language semantics is no longer purely applicative but includes references to a store as first class objects. We will also present some programming examples showing how familiar data structures like arrays and records with updatable fields can be adequately handled with this extension to the ML type discipline.

CHAPTER I

CHAPTER I

A type inference system for an applicative language

1. Introduction

In this chapter we study a type inference system for a purely applicative programming language.

The programming language in question, which is only intended to illustrate the main features of type inference, is the λ-calculus to which a simple declarative construct is added.

The type inference system is essentially that of basic functionality theory in combinatory logic and λ-calculus [Curry & Feys 58]. However, apart from the extension neccessary to handle the extra declarative construct, we place an emphasis on the use of multiple assumptions about the type(s) of a free variable which is not present in basic functionality theory. The reason for this emphasis is twofold. First, it enables declarations to be type polymorphic. Secondly, it provides a basis for handling overloading of identifiers in programming languages.

Besides proving the soundness of type inference, i.e., that if a type can be derived for an expression the result of evaluating the expression is of the correct type, we also give an algorithm for deciding, when given an expression, if there is any type at all which can be inferred for the expression, and in the affirmative case computing a most general or principal type among those that can be derived for the expression. The existence of such an algorithm is in fact the main advantage, from a computer science point of view, of our system over the more permissive system of Coppo *et al* [Coppo *et al* 80] which has already been described in the introduction to this work.

The existence of a principal type for expressions for which some type can be derived, is, perhaps, the most interesting result of the chapter. This is more so because, by allowing multiple assumptions about the same variable, we are ensuring, in general, the non-existence of a principal or most general type (the precise meaning will be given later) amongst those types which can be derived for an expression from a particular set of assumptions. In this sense, when restricted to λ-terms, our existence theorem generalizes the one of Curry and of Hindley [Hindley 69] in combinatory logic and similar results for the λ-calculus [Yelles 79], by removing the restriction of only one assumption about each free variable.

In order to be able to give a computable characterization of the set of all the types which can be inferred for an expression from a particular set of assumptions we introduce type schemes. Type schemes are terms of the form $\forall\alpha_1\ldots\alpha_n\tau$ where τ is a type and the α_i are type variables, which are used to represent all the types which can be obtained by substituting any other types for the α_i in τ. It is then shown that, for any set of assumptions described by a finite number of type schemes, the set of types which can be derived for an expression from those assumptions can also be described by a finite set of type schemes.

Finally we discuss briefly how the type system can be used to handle overloading of identifiers in programming languages.

2. Expressions

The syntax of *expressions* is described by the following ambiguous BNF grammar

$$e ::= x \mid ee' \mid \lambda x.e \mid \text{let } x{=}e \text{ in } e'$$

where x ranges over a given set of *identifiers Ide*. We will use parentheses where necessary to avoid ambiguity.

Note that ee' denotes function application and $\lambda x.e$ functional abstraction.

Many concepts of the λ-calculus, e.g., free variable, substitution of an expression for a variable, etc., can be extended to expressions by treating an expression of the form

let x=e in e'

as if it was the β-redex

$(\lambda x.e')e$.

We also introduce a reduction rule red_{let} which acts on let-subexpressions as β-reduction does, i.e.,

let x=e in e' red_{let} $\{e/x\}e'$

where $\{e/x\}$ denotes substitution of e for x.

Given an expression e if we successively let-reduce one of the inner let-subexpressions, we obtain, in as many steps as the number of subexpressions of the let-form in e, an expression without any subexpressions of that form in it, i.e., a λ-term (one could prove that any sequence of let-reductions terminates in a finite number of steps with that λ-term). We will refer to the λ-term as the let-*free form* of e.

Finally we note that many usual constructs such as constants, conditionals, recursion, etc., which are not present in our simple language, can be replaced, at least as far as type inference is concerned, by identifiers which are assumed to have appropriate values bounded to them, e.g.,

if $e_1 e_2 e_3$

instead of

if e_1 then e_2 else e_3.

Similarly one could use an identifier *fix* bounded to the fixed point operator to allow the definition of recursive functions.

3. Types

Assuming we are given a set *Tv* of *type variables* α and a set *Pt* of *primitive types* ι, the syntax of *types* is given by

$$\tau ::= \alpha \mid \iota \mid \tau\to\tau'.$$

Again we will use parentheses, where necessary, to avoid ambiguity.

We will denote the set of all the types by *Ty*.

As with expressions, we kept types to an essential minimum. However we could introduce other type constructors like sums, products, lists, etc., without affecting the proof of all the results in this chapter.

A *substitution of types for type variables S* is a map from type variables to types.

For given type variables $\alpha_1,\ldots,\alpha_n$ and types $\tau_1,\ldots,\tau_n$

$$[\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n]$$

denotes the substitution which maps each of the α_i to the corresponding τ_i and which maps any other type variable into itself. We will often shorten the above notation to

$$[\tau_i/\alpha_i].$$

A substitution *S* extends naturally to a map from types to types by replacing (simultaneously) each occurrence of a variable α in a type with $S\alpha$(the same can be done when instead of types we have any larger syntactic class involving types). The *composition* of two substitutions *R* and *S* is defined by

$$(SR)\alpha = S(R\alpha)$$

It is easily realized that the composition of substitutions is associative.

We will say that a substitution S is *idempotent* iff $S\,S = S$.

For any substitution S we define

$$\mathrm{dom}\ S =_{\mathrm{def}} \{\ \alpha \mid S\alpha \neq \alpha\ \}$$

$$\mathrm{ran}\ S =_{\mathrm{def}} \{\ \alpha \mid \exists\beta \in \mathrm{dom}\ S \text{ s.t. } \alpha \text{ occurs in } S\beta\ \}$$

$$\mathrm{inv}\ S =_{\mathrm{def}} \mathrm{dom}\ S \cup \mathrm{ran}\ S.$$

Although somewhat inappropriately, we will refer to the above sets as the *domain*, *range* and the set of variables *involved* in S.

If S is a substitution and A is a set of type variables $S_{|A}$ will denote the *restriction* of S to A defined by

$$S_{|A}\alpha =_{\mathrm{def}} \begin{cases} S\alpha & \text{if } \alpha \in A \\ \alpha & \text{otherwise.} \end{cases}$$

If R and S are substitutions such that for any $\alpha \in \mathrm{dom}\ R \cap \mathrm{dom}\ S$, $R\alpha = S\alpha$, we define the *simultaneous composition* of R and S, $R+S$ by

$$(R+S)\alpha =_{\mathrm{def}} \begin{cases} R\alpha & \text{if } \alpha \in \mathrm{dom}\ R \\ S\alpha & \text{otherwise.} \end{cases}$$

Given a type τ if S and R are substitutions such that

$$S\tau = R\tau$$

then S and R must act on the type variables occurring in τ in the same way, i.e., if A is the set of type variables occurring in τ then $S_{|A}$ and $R_{|A}$ must be the same. Now assume we are given types τ and τ'. Then if there is any substitution S such that $S\tau = \tau'$ then there is also one with minimal domain among those which satisfy that equality. We will call that substitution the *minimal* substitution such that $S\tau = \tau'$.

The above discussion about minimal substitutions still applies when instead of just one equation we have any finite number of equations where a substitution S occurs applied to types or other larger syntactic structures involving types.

Finally a type τ' is said to be an *instance* of a type τ iff there is a substitution S such that $\tau' = S\tau$. If τ and τ' are instances of each other then we will say that τ' is a *trivial variant* of τ. Note that τ' is a trivial variant of τ iff there are distinct type variables $\beta_1, \ldots, \beta_n$ such that

$$\tau' = [\beta_i/\alpha_i]\ \tau$$

where the α_i are all the type variables which occur in τ.

4. Semantics

Since our semantic domains will be complete partial orders we will recall briefly the definitions of complete partial order and of other related notions.

A *complete partial order(cpo) D* is a partial order with a least or *bottom* element $\perp_D$ and such that every ascending ω-chain in D has a least upper bound(lub).

A map between cpos is *continuous* iff it is monotonic and also preserves lubs of ω-chains.

For any set S and cpo D the set $S \to D$ of all maps from S to D endowed with the extension order

$$f \sqsubseteq g \quad \leftrightarrow_{\mathrm{def}} \forall s \varepsilon S \quad fs \sqsubseteq gs$$

is itself a cpo. In the case where S is a cpo D' we will use $D' \to D$ to denote the cpo of all continuous maps from D' to D.

If S is a set then $S_\perp$ denotes the cpo obtained by taking the discrete order in S and adding a bottom element $\perp$. An example is the cpo of truth values T defined as $\{true, false\}_\perp$.

If $D_1, \ldots, D_n$ are cpos then the *coalesced sum* $D_1 + \ldots + D_n$ is defined by taking the disjoint union of $D_1, \ldots, D_n$ with the induced order and then identifying the bottom elements of $D_1, \ldots, D_n$. If a cpo D is a summand of another cpo V then

(i) if $d \varepsilon D$ then "d in V" denotes the image of d by the natural injection of D into V.

(ii) $isD : V \to T$ is defined by

$$isD(v) = \begin{cases} true & \text{if } v = d \text{ in } V \text{ for some } d \varepsilon D \\ \perp_T & \text{if } v = \perp_V \\ false & \text{otherwhise} \end{cases}$$

(iii) if $v \varepsilon V$ then $v_{|D}$ is defined by

$$v_{|D} = \begin{cases} d \text{ if } v = d \text{ in } V & \text{for some } d \varepsilon D \\ \perp_D & \text{otherwhise} \end{cases}$$

Given $t \in T$ and $v, v' \in V$ we will use

$$t \rightarrow v,\ v'$$

to denote the value defined by

$$t \rightarrow v, v' = \begin{cases} \perp_V & \text{if } t = \perp_T \\ v & \text{if } t = \mathit{true} \\ v' & \text{if } t = \mathit{false} \end{cases}$$

Starting with a given domain B of *basic values* we define the domains of *values* V, of *functions* F and of the *error value* W, by the following domain equations

$$V = B + F + W$$

$$F = V \rightarrow V$$

$$W = \{\circ\}_{\perp}$$

where $\{\circ\}$ is a set with only one element.

Note that the existence of a solution to the above equations follows from well known results (see, e.g., [Plotkin 76]).

We also define the domain of *environments* *Env* by

$$Env = Ide \rightarrow V$$

and we will use ρ to range over *Env*. Given an environment ρ, a value v and an identifier x we define the environment $\rho[v/x]$ by

$$(\rho[v/x])[\![y]\!] = y{=}x \rightarrow v,\ \rho[\![y]\!]$$

where, as it is usual in denotational semantics, we use the decorated brackets $[\![\ \ ]\!]$ to indicate syntactic arguments of a map.

Let *Exp* denote the set of all expressions, then we define a semantic function

$$E{:}Exp \rightarrow Env \rightarrow V$$

by the following equations where *wrong* denotes "∘ in *V*"

$$E[\![x]\!]\rho = \rho[\![x]\!]$$

$$E[\![e_1 e_2]\!]\rho = isF(v_1) \rightarrow (v_{1|F})v_2,\ wrong$$

$$\text{where } v_i \text{ is } E[\![e_i]\!]\rho \text{ (i=1,2)}$$

$$E[\![\lambda x.e]\!]\rho = (\lambda v.E[\![e]\!]\rho[v/x]) \text{ in } V$$

$$E[\![\text{let } x{=}e_1 \text{ in } e_2]\!]\rho = E[\![e_2]\!]\ \rho[E[\![e_1]\!]\rho/x]$$

The semantics above is an extension of the formal semantics of the λ-calculus defined in [Stoy 77]. Since one obviously has

$$E[\![\text{let } x{=}e_1 \text{ in } e_2]\!]\rho = E[\![(\lambda x.e_2)e_1]\!]\rho$$

it follows that the results of [Stoy 77] stating that the value denoted by a λ-term is not altered by any of the conversion rules, still holds for expressions and for let-conversion.

Comparing our semantics with the one defined by [Milner 78] for a similar language we see that the latter is more strict in its treatment of non-termination and of error values and more near actual implementations of applicative languages. We will refer to it as the *strict* semantics for expressions and we define it by replacing two of the equations defining *E* with

$$E[\![e_1 e_2]\!]\rho = isF(v_1) \rightarrow$$

$$v_2{=}wrong \rightarrow wrong,\ (v_{1|F})v_2,$$

$$wrong$$

$$\text{where } v_i = E[\![e_i]\!]\rho \text{ (i=1,2)}$$

$$\begin{aligned} E[\![\text{let } x{=}e_1 \text{ in } e_2]\!]\rho = \\ &\text{let } v_1 = E[\![e_1]\!]\rho \text{ in} \\ &\quad isW(v_1) \rightarrow v_1,\ E[\![e_2]\!]\rho[v_1/x] \end{aligned}$$

We now turn to the definition of a semantic for types and start by recalling the notion of ideal of a cpo.

A non-empty subset I of a cpo V is an *ideal* iff

(i) I is downwards closed, i.e., if $v \in I$ and $v' \sqsubseteq v$ then v' is also in I.

(ii) I is closed under lubs of ω-chains, i.e., the lub of any ω-chain in I is also in I.

Ideals of the domain of values proved very satisfactory as models for types [Shamir & Wadge 77].

Let $\bar{V}$ be the set of all ideals of V which do not contain *wrong*.

Clearly every intersection of ideals in $\bar{V}$ is also in $\bar{V}$ and in fact $\bar{V}$ becomes a complete lattice when ordered by

$$I \sqsubseteq I' \Leftrightarrow_{\text{def}} I' \text{ is a subset of } I.$$

For any I and I' in $\bar{V}$ the set

$$I \rightarrow I' =_{\text{def}} \{\, v \in V \mid isF(v) \text{ and } \forall v' \in I \ \ (v_{|F})v' \in I' \,\}$$

is also in $\bar{V}$.

A *valuation* of type variables is a map ψ which assigns an element of $\bar{V}$ to each type variable.

Let *Tval* be the domain of valuations, i.e,

$$Tval = Tv \rightarrow \overline{V}$$

and assume we are given a map

$$\Xi : Pt \rightarrow \overline{V}$$

which gives a meaning to each primitive type. We then define a semantic function for types

$$\mathrm{T} : Ty \rightarrow Tval \rightarrow \overline{V}$$

by

$$\mathrm{T}[\![\iota]\!]\psi = \Xi[\![\iota]\!]$$

$$\mathrm{T}[\![\alpha]\!]\psi = \psi[\![\alpha]\!]$$

$$\mathrm{T}[\![\tau\rightarrow\tau']\!]\psi = (\mathrm{T}[\![\tau]\!]\psi)\rightarrow(\mathrm{T}[\![\tau']\!]\psi).$$

Using T we can define, for each valuation ψ, a relation $v :_{\psi} \tau$ between a value v and a type τ, by

$$v :_{\psi} \tau \Leftrightarrow_{\mathrm{def}} v \varepsilon \mathrm{T}[\![\tau]\!]\psi.$$

Given a set A of assumptions of the form $x:\tau$ where x is an identifier and τ is a type, we can extend the above relation to a relation between environments and sets of assumptions by

$$\rho :_{\psi} A \Leftrightarrow_{\mathrm{def}} \forall\ x:\tau\ \varepsilon\ A\quad \rho[\![x]\!] :_{\psi} \tau.$$

Finally we can give a semantic meaning to assertions of the form $A \models e:\tau$, stating that if the assumptions A about the environment hold then e yields a value of type τ, by

$$A \models e:\tau \Leftrightarrow_{\mathrm{def}} \forall\psi\varepsilon Tval\ \forall\rho\varepsilon Env\quad \rho :_{\psi} A \Rightarrow E[\![e]\!]\rho :_{\psi} \tau.$$

A similar relation $A \overset{\mathrm{S}}{\models} e:\tau$ can be defined by using the strict semantics for expressions.

An alternative way of defining the semantic relation $\models$, avoiding valuations, can be found in [Milner 78] and in

[Damas & Milner 82] where a semantics is defined for monotypes(i.e. types without type variables in it) only. Then, using that semantics, $A\models e:\tau$ is defined, as above, for monotypes. Finally $\models$ is extended to the general case by requiring that it holds for each monotyped instance $A'\models e:\tau'$ of $A\models e:\tau$. However the approach followed here, using valuations of type variables, is more general since it can be easily extended to cope with other type constructs like recursively defined types.

5. Type inference

Assuming A is a set of *assumptions* as in the previous section we say that we can infer the type τ for e from A, and write $A\vdash e:\tau$, iff this can be derived from the following inference rules where A_x denotes the result of excluding from A any assumptions about x:

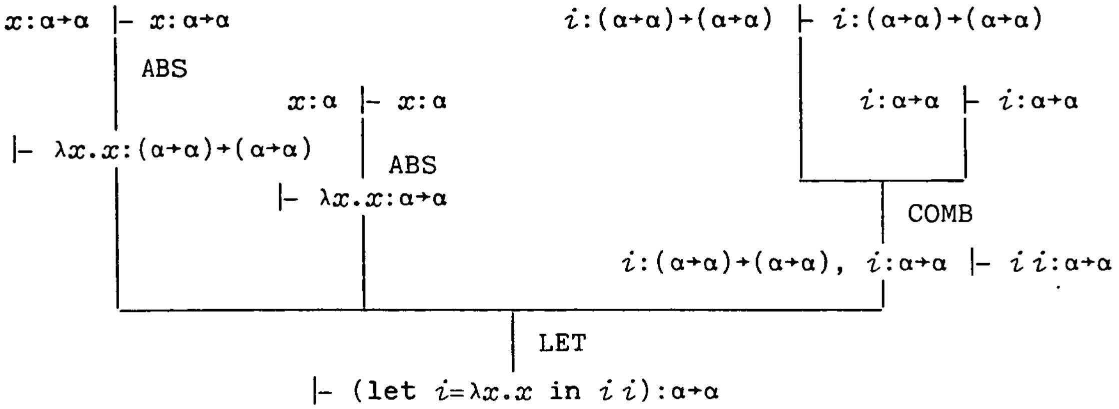
TAUT: $A \vdash x:\tau$ $\qquad$ $(x:\tau \text{ in } A)$

COMB: $$\frac{A \vdash e:\tau'\to\tau,\ A \vdash e':\tau'}{A \vdash ee':\tau}$$

ABS: $$\frac{A_x\cup\{x:\tau'\} \vdash e:\tau}{A \vdash \lambda x.e:\tau'\to\tau}$$

LET: $$\frac{A \vdash e:\tau_1,\ \ldots,\ A \vdash e:\tau_n,\ A_x\cup\{x:\tau_1,\ldots,x:\tau_n\} \vdash e':\tau}{A \vdash (\text{let } x=e \text{ in } e'):\tau}$$

The following example of a derivation is organised as a tree, in which each node follows from those immediately above it by an inference rule.


x:α→α |- x:α→α
ABS
x:α |- x:α
|- λx.x:(α→α)→(α→α)
ABS
|- λx.x:α→α
i:(α→α)→(α→α) |- i:(α→α)→(α→α)
i:α→α |- i:α→α
COMB
i:(α→α)→(α→α), i:α→α |- i i:α→α
LET
|- (let i=λx.x in i i):α→α

So far we have not imposed any constraint on the integer n in the inference rule LET. In fact we will allow n to be any non-negative integer. However if we require n to be non-zero we obtain a stronger inference relation which we will denote by $\overset{S}{\vdash}$. This stronger relation has an advantage from a pragmatic point of view since it forces every subexpression of a well-typed program to be well-typed, i.e., to be such that we may infer a type for it. Note that this is not the case for the weaker relation $\vdash$ since if we consider an expression of the form

let x=e in e'

such that x does not occur free in e', then to derive a type for the whole expression it is not necessary to derive a type for e. Furthermore one requires this stricter relation if one wants type inference to be sound for those semantics such as the strict semantics for expressions we have defined in the previous section, which evaluate e even when x does not occur free in e'. In other words, the stricter type inference relation is semantically sound for the strict semantics of expressions, or more precisely the following theorem also holds when we substitute $\vdash^{S}$ for $\vdash$ and $\models^{S}$ for $\models$.

Theorem 1 (Semantic soundness of type inference). For any expression e, type τ and assumptions A if

$$A \vdash e:\tau$$

holds then

$$A \models e:\tau$$

also holds.

proof: we will use induction on the structure of the derivation tree of $A \vdash e:\tau$.

basis: if the derivation consists of just one step then it must be an instance of rule TAUT. Thus e is a variable x and $x:\tau \in A$. Then, for any valuation ψ and environment ρ such that $\rho:_{\psi}A$, $\rho[\![x]\!]:_{\psi}\tau$ and the conclusion follows since $E[\![x]\!]\rho=\rho[\![x]\!]$.

induction step: we now assume that the derivation consists of more than one step and that the result holds for the subderivations or antecedents of the last step. We have three possible cases accordingly to the rule of inference used in the last step of the derivation.

case COMB: here one has $e\equiv e_1e_2$ for some expressions e_1 and e_2 and the antecedents are $A \vdash e_1:\tau'\rightarrow\tau$ and $A \vdash e_2:\tau'$ for some type τ'. Now, by the induction hypothesis, both $A \models e_1:\tau'\rightarrow\tau$ and $A \models e_2:\tau'$ hold. So, for any valuation ψ and environment ρ such that $\rho:_{\psi}A$, one has, letting $v_i=E\llbracket e_i\rrbracket\rho$ for $i=1,2$, $v_1:_{\psi}\tau'\to\tau$ and $v_2:_{\psi}\tau'$. But then $isF(v_1)=true$ and thus, by the definitions of E and of T, $E\llbracket e_1e_2\rrbracket\rho=(v_1|_F)v_2$ is in $\mathrm{T}\llbracket\tau\rrbracket\psi$ as we wanted.

case ABS: $e\equiv\lambda x.e'$ and $\tau\equiv\upsilon\to\upsilon'$ for some x, e', υ and υ' and the antecedent is $A_x\cup\{x:\upsilon\}\vdash e':\upsilon'$. To prove that $E\llbracket\lambda x.e'\rrbracket\rho$ is in $\mathrm{T}\llbracket\upsilon\to\upsilon'\rrbracket\psi$ for any valuation ψ and environment ρ such that $\rho:_{\psi}A$, it is enough to prove that $E\llbracket e'\rrbracket\rho[v/x]$ is in $\mathrm{T}\llbracket\upsilon'\rrbracket\psi$ for any v in $\mathrm{T}\llbracket\upsilon\rrbracket\psi$. Now, by the induction hypothesis $A_x\cup\{x:\upsilon\}\models e':\upsilon'$, and since one obviously has $\rho[v/x]:_{\psi}A_x\cup\{x:\upsilon\}$ the conclusion follows.

case LET: $e\equiv$let $x=e_1$ in e_2 and the antecedents are $A_x\cup\{x:\tau_1,\ldots,x:\tau_n\}\vdash e_2:\tau$ and $A\vdash e_1:\tau_1$, ..., $A\vdash e_1:\tau_n$. We will treat the case where n is zero separately from the case where it is non-zero because in doing so it becomes clear why it is necessary to take the restriction $\overset{s}{\vdash}$ to achieve semantic soundness for the strict semantics for expressions. So let us assume that n is zero and let ψ be any valuation and ρ be any environment such that $\rho:_{\psi}A$. Then, since $E\llbracket e\rrbracket\rho=E\llbracket e_2\rrbracket\rho[E\llbracket e_1\rrbracket\rho/x]$, it is enough to notice $\rho[E\llbracket e_1\rrbracket\rho/x]:_{\psi}A_x$ holds. Notice that it is at this point that the proof would break if we were considering the strict semantics because then we would have to show that $E\llbracket e_1\rrbracket\rho\neq wrong$ and we would be unable to do it since we would not have any antecedent about the type of e_1. Notice also that it is because this case does not arise when we consider $\overset{s}{\vdash}$ instead of $\vdash$ that the theorem still holds when we replace $\vdash$ and $\models$ with $\overset{s}{\vdash}$ and $\overset{s}{\models}$. Assuming now that n is non-zero we have by the induction hypothesis, for any valuation ψ and any environment ρ such that $\rho:_{\psi}A$, and letting $v_1=E\llbracket e_1\rrbracket\rho$, $v_1:_{\psi}\tau_1$, ...,

$v_1:_\psi\tau_n$ and thus $\rho[v_1/x]:_\psi A_x\cup\{x:\tau_1,\ldots,x:\tau_n\}$ would also hold from which the result follows by applying the induction hypothesis to $A_x\cup\{x:\tau_1,\ldots x:\tau_n\}\vdash e_2:\tau$. Note also that, and this is only relevant for the strict semantics, then $v_1\neq wrong$ since $v_1:_\psi\tau_1$.□

The following proposition collects together properties of type inference which are immediate consequences of its definition.

