JOURNAL OF COMPUTER AND SYSTEM SCIENCES 17, 348–375 (1978)

A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types, entails defining procedures which work well on objects of a wide variety. We present a formal type discipline for such polymorphic procedures in the context of a simple programming language, and a compile time type-checking algorithm $\mathscr{W}$ which enforces the discipline. A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong" and a Syntactic Soundness Theorem states that if $\mathscr{W}$ accepts a program then it is well typed. We also discuss extending these results to richer languages; a type-checking algorithm based on $\mathscr{W}$ is in fact already implemented and working, for the metalanguage ML in the Edinburgh LCF system.

1. INTRODUCTION

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types (LISP is a perfect example), entails defining procedures which work well on objects of a wide variety (e.g., on lists of atoms, integers, or lists). Such flexibility is almost essential in this style of programming; unfortunately one often pays a price for it in the time taken to find rather inscrutable bugs—anyone who mistakenly applies CDR to an atom in LISP, and finds himself absurdly adding a property list to an integer, will know the symptoms. On the other hand a type discipline such as that of ALGOL 68 [22] which precludes the flexibility mentioned above, also precludes the programming style which we are talking about. ALGOL 60 was more flexible—in that it required procedure parameters to be specified only as "*procedure*" (rather than say "*integer to real procedure*") —but the flexibility was not uniform, and not sufficient.

An early discussion of such flexibility can be found in Strachey [19], who was probably the first to call it polymorphism. In fact he qualified it as "parametric" polymorphism, in contrast to what he called "adhoc" polymorphism. An example of the latter is the use of "+" to denote both integer and real addition (in fact it may be further extended to denote complex addition, vector addition, etc.); this use of an identifier at several distinct types is often now called "overloading," and we are not concerned with it in this paper.

In this paper then, we present and justify one method of gaining type flexibility, but also retaining a discipline which ensures robust programs. We have evidence that this

0022-0000/78/0173-0348$02.00/0
Copyright © 1978 by Academic Press, Inc.
All rights of reproduction in any form reserved.

work is not just a theoretical exercise; the polymorphic type discipline which we discuss here has been incorporated in the LCF metalanguage ML [2, 3], and has been in use for nearly 2 years. The compile-time type checker for this language has proved to be a valuable filter which traps a significant proportion of programming errors.

The main body of the present paper is concerned with a technical account—both semantic and syntactic—of our discipline of types in the context of a simple illustrative language, but at this point it is helpful to characterize the approach informally. We outline its predominant features.

First, everything concerning types is done at compile time; once the type checker (part of the compiler) has accepted a program or program phrase, code may be generated which assumes that no objects carry their types at run-time. This is widely accepted as yielding efficient object code, though it does impose constraints on the use of types compared with, for example, the approach in EL1 [21].

Second, many nontrivial programs can avoid mentioning types entirely, since they be inferred from context. (In ML however, as in other languages, the user may—indeed often should—define his own types together with operations over these types. Recent languages which allow the user to define his own types in this manner are CLU [8], ALPHARD [23] and Euclid [6]). Although it can be argued convincingly that to demand type specification for declared variables, including the formal parameters of procedures, leads to more intelligible problems, it is also convenient—particularly in on-line programming—to be able to leave out these specifications. In any case, the type checker which we present is quite simple and could not be made much simpler even if the types of variables were always specified in declarations.

Third, polymorphism plays a leading role. For example, a procedure is assigned a polymorphic type (which we abbreviate to *polytype*) in general; only when the types of its arguments and result can be uniquely determined from the context is it monomorphic (i.e., assigned a *monotype*). Gries and Gehani [4], among others, have made a convincing case for controlled polymorphic programming (in contrast with the typeless programming in LISP or in SNOBOL); for them however, and also for Tennent [20], the presence of type variables or identifiers is needed to specify polymorphic types. For us, the polymorphism present in a program is a natural outgrowth of the primitive polymorphic operators which appear to exist in every programming language; such operators are assignment, function application, pairing and tupling, and list-processing operators. It is principally the type constraints on these operators, and in the declaration and use of variables, which determine for us the types of a program phrase and its subphrases.

We do not discuss in this paper—except briefly at the end—either coercions or the "overloading" of identifiers. Our view is that these concepts, and also run-time type manipulation, are somewhat orthogonal to a compile-time polymorphic type discipline, and may (to some extent) be incorporated without invalidating it.

In Section 2 we illustrate our type discipline by examples in a fragment of ML. This fragment should be self-explanatory, but an outline of ML is given in [3] and a full description appears in [2]. These illustrations should serve to make the point that we are able to handle useful languages. The remainder of the paper justifies the discipline using a very simple applicative language, Exp. The justification factors into two parts.

In Section 3 we define the notion of *well typing* (correct type assignment) and prove the Semantic Soundness Theorem, which says that a well-typed program is semantically free of type violation. If we were to give an operational definition of the language, this would imply that, for example, an integer is never added to a truth value or applied to an argument, and consequently need not carry its type around for run-time checking. In Section 4 we present a well-type algorithm $\mathscr{W}$ and prove the Syntactic Soundness Theorem, which states that $\mathscr{W}$, if it succeeds, produces a well typing of a program. We also give a more efficient algorithm $\mathscr{J}$, which simulates $\mathscr{W}$.

The types in Exp are just the hierarchy of purely functional types over a set of basic types. That is, the polymorphism in Exp is the natural outgrowth of a single primitive polymorphic operator, function application, together with variable binding. To add other primitive polymorphic operators, such as pairing and list-processing operators (as in ML), together with types built from basic ones $\times$ (Cartesian Product), *list* (list-forming), and $+$ (disjoint sum) in addition to (function type), presents no extra difficulty in the two soundness theorems. Indeed, adding an assignment operator is also easy as far as the Syntactic Soundness Theorem is concerned, but the Semantic Soundness Theorem is harder to extend in this case, due to the extra semantic complication of a memory or store which holds the current values of assignable variables. We discuss this further in Section 5.

Our work is a step towards solving the problem expressed by Morris [10] in his thesis as follows: "to design a language and a type system in which a programmer may define functions whose parameters may have different types for different calls of the function." We recommend Chapter 4 of this thesis as a lucid introduction to the problem. Although Morris does not discuss the semantics of types formally, or give a polymorphic type system, he describes how a valid type assignment may be found for a term of the λ-calculus by solving a set of simultaneous linear equations; we take this idea further in the next section.

After doing this work we became aware of Hindley's [5] method for deriving the "principal type scheme" (which is what we call a polymorphic type) for a term in combinatory logic. Hindley appears to have been the first to notice that the Unification Algorithm of Robinson [14] is appropriate to this problem. Our work can be regarded as an extension of Hindley's method to programming languages with local declarations, and as a semantic justification of the method.

In summary, we present a polymorphic type discipline which is syntactically well understood and justified for a currently used programming language with imperative features, and is also semantically explained for a nontrivial, though nonimperative, sublanguage.

2. Illustrations of the Type Discipline

We illustrate our notion of polymorphism by means of some simple examples. They are written in a fragment of ML which we hope is self-explanatory; this fragment is indeed no more than Landins ISWIM [7], and we refer the reader to Burge's book [1] in

which he uses this style of programming almost exactly. We use no imperative constructs here (assignments or jumps). The constructs

$$\begin{aligned} &\textit{let } x = e \textit{ in } e', \\ &\textit{let } f(x_1, \ldots, x_n) = e \textit{ in } e' \end{aligned}$$

are used to give x the value of e, and to give f the value of the abstraction $\lambda(x_1, \ldots, x_n) \cdot e$, throughout e'. For recursive functions *letrec* is used in place of *let*, and when the part *in* e' is omitted we have a declaration.

The fully determined types (i.e., the monotypes) of ML are built from a set of basic types (*int*, *bool*, etc.) by the binary infixed operators $\times$ (Cartesian product), $+$ (disjoint sum) and $\rightarrow$ (function type), and the unary postfixed operator *list*. Polymorphic types (polytypes) are obtained by admitting *type variables*, which here are represented by $\alpha, \beta, \gamma \cdots$. We represent arbitrary types by ρ, σ, τ. For this section we leave the meaning of types to the reader's intuition; it is made precise in the next section.