Proposition 1. If $A \vdash e:\tau$ holds then

(i) If A' contains A as a subset then $A' \vdash e:\tau$ also holds;

(ii) If A' is the result of excluding from A assumptions about variables which do not occur free in e then $A' \vdash e:\tau$ holds;

(iii) If e' is obtained from e by renaming some of the variables bounded in e, then $A \vdash e':\tau$ holds.

proof: suitable derivations are easly constructed in each of the clauses by modifying the original derivation. □

The fact that derivations are preserved under substitution of types for type variables is expressed by

Proposition 2. If $A \vdash e:\tau$ then for any substitution S, $SA \vdash e:S\tau$ also holds.

proof: We will use induction on the derivation tree of $A \vdash e:\tau$.

basis: There is only the case where the derivation consists of a simple application of rule TAUT, i.e, e is a variable x and $x:\tau$ is in A; but then $x{:}S\tau$ is also in SA and so $SA \vdash x{:}S\tau$ can be infered by the TAUT rule.

induction step: we have three differents cases to consider accordingly to the inference rule used in the last step of the derivation.

case COMB: Then e is of the form e_1e_2, and there are derivations of $A \vdash e_1{:}\tau'\to\tau$ and of $A \vdash e_2{:}\tau'$, but then, by the induction hypothesis, $SA \vdash e_1{:}S\tau'\to S\tau$ and $SA \vdash e_2{:}S\tau'$ hold, and so we can infer $SA \vdash e{:}S\tau$.

case ABS: e is of the form $\lambda x.e'$, τ is of the form $\tau_1'\to\tau_1$ and there is a derivation of $A_x\cup\{x{:}\tau_1'\} \vdash e'{:}\tau_1$ but then, again by induction, there is a derivation of $SA_x\cup\{x{:}S\tau_1'\} \vdash e'{:}S\tau_1$ and so we can infer $SA \vdash \lambda x.e_1{:}S(\tau_1'\to\tau_1)$ as we wanted.

case LET: e is of the form let $x=e_1$ in e_2 and there are types τ_1, ..., τ_n such that $A \vdash e_1{:}\tau_i$ and $A_x\cup\{x{:}\tau_1, \ldots, x{:}\tau_n\} \vdash e_2{:}\tau$ hold. But then, by induction, $SA \vdash e_1{:}S\tau_i$ holds for each i and so does $SA_x\cup\{x{:}S\tau_1, \ldots, x{:}S\tau_n\} \vdash e_2{:}S\tau$ and we can infer

$SA \vdash (\text{let } x=e_1 \text{ in } e_2){:}S\tau$. □

Note that the two previous propositions(and their proofs) also hold for $\overset{S}{\vdash}$.

The following result expresses the fact that an expression of the form let $x = e$ in e' has a type iff its reduced form has the same type.

Proposition 3. For any set of assumptions A , type τ and expression e

$$A \vdash (\text{let } x{=}e \text{ in } e'):\tau$$

holds iff

$$A \vdash \{e/x\}e':\tau$$

holds.

proof: We can assume, if necessary by renaming them, that all variables bounded in e and e' and x are distinct from each other and from variables occurring in A. This supposition simplifies the proof because, in any of the deductions involved in it, the use of rules ABS and LET does not lead to the exclusion of any assumption and so, at any node of the derivation tree, the set of assumptions will contain A.

($\Rightarrow$): if $A\cup\{x:\tau_1,\dots,x:\tau_n\} \vdash e':\tau$ and for each i there is a derivation of $A \vdash e:\tau_i$ we construct a derivation of $A \vdash \{e/x\}e':\tau$ by replacing each TAUT step involving $x:\tau_i$ in the derivation of $A \cup\{x:\tau_1,\dots,x:\tau_n\} \vdash e':\tau$ with the derivation for $A \vdash e:\tau_i$.

($\Leftarrow$): for each occurrence of x in e' there is, in the derivation of $A \vdash \{e/x\}e':\tau$, a subderivation of $A_i \vdash e:\tau_i$. But then, by Proposition 1 and by our initial remarks, there is also a derivation of $A \vdash e:\tau_i$ for each of the n distinct occurrences, and so it is possible to construct a derivation of $A \cup\{x:\tau_1, \dots,x:\tau_n\} \vdash e':\tau$ and then, using LET, of $A \vdash (\text{let } x{=}e \text{ in } e'):\tau$. □

If, in the previous proposition, we consider $\vdash^{S}$ instead of $\vdash$

then only the ⇒ part holds. However, if x occurs free in e and $A \overset{S}{\vdash} \{e/x\}e'$ then there must be at least one subderivation $A' \overset{S}{\vdash} e:\tau'$ and so the other half also holds in this case. Note also that this argument holds only for $\overset{S}{\vdash}$.

To show that type inference is preserved by let-reduction we need the concept of *context*.

Definition. Contexts $C[\ ]$ are described by

$$C[\ ] \ ::= \ [\ ] \ \mid \ e\, C[\ ] \ \mid \ C[\ ]\, e \ \mid \ \lambda x.C[\ ]$$
$$\mid \ \text{let } x{=}e \text{ in } C[\ ] \ \mid \ \text{let } x{=}C[\ ] \text{ in } e$$

where e is an expression.

That is, a context is essentially an expression with one of its subexpressions replaced by one hole [ ].

If e is an expression and $C[\ ]$ a context then $C[e]$ is the result of replacing the hole in $C[\ ]$ with e. Note that this operation is different from that of substitution because no renaming of bounded variables is allowed.

Proposition 4. Let $C[\ ]$ be a context, e be an expression and assume that for some A and τ there is a derivation of

$$A \vdash C[e]:\tau.$$

and that $A_i' \vdash e:\tau_i$ (i=1,...,n), are the steps in that derivation pertaining to the distinguished occurrence of e (there can be more than one if the hole is in a subexpression e'' occurring in the context let $x{=}e''$ ...).

Let e' be another expression such that there is a derivation of $A_i' \vdash e':\tau_i$ for each i=1,...,n. Then

$$A \vdash C[e']:\tau$$

also holds.

proof: We obtain a derivation of $A \vdash C[e']:\tau$ by replacing in the derivation of $A \vdash C[e]:\tau$ each of the subderivations $A_i' \vdash e:\tau_i$ with a derivation of $A_i' \vdash e':\tau_i$. □

From the two last propositions we can infer the invariance of type inference with respect to let-conversion.

Proposition 5. Let e be an expression and e' be obtained from e by let-conversion. Then

$$A \vdash e:\tau \text{ iff } A \vdash e':\tau.$$

A similar result holds for $\overset{S}{\vdash}$ if we assume that every let-reduced subexpression

let x=e in e'

is such that x occurs free in e'.

The above result shows that the type discipline imposed by type inference overcomes the limitation of monotyped programming languages where, for example, we can not have a single declared function for reversing lists of integers and lists of reals but instead we have to declare two functions which differ only by the types declared for their arguments.

6. A type assignment algorithm

An algorithm for, given an expression e, deciding if there is any type which can be inferred for the expression, is introduced in this section.

Although the concluding results of the previous section show that we could concentrate on λ-terms(at least as far as $\vdash$ is concerned) by applying let-reduction to expressions until a let-free expression is obtained, there are many advantages, when it comes to practical applications, in typing expressions directly. One such advantage, exploited in the ML system [Gordon *et al*, 79], is that it is then possible to type declarations more or less independently of the expressions in which they are used. As a matter of fact this can also be taken as a good argument to include declarative constructs in any language used to model type checking in programming languages.

To define our algorithm we need the unification algorithm of Robinson [Robinson, 65].

Proposition 6.(Robinson) There is an algorithm U that, when given two types τ and τ', either fails,or succeeds with a substitution U which unifies τ and τ', i.e., such that $U\tau = U\tau'$.

Further

(i) U fails, iff there is no such unifying substitution;

(ii) If S unifies τ and τ' then there is another substitution R such that $S = R\,U$;

(iii) U is idempotent and involves only variables occurring in τ or in τ'.

We will say that a type τ is a *common instance* of τ_1 and of τ_2 iff there is a substitution S such that $S\tau_1=S\tau_2=\tau$. It follows immediately from the properties of the unification algorithm that if two types have a common instance than they also have an *highest common instance*(h.c.i), i.e., a common instance of which any other common instance is an instance.

The unification algorithm and the concept of common instance are easily extended to more than two types and to tuples of types.

The type assignment algorithm to be introduced below requires the ability to generate new type variables. The somehow vague term "new type variable" could be made more precise by passing to the algorithm, as an extra argument, a sequence of type variables from which to pick new ones. However, all that is really needed is that we may assume, when combining the results of distinct invocations of the algorithm, that the set of new type variables created by each of the invocations, is disjoint from the set of new type variables created by the others; we have therefore chosen to keep the extra argument implicit rather then burdening the exposition and definitions with inessential detail.

We will also use the term "new trivial variant" of a syntactic construct involving type variables to denote the result of renaming each of the type variables occurring in it with a new type variable.

We are now able to give a recursive definition of the type assignment algorithm T which takes as argument an expression and returns, when it succeeds, a type and a set of assumptions.

Definition. $T(e)=(A,\tau)$, where

(i) If e is x then $A=\{x:\beta\}$ and $\tau=\beta$ where β is a new type variable.

(ii) If e is (e_1e_2), let $(A_i,\tau_i)=T(e_i)$ for i=1,2 and

$U=U(\tau_1,\tau_2\to\beta)$ where β is new

then $A=U(A_1 \cup A_2)$ and $\tau=U\,\beta$.

(iii) If e is $\lambda x.e'$ let $(A',\tau')=T(e')$, then:

- if there are no assumptions about x in A' then

  $A=A'$ and $\tau=\beta\to\tau'$ where β is new.

- if there is exactly one assumption $x:\upsilon$ in A' about x then

  $A=A'_x$ and $\tau=\upsilon\to\tau'$.

- if $x:\upsilon_1$, ..., $x:\upsilon_n$ are the assumptions in A' about x

  let $U=U(\upsilon_1,\ldots,\upsilon_n)$

  then $A=U\,A'_x$ and $\tau=U(\upsilon_1\to\tau')$.

(iv) If e is let $x=e_1$ in e_2

let $(A_2,\tau_2)=T(e_2)$ then:

- if there are no assumptions in A_2 about x

  then $A=A_2$ and $\tau=\tau_2$.

- if $x:\upsilon_1$, ..., $x:\upsilon_n$ are the assumptions about x in A_2

  let $(A_1,\tau_1)=T(e_1)$ and

  $(A_1^1,\tau_1^1),\ldots,(A_1^n,\tau_1^n)$ be new trivial variants of (A_1,τ_1)

  and $U=U(<\upsilon_1,\ldots,\upsilon_n>,<\tau_1^1,\ldots,\tau_1^n>)$

  then $A = U\,(A_{2_x} \cup A_1^1 \cup \ldots \cup A_1^n)$

  and $\tau = U\,\tau_2$.

remark: T fails iff one of the invocations of U fails.

The following example, where we sketch an application of T to an expression by specifying the arguments and results of T and U, ilustrates the main features of the type assignment algorithm.

- $T(\text{let } i=\lambda x.f\,x \text{ in } i\,i)$:
- $T(i\,i)$:

  $T(i)=(\{i:\beta_1\},\beta_1)$

  $T(i)=(\{i:\beta_2\},\beta_2)$

  $U(\beta_1,\beta_2\to\beta_3)=[\beta_2\to\beta_3/\beta_1]$

  $T(i\,i)=(\{i:\beta_2,\ i:\beta_2\to\beta_3\},\beta_3)$

- $T(\lambda x.f\,x)$:

  - $T(f\,x)$:

    $T(f)=(\{f:\beta_4\},\beta_4)$

    $T(x)=(\{x:\beta_5\},\beta_5)$

    $U(\beta_4,\beta_5\to\beta_6)=[\beta_5\to\beta_6/\beta_4]$

  $T(f\,x)=(\{f:\beta_5\to\beta_6,\ x:\beta_5\},\beta_6)$

  $T(\lambda x.f\,x)=(\{f:\beta_5\to\beta_6\},\beta_5\to\beta_6)$

  $U(<\beta_2,\beta_2\to\beta_3>,<\beta_7\to\beta_8,\beta_9\to\beta_{10}>)=[\beta_7\to\beta_8/\beta_2,\ \beta_7\to\beta_8/\beta_9,\ \beta_3/\beta_{10}]$

$T(\text{let } i=\lambda x.f\,x \text{ in } i\,i)=(\{f:\beta_7\to\beta_8,\mathrm{f}:(\beta_7\to\beta_8)\to\beta_3\},\beta_3)$

A comparison of our algorithm with the Curry-Hindley one for the λ-calculus[Yelles, 79] reveals that, as far as λ-terms are concerned, the second returns a single assumption for each free variable in the term while ours returns a different assumption for each occurrence of a free variable in the term. It is also not difficult to realise that, for an expression, T returns an assumption for each occurrence of a free variable in the let-free form of the expression.

We will now prove that T satisfies one of our initial claims, more precisely

Proposition 7 (Soundness of T). If $\mathrm{T}(e)$ succeeds with (A,τ) then there is a derivation of

$$A \vdash e:\tau.$$

proof: by structural induction on e:

basis: e is some variable x and so $A=\{x:\beta\}$, $\tau=\beta$, for some type variable β, and there is a derivation consisting of just a TAUT step.

induction step: there are three cases to consider:

case $e=e_1e_2$: since both $\mathrm{T}(e_1)$ and $\mathrm{T}(e_2)$ succeed we know, by the induction hypothesis, that if $(A_i,\tau_i)=\mathrm{T}(e_i)$ for i=1,2, there are derivations of

$$A_i \vdash e_i:\tau_i \qquad \text{for } i=1,2.$$

Also, since T(e) succeeds, $\mathrm{U}(\tau_1,\tau_2\rightarrow\beta)$, for some type variable β, succeeds with a substitution U and $A = U\,(A_1 \cup A_2)$ and $\tau=U\beta$. Now, since, for i=1,2, UA_i is a subset of A, we know, by (i) prop.1 and prop.2, that there are derivations of

$$A \vdash e_1:U\tau_2\rightarrow U\,\beta$$

and of

$$A \vdash e_2:U\tau_2$$

and we can then apply rule COMB to obtain the desired derivation.

case: $e=\lambda x.e'$: T(e) succeeds with some pair (A',τ') and, as in the definition of T, we have three subcases:

(i) If there are no assumptions in A' about x then $A=A'$ and $\tau=\beta\to\tau'$ for some type variable β. Now by induction there is a derivation of $A \vdash e':\tau'$ and then, since there is no assumption about x in A, we can infer that there is also a derivation of $A_x \cup \{x:\beta\} \vdash e:\tau'$ and then finally use rule ABS to obtain the desired derivation.

(ii) If there is exactly one assumption $x:\upsilon$ about x in A' then $A'=A \cup \{x:\upsilon\}$ and $\tau=\upsilon\to\tau'$ and, since there is a derivation of $A' \vdash e':\tau'$, we can immediately apply rule ABS to get a derivation Of $A \vdash \lambda x.e':\upsilon\to\tau'$.

(iii) there are two or more assumptions $x:\upsilon_1, \ldots, x:\upsilon_n$ about x in A', $U(\upsilon_1,\ldots,\upsilon_n)$ succeeds with a substitution U and $A=U\,A'_x$ and $\tau=U\,(\upsilon_1\to\tau')$. Now, by prop.2 and the induction hypothesis, there is a derivation of $U\,A' \vdash e':U\,\tau'$ and then, since

$$\begin{aligned} U\,A' &= U(A'_x \cup \{x:\upsilon_1,\ldots,x:\upsilon_n\}) \\ &= UA'_x \cup \{x:U\,\upsilon_1,\ldots,x:U\,\upsilon_n\} \\ &= A \cup \{x:U\,\upsilon_1\} \end{aligned}$$

we can infer $A \vdash \lambda x.e':U(\upsilon_1\to\tau')$ by applying rule ABS.

case e = let $x=e_1$ in e_2: T(e_2) succeeds with (A_2,τ_2) and, by the induction hypothesis, there is a derivation of $A_2 \vdash e_2:\tau_2$. Then either

(i) there are no assumptions about x in A_2 and then $A=A_2=A_{2_x}$, $\tau=\tau_2$ and it follows immediately from rule LET with n=0 that

$$A \vdash (\text{let } x=e_1 \text{ in } e_2):\tau$$

holds.

(ii) $x:\upsilon_1, \ldots, x:\upsilon_n$ are all the assumptions about x in A_2, T(e_1)

succeeds with (A_1,τ_1) and letting (A_1^1,τ_1^1), ..., (A_1^n,τ_1^n) be new trivial variants of (A_1,τ_1) then

$$\mathrm{U}(<\upsilon_1,\ldots,\upsilon_n>,<\tau_1^1,\ldots,\tau_1^n>)$$

succeeds with U and

$$A = U\,(A_{2_x} \cup A_1^1 \cup \ldots \cup A_1^n)$$

and $\tau = U\,\tau_2$.

Now, by the induction hypothesis, (i) prop. 1 and prop. 2, there are derivations of

$$A \vdash e_1 : U\,\tau_1^i$$

for i=1,...,n, and since

$$\begin{aligned} U A_2 &= U A_{2_x} \cup \{x{:}U\,\upsilon_1,\ldots,x{:}U\,\upsilon_n\} \\ &= U A_{2_x} \cup \{x{:}U\,\tau_1^1,\ldots,x{:}U\,\tau_1^n\} \end{aligned}$$

there is also a derivation of

$$A_x \cup \{x{:}U\,\tau_1^1,\ldots,x{:}U\,\tau_1^n\} \vdash e_2 : U\,\tau_2$$

and the conclusion follows again from rule LET.□

Finally we note that it is possible to change T to suit $\overset{S}{\vdash}$ instead of $\vdash$. In fact all we need to change is clause (iv) of the definition of T to

(iv) If e is let x=e_1 in e_2

let (A_i,τ_i)=T(e_i) for i=1,2, then:

- if there are no assumptions in A_2 about x

  then A=$A_1 \cup A_2$ and $\tau=\tau_2$.

- if $x{:}\upsilon_1$, ..., $x{:}\upsilon_n$ are the assumptions about x in A_2

  let (A_1^1,τ_1^1),...,(A_1^n,τ_1^n) be new trivial variants of (A_1,τ_1)

  and $U = \mathrm{U}(<\upsilon_1,\ldots,\upsilon_n>,<\tau_1^1,\ldots,\tau_1^n>)$

then $A = U\,(A_{2_x} \cup A_1^1 \cup \ldots \cup A_1^n)$

and $\tau = U\,\tau_2$.

It is then easy to see that proposition 7 holds for this new algorithm when we replace $\vdash$ with $\overset{s}{\vdash}$.

7. Principal types and completeness of T.

This section is concerned with proving that the type assignment algorithm succeeds, when applied to an expression, if any type at all can be derived for the expression. We will also show that T computes a most general type, in a sense to be made precise below, amongst those that can be derived for the expression.

Definition. A type τ is a *principal type* of an expression e iff there is a set of assumptions A such that

(i) $A \vdash e{:}\tau$

(ii) For any derivation $B \vdash e{:}\tau'$ there is a subset B' of B such that $B' \vdash e{:}\tau'$ is an instance of $A \vdash e{:}\tau$.

One of the interests of principal types is that they give a simple characterization of all the other types that can be derived for the expression, and, in particular, that when known, they provide a mean of deciding whether or not a particular type can also be inferred for the expression. As we shall see latter, if an expression has a type at all then it has a principal type. It is interesting to notice that if we had concentrated on those types which can be derived for the expression from a particular set of assumptions, then this property would no longer hold. More precisely

Definition. A type τ is *a principal type of e under A* iff

(i) $A \vdash e{:}\tau$

(ii) If $A' \vdash e{:}\tau'$ holds for some instance A' of A then $A' \vdash e{:}\tau'$ is an instance of $A \vdash e{:}\tau$.

To see that, in general, an expression does not have a principal type under a particular set of assumptions, let A be $\{x{:}\iota_1, x{:}\iota_2\}$ and the expression be just x, then the possible derivations are $A \vdash x{:}\iota_1$ and $A \vdash x{:}\iota_2$ and neither of these is an instance of the other. In fact, if the type inference system is to be used to handle overloading, we do not want it to have that property. However, we shall see that, when A is finite, the role of principal type can be played by a finite number of types.

The following lemma will be useful when proving the completeness of the type assignment algorithm.

Lemma. If υ_1 is an instance of τ_1 and υ_2 is also an instance of τ_2 and all the type variables in τ_1 are distinct from those in τ_2 then $\langle\upsilon_1,\upsilon_2\rangle$ is an instance of $\langle\tau_1,\tau_2\rangle$.

proof: Let S_1 and S_2 be minimal substitutions such that $S_1\tau_1=\upsilon_1$ and

$S_2\tau_2=\upsilon_2$. Now, since S_1 and S_2 are minimal, dom S_1 is a subset of those type variables which occur in τ_1 and similarly for S_2. Thus S_1 and S_2 have disjoint domains and we can define their simultaneous composition S. But then

$$S\langle\tau_1,\tau_2\rangle = \langle\upsilon_1,\upsilon_2\rangle$$

as required.□

The previous result is easily extended to any number of types or to larger syntactic constructs involving type variables, e.g., sets of assumptions or statements like $A \vdash e:\tau$.

A particular case of special interest to us arises when, in the previous result, υ_1 and υ_2 are the same type υ because then it follows that υ is a common instance of τ_1 and of τ_2.

The following result states the completeness of T

Proposition 8(Completeness of T). Given an expression e if there is a type υ and a set of assumptions B such that

$$B \vdash e:\upsilon$$

holds, then

(i) T(e) succeeds;

(ii) If T(e)=(A,τ) then there is a subset B' of B such that $B' \vdash e:\upsilon$ is an instance of $A \vdash e:\tau$.

proof: by structural induction on the derivation tree of $B \vdash e:\upsilon$.

basis: the derivation consists of just a TAUT step and e is a variable x, i.e., $B \vdash x:\upsilon$ and $x:\upsilon$ is in B. Now T(x) succeeds with $(\{x:\beta\},\beta)$, for some new variable β, and so, if we take $B'=\{x:\upsilon\}$,

$B' \vdash x\!:\!\upsilon$ is obviously an instance of $A \vdash x\!:\!\beta$.

induction step: we have three different cases to consider accordingly to the inference rule used in the last step of the derivation of $B \vdash e\!:\!\upsilon$

case COMB: e is e_1e_2 and there are derivations of $B \vdash e_1\!:\!\upsilon'\to\upsilon$ and of $B \vdash e_2\!:\!\upsilon'$. Now, by the induction hypothesis, $\mathrm{T}(e_1)$ and $\mathrm{T}(e_2)$ succeed and, letting $(A_i,\tau_i)=\mathrm{T}(e_i)$ for i=1,2, there are subsets B_1' and B_2' of B' such that $B_1' \vdash e_1\!:\!\upsilon'\to\upsilon$ is an instance of $A_1 \vdash e_1\!:\!\tau_1$ and $B_2' \vdash e_2\!:\!\upsilon'$ is an instance of $A_2 \vdash e_2\!:\!\tau_2$. To show that $\mathrm{U}(t_1,\tau_2\to\beta)$, where β is new, succeeds it is suficient to notice that, since we assume that all the type variables in $A_1 \vdash e_1\!:\!\tau_1$ are different from those in $A_2 \vdash e_2\!:\!\tau_2$ and from β, $\upsilon'\to\upsilon$ is a common instance of τ_1 and of $\tau_2\to\beta$. Thus $\mathrm{T}(e_1e_2)$ succeeds with $A=U(A_1 \cup A_2)$ and $\tau=U\beta$ where $U=\mathrm{U}(\tau_1,\tau_2\to\beta)$. Further, by (ii) prop. 6, $B_1' \cup B_2' \vdash e_1e_2\!:\!U\beta$ is an instance of $U(A_1 \cup A_2) \vdash e_1e_2\!:\!U\beta$ and part (ii) follows by taking $B' = B_1' \cup B_2'$.

case ABS: e is $\lambda x.e'$, υ is $\upsilon_1\to\upsilon'$ and there is a derivation of $B_x \cup \{x\!:\!\upsilon_1\} \vdash e'\!:\!\upsilon'$. Now, by the induction hypothesis, $\mathrm{T}(e')$ succeeds with (A',τ') and there is a subset B_1' of $B_x \cup \{x\!:\!\upsilon_1\}$ such that $B_1' \vdash e'\!:\!\upsilon'$ is an instance of $A' \vdash e'\!:\!\tau'$. There are three possible alternatives to consider:

(i) there are no assumptions about x in A' and then $\mathrm{T}(\lambda x.e')$ succeeds with $A=A_x'=A'$ and $\tau=\beta\to\tau'$. Now, since B_1' is an instance of A' it does not contain any assumption about x and thus is a subset of B and so, if we take $B'=B_1'$, $B' \vdash \lambda x.e'\!:\!\upsilon_1\to\upsilon'$ is still an instance of $A_x' \vdash \lambda x.e'\!:\!\beta\to\tau'$ because β does not occur in either τ' or in A'.

(ii) there is exactly one assumption $x\!:\!\nu$ about x in A'. Then

$\mathrm{T}(\lambda x.e')$ succeeds with $A=A'_x$ and $\tau=\nu\to\tau'$. Now, since $A'=A \cup \{x:\nu\}$, B'_1 must be the same as $B'_{1_x} \cup \{x:\upsilon_1\}$ and thus, since $B'_{1_x} \cup \{x:\upsilon_1\} \vdash e':\upsilon'$ is an instance of $A \cup \{x:\nu\} \vdash e':\tau'$, then $B'_{1_x} \vdash \lambda x.e':\upsilon_1\to\upsilon'$ is also an instance of $A \vdash \lambda x.e':\nu\to\tau'$ and part (ii) follows by noticing that B'_{1_x} is a subset of B.

(iii) there are two or more assumptions $x:\nu_1, \ldots, x:\nu_k$ about x in A'. Now, since B'_1 is an instance of A', $x:\upsilon_1$ must be in B'_1 and further υ_1 must be a common instance of $\nu_1, \ldots, \nu_k$, but then $\mathrm{U}(\nu_1,\ldots,\nu_k)$ succeeds and thus, if U is the unifying substitution returned by U, $\mathrm{T}(\lambda x.e')$ succeeds with $(UA'_x, U(\nu_1\to\tau'))$. It also follows from (ii) Prop. 6 that $B'_1 \vdash e':\upsilon'$ is still an instance of $U\,A' \vdash e':U\,\tau'$ but then, taking $B'=B'_{1_x}$, $B' \vdash \lambda x.e':\upsilon_1\to\upsilon'$ is also an instance of $U A'_x \vdash \lambda x.e':U(\nu_1\to\tau')$.

case LET: e is let $x=e_1$ in e_2 and there are derivations of $B \vdash e_1:\upsilon_1$, $\ldots$, $B \vdash e_1:\upsilon_n$ and of $B_x \cup \{x:\upsilon_1,\ldots,x:\upsilon_n\} \vdash e_2:\upsilon$. Also, by the induction hypothesis, $\mathrm{T}(e_2)$ succeeds and, letting $(A_2,\tau_2)=\mathrm{T}(e_2)$, there is a subset B_2 of $B_x \cup \{x:\upsilon_1,\ldots,x:\upsilon_n\}$ such that $B_2 \vdash e_2:\upsilon$ is an instance of $A_2 \vdash e_2:\tau_2$. We have two possible alternatives:

(i) there are no assumptions in A_2 about x, and so T(let $x=e_1$ in e_2) succeeds with $A=A_{2_x}=A_2$ and $\tau=\tau_2$ as we wanted. Furthermore B_2 does not contain any assumptions about x since B_2 is an instance of A_2 and there are no assumptions about x in A_2. So B_2 is a subset of B and, since $B_2 \vdash$ (let $x=e_1$ in e_2)$:\upsilon$ must still be an instance of $A_2 \vdash$ (let $x=e_1$ in e_2)$:\tau_2$, clause (ii) also holds.

(ii) there is one or more assumptions $x:\nu_1, \ldots, x:\nu_k$ about x in A_2.

In this case B_2, being an instance of A_2, contains at least one assumption about x and thus n is non-zero. So, by induction, T(e_1) succeeds with some pair (A_1,τ_1). Now let (A_1^1,τ_1^1), ..., (A_1^k,τ_1^k) be new trivial variants of (A_1,τ_1). To prove that $U(<\nu_1,\ldots,\nu_k>,<\tau_1^1,\ldots,\tau_1^k>)$ succeeds it is enough to show that its arguments have a common instance. Now, since B_2 is an instance of A_2, for each j=1,...,k there is i_j such that $<\upsilon_{i_1},\ldots,\upsilon_{i_k}>$ is an instance of $<\nu_1,\ldots,\nu_k>$. Also, by induction, there is a subset B_j' of B such that $B_j' \vdash e_1:\upsilon_{i_j}$ is an instance of $A_1 \vdash e_1:\tau_1$ and, then, also of $A_1^j \vdash e_1:\tau_1^j$ since the last two are trivial variants of each other. So, noticing that the variables in each of τ_1^1, ..., τ_1^k are all different from the variables in the others, $<\upsilon_{i_1},\ldots,\upsilon_{i_k}>$ is an instance of $<\tau_1^1,\ldots,\tau_1^k>$. Also, since the variables in $<\nu_1,\ldots,\nu_k>$ are all distinct from the ones in $<\tau_1^1,\ldots,\tau_1^k>$, it follows that $<\upsilon_{i_1},\ldots,\upsilon_{i_k}>$ is a common instance of the last two as we wanted. So U succeeds and then, if U is the substitution returned by U, T(let x=e_1 in e_2) succeeds with $A = U(A_{2_x} \cup A_1^1 \cup \ldots \cup A_1^k)$ and $\tau = U\,\tau_2$ as we wanted. Now, to prove part (ii), the discussion above proves the existence of a substitution S such that

$$S(A_2 \vdash e_2:\tau_2) = B_2 \vdash e_2:\upsilon$$

and

$$S(A_1^j \vdash e_1:\tau_1^j) = B_j' \vdash e_1:\upsilon_j$$

for j=1,...,k. But then, by part (ii) of the unification lemma, there is a substitution R such that $S = R\,U$ and then

$$B_{2_x} \cup B_1' \cup \ldots \cup B_k' = R\,A$$

is a subset of B and, since we also have

$$\upsilon = R(U\,\tau_2) = R\,\tau,$$

part (ii) holds.□

It is clear that the proof above could be easily adapted to suit $\vdash^{S}$ and the alternative type assignment algorithm defined at the end of the last section. Thus the following theorem holds for both $\vdash$ and $\vdash^{S}$ provided we define principal types with respect to $\vdash^{S}$ in a similar way to what we did using $\vdash$.

Theorem 2. If for some type υ and assumptions B it is possible to infer

$$B \vdash e:\upsilon$$

then there is a principal type of e.

8. Type schemes, assumption schemes and type inference.

This section is concerned with providing a description of the set of types which can be inferred for an expression from a given set of assumptions. In fact we shall see that it is possible to define an algorithm which achieves that purpose for any expression and for a large class of sets of assumptions which can be described in a finite way.

We start by noting that if a type τ can be inferred for an expression *e* from a set of assumptions *B* then for any substitution $[\tau_i/\alpha_i]$ such that the α_i do not occur in *B* we can also infer the type $[\tau_i/\alpha_i]\tau$ for *e* from *B* by prop. 2 since the substitution leaves *B* unchanged. That is, any type of the set

$$\{\ [\tau_i/\alpha_i]\tau \mid \text{for any types } \tau_1,\ldots,\tau_n\}$$

where the α_i are the variables which occur in τ but not in *B*, can be inferred for *e* from *B*.

The above discussion provides part of the motivation for introducing a new class of terms to describe sets of types of that form.

Definition. A *type scheme* η is either a type τ or a term of the form

$$\forall\alpha_1\ldots\alpha_n\tau$$

where τ is a type and $\alpha_1,\ldots,\alpha_n$ are type variables which will be called the *generic variables* of η.

Note that types form a subclass of the class of type schemes.

We will say that a type υ is a *generic instance* of a type scheme η or that η *subsumes* υ and write $\eta>\upsilon$ iff either η is υ or η is of the form $\forall\alpha_1\ldots\alpha_n\tau$ and there are types τ_1, ..., τ_n such that $\upsilon=[\tau_i/\alpha_i]\tau$. Note that the set of types which are generic instances of a type scheme $\forall\alpha_1\ldots\alpha_n\tau$ is precisely

$$\{\ [\tau_i/\alpha_i]\tau \mid \text{for all types } \tau_1,\ldots,\tau_n\}.$$

We extend > to a relation between type schemes η and η' by saying that η subsumes η' iff every type subsumed by η' is also subsumed by η. Note that > is reflexive and transitive. From now on we will identify type schemes which are equivalent under >.

We will say that a type variable *occurs free* in a type scheme $\forall\alpha_1\ldots\alpha_n\tau$ iff it occurs in τ and is not one of the generic variables $\alpha_1,\ldots,\alpha_n$. Note that if $\beta_1,\ldots,\beta_n$ are distinct type variables which do not occur free in $\forall\alpha_1\ldots\alpha_n\tau$ then $\forall\alpha_1\ldots\alpha_n\tau$ and $\forall\beta_1\ldots\beta_n([\beta_i/\alpha_i]\tau)$ are equivalent under >.

Given a type scheme η we will let a substitution S act on η by acting on the free variables of η while renaming, if necessary, the generic variables of η to avoid clashes with variables involved in S.

The following result provides a syntactic characterization of >.

Proposition 9. A type scheme $\forall\alpha_1\ldots\alpha_n\tau$ subsumes a type scheme $\forall\beta_1\ldots\beta_m\upsilon$ iff

(i) there are types $\tau_1,\ldots,\tau_n$ such that $\upsilon=[\tau_i/\alpha_1]\tau$

(ii) the β_j do not occur free in $\forall\alpha_1\ldots\alpha_n\tau$.

proof: assume $\forall\alpha_1\ldots\alpha_n\tau$ subsumes $\forall\beta_1\ldots\beta_m\upsilon$. Then, in particular, υ is a generic instance of $\forall\alpha_1\ldots\alpha_n\tau$ and thus there are types $\tau_1,\ldots,\tau_n$ such that $\upsilon=[\tau_i/\alpha_i]\tau$. Now every variable occurring free in $\forall\alpha_1\ldots\alpha_n\tau$ also occurs in each type subsumed by the type scheme. But then since $[\alpha/\beta_1,\ldots,\alpha/\beta_m]\upsilon$ is a generic instance of $\forall\beta_1\ldots\beta_m\upsilon$ for any variable α, none of the β_j occurs free in $\forall\alpha_1\ldots\alpha_n\tau$.

Reciprocally if $\upsilon=[\tau_i/\alpha_i]\tau$ and the β_j do not occur free in $\forall\alpha_1\ldots\alpha_n\tau$ then for any types $\upsilon_1,\ldots,\upsilon_m$

$$\begin{aligned}[\upsilon_j/\beta_j]\upsilon &= [\upsilon_j/\beta_j]\ [\tau_i/\alpha_1]\ \tau\\ &= [[\upsilon_j/\beta_j]\tau_i/\alpha_i]\ \tau\end{aligned}$$

since each β_j either is one of the α_i or does not occur in τ at all, and so $\forall\alpha_1\ldots\alpha_n\tau$ subsumes $\forall\beta_1\ldots\beta_m\upsilon$ as we wanted.□

The following proposition shows that $>$ is preserved by substitutions.

Proposition 10. For any type schemes η and η' and for any substitution $\mathcal{S}$ if η subsumes η' then $\mathcal{S}\eta$ also subsumes $\mathcal{S}\eta'$.

proof: Let η be $\forall\alpha_1\ldots\alpha_n\tau$ and η' be $\forall\beta_1\ldots\beta_m\upsilon$. Without loss of generality we can assume that neither the α_i nor the β_j are involved in $\mathcal{S}$. But then

$$\mathcal{S}\eta = \forall\alpha_1\ldots\alpha_n\mathcal{S}\tau$$

and

$$\mathcal{S}\eta' = \forall\beta_1\ldots\beta_n\mathcal{S}\upsilon$$

and the conclusion follows from the previous result because since the β_j are not involved in $\mathcal{S}$ and do not occur free in η they also do not occur free in $\mathcal{S}\eta$. Further since there are types $\tau_1,\ldots,\tau_n$ such that $\upsilon = [\mathcal{S}\tau_i/\alpha_i]\tau$ we still have

$$\mathcal{S}\upsilon = [\tau_i/\alpha_i]\mathcal{S}\tau$$

because the α_i are not involved in $\mathcal{S}$.□

Given a set of variables V and a type τ we define the *closure* of τ under V, $\overline{V}(\tau)$, as being the type scheme $\forall\alpha_1\ldots\alpha_n\tau$ where $\alpha_1,\ldots,\alpha_n$ are all the type variables occurring in τ which are not in V. For any assumptions $\mathcal{A}$ we will use $\overline{\mathcal{A}}(\tau)$ to denote the closure of τ under the set of type variables which occur in $\mathcal{A}$.

Returning to the set of types which can be inferred for an expression e from a set of assumptions B we note that the initial result of this section can now be expressed by saying that if we can infer υ for e from B then we also can infer any generic instance of $\overline{B}(\upsilon)$ for e.

Using the above definitions we can now provide a characterization of the types that can be inferred for an expression from a finite set of assumptions.

Proposition 11. There is an algorithm which, when given a finite set of assumptions B and an expression e, returns a (possibly empty) finite set of type schemes $\eta_1,\ldots,\eta_k$ such that a type υ can be inferred for e from B iff it is subsumed by one of the η_i.
proof: we first note that there is an algorithm, similar to the unification algorithm, which when given two types (or two tuples of types) decides whether or not the second type is an instance of the first and, in the affirmative case, yields a minimal substitution which when applied to the first argument gives it second argument. Using this algorithm we can define the desired algorithm as follows:

Given e and B we first apply the type assignment algorithm T to e. If T fails then we return the empty set and the proposition holds in this case because then there are no types which can be inferred for e from B. Otherwise let T succeed with A and τ and assume that A consists of the assumptions

$$x_1:\tau_1,\ \ldots,\ x_n:\tau_n$$

where the x_i are not necessarily distinct. Similarly assume that B

consists of the assumptions

$$y_1{:}\upsilon_1,\ \ldots,\ y_m{:}\upsilon_m.$$

Now there are at most a finite number of tuples $<j_1,\ldots,j_n>$ such that x_i is identical to y_{j_i} for each i. For each of these tuples if there is a minimal substitution S such that

$$S<\tau_1,\ldots,\tau_n> = <\upsilon_{j_1},\ldots,\upsilon_{j_n}>$$

we add the closure of $S\tau$ under SA to the set of type schemes computed by the algorithm.

Now to see that the algorithm meets our claims we note that for each of the substitutions S above, we can infer $S\tau$ for e from SA as well as any other type in the closure of $S\tau$ under SA. Also, since SA is a subset of B, we can also infer all those types for e from B instead of SA. Reciprocally assume we can infer υ for e from B. Then, since τ is a principal type of e, there is a substitution R such that RA is a subset of B and $R\tau=\upsilon$. Now the restriction of R to the type variables occurring in A must be identical to one of the substitutions S considered in the algorithm and, to show that υ can be obtained from $S\tau$ by instantiating some of the variables of the latter which do not occur in SA , we can assume that all the types variables in A and in τ are distinct from the type variables in B. Then any variable in τ not occurring in A is left unchanged in $S\tau$ and also does not occur in SA and those are precisely the variables that have to be instantiated in $S\tau$ to obtain υ.□

The remainder of this section will be concerned with extending the previous proposition to a larger class of sets of assumptions of a finite character.

Definition. An *assumption scheme* is a term of the form

$$x{:}\eta$$

where x is an identifier and η is a type scheme.

The concept of subsumption extends naturally to assumption schemes. We will also say that a set $\underline{B}$ of assumption schemes subsumes another set $\underline{B}'$ of assumption schemes iff every assumption scheme in $\underline{B}'$ is subsumed by one of the assumption schemes in $\underline{B}$.

Given a set of assumption schemes $\underline{B}$ we say that we can infer the type τ for e from $\underline{B}$ iff we can infer τ for e from the set of all assumptions which are subsumed by assumption schemes in $\underline{B}$. It is easy to see that propositions 1 and 2 still hold for type inference from a set of assumption schemes. In the case of proposition 2 this follows from the invariance of $>$ under substitutions. From this it also follows that if we can infer τ for e from $\underline{B}$ we then can also infer any generic instance of the closure of τ under the set of type variables which occur free in $\underline{B}$. We will let $\overline{\underline{B}}(\tau)$ denote that closure.

To prove that the proposition 11 still holds for finite sets of assumption schemes we need the following auxiliary result.

Proposition 12. There is an algorithm which when given a type scheme η and a type τ either fails or returns an instance τ' of τ which is also a generic instance of η. Further

(i) The algorithm fails iff there is no such instance of τ.

(ii) If the algorithm succeeds with τ' then for any instance ν of τ subsumed by η there is a substitution R such that $\nu=R\tau'$. Moreover R leaves unchanged any variable occurring free in η.

proof: the algorithm can be defined by:

(i) let τ_1 be a trivial variant of τ such that the variables occurring in τ_1 do not occur free in η and are distinct from the generic variables of η.

(ii) assuming η to be $\forall\alpha_1\ldots\alpha_n\upsilon$ apply the unification algorithm to υ and τ_1 but treating variables in υ other then the α_i as if they were primitive or constant types. If the unification algorithm fails then the algorithm fails. Otherwhise let U be the unifying substitution. Then the algorithm succeeds with $U\tau_1$.

Now, if the algorithm succeeds, $\tau'=U\tau_1$ is clearly an instance of τ. Also, since $\tau'=U\upsilon$ and the only variables instanciated by U in υ are the generic variables $\alpha_1,\ldots,\alpha_n$, τ' is a generic instance of η.

To prove (i) and (ii) let ν be an instance of τ which is subsumed by ν. Then ν is also an instance of τ_1. Let S be the minimal substitution such that $\nu=S\tau_1$ and let $\upsilon_1,\ldots,\upsilon_n$ be types such that $\nu=[\upsilon_i/\alpha_i]\upsilon$. Now S and $[\upsilon_i/\alpha_i]$ have disjoint domains and so we can take their simultaneous composition U_0. But U_0 unifies τ_1 and υ and so, since it leaves unchanged any variable occurring free in η, the unification algorithm, when applied in the conditions above, succeeds with some substitution U. To prove (ii) we note that there is a substitution R such that $U_0=RU$ and thus $\nu=U_0\tau_1=R\tau'$. Further since U involves only variables occurring in τ_1 and the α_i it must leave unchanged the variables occurring free in η. But then for any variable α occurring free in η one has

$$\alpha = U_0\alpha = RU\alpha = R\alpha$$

as we wanted.□

The above algorithm is easily extended to the case of tuples of type schemes and tuples of types by renaming, if necessary, the generic variables of the type schemes to ensure that each generic variable occurs in only one of the type schemes, after which the unification algorithm can be used in a way similar to the one above but applied to tuples of types.

We can now generalize proposition 11.

Theorem 3. There is an algorithm which, when given a finite set of assumption schemes $\underline{B}$ and an expression e, returns a (possibly empty) finite set of type schemes $\eta_1,\ldots,\eta_k$ such that a type υ can be inferred for e from $\underline{B}$ iff it is subsumed by one of the η_i.

proof: the proof is similar to that of prop. 11 and involves the use of the algorithm of the previous proposition. The algorithm is defined as follows:

We first apply the type assignment algorithm to e. If it fails the algorithm returns the empty set. Otherwhise let T succeed with A

and τ and assume A consists of the assumptions

$$x_1:\tau_1, \dots, x_n:\tau_n$$

and assume also that $\underline{\underline{B}}$ consists of the assumption schemes

$$y_1:\mu_1, \dots, y_m:\mu_m.$$

Now for each tuple $<j_1,\dots,j_n>$ such that y_{j_i} is x_i for each i, we apply the algorithm of prop. 12 to $<\mu_{j_1},\dots,\mu_{j_n}>$ and $<\tau_1,\dots,\tau_n>$. If it succeeds with an instance $<\tau_1',\dots,\tau_n'>$ of $<\tau_1,\dots,\tau_n>$ let S be the minimal substitution which when applied to the latter gives the first. We then add the closure of $S\tau$ under the variables occurring free in the μ_{j_i} to the set of type schemes returned by the algorithm.

The same argument used in the proof of prop. 11 can be used here to prove that any type subsumed by one of the η_i can be derived for e from $\underline{\underline{B}}$. Conversely assume we can infer υ for e from $\underline{\underline{B}}$. Then, since τ is principal, there is a substitution S' such that $\upsilon=S'\tau$ and SA is a subset of the set of all assumptions which are subsumed by at least one of the assumption schemes in B. So there is a tuple $<j_1,\dots,j_n>$ such that $S'\tau_i$ is subsumed by μ_{j_i}. But then the application of the algorithm of prop. 12, in the algorithm above, for this tuple $<j_1,\dots,j_n>$ succeeds. Further there is a substitution R, which leaves unchanged any type variable occurring free in the μ_{j_i}, such that $S'\tau_i = RS\tau_i$ for each i. Let η_p be the type scheme returned by the algorithm corresponding to the tuple $<j_1,\dots,j_n>$. We will now show that $S'\tau$ is a generic instance of η_p. For that purpose let $\alpha_1,\dots,\alpha_l$ be the type variables occurring in τ but not in A nor in SA, then we claim that

$$S'\tau = (R+[S'\alpha_j/\alpha_j])S\tau.$$

In fact for any variable α occurring in τ either α is one of the α_j in which case it is left unchanged by both S and R or occurs in A in which case $RS\alpha = S'\alpha$ since $RSA = S'A$, or it does not occur in A, in which case it is left unchanged by S, but occurs in SA and then we have by the same argument $RS\alpha = S'\alpha$. Further we already know that any variable in the domain of R does not occur free in the μ_{j_i} and the same applies to the α_j since they do not occur in SA. Thus any variable instantiated by $R+[S'\alpha_j/\alpha_j]$ in $S\tau$ must be a generic variable of η_p.□

The corollary bellow follows from noting that, in the previous proof, if $\underline{\underline{B}}$ contains at most one assumption about each identifier x, then there is at most one tuple $<j_1,\ldots,j_n>$ such that y_{j_i} is x_i.

Corollary 3. Let $\underline{\underline{B}}$ be a finite set of assumption schemes containing at most one assumption scheme about each identifier, and let e be an expression. Then, if some type can be inferred for e from $\underline{\underline{B}}$, there is a type scheme η such that any type which can be inferred for e from $\underline{\underline{B}}$ is a generic instance of η.

9. Type assignment and overloading.

By overloading an identifier we will mean using that identifier to denote more than one value.

The most common example of overloading is provided by arithmetic operators like + being used on both reals and integers. From a strict point of view this does not imply that + denotes two different values. As a matter of fact if, e.g., the set of integers denotations is disjoint from the set of real denotations, one could use a map which would work for both types of operands. Nevertheless if one wants to allow, like Algol 68 does, the extension of an operator like + to other arbitrary types one can not rely on such assumptions and one is forced to regard + as denoting more than one value.