EXAMPLE 1. Mapping a function over a list.

$$\begin{aligned} \textit{letrec } \mathrm{map}(f, m) = \textit{if } &\mathrm{null}\,(m) \textit{ then } \mathrm{nil} \\ &\textit{else } \mathrm{cons}\,(f(\textit{hd}(m)), \mathrm{map}\,(f, \textit{tl}(m))). \end{aligned}$$

Intuitively, the function map so declared takes a function from things of one sort to things of another sort, and a list of things of the first sort, and produces a list of things of the second sort. So we say that map has type

$$((\alpha \rightarrow \beta) \times \alpha\, \textit{list}) \rightarrow \beta\, \textit{list},$$

where α, β are type variables.

How is this type determined from the bare declaration of map? First, the *generic* types (we discuss "generic" later) of the identifiers occurring free in the declaration are given by

$$\begin{aligned} \mathrm{null}&: \alpha\, \textit{list} \rightarrow \textit{bool}, \\ \mathrm{nil}&: \alpha\, \textit{list}, \\ \mathrm{hd}&: \alpha\, \textit{list} \rightarrow \alpha, \\ \mathrm{tl}&: \alpha\, \textit{list} \rightarrow \alpha\, \textit{list}, \\ \mathrm{cons}&: (\alpha \times \alpha\, \textit{list}) \rightarrow \alpha\, \textit{list}, \end{aligned}$$

that is, they are polymorphic, because their types contain one or more type variables, and our rule is: To every occurrence of such an identifier is assigned a type which is a substitution instance (substituting types for type variables) of its generic type.

Now each of these identifiers occurs just once in the declaration, so if we denote by σ_{id} the type assigned to an identifier id we must have for some types $\tau_1, \ldots, \tau_5$,

$$\begin{aligned} \sigma_{\mathrm{null}} &= \tau_1\, \textit{list} \rightarrow \textit{bool}, \\ \sigma_{\mathrm{nil}} &= \tau_2\, \textit{list}, \\ \sigma_{\mathrm{hd}} &= \tau_3\, \textit{list} \rightarrow \tau_3, \\ \sigma_{\mathrm{tl}} &= \tau_4\, \textit{list} \rightarrow \tau_4\, \textit{list}, \\ \sigma_{\mathrm{cons}} &= (\tau_5 \times \tau_5\, \textit{list}) \rightarrow \tau_5\, \textit{list}. \end{aligned}$$

The other identifiers (map, f, m) each occur more than once, and our rules demand that each occurrence is assigned the same type. The rules also demand that the following equations are satisfied for some types $\rho_1, \rho_2, \ldots,$

$$\begin{aligned}
\sigma_{\text{map}} &= \sigma_{\text{f}} \times \sigma_{\text{m}} \rightarrow \rho_1, \\
\sigma_{\text{null}} &= \sigma_{\text{m}} \rightarrow \textit{bool}, \\
\sigma_{\text{hd}} &= \sigma_{\text{m}} \rightarrow \rho_2, \\
\sigma_{\text{tl}} &= \sigma_{\text{m}} \rightarrow \rho_3, \\
\sigma_{\text{f}} &= \rho_2 \rightarrow \rho_4, \\
\sigma_{\text{map}} &= \sigma_{\text{f}} \times \rho_3 \rightarrow \rho_5, \\
\sigma_{\text{cons}} &= \rho_4 \times \rho_5 \rightarrow \rho_6, \\
\rho_1 &= \sigma_{\text{nil}} = \rho_6 .
\end{aligned}$$

The first of these conditions relates the type of a function to those of its formal parameters; each of the other conditions arises from some subterm which is a function application, except the last, which is because a conditional expression has the same type as its two arms, and because the definiens and definiendum of a declaration have the same type.