The way the ambiguities that arise from overloading are handled is by using typechecking considerations to decide which value is intended by each occurrence of an overloaded identifier. This implies that each of the values represented by an overloaded identifier has type(s) distinct from those of the others. The process of identification just referred can be regarded as a transformation which produces a copy of the original program but with overloaded identifiers replaced by normal ones, e.g., each occurrence of + would be replaced by either $+_{\text{int}}$ or $+_{\text{real}}$.

In this section we take the above point of view about handling overloading by means of a program transformation a step further to allow declarations like let *twice* = $\lambda x.x+x$ in ...

to be treated, if neccessary, as overloading *twice* as a consequence of the overloading of +.

Let A be any set of assumptions and Φ be a function which maps each assumption in A to an identifier. The purpose of Φ is to associate with each assumption in A about an overloaded identifier a non-overloaded identifier specifying the value to which that assumption pertains, i.e., in the case of example above, Φ would map $+:\mathrm{int}\to\mathrm{int}\to\mathrm{int}$ to $+_{\mathrm{int}}$ and $+:\mathrm{real}\to\mathrm{real}\to\mathrm{real}$ to $+_{\mathrm{real}}$.

We will now introduce an algorithm which, when given a derivation tree of $A \vdash e:\tau$ and a map Φ produces another expression $\tilde{e}$ such that there are no overloaded identifiers in $\tilde{e}$ and $\Phi A \vdash \tilde{e}:\tau$ still holds where $\Phi A = \{\Phi(x:\tau):\tau \mid x:\tau \text{ in } A\}$.

We will define the algorithm recursively and accordingly to the rule of inference used in the last step of the derivation.

In the case where the derivation consists of just a TAUT step e is an identifier x and $x:\tau$ is one of the assumptions in A. In this case we take $\Phi(x:\tau)$ as $\tilde{e}$.

In the case where the last step of the derivation is a COMB step e is an expression of the form e_1e_2 and the subderivations are $A \vdash e_1:\tau'\to\tau$ and $A \vdash e_2:\tau'$ for some type τ'. We then take $\tilde{e}_1\tilde{e}_2$ as $\tilde{e}$.

If the last step is an ABS step then e is $\lambda x.e'$, τ is $\upsilon'\to\upsilon$ and the subderivation is $A_x\cup\{x:\upsilon'\} \vdash e':\upsilon$. Let $\tilde{e}'$ be the tranformed expression of e' by $\Phi[x/x:\upsilon']$, then $\tilde{e}$ is $\lambda x.\tilde{e}'$.

Finally if the last step is a LET step then e is of the form let $x{=}e'$ in e'' and the subderivations are $A \vdash e':\tau_i$ for i=1,...,n and $A_x \cup \{x:\tau_1,\ldots,x:\tau_n\} \vdash e'':\tau$. We then apply the algorithm to each of the $A \vdash e':\tau_i$ to obtain $\tilde{e}'_1,\ldots,\tilde{e}'_n$. Since many of these may be identical we will, to avoid unnecessary growth of the transformed expression, assume that, without loss of generality,

$$\tilde{e}'_1 = \tilde{e}'_2 = \ldots = \tilde{e}'_{k_1-1}$$

$$\tilde{e}'_{k_1} = \ldots = \tilde{e}''_{k_2-1}$$

$$\ldots$$

$$\tilde{e}'_{k_p} = \ldots = \tilde{e}'_n.$$

Then we also apply the algorithm to the derivation of

$$A_x \cup \{x:\tau_1,\ldots,x:\tau_n\} \vdash e'':\tau$$

and to

$$\Phi[x_1/x:\tau_1,\ldots,x_1/x:\tau_{k_p-1},\ldots,x_p/x:\tau_{k_p},\ldots,x_p/x:\tau_n]$$

where the x_j are new identifiers, to obtain $\tilde{e}''$. Finally the transformed expression will be

let $x_1{=}\tilde{e}'_1$ in

...

let $x_p{=}\tilde{e}'_p$ in $\tilde{e}''$.

It is easily seen, by induction, that the transformed expression satisfies the requirements stated previously. We further note that the number of versions introduced by the transformation of a declaration like that of *twice* is kept to a minimum.

We note on passing that we believe it is possible to use derivation trees to perform a transformation, perhaps more interesting from a practical point of view, that rather than introducing several different versions of a declaration like that of *twice* would instead parametrize it on the overloaded objects present on it, i.e., something like

let $ptwice = \lambda f.\lambda x.fxx$ in ...

and transform every occurrence of *twice* in the scope of the declaration into either $ptwice\ +_{\text{int}}$ or $ptwice\ +_{\text{real}}$ as appropriate.

Before proceeding we note that two different derivations may lead to different transformed expressions as can be seen from the following example (where K is assumed to denote the constant combinator):

$K\ 1\ +.$

In fact, taking the usual assumptions about the types of the identifiers involved, there are two possible derivations of the type int for the expression but which lead to transformed expressions which in one case are obtained by replacing + with $+_{\text{int}}$ and in the other replacing it with $+_{\text{real}}$.

We will now discuss type assignment in the presence of overloading. From the example above we realize that only expressions for which all possible derivations lead to equivalent transformed expressions should be accepted. As a matter of fact some restriction of this kind was to be expected since some expressions like, for instance, + do not even have a definite value in the presence of overloading. We will alter the algorithm of Theorem 3 to ensure that the condition refered above is satisfied.

Assume we are given a finite set of assumption schemes $\underline{\underline{B}}$, a map Φ as before, and an expression e. We would start by applying the type assignment algorithm T to e to obtain a principal type τ of e and an associated set of assumptions A. Then, assuming that A consists of the assumptions

$$x_1:\tau_1,\ldots,x_n:\tau_n$$

and that $\underline{\underline{B}}$ consists of

$$y_1:\mu_1,\ldots,y_m:\mu_m$$

we would proceed as in the algorithm of theorem 3 by considering all the tuples $<j_1,\ldots,j_n>$ such that y_{j_i} is x_i for each i and such that the algorithm of Proposition 12 succeeds when applied to $<\mu_{j_1},\ldots,\mu_{j_n}>$ and $<\tau_1,\ldots,\tau_n>$. However we would reject the expression, as leading to ambiguity, if for two such tuples $<j_1,\ldots,j_n>$ and $<j'_1,\ldots,j'_n>$ we had $\Phi(x_i:\mu_{j_i}) \neq \Phi(x_i:\mu_{j'_i})$ for some i. Intuitively we are forcing every occurrence of an overloaded identifier in the let-free form of e to be associated with the same particular value of that identifier independently of whatever derivation tree we consider. This in turn garantees that whatever derivations we take we always obtain equivalent transformed expressions.

Now, since there are many useful expressions which may lead to ambiguity it is convenient to enable the user to eliminate the ambiguity in those cases where it can not be resolved in the way outlined above. To achieve this we can introduce type constraints similar to those of ML of the form $e:\tau$. The meaning of such term is to constrain any type assigned to e to be an instance of τ. In the presence of those constraints we would start by replacing each term $e:\tau$ with $constrain_\tau\, e$, where $constrain_\tau$ is a new identifier, and add the assumption scheme $\mathit{constrain}_\tau{:}\forall\alpha_1\ldots\alpha_n\tau\to\tau$ to $\tilde{B}$ where the α_i are all the type variables in τ. After this we could proceed as above.

So far we have only considered the overloading which arises from overloaded variables occurring free in the expression. We will now extend the language to enable the arbitrary overloading of an identifier. For this purpose we will introduce a construct of the form

let $x = e_1$
and $x = e_2$
in e_3

which declares x as denoting both e_1 and e_2 in e_3. We will now show how to type such declarations provided that we restrict them to the outermost chain of declarations in an expression. Thus to type the expression above under a set of assumption schemes $\underline{\underline{B}}$ we would start by computing the type schemes that could be inferred for e_1 and for e_2. Now if e_1 and e_2 had a type in common we would reject the expression as leading to ambiguity. Otherwise we would add to $\underline{\underline{B}}$ the assumption scheme $x{:}\eta$ for each of the type schemes η computed for e_1 and e_2. Finally we would type e_3 using this extended set of assumption schemes.

CHAPTER II

CHAPTER II

A type scheme inference system

1. Introduction

The theory supporting the polymorphic type discipline of ML, the metalanguage of the LCF system [Gordon *et al* 79], was studied in [Milner 78]. Here we reformulate that theory by using quantifiers to make the *generic type variables* of [Milner 78] explicit. This leads to a set of rules for inferring type schemes for expressions which is in contrast with chapter I where the inference system only dealt with types although type schemes were used to describe the set of types that could be derived for an expression. The main advantage of such a reformulation is that it enables us to prove both the completeness, conjectured in [Milner 78], of the type assignment algorithm W and the existence of principal type schemes of an expression under particular sets of assumptions. Those constitute the main results of this chapter. Finally we will also study the relation between the inference system of chapter I and the one defined here.

2. Preliminaries

We will use the same programming language as in the previous chapter and the same set of types as before. However, for a smoother treatment lat~~t~~er, we will change the syntax of type schemes which will now be described by

$$\eta ::= \tau \mid \forall\alpha.\eta.$$

Nevertheless we will abreviate

$$\forall\alpha_1 \ldots \forall\alpha_n.\tau$$

to

$$\forall\alpha_1 \ldots \alpha_n.\tau$$

where convenient.

Let Ts denote the set of type schemes. The semantic function $\mathbf{T}$ of chapter I can be extended to type schemes by

$$\mathbf{T}[\![\forall\alpha.\eta]\!]\psi = \bigcap_{I\in V} \mathbf{T}[\![\eta]\!]\psi[I/\alpha].$$

Using this semantic function for type schemes one extends the relations $v:_{\psi}\tau$, $\rho:_{\psi}A$, $A \models e:\tau$ and $A \overset{S}{\models} e:\tau$ to terms involving type schemes instead of just types.

To conclude this section we present some properties of $\mathbf{T}$ that will be required later.

Lemma 1. $T[\![\{\tau/\alpha\}\eta]\!]\psi = T[\![\eta]\!]\psi[T[\![\tau]\!]\psi/\alpha]$.

proof: the proof is easily obtained by structural induction on η.□

Lemma 2. If $\eta>\eta'$ then $T[\![\eta]\!]\psi \subseteq T[\![\eta']\!]\psi$.

proof: We first note that, by lemma 1, given any type scheme

$$\forall\alpha_1 \ldots \alpha_n\tau$$

if $\beta_1, \ldots, \beta_n$ are distinct type variables which do not occur in τ then

$$T[\![\forall\alpha_1 \ldots \alpha_n\tau]\!]\psi = T[\![\forall\beta_1 \ldots \beta_n\{\beta_i/\alpha_i\}\tau]\!]\psi.$$

Now let η be $\forall\alpha_1 \ldots \alpha_n\tau$ and η' be $\forall\beta_1 \ldots \beta_m\tau'$. In view of what has just been said we can assume without loss of generality that the β_j do not occur in τ and are all distinct from the α_i. Note also that, since $\eta>\eta'$, there are types τ_i such that $\tau' = \{\tau_i/\alpha_i\}\tau$. But then

$$\begin{aligned} T[\![\forall\beta_1 \ldots \beta_m\tau']\!]\psi &= \bigcap \{T[\![\tau']\!]\psi[J_j/\beta_j] \mid J_j \varepsilon \bar{V}\} \\ &= \bigcap \{T[\![\{\tau_i/\alpha_i\}\tau]\!]\psi[J_j/\beta_j] \mid J_j \varepsilon \bar{V}\} \\ &= \bigcap \{T[\![\tau]\!]\psi[(T[\![\tau_i]\!]\psi[J_i/\beta_i])/\alpha_i; J_j/\beta_j] \mid J_j \varepsilon \bar{V}\}. \end{aligned}$$

But since we assumed the β_j do not occur in τ we have

$$\begin{aligned} T[\![\forall\beta_1 \ldots \beta_m\tau']\!]\psi &= \bigcap \{T[\![\tau]\!]\psi[(T[\![\tau_i]\!]\psi[J_j/\beta_j])/\alpha_i] \mid J_j \varepsilon \bar{V}\} \\ &\supseteq \bigcap \{T[\![\tau]\!]\psi[I_i/\alpha_i] \mid I_i \varepsilon \bar{V}\} \\ &= T[\![\forall\alpha_1 \ldots \alpha_n\tau]\!]\psi \end{aligned}$$

as we wanted.□

3. Type inference

From now on, and in contrast with chapter I, we shall assume that A contains at most one assumption about each identifier x.

For assumptions A, expression e and type scheme η we write $A \vdash e{:}\eta$ iff this sentence may be derived form the following inference rules:

TAUT: $$A \vdash x{:}\eta \qquad (x{:}\eta \text{ in } A)$$

INST: $$\frac{A \vdash e{:}\eta}{A \vdash e{:}\eta'} \qquad (\eta > \eta')$$

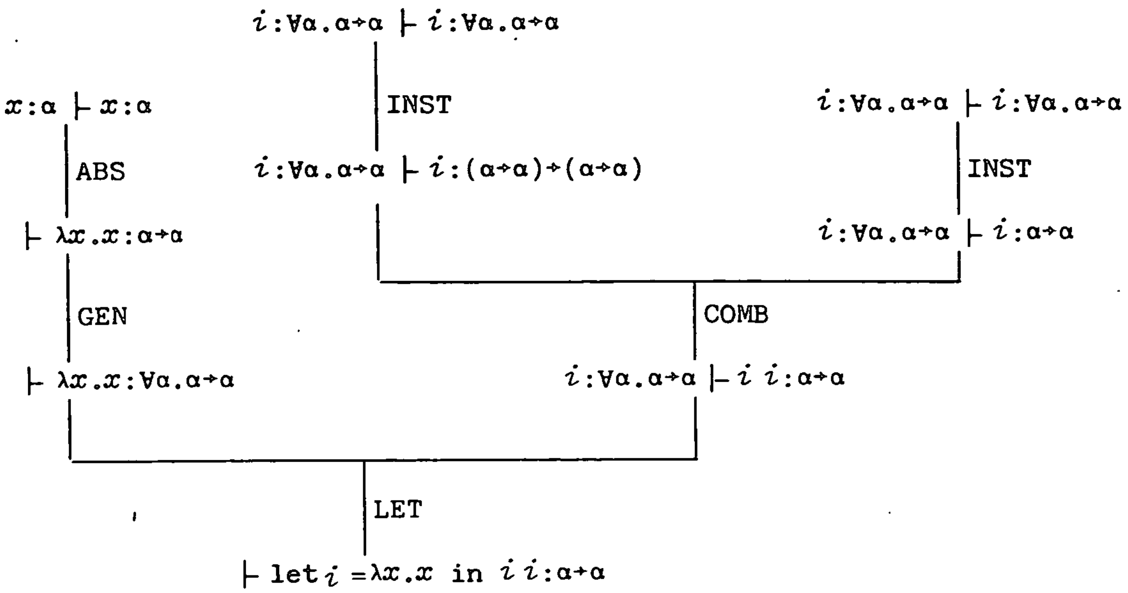
GEN: $$\frac{A \vdash e{:}\eta}{A \vdash e{:}\forall\alpha.\eta} \qquad (\alpha \text{ does not occur free in } A)$$

COMB: $$\frac{A \vdash e{:}\tau'\to\tau,\ A \vdash e'{:}\tau'}{A \vdash ee'{:}\tau}$$

ABS: $$\frac{A_x \cup \{x{:}\tau'\} \vdash e{:}\tau}{A \vdash \lambda x.e{:}\tau'\to\tau}$$

LET: $$\frac{A \vdash e{:}\eta,\ A_x \cup \{x{:}\eta\} \vdash e'{:}\tau}{A \vdash \text{let } x{=}e \text{ in } e'{:}\tau}$$

As in chapter I, the following example of a derivation is organized as a tree, in which each node follows from those immediately above it by an inference rule.



The semantic soundness of type inference is expressed by the following result which also holds for the stricter relation $\overset{s}{\models}$.

Theorem 1 (Semantic soundness of type inference). For any expression e, type scheme η and assumptions A if

$$A \vdash e:\eta$$

holds then

$$A \models e:\eta$$

also holds.

proof: since the proof, by induction on the structure of derivation tree of $A \vdash e:\eta$, is an extension of the proof of the similar result of chapter I we will only consider here the cases where the the last step of the derivation is either an INST or a GEN step.

case INST: in this case the antecedent is $A \vdash e:\eta'$ and η is a generic instance of η'. Then the result follows immediately since then $T[\![\eta']\!]\psi$ is a subset of $T[\![\eta]\!]\psi$ for any valuation ψ.

case GEN: in this case η is $\forall\alpha.\eta'$ and the antecedent is $A \vdash e:\eta'$ where α is such that it does not occur free in A. Now, let ψ be any valuation and let ρ be any environment such that $\rho:_{\psi}A$. Then, since α does not occur free in A, we also have $\rho:_{\psi[I/\alpha]}A$ for every I in $\overline{V}$. But then, by the induction hypothesis, $E[\![e]\!]\rho \in \mathbf{T}[\![\eta']\!]\psi[I/\alpha]$ for every I and thus $E[\![e]\!]\rho \in \mathbf{T}[\![\forall\alpha.\eta']\!]\psi$ as we wanted.□

It is interesting to point out that if we had restricted our attention to the relation $\models$ then the inference rule LET could be relaxed, by considering the case where x does not occur free in e', without affecting the semantic soundness of type inference.

We will need the following lemma latter.

Lemma 3. If $A_x \cup \{x:\eta'\} \vdash e:\eta_0$ holds then $A_x \cup \{x:\eta\} \vdash e:\eta_0$ also holds for any type scheme η such that $\eta > \eta'$.

proof: We construct a derivation of $A_x \cup \{x:\eta\} \vdash e:\eta_0$ from that of $A_x \cup \{x:\eta'\} \vdash e:\eta_0$ by substituting each use of rule TAUT for $x:\eta'$ with $x:\eta$ followed by an INST step. Note that GEN steps remain valid since if α occurs free in η then it also occurs free in η'.□

As in the previous chapter the invariance of type inference under substitution of types for type variables plays a major role in the proof of the existence of principal type schemes.

Proposition 1. If S is a substitution and $A \vdash e{:}\eta$ holds then $SA \vdash e{:}S\eta$ also holds. Moreover if there is a derivation of $A \vdash e{:}\eta$ of height n then there is also a derivation of $SA \vdash e{:}S\eta$ of height less or equal to n.

proof: by induction on the height n of the derivation of $A \vdash e{:}\eta$.
basis: in this case the derivation must consist simply of an instance of rule TAUT, i.e. $A \vdash x{:}\eta$ for $x{:}\eta$ in A. Then $x{:}S\eta$ is also in SA and so $SA \vdash x{:}S\eta$ also holds.
induction step: we have several different cases to consider accordingly to the rule of inference used in the last step of the derivation.
case INST: the result follows from the induction hypothesis and from the fact that $>$ is preserved by substitutions.
case GEN: in this case η is of the form $\forall\alpha.\eta'$, the antecedent is $A \vdash e{:}\eta'$ and α does not occur free in A. Let α' be a type variables not occurring free in A nor in η' and such that S does not involve α'. Now by the induction hypothesis there is a derivation of

$$S[\alpha'/\alpha]A \vdash e{:}S[\alpha'/\alpha]\eta'$$

which, since neither α nor α' occur free in A, is just

$$SA \vdash e{:}S[\alpha'/\alpha]\eta'.$$

Now, since α' does not occur free in SA, we can infer

$$SA \vdash e{:}\forall\alpha'.S[\alpha'/\alpha]\eta'$$

and the desired conclusion follows by noting that $S(\forall\alpha.\eta')$ and $\forall\alpha'.S[\alpha'/\alpha]\eta'$ are identical up to renaming of generic variables and, thus, equivalent under $>$.
other cases: follow by a direct application of the induction hypothesys to the antecedents.□

4. The type assignment algorithm W

In contrast with chapter I where it was shown that if a type could be inferred for an expression then there was also a most general or principal type, this is no longer true for the type scheme inference system considered in this chapter. To see this consider the expression $K(fx)(fy)$ where K is the constant combinator. Then we can derive type schemes for the expression if we assume, for instance, $f:\forall\alpha.\alpha\rightarrow\alpha$ or $f:\forall\alpha.\alpha\rightarrow\beta$. It is however easy to see that there is not a more general assumption scheme about f, except for the trivial $f:\forall\alpha.\alpha$, from which a type for the expression can be derived, and which includes as "instances", in some sense, both the assumptions considered above.

Nevertheless, again in contrast with chapter I, we will see that if a type scheme can be inferred for an expression from a particular set of assumptions then it admits a principal type scheme under those assumptions.

From what has been said it is then natural that a type assignment algorithm should take as arguments not only an expression but also a set of assumptions. In fact the type assignment algorithm W to be presented now, and which is essentially a translation into our notation of the one in [Milner 78], takes as arguments an expression e and a set of assumptions A and returns, when it succeeds, a type τ and a substitution S such that $SA \vdash e:\tau$. As a matter of fact we shall prove later that not only τ is the most general type amongst those which can be inferred for e from SA but also that SA itself is most general amongst those instances of A which make the derivation of a type scheme for e possible.

Definition. $W(A,e) = (S,\tau)$ where

(i) If e is x and there is an assumption $x{:}\forall\alpha_1\ldots\alpha_n\tau'$ in A

then $S = \mathrm{Id}$ and $\tau = [\beta_i/\alpha_i]\tau'$ where the β_i are new.

(ii) If e is e_1e_2 then

let $W(A,e_1) = (S_1,\tau_1)$

and $W(S_1A,e_2) = (S_2,\tau_2)$

and $U(S_2\tau_1,\tau_2\to\beta) = U$ where β is new;

then $S = US_2S_1$ and $\tau = U\beta$.

(iii) If e is $\lambda x.e_1$ then let β be a new type variable

and $W(A_x\cup\{x{:}\beta\},e_1) = (S_1,\tau_1)$;

then $S = S_1$ and $\tau = S_1\beta\to\tau_1$.

(iv) If e is let $x{=}e_1$ in e_2 then

let $W(A,e_1) = (S_1,\tau_1)$

and $W(S_1A_x\cup\{x{:}\overline{S_1A}(\tau_1)\},e_2) = (S_2,\tau_2)$;

then $S = S_2S_1$ and $\tau = \tau_2$.

remark: when any of the conditions above is not met W fails.

The following theorem, which can be easily proved by induction on e using proposition 1, shows that W meets one of our claims.

Theorem 2 (Soundness of W). If $W(A,e)$ succeeds with (S,τ) then there is a derivation of $SA \vdash e{:}\tau$.

Finally we note that since $SA \vdash e{:}\tau$ holds then $SA \vdash e{:}\overline{SA}(\tau)$ also holds. We will refer to $\overline{SA}(\tau)$ as the type scheme computed by W for e.

5. The completeness of W and principal types schemes

In this section we prove the completeness of the type assignment algorithm W and use this result to prove the existence of principal type schemes.

Theorem 3(Completeness of W). Given A and e, let A' be an instance of A and η be a type scheme such that

$$A' \vdash e{:}\eta$$

Then (i) $W(A,e)$ succeeds

(ii) If $W(A,e) = (P,\Pi)$ then, for some substitution R,

$$A' = RPA \text{ and } \overline{RPA}(\Pi) > \eta.$$

proof: The proof consists of two parts. We first prove that the theorem holds in general if it holds when the last step of a derivation of $A' \vdash e{:}\eta$ is not a GEN nor an INST step. In the second part we use structural induction on the expression e to show that the result holds for that special case.

The first part of the proof can be done by induction on the lenght of the sequence of GEN and INST steps which constitute the last steps of the derivation of $A' \vdash e{:}\eta$. It is enough to show that the theorem holds for $A' \vdash e{:}\eta$ when it holds for $A' \vdash e{:}\eta'$ and $A' \vdash e{:}\eta$ is obtained from $A' \vdash e{:}\eta'$ by an INST or a GEN step.

In the case of an INST step we have $\eta' > \eta$. But then, by the transitivity of $>$, we have $R\overline{PS}(\Pi) > \eta$ as required.

Assume now that $A' \vdash e{:}\eta$ is obtained from $A' \vdash e{:}\eta'$ by a GEN step, i.e. η is $\forall\alpha.\eta'$ for some type variable α which does not occur in A'. It is enough to show that α does not occur free in $R\overline{PA}(\Pi)$ because then, since $R\overline{PA}(\Pi) > \eta'$, we will also have $R\overline{PA}(\Pi) > \forall\alpha.\eta'$. Now, by definition, every type variable occurring free in $\overline{PA}(\Pi)$ also occurs free in PA. But then any type variable occurring free in $R\overline{PA}(\Pi)$ also occurs free in $A' = RPA$ and thus α can not occur free in $R\overline{PA}(\Pi)$ as we wanted.

We now turn to the second part of the proof. All we have to do, after taking in consideration the first part of the proof, is to prove that if the theorem holds for every proper subexpression of e, then it holds for e when the last step of the derivation of $A' \vdash e{:}\eta$ is neither an INST nor a GEN step. We have several distinct cases to consider accordingly to e.

case $e = x$: in this case the derivation is just an instance of rule TAUT, i.e. $x{:}\eta$ is one of the assumptions in A'. Now, since A' is an instance of A, let R be the minimal substitution such that $A' = RA$. Then there is an assumption $x{:}\eta_0$ in A such that $\eta = R\eta_0$. Assume η_0 is

$$\forall\alpha_1 \ldots \alpha_n \tau.$$

Then $W(A,x)$ succeeds, as we wanted, with $(Id, [\beta_i/\alpha_i]\tau)$ where the β_i are new type variables. Further $\overline{A}([\beta_i/\alpha_i]\tau)$ and η_0 are identical up to renaming of generic variables, and since

$$R\overline{A}([\beta_i/\alpha_i]\tau) = R\eta_0 = \eta$$

we also have $R\eta_0 > \eta$ trivially.

case $e = e_1e_2$: in this case the last step of the derivation of $A' \vdash e:\eta$ must be an instance of rule COMB. Then η is a type τ and the antecedents are $A' \vdash e_1:\tau'\to\tau$ and $A' \vdash e_2:\tau'$ for some type τ'. By the induction hypothesis $W(A,e_1)$ succeeds. Further if we let $(P_1,\Pi_1) = W(A,e_1)$, there is a minimal substitution R_1 such that

$$A' = R_1P_1A$$

$$R_1\overline{P_1A}(\Pi_1) > \tau'\to\tau.$$

Now let α_1, ..., α_n be the generic variables of Π_1, i.e. all the type variables which occur in Π_1 but which do not occur free in P_1A. Then R_1 leaves the α_i unchanged because we have assumed R_1 to be minimal and so dom $R_1 \subseteq \text{Fvars}(P_1A)$. Now, since $\tau'\to\tau$ is a generic instance of $R_1(\forall\alpha_1\ldots\alpha_n\Pi_1)$, it follows from Proposition 9, chapter I, and because R_1 leaves the α_i unchanged, that there are types τ_1, ..., τ_n such that

$$\tau'\to\tau = (R_1+[\tau_i/\alpha_i])\Pi_1.$$

Now A' is also an instance of P_1A and so the induction hypothesis implies that $W(P_1A,e_2)$ also succeeds. Further, if we let $(P_2,\Pi_2) = W(P_1A,e_2)$, there is a minimal substitution R_2 such that

$$A' = R_2P_2P_1A$$

$$R_2\overline{P_2P_1A}(\Pi_2) > \tau'.$$

As before, if β_1, ..., β_m are the generic variables of Π_2, we can prove that R_2 leaves the β_i unchanged and that there are types τ_1', ..., τ_m' such that

$$\tau' = (R_2+[\tau'_j/\beta_j])\Pi_2.$$

To prove that $\mathcal{W}(A,e_1e_2)$ succeeds all we have left to do is to prove that, if β is a new type variable, $\mathrm{U}(P_2\Pi_1,\Pi_2\to\beta)$ succeeds. We will do this by showing that

$$U_0 = [\tau_i/\alpha_i;\ \tau'_j/\beta_j;\ \tau/\beta] + R_2$$

is an unifying substitution. To show that U_0 is well defined we note that

$$\{\alpha_1,\ \ldots,\ \alpha_n\} \subseteq (\mathrm{Fvars}(A)\cup New_1)\setminus\mathrm{Fvars}(P_1A)$$

$$\{\beta_1,\ \ldots,\ \beta_m\} \subseteq (\mathrm{Fvars}(P_1A)\cup New_2)\setminus\mathrm{Fvars}(P_2P_1A)$$

$$\mathrm{dom}\ R_2 \subseteq \mathrm{Fvars}(P_2P_1A) \subseteq \mathrm{Fvars}(P_1A)\cup New_2$$

where New_1 is the set of new type variables used by $W(A,e_1)$ and New_2 is the one used by $\mathcal{W}(P_1A,e_2)$. From the inclusions above, and because β is new, it follows that the α_i, β_j and β are distinct from each other and are left unchanged by R_2 which proves that U_0 is well defined. Now

$$U_0(\Pi_2\to\beta) = (U_0\Pi_2)\to\tau$$

but since β and the α_i do not occur in Π_2 we have

$$U_0\Pi_2 = ([\tau'_j/\beta_j]+R_2)\Pi_2 = \tau'$$

and so $U_0(\Pi_2\to\beta) = \tau'\to\tau$. Now, to see that we also have $U_0P_2\Pi_1 = \tau'\to\tau$, we first note that, since

$$A' = R_2P_2P_1A = R_1P_1A$$

we have $R_1\alpha = R_2P_2\alpha$ for any type variable α occurring free in P_1A. But then, since every type variable occurring in $P_2\Pi_1$ is either one of the α_i or occurs free in P_2P_1A, we have

$$\begin{aligned} U_0P_2\Pi_1 &= ([\tau_i/\alpha_i;\ \tau'_j/\beta_j;\ \tau/\beta]+R_2)P_2\Pi_1 \\ &= ([\tau_i/\alpha_i]+R_2)P_2\Pi_1 \\ &= ([\tau_i/\alpha_i]+R_1)\Pi_1 \end{aligned}$$

$$= \tau'\to\tau$$

because P_2 leaves the α_i unchanged.

Since U_0 unifies $P_2\Pi_1$ and $\Pi_2\to\beta$, $\mathrm{U}(P_2\Pi_1,\Pi_2\to\beta)$ succeeds with some substitution U and thus $\mathrm{W}(A,e_1e_2)$ also succeeds, with $(UP_2P_1,U\beta)$, as we wanted. To prove clause (ii) of the theorem let R be a substitution such that $U_0=RU$. Then

$$RUP_2P_1A = U_0P_2P_1A$$

$$= R_2P_2P_1A = A'$$

since none of the α_i, β_j and β occurs free in P_2P_1A. Note also that we have trivially

$$\overline{UP_2P_1A}(U\beta) > U\beta$$

and so, by the invariance of > under substitutions, we must also have

$$\overline{RUP_2P_1A}(U\beta) > RU\beta = U_0\beta = \tau$$

as required.

case $e=\lambda x.e'$: in this case the last step of the derivation of $A'\vdash e:\eta$ must be an instance of rule ABS and thus η is a type $\tau\to\tau'$ and the antecedent is $A'_x\cup\{x:\tau\}\vdash e':\tau'$. Now let β be a new type variable, then, since $A'_x\cup\{x:\tau\}$ is an instance of $A_x\cup\{x:\beta\}$, it follows from the induction hypothesis that $\mathrm{W}(A_x\cup\{\mathrm{x}:\beta\},e')$ succeeds with some pair (P',Π') and that there is a minimal substitution R' such that

$$A'_x\cup\{x:\tau\} = R'P'(A_x\cup\{x:\beta\})$$

$$R'\overline{P'(A_x\cup\{x:\beta\})}(\Pi') > \tau'$$

So $\mathrm{W}(A,\lambda x.e')$ succeeds with $(P',P\beta\to\Pi')$ as we wanted. All that is left to show is that there is a substitution R such that

$$A' = RP'A$$

$$R\overline{P'A}(P\beta\to\Pi') \geq \tau\to\tau'.$$

Let S be the minimal substitution such that $A' = SA$ and let S_0 be the restriction of S to those type variables which occur free in A but not in A_x. We will show that

$$R = R'+S_0$$

satisfies the requirements above. First of all we note that R is well defined because R' is minimal and β, since it is new, does not occur free in A'. Further for every type variable α occurring free in A_x we have $S\alpha = R'P'\alpha$. Also if α occurs free in A but not in A_x we must have $P'\alpha = \alpha$ since

$$\text{inv } P' \subseteq \text{Fvars}(A_x \cup \{x:\beta\}) \cup New'$$

where New' is the set of new type variables used by $W(A_x \cup \{x:\beta\}, e')$. Thus, for any type variable α occurring free in A but not in A_x, we have

$$SP'\alpha = S\alpha = S_0\alpha$$

and so we have proved

$$RP'A = (R'+S_0)P'A = SA = A'$$

as we wanted. Note also that

$$\overline{P'A}(P'\beta\to\Pi') > \overline{P'(A_x \cup \{x:\beta\})}(P'\beta\to\Pi')$$

since if α is a type variable occurring in $P'\beta\to\Pi'$ and if α occurs free in $P'A$ then α also occurs free in $P'(A_x \cup \{x:\beta\})$ because

$$\text{vars}(P'\beta\to\Pi') \subseteq \text{Fvars}(A_x \cup \{x:\beta\}) \cup New'.$$

More precisely if α occurs free in $P'A$ then α occurs in $P'\gamma$ for some variable γ occurring free in A. Now if γ does not occur free in A_x then $P'\gamma = \gamma$ because P' involves only variables in $\text{Fvars}(A_x \cup \{x:\beta\}) \cup New'$. But then α and γ are the same type variable which is absurd because we have assumed that α occurs in $P'\beta\to\Pi'$.

Finnaly to show that $R\overline{P'A}(P\beta\to\Pi') > \tau\to\tau'$ let $\alpha_1, \ldots, \alpha_n$ be those type variables which occur in Π' but not(free) in $P'(A_x \cup \{x:\beta\})$. Then there are types $\tau_1, \ldots, \tau_n$ such that

$$\tau' = ([\tau_i/\alpha_i]+R')\Pi' = ([\tau_i/\alpha_i]+R)\Pi'.$$

It is easly seen that the α_i are also the only type variables which occur in $P'\beta\to\Pi'$ but not in $P'(A_x \cup \{x:\beta\})$ and thus

$$\overline{P'(A_x \cup \{x:\beta\})}(P'\beta\to\Pi') = \forall\alpha_1\ldots\alpha_n(P\beta\to\Pi').$$

Further, since $R'P'\beta = \tau'$, we have

$$\tau\to\tau' = ([\tau_i/\alpha_i]+R)(P\beta\to\Pi').$$

Finnaly, by the invariance of > under substitutions,

$$R\overline{P'A}(P'\beta\to\Pi') > R\overline{P'(A_x \cup \{x:\beta\})}(P'\beta\to\Pi')$$

and then it follows from the transitivity of > that also

$$R\overline{P'A}(P'\beta\to\Pi') > \tau\to\tau'$$

as we wanted.

case e = let $x=e_1$ in e_2: in this case the last step of the derivation of $A' \vdash$ let $x=e_1$ in $e_2:\eta$ must be an instance of rule LET. Thus η is a type τ and the antecedents are $A' \vdash e_1:\eta_1$ and $A'_x \cup \{x:\eta_1\} \vdash e_2:\tau$ for some type scheme η_1. Then, by the induction hypothesis, $W(A,e_1)$ succeeds with some pair (P_1,Π_1) and there is a minimal substitution R_1 such that

$$A' = R_1P_1A$$

$$R_1\overline{P_1A}(\Pi_1) > \eta_1.$$

Now, by lemma 3, since $A'_x \cup \{x:\eta_1) \vdash e_2:\tau$ and $R_1\overline{P_1A}(\Pi_1)>\eta_1$ there is also a derivation of $A'_x \cup \{x:R_1\overline{P_1A}(\Pi_1)\} \vdash e_2:\tau$. But

$$A'_x\cup\{x:R_1\overline{P_1A}(\Pi_1)\} = R_1(P_1A_x\cup\{x:\overline{P_1A}(\Pi_1)\})$$

and thus, again by induction, $W(A_x\cup\{x:\overline{P_1A}(\Pi_1)\},e_2)$ succeeds with (P_2,Π_2) as we wanted. Now there is a minimal R_2 such that

$$R_2\overline{P_2(P_1A_x\cup\{x:\overline{P_1A(\Pi_1)}\})}(\Pi_2) > \tau$$

$$A'_x\cup\{x:R_1\overline{P_1A}(\Pi_1)\} = R_2P_2(P_1A_x\cup\{x:\overline{P_1A}(\Pi_1)\}).$$

But then

$$A'_x = R_2P_2P_1A_x$$

and

$$R_1\overline{P_1A}(\Pi_1) = R_2P_2\overline{P_1A}(\Pi_1).$$

Finnaly we must prove the existence of an R such that

$$R\overline{P_2P_1A}(\Pi_2) > \tau$$

$$A' = RP_2P_1A.$$

Let R_0 be the restriction of R_1 to those type variables which occur free in P_1A but not in P_1A_x. We will show that

$$R = R_0+R_2$$

satisfies the above requirements. We start by noting the following inclusions

$$\text{Fvars}(P_1A)\subseteq \text{Fvars}(A)\cup New_1$$

$$\text{Fvars}(P_2P_1A_x)\subseteq \text{Fvars}(P_1A_x)\cup New_2$$

$$\text{dom}(P_2)\subseteq \text{Fvars}(P_1A_x)\cup New_2$$

$$\text{dom}(R_2)\subseteq \text{Fvars}(P_2P_1A_x)$$

$$\text{dom}(R_1)\subseteq \text{Fvars}(P_1A).$$

Now let α be a type variable. If α occurs free in P_1A_x then $RP_2\alpha = R_1\alpha$. Similarly if α occurs free in P_1A but not in P_1A_x then $P_2\alpha = \alpha$ and thus

$$RP_2\alpha = R\alpha = R_0\alpha = R_1\alpha$$

and thus we have proved $RP_2A = A'$.

Now, from

$$\mathrm{vars}(\Pi_2)\quad \mathrm{Fvars}(P_1A_x)\cup New_1\cup New_2$$

it follows that

$\overline{P_2P_1A}(\Pi_2) > \overline{P_2(P_1A_x\cup\{x:\overline{P_1A(\Pi_1)}\})}(\Pi_2)$ and thus, by the invariance of > under substitutions,

$$R\overline{P_2P_1A}(\Pi_2) > R\overline{P_2(P_1A_x\cup\{x:\overline{P_1A(\Pi_1)}\})}(\Pi_2).$$

Also, since

$$\mathrm{Fvars}(\overline{P_2(P_1A_x\cup\{x:\overline{P_1A(\Pi_1)}\})}(\Pi_2))\subseteq \mathrm{Fvars}(P_2P_1A_x)\cup New_1\cup New_2$$

we have

$$R\overline{P_2(P_1A_x\cup\{x:\overline{P_1A(\Pi)}\})}(\Pi_2) = R_2\overline{P_2(P_1A_x\cup\{x:\overline{P_1A(\Pi)}\})}(\Pi_2)$$

and $R\overline{P_2P_1A}(\Pi_2) > \tau$ follows from the transitivity of >.□

The above result proves that SA is the least instance of A for which it is possible to derive a type for e. To complete the proof of our claims about W we will now formalize the notion of principal type scheme.

We will say that η is a *principal type scheme of e under A* iff

(i) $A \vdash e:\eta$ holds;

(ii) if $A \vdash e:\eta'$ holds then $\eta>\eta'$.

Corollary 1. If $W(A,e)$ succeeds with (P,Π) then $\overline{PA}(\Pi)$ is a principal type scheme of e under PA.

proof: assume $PA \vdash e{:}\eta'$. Then, by the previous theorem there is a substitution R such that $RPA = PA$ and $R\overline{PA}(\Pi) > \eta'$. Now, since the only type variables which occur free in $\overline{PA}(\Pi)$ are those which occur free in PA and since R leaves those variables unchanged because $RPA = PA$, we have $R\overline{PA}(\Pi) = \overline{PA}(\Pi)$ and thus $\overline{PA}(\Pi) > \eta'$ as required.□

Corollary 2. If it is possible to derive a type scheme for e from assumptions A then there is a principal type scheme of e under A.

proof: we start by noting that if $A' \vdash e{:}\eta'$ is a trivial variant of $A \vdash e{:}\eta$ then η is a principal type scheme of e under A iff η' is a principal type scheme of e under A'. Now, since it is possible to derive a type scheme for e from A, $W(A,e)$ succeeds with some pair (P,Π) and there is a substitution R such that $RPA = A$. But then PA is a trivial variant of A, and since by the previous corollary, there is a principal type scheme of e under PA it follows that there is also a principal type scheme of e under A.□

6. Comparison with the inference system of chapter I.

We will close this chapter with a comparison of the inference system defined in this chapter with the one defined in chapter I.

We start by noting that the main differences between the two inference systems reside in the way let-expressions are typed and in the use of type schemes in the inference system studied in this chapter.

Concerning type schemes it is interesting to note that rule INST is in accordance with the point of view, taken in section 8 of chapter I, of regarding type schemes as describing sets of types. Similarly rule GEN corresponds to the initial remark made in that section, showing that if one could infer τ for e from B and if α was a type variable which did not occur in B then, for any other type τ', one could also infer $[\tau'/\alpha]\tau$ for e from B.

Turning now our attention to rule LET we see that in chapter I, in order to infer a type for

let x=e in e'

one has to infer for e all the types which are needed to make e' well typed. Now, if the sets of assumption schemes involved contain at most one assumption about each identifier, then by the corollary to theorem 3 of chapter I, we know that all the types one can infer for e are generic instances of a same type scheme. This reasoning can been taken as a justification for the form rule LET takes in the inference system studied in this chapter.

To make the above remarks more precise let $\overset{+}{\vdash}$ denote the extension of the relation $\overset{S}{\vdash}$ of chapter I to sets of assumption schemes as was done in section 8 of that chapter. We will extend $\overset{+}{\vdash}$ to type schemes by writing

$$A \overset{+}{\vdash} e{:}\eta$$

iff

$$A \overset{+}{\vdash} e{:}\tau$$

holds for every generic instance τ of η. Then

Proposition 2. Let A be any set of assumption schemes containing at most one assumption scheme about each identifier . Then for any expression e and type scheme η

$$A \vdash e{:}\eta \Leftrightarrow A \overset{+}{\vdash} e{:}\eta.$$

proof:($\Rightarrow$) we will use induction on the derivation of $A \vdash e{:}\eta$. We will only consider here the case where the last step of the derivation is an instance of rule LET since in all the other cases the desired conclusion follows directly from the definitions and from the induction hypothesis. So we will assume that e is of the form let x=e_1 in e_2, η is a type τ and that, for some type scheme η', the antecedents are $A \vdash e_1{:}\eta'$ and $A_x \cup \{x{:}\eta\} \vdash e_2{:}\tau$. Now by the induction hypothesis there are derivations of $A \overset{+}{\vdash} e_1{:}\eta'$ and of $A_x \cup \{x{:}\eta'\} \overset{+}{\vdash} e_2{:}\tau$. But from the definition of $\overset{+}{\vdash}$ it follows that there are a finite number of generic instances $\tau_1, \ldots, \tau_n$ of η' such that $A_x \cup \{x{:}\tau_1, \ldots, x{:}\tau_n\} \overset{+}{\vdash} e_2{:}\tau$. But then, since there are derivations of $A \overset{+}{\vdash} e_1{:}\tau_i$ for i=1,...,n, we can also infer $A \overset{+}{\vdash}$ (let x=e_1 in e_2)$:\tau$ as we wanted.

($\Leftarrow$) we will use structural induction on e. Again the only case which presents some degree of difficulty is the one where e is of the form let $x=e_1$ in e_2. Now all we have to show is that if $A \overset{+}{\vdash} (\text{let } x=e_1 \text{ in } e_2):\tau$ holds then $A \vdash (\text{let } x=e_1 \text{ in } e_2):\tau$ also holds. But if $A \overset{+}{\vdash} (\text{let } x=e_1 \text{ in } e_2):\tau$ then there are types $\tau_1, \ldots, \tau_n$ and derivations $A \overset{+}{\vdash} e_1:\tau_i$ for i=1,...,n, such that

$$A_x \cup \{x:\tau_1,\ldots,x:\tau_n\} \overset{+}{\vdash} e_2:\tau.$$

Now, since A contains at most an assumption about each identifier, there is, by the corollary to theorem 3 of chapter I, a type scheme η such that $A \overset{+}{\vdash} e_1:\eta$ and such that each of the τ_i is a generic instance of η. But then we have $A_x \cup \{x:\eta\} \overset{+}{\vdash} e_2:\tau$ and thus, by the induction hypothesis, both $A \vdash e_1:\eta$ and $A_x \cup \{x:\eta\} \vdash e_2:\tau$ hold and so we can infer $A \vdash (\text{let } x=e_1 \text{ in } e_2):\tau$ as we wanted.□

It follows from the above result that the set of types that can be inferred for an expression, from a set of assumption schemes involving at most one assumption about each identifier, is the same for both inference systems. On the other hand the type discipline of chapter I can cope with sets of assumption involving more then one assumption scheme about each identifier and thus, that type discipline can be regarded as a proper extension of the one studied in the present chapter.

Turning now our attention to the type assignment algorithms defined for each inference system and in particular comparing the type assignment algorithm W with the one defined in theorem 3 of chapter I we see that the main difference is that the former accepts as arguments a set of assumption schemes A and an expression e and returns, when it succeeds, a substitution S and a principal type scheme of e under SA, while the latter accepts as arguments a set of assumption schemes A and an expression e and returns, when it succeeds, a set of types which describe all the types which can be inferred for e from A.

The above difference is however of no importance from a practical point of view since in practical applications there are no free type variables in the assumptions A. As a matter of fact W returns a substitution S only to enable its recursive definition.

A subject that we will not discuss in this work and which is important from a practical point of view is a comparison, from a point of view of efficiency, of practical implementations of the two algorithms and in particular a study of the extra cost involved in handling multiple assumptions concerning a same identifier which forced the combinatorial nature of the algorithm defined in theorem 3 of chapter I.

CHAPTER III

CHAPTER III

References to a store and type inference

1. Introduction

In this chapter we extend the methods and results of chapter II to the case where the language semantics is no longer purely applicative but includes references to an updatable store as first class objects.

First of all we will use an example to show that the semantic soundness of the type inference system of the previous chapter does not hold for this extended semantics. Assuming we extend types to include terms of the form

τ ref

to mean the type of a reference to a store location used to hold objects of type τ, and that store access functions

newref:$\forall\alpha.\alpha \rightarrow \alpha$ ref

$contents{:}\forall\alpha.\alpha$ ref $\rightarrow \alpha$

$update{:}\forall\alpha.\alpha$ ref $\rightarrow \alpha \rightarrow \alpha$ ref

are available, and that list functions are also available, we could infer, using the inference system of chapter II, the type int list for the following expression

let x = *newref* [ ]

in let y = *update* x *(cons true* [ ]*)*

in *contents (update* x *(cons* 1 *(contents* x*)))*

even though the result is the list [1;*true*] which is not of that type.

The inadequacy of the inference system of chapter II to cope with references could be expected since it pays no attention to the side effects which may result, in the extended semantics, from the evaluation of an expression.

In the example above the semantic failure of the type inference can be traced to the fact that it allows, in the declaration of x, the generalization of the type of the expression

newref [ ]

to infer the type scheme

$\forall\alpha.\alpha$ list ref

for x. Our solution, which will prevent such generalizations, is based on considering not only the type of the result of an expression but also the types of any new references to the store which are created as a side effect of the evaluation of the expression. In particular the type inference relation will now have the extended form

$$A \vdash e{:}\tau^{*}\Delta$$

where Δ is a finite set of types, to mean that not only the result of evaluating *e* is of type τ but also that the type of any of the references created as a side effect of the evaluation is one of the types in Δ.