Now these equations may be solved for the variables ρ_i, τ_i, and σ_{id}; Morris [10] discusses the solution of such equations. Indeed, the situation is entirely appropriate for the use of the Unification Algorithm of Robinson [14]; our well-typing algorithm is based upon this algorithm, and (since in this case nothing more than unification is needed) we may conclude from Robinson's work that the most general type of map is obtained, i.e., any other type σ_{map} which satisfies the equations must be a substitution instance of the type obtained. In fact, the solution of the above equations is

$$\sigma_{\text{map}} = (\gamma \rightarrow \delta) \times \gamma\ \textit{list} \rightarrow \delta\ \textit{list},$$

where γ, δ are any distinct type variables. So this is the *generic* type of map, that is, to any occurrence of map within the scope of this declaration must be assigned some substitution instance of this type.

These instances need not be the same. Suppose that *tok* is a basic type (a token being a sequence of characters) and that we have available the identifiers (with their types)

tokl: *tok list* (a variable),

and length: *tok* → *int*,

sqroot: *int* → *real*, two obvious functions.

Then in the expression

map(sqroot, map(length, tokl))

the two occurrence of map will have types

$$((\textit{tok} \rightarrow \textit{int}) \times \textit{tok list}) \rightarrow \textit{int list},$$
$$((\textit{int} \rightarrow \textit{real}) \times \textit{int list}) \rightarrow \textit{real list}.$$

Similarly, if null, for example, had occurred twice in the definition of map, its types could have been different instances of

$$\alpha\ list \rightarrow bool$$

but our rules demand that different occurrences of a formal parameter (f, for example), or of an identifier (map) being recursively defined, shall have the same type.

In passing, note that the occurrences of map mentioned above can be regarded as uses of two separately declared (and monomorphic!) map functions, which differ only in that different types are explicitly provided for their arguments and results. As Gries and Gehani remark, the compiler could be given the task of generating these distinct declarations—or more accurately (since the programmer need not see the replication or even be aware of it), the task of generating different code for the body of the map function for use at distinct types. This would indeed be necessary in the above example if, for efficiency, token lists were implemented differently from integer lists (and the primitive polymorphic functions hd, tl etc., were correspondingly different). We are concerned with a conceptual framework in which these map functions may all be regarded semantically as the same object; then the implementor is left with the freedom to implement as few or many variants as he wishes.

It is clear from our example that the rules of typing need to be carefully framed. We leave this task until the next section, but here we look at one more example to illustrate what happens when *let* or *letrec* is used locally.

EXAMPLE 2. Tagging. Suppose we want a function tagpair, such that tagpair (a) is a function under which

$$(b, c) \mapsto ((a, b), (a, c)).$$

Of course, we can easily write

$$let\ \text{tagpair}(a) = \lambda(b, c) \cdot ((a, b), (a, c)).$$

Now we can explain, without setting up equations, how our well-typing algorithm tackles this declaration. It first assigns "unknown" types (i.e., type variables) α, β, and γ to a, b, and c. Then $((a, b), (a, c))$ acquires type $(\alpha \times \beta) \times (\alpha \times \gamma)$, and the λ-expression acquires $\beta \times \gamma \rightarrow (\alpha \times \beta) \times (\alpha \times \gamma)$; finally tagpair acquires

$$\alpha \rightarrow (\beta \times \gamma \rightarrow (\alpha \times \beta) \times (\alpha \times \gamma)) \qquad (*)$$

(no type equations have placed any constraint upon the types of a, b, and c).

But consider another way of defining tagpair, using the (infixed) function

$$\#\colon (\alpha \rightarrow \beta) \times (\gamma \rightarrow \delta) \rightarrow ((\alpha \times \gamma) \rightarrow (\beta \times \delta))$$

such that $(f \# g)(a, c) = (f(a), g(c))$, and the pairing function

$$\text{pair}\colon \alpha \rightarrow (\beta \rightarrow \alpha \times \beta)$$

such that pair(a)(b) = (a, b). We could write

$$\textit{let}\ \text{tagpair} = \lambda a \cdot (\textit{let}\ \text{tag} = \text{pair}(a)\ \textit{in}\ \text{tag}\ \#\ \text{tag})$$