What was said above about the evaluation of expressions applies also to functions. In particular a function like *newref* should be described by the type scheme

$$\forall\alpha.\alpha \to \alpha \text{ ref}^{*}\{\alpha\}$$

to mean that when it is applied to an object of type α it returns an object of type α ref and that any new store location created by that evaluation, in this case just one, is of type α. In the case of the expression *newref* [ ], assuming *A* consists of the assumptions

$$\{\mathit{newref}{:}\forall\alpha.\alpha \to \alpha \text{ ref},\ [\,]{:}\forall\alpha.\alpha \text{ list}\}$$

it will be possible to infer

$$A \vdash \mathit{newref}\ [\,]{:}\alpha \text{ list ref} * \{\alpha\}$$

but then it will not be possible to generalize this to

$$A \vdash \mathit{newref}\ [\,]{:}(\forall\alpha.\alpha \text{ list ref})^{*}\{\alpha\}$$

since α occurs in $\{\alpha\}$. Note also that, for the moment, values do not have sets Δ associated with their type but that those sets are used to describe side effects which result from the evaluation of expressions. That is why we will not be interested in terms like

$$A \vdash \mathit{newref}\ [\,]{:}\forall\alpha.(\alpha \text{ list ref} * \{\alpha\}).$$

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. That is so because the algorithm relies heavily on the properties of the unification algorithm and the latter can not be extended to unify terms involving sets of types while preserving those properties. For this reason our functional types will still have the form

$$\tau' \rightarrow \tau$$

but we will extend the type schemes of chapter II to include terms of the form

$$\tau' \rightarrow \tau{*}\Delta.$$

To overcome the absence of Δ in function types, the set Δ in a statement like

$$A \vdash e{:}(\tau'{\rightarrow}\tau)\mathrm{ref}{*}\Delta$$

will play a dual role. On one hand it means, as above, that any of the new references created when the expression is evaluated is of one of the types in Δ. On the other hand it also means, roughly, that any reference created when the function, referenced by the result of the expression, is applied to an argument of type τ' is of one of the types in Δ.

2. The language and its semantics

We start by extending the syntax of the simple programming language used in the previous chapters to include expressions of the form

$$\text{rec } f\ x.e$$

to enable the recursive definition of a function f with argument x. Note that this is necessary due to the presence of a store in the semantics which prevents the handling of recursive definitions through the introduction of an identifier bounded to the fixpoint operator as was done in the purely applicative case.

Given a set A we will use A^* to denote the set of all finite (possibly empty) sequences of elements of A. Given two such sequences s and s' we will use ss' to denote the concatenation of those sequences. We will also write $s{>}s'$ and say that s is an *extension* of s' iff there is another sequence s'' such that s is $s's''$. Finally $|s|$ will denote the length of the sequence s.

In the case of a cpo V we will use V^* to denote the cpo formed by taking all the finite sequences of elements of V together with a least element $\perp$ and ordering them by

$$s \subseteq s' \quad \Leftrightarrow_{\text{def}} s{=}\perp \text{ or } |s|{=}|s'| \text{ and } \forall n{\le}|s| \ s_n \sqsubseteq s'_n .$$

Starting from a given domain B of basic values we define the domain of *values* V, of *functions* F, of *stores* S, of *locations* L and of the *error value* W, by the following domain equations

$$V = B + L + F + W$$

$$F = V \rightarrow S \rightarrow V{\times}S$$

$$S = V^{*}$$

$$L = \mathrm{N}_{\perp}$$

$$W = \{.\}_{\perp}$$

where N stands for the set of non-negative integers and where $V{\times}S$ denotes the smashed product of V and S.

The above choice of N as our set of locations and the decision to consider only finite stores were made to simplify the definition of a semantics for types. Nevertheless we could have taken our domain of locations to be any flat domain and $L \rightarrow V$ as our domain of stores. Note also that the above assumptions are general enough for any practical purposes.

As before the domain of environments is defined by

$$Env = Ide \rightarrow V$$

and we define a semantic function

$$E : Exp \rightarrow Env \rightarrow S \rightarrow V{\times}S$$

by the following recursive equations

```
$E[\![ x ]\!]\rho s = \langle \rho[\![ x ]\!], s \rangle$
$E[\![ e_1 e_2 ]\!]\rho s =$
    let $\langle v_1, s_1 \rangle = E[\![ e_1 ]\!]\rho s$ in
      $isW(v_1) \rightarrow \langle v_1, s_1 \rangle,$
      $\neg isF(v_1) \rightarrow \langle wrong, s_1 \rangle,$
          let $\langle v_2, s_2 \rangle = E[\![ e_2 ]\!]\rho s_1$ in
            $isW(v_2) \rightarrow \langle v_2, s_2 \rangle,\ v_1|_F v_2 s_2$
$E[\![ \lambda x.e ]\!]\rho s = \langle (\lambda vs.E[\![ e ]\!]\rho[v/x]s)$ in $V,\ s \rangle$
$E[\![$ rec $f\ x.e ]\!]\rho s = \langle (\mathrm{Y}(\lambda u.\lambda vs.E[\![ e ]\!]\rho[u/f, v/x]s)$ in $V,\ s \rangle$
```

$$E[\![\text{let } x = e_1 \text{ in } e_2]\!]\rho s =$$
$$\text{let } \langle v_1,s_1\rangle = E[\![e_1]\!]\rho s \text{ in}$$
$$isW(v_1) \rightarrow \langle v_1,s_1\rangle,\ E[\![e_2]\!]\rho[v_1/x]s_1$$

where Y is the fixed point operator and, as before, *wrong* stands for "∘ in V".

The store access functions are defined as

$$newref\ v\ s = s{=}\bot \rightarrow \langle\bot,\bot\rangle, \langle\ |s|{+}1 \text{ in } V,\ sv\rangle$$

$$contents\ v\ s = \neg isL(\text{v}) \rightarrow \langle wrong,s\rangle,$$
$$v = \bot \rightarrow \langle\bot,\bot\rangle,\ \langle s_v,s\rangle$$

$$update\ v\ s =$$
$$\neg isL(v) \rightarrow \langle wrong,s\rangle,$$
$$v = \bot \rightarrow \langle\bot,\bot\rangle,\ \langle u \text{ in } V,s\rangle$$
$$\text{where } u\ v'\ s' = |\text{s}'|<v \rightarrow \langle wrong,s'\rangle,$$
$$\langle v,s'_1\ldots s'_{v-1}v's'_{v+1}\ldots s'_{|s'|}\rangle.$$

3. Types, type schemes and their semantics.

Given a set *Pt* of *primitive types* ι and a set *Tv* of *type variables* α, the syntax of types is described by

$$\tau ::= \iota \mid \alpha \mid \tau \text{ ref} \mid \tau\rightarrow\tau'.$$

We will use *Ty* to denote the set of all types and *Mty* to denote the set of *monotypes*, i.e., types without type variables in them.

The syntax of *type schemes* η is described by

$$\eta ::= \tau \mid \tau'\rightarrow\tau^*\Delta \mid \forall\alpha.\eta$$

where Δ ranges over finite sets of types. We will use θ to range over non-quantified type schemes, i.e., a term which is either a type or is of the form $\tau'\to\tau*\Delta$.

The concepts of generic variables, free occurrence, substitution of types for type variables, ..., are defined as before.

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

(i) both θ and θ' are types and $\theta' = [\tau_i/\alpha_i]\theta$

(ii) θ is $\tau'\to\tau*\Delta$, θ' is $\upsilon'\to\upsilon*\Delta'$, $\upsilon'\to\upsilon = [\tau_i/\alpha_i](\tau'\to\tau)$ and $[\tau_i/\alpha_i]\Delta$ is a subset of Δ'.

It can be proved, although we will not need it, that the relation $>$ between type schemes is the minimal reflexive and transitive relation such that

(i) $\forall\alpha.\eta > [\tau/\alpha]\eta$ (for any type τ)

(ii) $\eta>\eta'$ and α does not occur free in $\eta \Rightarrow \eta>\forall\alpha.\eta'$

(iii) $\tau'\to\tau*\Delta > \tau'\to\tau*\Delta'$ (for $\Delta'\supseteq\Delta$).

Note that the above notion of generic instance is similar to the one used in previous chapters except for the extension required to cope with the extended function types.

Proposition 1. For any substitution S and type schemes η and η'

$$\eta>\eta' \Rightarrow S\eta>S\eta'.$$

Proof: the result can be proved using the same argument as in the proof of proposition 10 of chapter I.□

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

(i) $\eta>\eta'$

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

Again, although we will not need it, it can be proved that $\eta{*}\Delta>\eta'{*}\Delta'$ is the minimal reflexive and transitive relation such that

(i) $\eta>\eta' \Rightarrow \eta{*}\Delta>\eta'{*}\Delta$

(ii) $\Delta\subseteq\Delta' \Rightarrow \eta{*}\Delta>\eta{*}\Delta'$

(iii) $(\tau'\to\tau{*}\Delta){*}\Delta > (\tau'\to\tau){*}\Delta$

(iv) if $\eta{*}\Delta>\eta'{*}\Delta'$ and α does not occur free in η nor in Δ' then $\eta{*}\Delta>(\forall\alpha.\eta'){*}\Delta'$.

The following proposition can be proved in the same way as proposition 1 above

Proposition 2. For any substitution S

$$\eta{*}\Delta>\eta'{*}\Delta' \quad\Rightarrow\quad S(\eta{*}\Delta)>S(\eta'{*}\Delta').$$

We will say that a type scheme η is *closed* iff no type variable occurs free in it. We will use *Cts* to denote the set of all closed type schemes.

We will now define a semantics for closed type schemes. In the purely applicative case we have used ideals of the domain of values to model types. However, in the presence of a store, it does not make much sense to say that, e.g., a location l belongs to the type int ref; to make this statement meaningful we must say it in the context of a store s such that s_l is of type int, or, better, in the context of a class of stores such that their location l holds an integer. To describe such classes of stores we will use finite sequences of monotypes which we will call *store typings*. So let $Sty = Mty^*$ be the set of all store typings. Thus for a particular store typing σ the set of locations having type int ref would consist of those locations l such that σ_l = int. The reason for taking only monotypes for the definition of store typings is that, as implied in the introduction, each store location is used to store objects of only one type. Further, since store typings are not involved in deductions, there was no advantage in having type variables in them. Thus maps $d:Sty\to\overline{V}$ offer a better model for types in the presence of a store. Note also that we surely want to preserve the types of values when the store typing is extended, so we impose the following constraint on the maps d

$$\sigma<\sigma' \Rightarrow \quad d\sigma \subseteq d\sigma'.$$

Finally there is the question of the types of new references which may be produced when evaluating functions. Note that this is relevant even for nonfunctional values like references to a function. Since we decided not to include the sets of types of new references to the store, in the types of functions, which may be created when the the function is evaluated, we will have to introduce then in our model for types. More precisely we will use maps $d:FP(Mty)\to Sty\to\overline{V}$ as model for types where $FP(Mty)$ stands for the set of all finite subsets of Mty. We will also impose the new constraint

$$\Delta\subseteq\Delta' \;\Rightarrow\; d\Delta\sigma\subseteq d\Delta'\sigma.$$

Let D be the set of all such maps.

Now let $\Xi:Pt\to\overline{\mathcal{B}}$ be a given semantic function for primitive types. The following result shows we can extend Ξ to closed type schemes.

Theorem 1. There exists a semantic function

$$\mathrm{T}:Cts\to D$$

such that

$$\mathrm{T}[\![\,\iota\,]\!]\Delta\sigma = \Xi[\![\,\iota\,]\!] \text{ in } \overline{V}$$

$$\mathrm{T}[\![\,\tau\text{ ref}\,]\!]\Delta\sigma = \{l \text{ in } V \mid l\varepsilon L \text{ and } \sigma_1=\tau\}\cup\{\bot\}$$

$$\mathrm{T}[\![\,\tau'\to\tau\,]\!]\Delta\sigma =$$

$$\bigcap_{\Delta'\supseteq\Delta}\;\bigcap_{\sigma'>\sigma}\{\, f \text{ in } V \mid f\varepsilon F \;\; \mathrm{V}v\varepsilon\mathrm{T}[\![\,\tau'\,]\!]\Delta'\sigma' \quad \forall s\varepsilon\Sigma\Delta'\sigma'$$

$$\exists\sigma''\varepsilon\Delta^{*} \text{ s.t. } v'\varepsilon\mathrm{T}[\![\,\tau\,]\!]\Delta'(\sigma'\sigma'') \text{ and } s'\varepsilon\Sigma\Delta'(\sigma'\sigma'')$$

$$\text{where } \langle v',s'\rangle = fvs\,\}$$

$$\mathrm{T}[\![\,\tau'\to\tau*\Delta'\,]\!]\Delta\sigma = \mathrm{T}[\![\,\tau'\to\tau\,]\!](\Delta\cup\Delta')\sigma$$

$$\mathrm{T}[\![\,\forall\alpha.\eta\,]\!]\Delta\sigma = \bigcap_{\tau\varepsilon Mty}\mathrm{T}[\![\,[\tau/\alpha]\eta\,]\!]\Delta\sigma$$

where $\Sigma:FP(Mty)\to Sty\to\overline{S}$ is defined by

$$\Sigma\Delta\sigma = \{\, s\varepsilon S \mid \; |s|=|\sigma| \text{ and } \forall l\leq|s| \;\; s_l \;\varepsilon\; \mathrm{T}[\![\,\sigma_l\,]\!]\Delta\sigma\}\cup\{\bot\}.$$

Before we turn to the proof of the existence of T we wish, using both the semantic functions E and T, to attach meaning to assertions of the form

$$A \models e{:}\eta^*\Delta.$$

If the sentence is closed, i.e. if no type variable occurs free in it, then it is said to hold iff for every store typing σ, for every environment ρ such that $\rho[\![x]\!]\in T[\![\eta']\!]\Delta\sigma$ for each $x{:}\eta'$ in A, and for any store $s\in\Sigma\Delta\sigma$ there is a store typing $\sigma'\in\Delta^*$ such that $v'\in T[\![\eta']\!]\Delta(\sigma\sigma')$ and $s'\in\Sigma\Delta(\sigma\sigma')$ where $<v',s'>=E[\![e]\!]\rho s$. In the general case $A \models e{:}\eta^*\Delta$ is said to hold iff each of its closed instances holds.

The following result will be needed when proving the semantic soundness of type inference.

Proposition 3. Let S be any substitution such that every type variable is mapped by S into a monotype. Then

(i) $\eta>\eta' \Rightarrow T[\![S\eta]\!]\Delta\sigma\subseteq T[\![S\eta']\!]\Delta\sigma$ for every set of monotypes Δ and for any store typing σ;

(ii) $\eta^*\Delta>\eta'^*\Delta' \Rightarrow T[\![S\eta]\!](S\Delta)\sigma\subseteq T[\![S\eta']\!](S\Delta')\sigma$ for any store typing σ.

We now turn to the proof of Theorem 1. First of all we remark that in the previous chapters the equations defining a semantic for types provided what was tantamount to a definition of T by finite induction on types. Due to the presence of Σ in the right hand side of the equation for functional types this is no longer the case here. Secondly, although one can endow D with a complete partial order by ordering $\overline{V}$ by either the subset relation or its reverse and then taking the extension order in $Mty \to \overline{V}$ and in $FP(Mty) \to Sty \to \overline{V}$, the function from $Cts \to D$ onto itself defined by the equations above is not continuous so we will have to use methods other then those of fixed point theory to prove the existence of T.

Before going on we shall start by proving that the right hand sides of the equations above are indeed ideals of V. This being trivial for most of the equations we will prove it for the one concerning functional types. Note that, since any intersection of ideals is also an ideal, it is enough to prove that the set

$$\{\, f \varepsilon F \mid \forall v \varepsilon \mathrm{T}[\![\tau']\!]\Delta\sigma \quad \forall s \varepsilon \Sigma\Delta\sigma$$
$$\exists \sigma' \varepsilon \Delta^{*} \text{ s.t. } v' \varepsilon \mathrm{T}[\![\tau]\!]\Delta(\sigma\sigma') \text{ and } s' \varepsilon \Sigma\Delta(\sigma\sigma')$$
$$\text{where } \langle v', s'\rangle = fvs \}$$

is an ideal (provided $\mathrm{T}[\![\tau]\!]\Delta\sigma$ is an ideal for all τ, Δ and σ). It is easy to see that the above set is downwards closed. Now to prove it is also closed under lubs of ω-chains let (f_i) be an ω-chain of elements of the above set and let f be its lub. Now for every $v \varepsilon \mathrm{T}[\ \tau'\ ]\Delta\sigma$ and $s \varepsilon \Sigma\Delta\sigma$ there exists, for each i, a store typing $\sigma_i' \varepsilon \Delta^{*}$ such that $v_i' \varepsilon \mathrm{T}[\![\tau]\!]\Delta(\sigma\sigma_i')$ and $s_i' \varepsilon \Sigma\Delta(\sigma\sigma_i')$ where $\langle v_i', s_i'\rangle = f_i vs$. Now either $s_i = \perp$ for every i, in which case since we are taking smashed products one has also $v_i' = \perp$ for every i, and then $fvs = \langle\perp,\perp\rangle$ and the condition above is trivially satisfied by taking σ' to be the empty store typing, or the s_i' are distinct from $\perp$ for all but a finite number of i. But then, since they form an ω-chain, they all have, except for a finite number, the same length, and, by the definition of Σ, the same applies to the σ_i'. Now, since the σ_i' belong to Δ^{*} and

Δ is finite and all but a finite number of the σ'_i have the same length, there must be a subsequence (σ'_{i_j}) such that the σ'_{i_j} are all the same store typing σ'. Now if $<v',s'>=fvs$ we still have $v'=\bigsqcup v'_{i_j}$ and $s'=\bigsqcup s_{i_j}$ and since for every j, $v'_{i_j}\epsilon T[\![\tau]\!]\Delta(\sigma\sigma')$ and $s'_{i_j}\epsilon\Sigma\Delta(\sigma\sigma')$ we have also $v'\epsilon T[\![\tau]\!]\Delta(\sigma\sigma')$ and $s'\epsilon\Sigma\Delta(\sigma\sigma')$ as we wanted. At this point is interesting to point out the vital role played by the finiteness of Δ and by Δ itself in the above proof.

The method we will use to prove the existence of T is essentially the one developed in [Milne 75, Milne & Strachey 76] for *inclusive predicates*(see also [Stoy 77]). However, instead of a direct proof, we prefer to present a general method based on the theory of O-categories and of initial fixed points of ω-continuous functors [Smyth & Plotkin 78]. This will give a general result concerning the existence of (a generalization of) inclusive predicates and is similar to what was done by [Reynolds 74a] for *direct complete relations*.

We start by recalling that the categorical approach to the existence of solutions of recursive domain equations like those in section 2, is based on associating a functor F with those equations and then proving the existence of an object C such that $FC\cong C$. We use essentially the same method, by extending the category in question, to prove simultaneously the existence of a solution to the recursive domain equations and of a map T satisfying the equations of Theorem 1.

We recall that an O-category is one whose *hom* sets are

ω-complete partial orders and such that composition is ω-continuous in each argument.

Our starting point is a generalization of the notion of *comma category* for O-categories.

Definition. Let C be an O-category and D be an object of C. The category C$\downarrow D$ consists of

objects: arrows $p:C\rightarrow D$ of C;

arrows: diagrams $<p:C\rightarrow D, p':C'\rightarrow D, f:C\rightarrow C'>$ of C such that $p'\circ f\subseteq p$.

$$\begin{tikzcd} C \arrow[rr, "f"] \arrow[dr, "p"'] & & C' \arrow[dl, "p'"] \\ & D & \end{tikzcd} \qquad p'\circ f\subseteq p$$

The composition in C$\downarrow D$ is just the one induced from C. It is also easy to realize that if we take the order in the hom sets of C$\downarrow D$ induced from that in the hom sets of C then C$\downarrow D$ is also an O-category and that if C has an initial object then so does C$\downarrow D$.

Lemma 1: If C is ω^{op}-complete then C$\downarrow D$ is also ω^{op}-complete. Morever a cone $<u_n:p\rightarrow p_n>$ to an ω-chain $<p_n,f_n>$ is universal iff

(i) $<u_n>$ is a limiting cone to the ω-chain $<C_n,f_n>$ in C, where C_n is the domain of p_n;

(ii) $p = \bigsqcup_n p_n\circ u_n$.

proof: we have to prove that for every ω-chain $<p_n,f_n>$ there is a limiting cone.

$$\begin{tikzcd}
C_0 \arrow[dr, "p_0"'] & C_1 \arrow[l, "f_0"'] \arrow[d, "p_1"] & C_2 \arrow[l, "f_1"'] \arrow[dl, "p_2"] & \cdots \arrow[l] \\
& D & &
\end{tikzcd}$$

Now $\langle C_n, f_n \rangle$ is an ω-chain in C so there is a limiting cone $\langle u_n : C \to C_n \rangle$

$$\begin{tikzcd}
& C \arrow[dl, "u_0"'] \arrow[d, "u_1"] \arrow[dr, "u_2"] & & \\
C_0 & C_1 \arrow[l, "f_0"] & C_2 \arrow[l, "f_1"] & \cdots \arrow[l]
\end{tikzcd}$$

Now for each n

$$p_n \circ f_n \sqsubseteq p_{n+1}$$

so

$$p_n \circ f_n \circ u_{n+1} \sqsubseteq p_{n+1} \circ u_{n+1}$$

or since $u_n = f_n \circ u_{n+1}$

$$p_n \circ u_n \sqsubseteq p_{n+1} \circ u_{n+1}.$$

But then, if we define $p = \bigsqcup_n p_n \circ u_n$, the cone $\langle u_n \rangle$ lifts to a cone from p to the chain $\langle p_n, f_n \rangle$ in $\mathrm{C} \downarrow D$ since one obviously has for each n

$$p_n \circ f_n \sqsubseteq p.$$

Let us check that it is a limiting cone. Let $\langle u'_n \rangle$ be any other cone; then there is a unique arrow f such that the following diagram comutes

$$\begin{tikzcd}
 & & C' \arrow[dll, "u'_0"'] \arrow[dl, "u'_1"'] \arrow[dr, "u'_2"] \arrow[dd, dashed, "f"] & & \\
C_0 & C_1 \arrow[l] & & C_2 \arrow[ll] & \cdots \arrow[l] \\
 & & C \arrow[ull, "u_0"] \arrow[ul, "u_1"] \arrow[ur, "u_2"'] & &
\end{tikzcd}$$

but then $u_n \circ f = u'_n$ for each n, and also, by hypothesis,

$$p_n \circ u'_n \subseteq p'$$

so

$$p_n \circ u_n \circ f \subseteq p'$$

and

$$\bigcup_n p_n \circ u_n \circ f = (\bigcup_n p_n \circ u_n) \circ f = p \circ f \subseteq p'$$

so the arrow f lifts to an arrow in $\mathrm{C}{\downarrow}D$ with domain p and codomain p'. Now suppose $\langle u'_n \rangle$ is also a limiting cone. Then f is an isomorphism and

$$p' = p \circ f$$

but then

$$p' = p \circ f = (\bigcup_n p_n \circ u_n) \circ f = \bigcup_n p_n \circ u'_n$$

as we wanted.□

Let F be an endofunctor of C^{E}, the category of embeddings of C. A *lifting* of F to $(\mathrm{C}{\downarrow}D)^{\mathrm{E}}$ is an endofunctor $\tilde{F}$ of $(\mathrm{C}{\downarrow}D)^{\mathrm{E}}$ such that

$$F \circ U = U \circ \tilde{F}$$

where U is the obvious forgetful functor from $(\mathrm{C}{\downarrow}D)^{\mathrm{E}}$ to C.

Lemma 2. If $\hat{F}$ is a lifting of F then $\hat{F}$ is ω-continuous iff F is ω-continuous.

proof: we will only prove that F is ω-continuous when F is ω-continuous. Let $<u_n:p_n \rightarrow p>$ be a colimiting cone to an ω-cochain $<p_n,f_n>$ in $(C\downarrow D)^E$. By Theorem 2 of [Smyth & Plotkin 78] it is sufficient to show that $<(Fu_n)^R>$ is a limiting cone to the ω-chain $<Fp_n,(Ff_n)^R>$ in $C\downarrow D$. Now if we let C_n denote the domain of p_n, for each n, and C denote the domain of p then, by lemma 1 above, it is enough to prove that $<(Fu_n)^R>$ is a limiting cone to the ω-chain $<FC_n,(Ff_n)^R>$ in C and that

$$Fp = \bigcup_n Fp_n \circ (Fu_n)^R.$$

The first part follows from the continuity of F. For the second part we have

$$Fp \circ Fu_n \subseteq Fp_n,$$
$$Fp_n \circ (Fu_n)^R \subseteq Fp.$$

From the second innequality we can infer

$$\bigcup_n Fp_n \circ (Fu_n)^R \subseteq Fp.$$

From the first we also have

$$Fp \circ Fu_n \circ Fu_n^R \subseteq Fp_n \circ Fu_n^R.$$

But then, since

$$\bigcup_n Fu_n \circ Fu_n^R = id_{FC},$$

we have

$$Fp = Fp \circ id_C \subseteq \bigcup_n Fp_n \circ Fu_n^R.\square$$

Now, to use the above results to prove the existence of our semantic function for types, let

- T be the cpo consisting of the two elements *true* and *false* ordered by *true* $\subseteq$ *false*;

- $Sty\to T$ be the cpo of all maps from Sty to T which are monotonic with respect to the reverse extension order in Sty and the order in T.

- $FP(Mty)\to Sty\to T$ be the cpo of all maps from $FP(Mty)$ to $Sty\to T$ which are monotonic with respect to the reverse subset relation in $FP(Mty)$;

- $\tilde{D}$ be the cpo of all maps from Cts to $FP(Mty)\to Sty\to T$;

- $\mathrm{CPO}_{\perp}$ be the category of cpos together with strict ω-continuous maps;

- F be the ω-continuous endofunctor of $\mathrm{CPO}_{\perp}^{E}$ associated with the recursive domain equation for the domain of values V.

Note that there is an isomorphism between $V\to_{\perp}\tilde{D}$ and $Cts\to FP(Mty)\to Sty\to\overline{V}$. So all we have to do is to define an appropriate lifting of F to $(\mathrm{CPO}_{\perp}\downarrow\tilde{D})^{E}$ and then the initial fixed point of this lifting will give us not only V again but also a "predicate" $\tilde{\mathrm{T}}:V\to\tilde{D}$ satisfying the equivalent to the equations in Theorem 1. Since the outlined above consists only of lenghty, but otherwhise trivial, definitions and checks of innequalities we will omit the details (see however Appendix I).

4. Type inference

We say that we can infer $\eta*\Delta$ for e from A and write $A \vdash e{:}\eta*\Delta$ iff this can be derived from the following inference rules where, as before, A_x denotes the result of excluding from A any assumption about x:

$$\text{TAUT:} \quad A \vdash x{:}\eta*\emptyset \qquad (x{:}\eta \text{ in } A)$$

$$\text{INST:} \quad \frac{A \vdash x{:}\eta*\Delta}{A \vdash e{:}\eta'*\Delta'} \qquad (\eta*\Delta>\eta'*\Delta')$$

$$\text{GEN:} \quad \frac{A \vdash e{:}\eta*\Delta}{A \vdash e{:}(\forall\alpha.\eta)*\Delta} \qquad (\alpha \text{ does not occur free in } A \text{ or in } \Delta)$$

$$\text{COMB:} \quad \frac{A \vdash e{:}(\tau'\to\tau)*\Delta,\ A \vdash e'{:}\tau'*\Delta}{A \vdash ee'{:}\tau'*\Delta}$$

$$\text{ABS:} \quad \frac{A_x \cup \{x{:}\tau'\} \vdash e{:}\tau*\Delta}{A \vdash \lambda x.e{:}(\tau'\to\tau*\Delta)*\emptyset}$$

$$\text{REC:} \quad \frac{(A_f)_x \cup \{f{:}\tau'\to\tau, x{:}\tau') \vdash e{:}\tau*\Delta}{A \vdash \text{rec } f\ x.e{:}(\tau'\to\tau*\Delta)*\emptyset}$$

$$\text{LET:} \quad \frac{A \vdash e{:}\eta*\Delta,\ A_x \cup \{x{:}\eta\} \vdash e'{:}\eta'*\Delta}{A \vdash \text{let } x{=}e \text{ in } e'{:}\eta'*\Delta}$$

The following example, where

$$A = \{newref{:}\forall\alpha.\alpha\to\alpha \text{ ref}*\{\alpha\}\}$$

and

$$B = A \cup \{x{:}\alpha\}$$

illustrates the main new features of type inference.

$$\begin{tikzcd}
B \vdash \mathit{newref}:(\forall\alpha{\to}\alpha\ \mathrm{ref}^*\{\alpha\})^*\emptyset \arrow[d, dash, "\mathrm{INST}"] & & B \vdash x:\alpha^*\emptyset \arrow[d, dash, "\mathrm{INST}"] \\
B \vdash \mathit{newref}:(\alpha{\to}\alpha\ \mathrm{ref})^*\{\alpha\} \arrow[dr, dash] & & B \vdash x:\alpha^*\{\alpha\} \arrow[dl, dash] \\
 & B \vdash \mathit{newref}\ x:\alpha\ \mathrm{ref}^*\{\alpha\} \arrow[from=u, phantom, "\mathrm{COMB}"] \arrow[d, dash, "\mathrm{ABS}"] & \\
 & A \vdash \lambda x.\mathit{newref}\ x:(\alpha{\to}\alpha\ \mathrm{ref}^*\{\alpha\})^*\emptyset \arrow[d, dash, "\mathrm{GEN}"] & \\
 & A \vdash \lambda x.\mathit{newref}\ x:(\forall\alpha.\alpha{\to}\alpha\ \mathrm{ref}^*\{\alpha\})^*\emptyset &
\end{tikzcd}$$

It is interesting to note the strong similarity between the above inference rules, apart the one for recursive function definitions, and those of chapter II. One of the consequences of that similarity is that most of the proofs of results in that chapter can be easily adapted to prove similar results for the new inference system. Note also that if $A \vdash e:\eta$ in the inference system of chapter II then we also can derive $A \vdash e:\eta^*\emptyset$ in the new inference system.

The following result corresponds, for the new inference system, to lemma 3 of chapter II and can be proved in a similar way.

Lemma 3: If $\mu > \mu'$ and $A_x \cup \{x:\mu'\} \vdash e:\eta^*\Delta$ then also $A_x \cup \{x:\mu\} \vdash e:\eta^*\Delta$.

To prove the semantic soundness of type inference it is convenient to restrict ourselves to a certain class of derivations.

We will say that a derivation of $A \vdash e:\eta$ is *standard* iff each LET step of the form

$$\frac{A' \vdash e_1:\eta_1{*}\Delta',\ A'_x \cup \{x:\eta_1\} \vdash e_2:\eta_2{*}\Delta'}{A' \vdash \text{let } x{=}e_1 \text{ in } e_2:\eta_2{*}\Delta'}$$

is such that if a type variable occurs free in η_1 then it also occurs free in either A' or in Δ'.

Lemma 4: If $A \vdash e:\eta{*}\Delta$ admits a derivation then it also admits a standard derivation.

proof: the proof consists of two parts. We first assume that the result holds when the last step of the derivation of $A \vdash e:\eta{*}\Delta$ is one of TAUT, COMB, ABS, REC or LET and note that it then holds in the general case. In the second part we will use structural induction on the expression e to prove the special case. For the first part it is enough to note that if we have a standard derivation of $A \vdash e:\eta{*}\Delta$ and follow it with either an INST or GEN step then the new derivation is also standard. For the second part the only non trivial case in the induction is when e is of the form let $x{=}e_1$ in e_2. Then the last step of the derivation is a LET step. Let $A \vdash e_1:\eta'{*}\Delta$ and $A_x \cup \{x:\eta'\} \vdash e_2:\eta{*}\Delta$ be the antecedents and assume that $\alpha_1, \ldots, \alpha_n$ are all the type variables which occur free in η' but not in A nor in Δ. We can use a sequence of GEN steps to construct a derivation of $A \vdash e_1:(\forall\alpha_1\ldots\forall\alpha_n.\eta'){*}\Delta$ and use the induction hypothesis to show that it also admits a standard derivation. Now η' is a generic instance of $\forall\alpha_1 \ldots \forall\alpha_n \circ \eta'$ and we can use lemma 3 and the induction hypothesis to show that there is also a standard derivation of $A_x \cup \{x{:}\forall\alpha_1 \ldots \forall\alpha_n . \eta'\} \vdash e_2{:}\eta^*\Delta$. Finally we can apply rule LET to those two standard derivations to obtain a standard derivation of $A \vdash e{:}\eta^*\Delta$.□

To prove the semantic soundness of type inference we have to prove the following, slightly more strong, result.

Proposition 4. If $A \vdash e{:}\eta^*\Delta$ then:

- for any substitution S which maps each type variable into a monotype,

- for any store typing σ,

- for any environment ρ such that $\rho[\![x]\!] \varepsilon \mathrm{T}[\![S\eta']\!](S\Delta)\sigma$ for every assumption $x{:}\eta'$ in A

- for any store $s \varepsilon \Sigma(S\Delta)\sigma$

there is an $\sigma' \varepsilon (S\Delta)^*$ such that $v' \varepsilon \mathrm{T}[\![S\eta]\!](S\Delta)(\sigma\sigma')$ and $s' \varepsilon \Sigma(S\Delta)(\sigma\sigma')$ where $<v', s'> = E[\![e]\!]\rho s$. Further if α does not occur free in A or in Δ then for any monotype υ one also has $v' \varepsilon \mathrm{T}[\![S[\upsilon/\alpha]\eta]\!](S\Delta)(\sigma\sigma')$.

proof: by induction on a standard derivation of $A \vdash e{:}\eta^*\Delta$.

basis: if the derivation consists of just one step then it must be an instance of rule TAUT. In this case, if we take σ' to be the empty store typing, the first part of the result holds by the hypothesis. The second part holds trivially since α does not occur free in η.

induction: we have several different possibilities accordingly to the rule of inference used in the last step of the derivation.

case INST: in this case the antecedent is $A \vdash e:\eta'{*}\Delta'$ for some type scheme η' and set of types Δ' such that $\eta'{*}\Delta'>\eta{*}\Delta$. By the induction hypothesis there is a $\sigma'\varepsilon(S\Delta)^{*}$ such that $v'\varepsilon\mathrm{T}[\![S\eta']\!](S\Delta')(\sigma\sigma')$ and $s'\varepsilon\Sigma(S\Delta')(\sigma\sigma')$. Now we have, by proposition 2 and 3, that $\mathrm{T}[\![S\eta']\!](S\Delta)(\sigma\sigma')$ is a subset of $\mathrm{T}[\![S\eta]\!](S\Delta)(\sigma\sigma')$ and thus $v'\varepsilon\mathrm{T}[\![S\eta]\!](S\Delta)(\sigma\sigma')$ as we wanted. Also if α does not occur free in A or in Δ we have, noting that Δ' is a subset of Δ and by the induction hypothesis, $v'\varepsilon\mathrm{T}[\![S\eta']\!](S\Delta')(\sigma\sigma')$ for any monotype υ. Now, by proposition 2, we also have

$$[\upsilon/\alpha](\eta'{*}\Delta') > [\upsilon/\alpha](\eta{*}\Delta).$$

But then by proposition 3 we have $v'\varepsilon\mathrm{T}[\![S[\upsilon/\alpha]\eta]\!](S\Delta)(\sigma\sigma')$ as we wanted.

case GEN: in this case η is of the form $\forall\alpha.\eta'$ and the antecedent is $A \vdash e:\eta'{*}\Delta$ where α does not occur free in A or in Δ. We have to prove the existence of a store typing $\sigma'\varepsilon(S\Delta)^{*}$ such that $v'\varepsilon\mathrm{T}[\![S[\upsilon/\alpha]\eta']\!](S\Delta)(\sigma\sigma')$ and $s'\varepsilon\Sigma(S\Delta)(\sigma\sigma')$ for every monotype υ. Now, by the induction hypothesis, there is an $\sigma'\varepsilon(S\Delta)^{*}$ such that $v'\varepsilon\mathrm{T}[\![S\eta']\!](S\Delta)(\sigma\sigma')$ and $s'\varepsilon\Sigma(S\Delta)(\sigma\sigma')$ and, further, such that $v'\varepsilon\mathrm{T}[\![S[\upsilon/\beta]\eta']\!](S\Delta)(\sigma\sigma')$ for every type variable β not occurring free in A or in Δ and for any monotype υ. But then $v'\varepsilon\mathrm{T}[\![S[\upsilon/\alpha]\eta']\!](S\Delta)(\sigma\sigma')$ for any monotype υ as we wanted. Note that it is at this point that the second part of the proposition is required to enable the proof by induction.

case COMB: in this case e is e_1e_2 for some expressions e_1 and e_2, η is a type τ and the antecedents are $A \vdash e_1:(\tau'\to\tau){*}\Delta$ and $A \vdash e_2:\tau'{*}\Delta$ for some type τ'. Now, by the induction hypothesis, there is a store typing $\sigma_1 \varepsilon (S\Delta)^*$ such that $v_1 \varepsilon T[\![S(\tau' \to \tau)]\!](S\Delta)(\sigma\sigma_1)$ and $s_1 \varepsilon \Sigma(S\Delta)(\sigma\sigma_1)$ where $<v_1, s_1> = E[\![e_1]\!]\rho s$. Now noticing that the hypothesis about ρ still holds, when we take $\sigma\sigma'$ instead of σ, there is again, by the induction hypothesis, a store typing σ_2 such that $v_2 \varepsilon T[\![S\tau']\!](S\Delta)(\sigma\sigma_1\sigma_2)$ and $s_2 \varepsilon \Sigma(S\Delta)(\sigma\sigma_1\sigma_2)$ where $<v_2, s_2> = E[\![e_2]\!]\rho s_1$. Now if $v_1 = \bot$ or $v_2 = \bot$ then $E[\![e_1 e_2]\!]\rho s = <\bot, \bot>$ and the result holds if we take any σ'. Otherwise note that both v_1 and v_2 are distinct from *wrong* and furthermore $isF(v_1) = true$ and so $E[\![e_1 e_2]\!]\rho s = v_1|_F v_2 s_2$. Now, by the definition of T, there is a store typing $\sigma_3 \varepsilon (S\Delta)^*$ such that $v' \varepsilon T[\![S\tau]\!](S\Delta)(\sigma\sigma_1\sigma_2\sigma_3)$ and $s' \varepsilon \Sigma(S\Delta)(\sigma\sigma_1\sigma_2\sigma_3)$ where $<v', s'> = v_1|_F v_2 s_2$. Thus the result holds when we take σ' to be $\sigma_1\sigma_2\sigma_3$. The second part of the result follows along the same lines from the induction hypothesis.

case ABS: here e is $\lambda x.e_1$, η is $\tau' \to \tau * \Delta'$, Δ is $\emptyset$ and the antecedent is $A_x \cup \{x:\tau'\} \vdash e_1 : \tau * \Delta'$. In this case

$$E[\![e]\!]\rho s = <\lambda vs.E[\![e_1]\!]\rho[v/x]s) \text{ in } V,\ s>$$

and we will prove that the result holds when we take σ' to be the empty store typing. Obviously all we have to prove is that

$$(\lambda vs.E[\![e_1]\!]\rho[v/x]s) \text{ in } F \ \varepsilon\ T[\![S(\tau' \to \tau * \Delta')]\!](S\Delta)\sigma.$$

Now it is enough to prove that for every set Δ'' containning Δ', for every extension σ_1 of σ, for every $v \varepsilon T[\![S\tau']\!](S\Delta'')\sigma_1$ and store $s \varepsilon \Sigma(S\Delta'')\sigma_1$ there is a store typing σ_2 such that $v' \varepsilon T[\![S\tau]\!](S\Delta'')(\sigma_1\sigma_2)$ and $s' \varepsilon \Sigma(S\Delta'')(\sigma_1\sigma_2)$ where $<v', s'> = E[\![e_1]\!]\rho[v/x]s$. Now $\rho[v/x][\![y]\!] \varepsilon T[\![S\eta]\!](S\Delta'')\sigma_1$ for every assumption $y:\eta$ in $A_x \cup \{x:\tau'\}$ and so, by the induction hypothesis, there is a $\sigma_2 \varepsilon (S\Delta'')^*$ which satisfies those requirements. Now assume that α does not occur free in A nor in Δ. To prove that

$$(\lambda vs.E[\![e_1]\!]\rho[v/x]s) \text{ in } F \in \mathrm{T}[\![S[\upsilon/\alpha]\tau'\to\tau^*\Delta']\!](S\Delta)\sigma$$

is enough to repeat the argument above but taking $S[\upsilon/\alpha]$ instead of S since $S[\upsilon/\alpha]\Delta = S\Delta$.

case REC: in this case e is rec $f\ x.e_1$, η is $\tau'\to\tau^*\Delta'$, Δ is $\emptyset$ and the antecedent is $(A_f)_x \cup \{x:\tau', f:\tau'\to\tau\} \vdash e_1:\tau^*\Delta'$. In this case we can prove that the result holds when we take σ' to be the empty store typing by proving, by induction on n and in a similar way as for ABS, that each element of the ω-chain $(G^n(\bot))$, where

$$Gu = \lambda vs.E[\![e_1]\!]\rho[v/x,u/f]\sigma,$$

is in $\mathrm{T}[\![S(\tau'\to\tau^*\Delta')]\!]\emptyset\sigma$ and remembering that the latter is closed under lubs of ω-chains.

case LET: here e is let $x=e_1$ in e_2 and the antecedents are $A \vdash e_1:\eta'^*\Delta$ and $A_x\cup\{x:\eta'\} \vdash e_2:\eta^*\Delta$ for some type scheme η'. By the induction hypothesis there is a store typing $\sigma_1\in(S\Delta)^*$ such that $v_1\in\mathrm{T}[\![S\eta']\!](S\Delta)(\sigma\sigma_1)$ and $s_1\in\Sigma(S\Delta)(\sigma\sigma_1)$ where $<v_1,s_1>=E[\![e_1]\!]\rho s$. If $v_1=\bot$ then $E[\![e]\!]\rho s = <\bot,\bot>$ and the result holds when we take σ to be, e.g., σ_1. Otherwise note that v_1 is distinct from *wrong* and thus $E[\![e]\!]\rho s = E[\![e_2]\!]\rho[v_1/x]s_1$. Now $\rho[v_1/x][\![y]\!]\in\mathrm{T}[\![S\eta_y]\!](S\Delta)(\sigma\sigma_1)$ for every assumption $y:\eta_y$ in $A_x\cup\{x:\eta'\}$ and thus, again by induction, there is a store typing $\sigma_2\in(S\Delta)^*$ such that $v_2\in\mathrm{T}[\![S\eta]\!](S\Delta)(\sigma\sigma_1\sigma_2)$ and $s_2\in\Sigma(S\Delta)(\sigma\sigma_1\sigma_2)$ where $<v_2,s_2> = E[\![e_2]\!]\rho[v_1/x]s_1$ as we wanted. The second part of the result follows also by induction because if α does not occur free in A nor in Δ it also does not occur free in η' since we assume that the derivation is standard.□

The following is an immediate corollary of the previous result.

Theorem 2 (Semantic soundness of type inference). For any expression e, type scheme η, assumptions A and finite set of types Δ if

$$A \vdash e{:}\eta^{*}\Delta$$

holds then

$$A \models e{:}\eta^{*}\Delta$$

also holds.

The following result shows that type inferability is preserved by substitutions and can be proved exactly in the same way as proposition 1 of chapter II.

Proposition 5. If $A \vdash e{:}\eta^{*}\Delta$ then for any substitution S also $SA \vdash e{:}S(\eta^{*}\Delta)$. Moreover if there is a derivation of $A \vdash e{:}\eta^{*}\Delta$ of height n then there is also a derivation of $SA \vdash e{:}S(\eta^{*}\Delta)$ of height less or equal to n.

To conclude this section we will examine a property of type inference which may be of use in practical applications. We first note that the information provided by the sets Δ is not very interesting from a programmer's point of view. We will show that, to a certain extent, we can forget about the types in the sets Δ provided we remember the variables which occur in those types. More precisely:

We define an equivalence relation $\simeq$ between finite sets of types by writing $\Delta\simeq\Delta'$ iff the type variables occurring in Δ are exactly those which occur in Δ'. Note that the equivalence classes under $\approx$ are in one to one correspondence with finite sets of type variables. We next extend $\approx$ to type schemes by writing $\eta\approx\eta'$ iff η is η' or η is $\forall\alpha_1\ldots\alpha_n.\tau'\to\tau^*\Delta$ and η' is $\forall\alpha_1\ldots\alpha_n.\tau'\to\tau^*\Delta'$ with $\Delta\approx\Delta'$. Similarly we extend $\approx$ to sets of assumptions by writing $A\approx A'$ iff for every assumption $x:\eta$ in A there is an assumption $x:\eta'$ in A' with $\eta\approx\eta'$, and reciprocally, for every assumption $x:\eta'$ in A' there is an assumption $x:\eta$ in A such that $\eta'\approx\eta$. The following result states that we can take the quotient of the inference relation $\vdash$ by $\approx$.

Proposition 6. If $A \vdash e:\eta^*\Delta$ and $A\approx A'$ then there are a type scheme $\eta'\approx\eta$ and a set of types $\Delta'\approx\Delta$ such that $A' \vdash e:\eta'^*\Delta'$ also holds.

proof: the proof can be obtained by straightforward induction on the derivation of $A \vdash e:\eta^*\Delta$.□

5. A type assignment algorithm

We will now define a type assignment algorithm similar to the one of chapter II.

To define the algorithm it is convenient to introduce some new notation.

Given a type scheme η, a set of assumptions A and a set of types Δ we will use $\overline{\Delta A}(\eta)$ to denote the closure of η under the set of type variables occurring free in A or in Δ.

Given a nonquantified type scheme θ we will use $Disc(\theta)$ to denote the pair (τ,Δ) obtained from θ as follows

(i) if θ is a type then $\tau=\theta$ and $\Delta=\emptyset$;

(ii) if θ is of the form $\upsilon'\to\upsilon*\Delta'$ then $\tau=\upsilon'\to\upsilon$ and $\Delta=\Delta'$;

The type assignment algorithm takes as arguments an expression e and a set of assumptions A and returns, when it succeeds, a substitution S, a non-quantified type scheme θ and a finite set of types Δ such that $SA \vdash e:\theta*\Delta$.

Definition. $R(A,e)=(S,\theta,\Delta)$ where

(i) If e is x and there is an assumption $x:\forall\alpha_1\ldots\alpha_n.\theta'$ in A then $S=Id$, $\theta=[\beta_i/\alpha_i]\theta'$ and $\Delta=\emptyset$ where the β_i are new type variables.

(ii) If e is e_1e_2 then

let $(S_1,\theta_1,\Delta_1) = R(A,e_1)$

and $(S_2,\theta_2,\Delta_2) = R(S_1A,e_2)$

and $(\tau_1,\Delta_3) = Disc(S_2\theta_1)$

and $(\tau_2,\Delta_4) = Disc(\theta_2)$

and $U = U(\tau_1,\tau_2\to\beta)$ where β is new;

then $S = US_2S_1$, $\theta = U\beta$ and $\Delta = U(S_2\Delta_1\cup\Delta_2\cup\Delta_3\cup\Delta_4)$.

(iii) If e is $\lambda x.e'$ then let β be a new type variable

and $(S',\theta',\Delta') = R(A_x\cup\{x:\beta\},e')$

and $(\tau,\Delta'') = Disc(\theta')$;

then $S = S'$, $\theta = (S'\beta)\to\tau*(\Delta'\cup\Delta'')$ and $\Delta = \emptyset$.

(iv) If e is rec $f\ x{\cdot}e'$ then let α and β be new type variables

and $(S',\theta',\Delta') = \mathrm{R}((A_x)_f \cup \{f:\beta\to\alpha, x:\beta\}, e')$

and $(\tau,\Delta'') = Disc(\theta')$

and $U = \mathrm{U}(\tau, S'\alpha)$;

then $S = US'$, $\theta = US'(\beta\to\alpha^*(\Delta'\cup\Delta''))$ and $\Delta = \emptyset$.

(v) If e is let $x{=}e_1$ in e_2

let $(S_1,\theta_1,\Delta_1) = \mathrm{R}(A,e_1)$

and $(S_2,\theta_2,\Delta_2) = \mathrm{R}(S_1A_x \cup \{x:\overline{\Delta_1S_1A}(\theta_1)\}, e_2)$;

then $S = S_2S_1$, $\theta = \theta_2$ and $\Delta = (S_2\Delta_1)\cup\Delta_2$.

note: when one of the conditions above is not met R fails.

The two following results can be proved in the same way as the similar results of chapter II.

Proposition 7 (Soundness of R). If $\mathrm{R}(A,e)$ succeeds with (S,θ,Δ) then there is a derivation of $SA \vdash e:\theta^*\Delta$.

Proposition 8 (Completeness of R). Given A and e let A' be an instance of A, η be a type scheme and Δ' be a finite set of types such that

$$A' \vdash e:\eta^*\Delta'.$$

Then

(i) $\mathrm{R}(A,e)$ succeeds.

(ii) If $\mathrm{R}(A,e) = (S,\theta,\Delta)$ then, for some substitution R,

$$A' = RSA \quad \text{and} \quad R(\overline{\Delta SA}(\theta)^{*}\Delta) > \eta^{*}\Delta'.$$

Finally we note that R can be adapted to compute type module $\simeq$ by treating the sets Δ as sets of type variables and taking $S\Delta$ to denote the set of type variables which occur in $S\alpha$ for some α in Δ.

6. Weak polymorphism and programming examples.

The typechecker of the ML system was modified by the author to handle references. Similar extensions were also included in the ML implementation for the VAX by L.Cardelli. However those implementations use a notation for functional type schemes which is different from the one used in this chapter. That notation, which is very similar to the one used in an early proposal to extend ML to handle references[Gordon 79], is based on the introduction of a second class of type variables called *weak variables*. To explain this designation consider two functions like

$$\mathit{newref}:\forall\alpha.\alpha \rightarrow \alpha\ \mathbf{ref} * \{\alpha\}$$

$$f:\forall\alpha.\alpha \rightarrow \alpha.$$

Now if we apply each of those functions to the null list we can infer the type scheme $\forall\alpha.\alpha$ list for $f[\,]$ but, although we can infer α list ref for *newref*$[\,]$ for every type variable α, we can not generalize this to $\forall\alpha.\alpha$ list ref. In this sense the α in the type scheme for *newref* is weaker then the α in the type scheme for f. In a similar sense we can say that the polymorphism of *newref* is weaker then that of f because if we apply f to a polymorphic object like [] we obtain a polymorphic result while for *newref* this is not true.

Returning to the notation used by the implementations referred above, and remembering what was mentioned about the possibility of replacing the sets of types Δ with sets of type variables, what is done is to represent a functional type scheme like $\forall\alpha\beta.\alpha\to\beta\to\alpha*\{\alpha\}$ as $\forall\underline{\alpha}\beta.\underline{\alpha}\to\beta\to\underline{\alpha}$ where we have underlined α to mean it is weak. Finally we note that even though this does not define a one to one correspondence because in a type scheme like $\tau'\to\tau*\Delta$ there might be a variable in Δ which does not occur in $\tau'\to\tau$ the fact remains that the type assignment algorithm only attempts to generalize on the variables which occur in $\tau'\to\tau$ and so we can discard those cases.

We will now present some examples of the extensions to the ML type discipline to handle references which are supported by the results of this chapter. In this examples *update* and *contents* are replaced with infix := and prefix @ and, contrary to both the implementations, we have explicitly quantified generic type variables.

```
% The following declaration results in a non-polymorphic
  type assignment                                           %

# let x=newref [];;
x = - : α list ref

% The following assignment causes α to be instantiated to int %

# x := 1.@x;;
- : int list ref
```

```
% as can be checked with %

# x;;
- : int list ref

% One of the consequences of weak polymorphism is that
  some expressions are assigned a non-polymorphic type
  even when this would be semantically valid                %

# map newref;;
- : α̲1 list → α̲1 ref list

% compare however with                                      %

# λl.map newref l;;
- : ∀α̲.α̲ list → α̲ ref list

%  The reason for the  difference  between the types assigned
   to the two  expressions  above  is due to the fact that the
   λ-abstraction on the  second expression ensures that no
   reference results from its  evaluation,  while  in
   the first expression the  information  conveyed by the type
   of map  is not  enough to guarantee  that it does not apply
   newref to some argument                                  %

%  the following function swaps the contents of two references %

# let swap r r' = let t=@r in r := @r'; r' := t;;
swap = - : ∀α.α ref → α ref → α ref

% the following is a definition of arrays as an abstract
  data type                                                 %

# abstype  α array = (α ref list) # int # int
# with
#     newarray(l,lb,ub) =
#             if length l = (ub-lb)+1
#             then absarray(map newref l,lb,ub)
#             else failwith `newarray`
# and
#     select a n =
#        let l, lb, ub = reparray a
#        in if n>ub or n<lb
#           then failwith `select`
#           else el(n+1-lb) l
#                wherec el n l = n=1 => hd l | el (n-1) (tl l)
# and
#     lwb = fst o snd o reparray
# and
#     upb = snd o snd o reparray
# and
#     arraytolist = (map $@) o fst o reparray;;
newarray = - : ∀α.((α list)#int*int) → α array
select = - : ∀α.α array → int → α ref
lwb = - : ∀α.α array → int
upb = - : ∀α.α array → int
arraytolist = - : ∀α.α array → α list


% using this definition for arrays one can define a function
  to sort arrays as follows                                  %

# let sort a pred =
#    let changes = newref true
#    and limit = newref upb(a)
#    in if @changes
#       loop (  let i = newref (lwb a)
#               in changes := false;
#                  if @i<@limit
#                  loop
#                    (  if not(pred @(select a @i) @(select a (@i+1)))
#                       then ( swap (select a @i) (select a (@i+1));
#                              changes := true
#                            )
#                       i := @i + 1
#                       )
#                     limit := @limit-1;
#               );;
sort = - :∀α.(α array) → (α → α → bool) → .

% we can now create an array and sort it                    %

# let a = newarray([3;7;4;6],1,4);;
a = - : int array

# sort a $<;;
() : .

# arraytolist a;;
[3;4;6;7] : int list

%  we will now define an abstract data type modelling
   updatable binary trees                                    %

# absrectype  α ubtree = . + α#(α ubtree ref)#(α ubtree ref)
# with emptytree = absubtree o inl
# and  nulltree = isl o repubtree
# and  mktree(tag,lson,rson) =
#                absubtree o inr (tag,newref lson,newref rson)
# and  tag t = isl o repubtree t => failwith `tag`
#                                   | fst o outr o repubtree t
# and  lson t = isl o repubtree t => failwith `lson`
#                                    | fst o snd o outr o repubtree
# and  rson t = isl o repubtree t => failwith `rson`
#                                    | snd o snd o outr o repubtree;;
emptytree = - : ∀α.α ubtree
nulltree = - : ∀α.α ubtree → bool
mktree = - : ∀α.α#(α ubtree)*(α ubtree) → α ubtree
tag = - : ∀α.α ubtree → α
lson = - : ∀α.α ubtree → (α ubtree ref)
rson = - : ∀α.α ubtree → (α ubtree ref)
```

```
% using the above definition for updatable binary trees we
  will now define, as an abstract data type, a  dictionary
  organized as a binary tree                                    %

# abstype (α,β)dic = (α*(β ref))ubtree ref # (α → α → bool)
# with  mkdic pred = absdic (newref emptytree, pred)
# and accessdic fn dic id =
#    let root,pred = repdic dic  in
#       letrec scan t =
#           if nulltree @t
#           then ( let ir = fn() in
#                      t:=mktree((id,ir), emptytree, emptytree);
#                      ir  )
#           if fst o tag (@t) = id then snd o tag (@t)
#           if pred (fst o tag (@t)) id
#           then scan ( rson @t)
#           else scan ( lson @t);;
mkdic = - : ∀α∀β.(α → α → bool) → (α,β) dic
accessdic = - : ∀α∀β (. → β ref) → (α,β) dic → α → β ref

% we can now define functions to search and insert new  ids in
  the dictionary                                                 %

# let search = accessdic (λ().failwith `search`)
# and insert dic id info =
#         (accessdic (λ().newref info) dic id) := info;;
search = - : ∀α∀β (α,β) dic → α → β ref
insert = - : ∀α∀β (α,β) dic → α → β → β ref
```

CHAPTER IV

CHAPTER IV

Conclusions and directions for further study

The main objective of this dissertation as set out in the intoduction was to complete and extend the work of R.Milner on polymorphic type assignment in programming. Here we examine the extent to which that objective was attained and also call the readers attention to some aspects which we think deserve a deeper study.

The type inference system studied in chapter 1 can be seen as an extension of the system of R.Milner to handle overloading. Apart from that it is interesting in itself because it provides a conceptually much more simple framework in which the theory of type assignment can be studied. One of the interesting points made in that chapter is not only that typechecking can be supported by type assignment theory but also that derivation trees and hence type assignment itself can be used to formulate and justify the program transformations which need to be done by compilers to support overloading. We believe this use of derivation trees can be also made in relation with other possible extensions to the language such as those involving run-time type manipulation. We should also point out that we have only presented a partial solution to the problem of handling user defined overloading and we think the way to overcome the limitation imposed is to design an algorithm that, like the one of chapter II, will go directly from a set of assumption schemes to the set of type schemes that can be inferred for the expression.

The reformulation of Milner's system presented in Chapter II provides a better insight on its features and in particular allowed us to give an affirmative answer to the questions left open in his work concerning the existence of principal types and the completeness of the type assignment algorithm.

Chapter III shows that type assignment and type polymorphism need not be confined to purely applicative languages. From a practical point of view it also provides a type discipline which is sufficiently powerful to handle a large class of programs involving references as can be judged by the examples provided there.

Nevertheless we should point out that many of the features of that system were forced on us by technical considerations such as the existence of a semantics for types and of a type assignment algorithm. This may explain why we were unable to do for this system what chapter I does for Milner's system. One of the consequences of this fact is that the question of how to handle references in the presence of overloading is still left open, at least from a theoretical point of view.

On the other hand we also should point that the technical problems associated with the definition of a semantics for types have nothing to do with polymorphism and would arise even if only monotypes were involved, so the methods and models presented there are also relevant to the study of the properties of typechecking for non purely applicative programming languages using the techniques of denotational semantics.

Finally the notion of weak polymorphism and the syntactical mechanisms used to enforce it, combined with program transformations based on derivation trees, seem to be promising in handling other extensions to the language such as those involving run-time type manipulation where those mechanisms could be used to decide what types needed to be dynamically passed to routines.

References

[Coppo 80]

M. Coppo
An Extended Polymorphic Type System for Applicative Languages.
Lecture Notes in Computer Science, vol. 88, pp 194-204
Springer-Verlag, 1980.

[Coppo *et al* 80]

M. Coppo, M. Dezani-Ciancaglini and B.Venneri
Principal Type Schemes and Calculus Semantics.
in *To H.B. Curry, Essays on Combinatory Logic, Lambda-Calculus and Formalism.*
eds. R.Hindley and J.P.Seldin
Academic Press, 1980.

[Curry 69]

H.B. Curry
Modified Functionality in Combinatory Logic.
Dialetica 23, pp.83-92, 1969.

[Curry & Feys 58]

H.B. Curry and R. Feys
Combinatory Logic, Vol. I
North Holland Co., 1958.

[Damas & Milner 82]

L. Damas and R. Milner
Principal Type-Schemes for Functional Programs.
Ninth Annual ACM Symposium on Principles of Programming Languages
Albuquerque NM, pp 207-212, 1982.

[Demers and Donahue 79]

A. Demers and J. Donahue
Revised Report on the Programming Language Russel.
TR79-389, Computer Science Departement
Cornell University, 1979.

[Gordon 79]

M. Gordon
Locations as First Class Objects in ML.
Unpublished, 1979.

[Gordon *et al* 79]

M. Gordon, R. Milner and C. Wadsworth
Edinburgh LCF.
Lecture Notes in Computer Science, Vol. 78
Springer-Verlag, 1979.

[Hindley 69]

R. Hindley
The Principal Type-Scheme of an Object in Combinatory Logic.
Trans AMS 146, pp 29-60, 1969.

[Liskov & Snyder 77]

B. Liskov and A. Snyder
Abstraction Mechanisms in CLU.
Comm. ACM 20, 3, pp. 564-583, 1977.

[McCracken 79]

N. McCracken
An Investigation of a Programming Language with a Polymorphic Type Structure.
PhD thesis, Syracuse University, June, 1979.

[Milne 75]

R. Milne
The Formal Semantics of Computer Languages and their Implementation.
PhD thesis, Programming Research Group.
University of Oxford, 1975.

[Milne & Strachey 76]

R. Milne and C. Strachey
A Theory of Programming Languages Semantics.
Chapman and Hall, London, 1976.

[Milner 78]

R. Milner
A Theory of Type Polymorphism in Programming.
J. of Comp. and Syst. Science 17, 3,pp 348-375, 1978.

[Plotkin 76]

G. Plotkin
A Power Domain Construction.
SIAM J. of Comp. 5, No.3 pp.452-487, 1976.

[Reynolds 74]

J.C. Reynolds
Towards a Theory of Type Structure.
Lecture Notes in Computer Science, vol. 19, pp. 408-425
Springer-Verlag, 1974.

[Reynolds 74a]

J.C. Reynolds
On the Relation between Direct and Continuation Semantics.
In J.Loeckx, editor, *Proc. 2nd Colloquion on Automata, Languages and Programming*, pp. 141-156
Gaarbrucken, 1974.

[Robinson 65]

J.A. Robinson

*A Machine-oriented Logic b*ɔ*sed on the Resolution Principle.*

JACM 12, 1, pp. 23-41, 1965.

[Shamir & Wadge 77]

A. Shamir and W. Wadge

Data Types as Objects.

Lecture Notes in Computer Science, Vol. 52 pp. 465-479

Springer-Verlag, 1977.

[Shultis 82]

J. Shultis

Type Checking in Exp: An Algebraic Approach.

CS/E-82-03, Oregon Graduate Center, 1982.

[Smyth & Plotkin 78]

M.B. Smyth and G.D. Plotkin

The Category-Theoretic Solution of Recursive Domain equations.

DAI report no. 60, Department of Artificial Intelegence

University of Edimburgh, 1978.

[Stoy 77]

J. Stoy

Denotational Semantics: The Scott-Strachey approach to Programming Language Theory.

MIT Press, 1977.

[Wijngaarden *et al* 75]

A. Wijngaarden *et al*

Revised Report on the Algorithmic Language ALGOL 68.

Acta Informatica, 5 pp. 1-236, 1975.

[Wulf *et al* 76]

A. Wulf, R.L. London and M. Shaw
Abstraction and Verification in ALPHARD:
Introduction to the Language and Methodology.
ISR/RR-76-46, University of California, 1976.

[Yelles 79]

C.B. Yelles
Type Assignment in the Lambda-Calculus:
Syntax and Semantics.
PhD thesis, University of Wales, 1979.

APPENDIX I

The purpose of this appendix is to present in more detail the procedure outlined in the last paragraph of section 3 of chapter III for defining a lifting $\tilde{F}$ of the endofunctor F associated with the domain equation for V.

We start by making the isomorphism between $V \to_{\perp} \tilde{D}$ and $\mathrm{Cts} \to \mathrm{FP}(\mathrm{Mty}) \to \mathrm{Sty} \to \overline{V}$ explicit.

Lemma 1. For any cpo V the map which sends each $p : V \to_{\perp} \tilde{D}$ to $\hat{p} : \mathrm{Cts} \to \mathrm{FP}(\mathrm{Mty}) \to \mathrm{Sty} \to \overline{V}$ defined by

$$\hat{p} [\![\eta]\!] \Delta\sigma = \{ v \in V \mid pv\,\eta\Delta\sigma \sqsubseteq \mathrm{true} \}$$

is an isomorphism with inverse given by

$$\check{q} v\,\eta\Delta\sigma = v\,\varepsilon q [\![\eta]\!] \Delta\sigma \to \mathrm{true},\ \mathrm{false}$$

for any q in $\mathrm{Cts} \to \mathrm{FP}(\mathrm{Mty}) \to \mathrm{Sty} \to \overline{V}$.

proof: to see that $\hat{p}$ is well defined we note that since p is strict and continuous the set $\{ v \in V \mid pv\,\eta\Delta\sigma \sqsubseteq \mathrm{true} \}$ is a nonempty ideal of V. Further for any $v \in V$, η, $\Delta' \supseteq \Delta$ and $\sigma < \sigma'$ we have by the assumptions on p, $pv\eta\Delta'\sigma' \sqsubseteq pv\,\eta\Delta\sigma$ and so if $v \in \hat{p} [\![\eta]\!] \Delta\sigma$ we must also have $v \in \hat{p} [\![\eta]\!] \Delta'\sigma'$ which shows that $\hat{p}$ also satisfies the required monotonicty requirements on Δ and σ The remainder of the proof follows along similar lines which we will omit here ▣

Using the above isomorphism we can now define how $\tilde{F}$ acts on objects of $(\mathrm{CPO}_{\perp} \downarrow \tilde{D})^{\mathcal{E}}$.

For any object $p:V\to\overset{\approx}{D}$ of $(CPO_{\perp}\downarrow\tilde{D})^{E}$ we take $\tilde{F}(p)$ to be $\check{q}:F(V)\to\tilde{D}$ where $q:Cts\to FP(Mty)\to Sty\to\overline{F(V)}$ is defined by

$$q\llbracket \iota\rrbracket\Delta\sigma = \{b \text{ in } F(V) \mid b\in \eqsim \llbracket \iota \rrbracket \}$$

$$q\llbracket \tau ref\rrbracket\Delta\sigma = \{ l \text{ in } F(V) \mid l\varepsilon L \text{ and } \sigma_{l}= \tau \} \cup \{\perp\}$$

$$q\llbracket \tau'\to\tau\rrbracket\Delta\sigma =$$

$$\bigcap_{\Delta'\supseteq\Delta}\ \bigcap_{\sigma'>\sigma} \{ f \text{ in } F(V) \mid f:V\to V^{*}\to V\times V^{*} \text{ s.t.}$$
$$\forall v\varepsilon\hat{p}\llbracket \tau' \rrbracket\Delta'\sigma' \ \ \forall\ s\varepsilon\Sigma_{p}\Delta'\sigma' \ \ \exists\sigma''\varepsilon\Delta^{*} \quad \text{s.t.}$$
$$v'\varepsilon\hat{p}\llbracket \tau \rrbracket\Delta'(\sigma'\sigma'') \text{ and } s'\varepsilon\Sigma_{p}\Delta'(\sigma'\sigma'')$$
$$\text{where } \langle v',s'\rangle = fvs \ \}$$

$$q\llbracket \tau'\to\tau*\Delta' \rrbracket\Delta\sigma = q\llbracket \tau'\to\tau \rrbracket(\Delta\cup\Delta')\sigma$$

$$q\llbracket \forall\alpha\eta \rrbracket\Delta\sigma = \bigcap_{\tau\varepsilon Mty} q\llbracket [\tau/\alpha]\eta \rrbracket\Delta\sigma$$

where

$$\Sigma_{p}\Delta\sigma = \{s\varepsilon V^{*} \mid |s|=|\sigma| \text{ and } \forall l\leq|s| \ s_{l}\varepsilon\ \hat{p}\llbracket \sigma_{l}\rrbracket\Delta\sigma \} \cup \{\perp\}.$$

One can easily prove that the above definition is sound by using exactly the same arguments as in pages 100-101.

Finally it only remains to define $\tilde{F}$ on arrows of $(CPO_{\perp}\downarrow\tilde{D})^{E}$. In fact given an arrow $\langle p_{1}:V_{1}\to\tilde{D},\ p_{2}:V_{2}\to\tilde{D},\ f:V_{1}\to V_{2}\rangle$ we take its image by $\tilde{F}$ to be $\langle\tilde{F}(p_{1}):F(V_{1})\to\tilde{D},\ \tilde{F}(p_{2}):F(V_{2})\to\tilde{D},\ F(f):F(V_{1})\to F(V_{2})\rangle$.

Now in order that the above definition is sound it is enough that $\tilde{F}(p_{2})\circ F(f)\subseteq\tilde{F}(p_{1})$ and that $\tilde{F}(p_{1})\circ F(f^{R})\subseteq\tilde{F}(p_{2})$. Since the two inequalit can be proved in exactly the same way we will only show how to prove the first one.

Now to prove $\tilde{F}(p_2)\circ F(f) \subseteq \tilde{F}(p_1)$ it is enough to show that for all $\eta,\Delta,$ σ and $u\varepsilon F(V_1)$ we have

$$u\,\varepsilon\,\widehat{F(\wp_1)}[\![\,\eta\,]\!]\Delta\sigma \quad\Rightarrow\quad F(f)(u)\ \varepsilon\ \widehat{F(p_2)}[\![\,\eta\,]\!]\Delta\sigma\ .$$

which can be done by structural induction on η. Since the only case whic presents some degree of difficulty is the one where η is a function typ $\tau'\to\tau$ this is the only one we will examine here.

Now if $u\,\varepsilon\,\widehat{F(p_1)}[\![\,\tau'\to\tau\,]\!]\Delta\sigma$ then either $u=\bot$ in which case the resul follows immediately or u is a map in $V_1\to V_1^*\to V_1\times V_1^*$ in which case $u_2=F(f)$ is the map in $V_2\to V_2^*\to V_2\times V_2^*$ defined by

$$u_2v_2s_2 \equiv (f\times f^*)\circ u\ (f^Rv_2)(f^{R^*}s_2)$$

where f^* denotes the natural extension of f to $V_1^*\to V_2^*$.

Now for any $\Delta'\supseteq\Delta$ and $\sigma'>\sigma$ assume

$$v_2\varepsilon\ \hat{p}_2[\![\,\tau'\,]\!]\Delta'\sigma' \text{ and } s_2\varepsilon\Sigma_{p_2}\Delta'\sigma'.$$

Now since $p_1\circ f^R \subseteq p_2$ by hypothesis we have $f^Rv_2\varepsilon\ \hat{p}_1[\![\,\tau'\,]\!]\Delta'\sigma'$ and similarly $f^{R^*}s_2\varepsilon\ \Sigma_{p_1}\Delta'\sigma'$, but then, if we let $<v',s'> = u(f^Rv_2)(f^{R^*}s_2)$ there is a $\sigma''\varepsilon\Delta^*$ such that $v'\varepsilon\hat{p}_1[\![\tau\,]\!]\Delta'(\sigma'\sigma'')$ and $s'\varepsilon\Sigma_{p_1}\Delta'(\sigma'\sigma'')$. Now, since $p_2\circ f\subseteq p_1$, we have $fv'\varepsilon\hat{p}_2[\![\tau]\!]\Delta'(\sigma'\sigma'')$ and similarly $f^*s'\varepsilon\Sigma_{p_2}\Delta'(\sigma'\sigma'')$ which proves F(f)(u) is in $\widehat{F(p_2)}[\![\tau'\to\tau]\!]\Delta\sigma$ as we wanted.