We might then expect the well-typing algorithm to proceed as follows. First, α is assigned to a. Then, using the generic type of pair, pair(a) acquires $\delta \to \alpha \times \delta$. This is then used as the local generic type of tag, and the two occurrences of tag in tag # tag are assigned $\beta \to \alpha_1 \times \beta$, $\gamma \to \alpha_2 \times \gamma$, respectively. The occurrence of # is assigned an instance of its generic type (again using new type variables) and the type equation for function application will cause the type

$$\beta \times \gamma \to (\alpha_1 \times \beta) \times (\alpha_2 \times \gamma)$$

to be assigned to tag # tag, and to the body of the λ-expression, so that tagpair acquires the type

$$\alpha \to (\beta \times \gamma \to (\alpha_1 \times \beta) \times (\alpha_2 \times \gamma)). \qquad (**)$$

Now comparing (*) with (**), something has gone wrong; the second type is too general. The problem is that tag and its generic type depend upon the λ-bound variable a and its type α, and we do not allow different bound occurrences of a λ-bound variable to have different type. Indeed, as far as type is concerned, we should get the same as if tagpair were defined in yet a third way, by

$$\textit{let}\ \text{tagpair} = \lambda a \cdot (\text{pair}(a)\ \#\ \text{pair}(a))$$

and the reader may be able to obtain (by setting up some equations as in Example 1) the expected type (*) in this case.

The solution is fortunately straightforward. We decree that in instantiating the type of a variable bound by *let* or by *letrec*. only those type variables which do *not* occur in the types of enclosing λ-bindings (or formal parameter bindings) may be instantiated. We call such instantiable variables (in a generic type) *generic* type variables.

Now in the second definition of tagpair, the locally defined tag acquired a generic type $\delta \to \alpha \times \delta$, in which δ is generic but α is not. Thus α should not have been instantiated in assigning types to the occurrences of tag, and then (**) would have been identical with (*). This example may appear a little contrived; indeed, our experience has been that almost always either *all* or *none* of the type variables of a generic type are generic. But there seem to be no simple syntactic constraints which would eliminate the exceptions, nor does it seem desirable to do so.

From the examples it becomes clear that the rules for typing variables bound, respectively, by *let* (or *letrec*) and by λ are going to be different. Thus, although our *semantics* for the two expressions

$$\textit{let}\ x = e\ \textit{in}\ e'; \qquad (\lambda x \cdot e')e$$

may be (and are) equivalent, it may be possible to assign types correctly to the former but not to the latter. An example is the pair

$$\textit{let}\ I = \lambda x \cdot x\ \textit{in}\ I(I); \qquad (\lambda I \cdot I(I))(\lambda x \cdot x).$$

A (partial) intuition for this is that a λ-abstraction may often occur without an argument; the second expression above contains a special (and rather unnecessary) use of abstraction, in that an explicit argument—$(\lambda x \cdot x)$—is present. Since the *let* construct (when translated) involves this restricted use of an abstraction, it is not unnatural that its rule for type assignment is less constrained. A compiler could, of course, treat all explicit occurrences of $(\lambda x \cdot e')e$ in the less constrained manner.

The treatment of types in the interaction between λ-bindings (i.e., formal parameter bindings) and *let* bindings is really the core of our approach. It gives a consistent treatment of the nonglobal declaration of a procedure which may contain, as a free variable, a formal parameter of an enclosing procedure. This appears to be one of the more crucial difficulties with polymorphism, and therefore we feel justified in presenting our analysis in terms of a simple language (Exp) which excludes as much as possible that is irrelevant to the difficulty.

The reader may still feel that our rules are arbitrarily chosen and only partly supported by intuition. We certainly do not claim that the rules are the only possible ones, but the results given later demonstrate that they are semantically justified. In fact, we show that a program which admits a correct type assignment cannot fail at run-time due to a type error—or more precisely, that type constraints encoded in its semantics are always satisfied. It follows from this that compile-time type checking (i.e., the attempt to discover a correct type assignment) obviates the need to carry types at run-time, with an obvious gain in the efficiency of implementation.

This is of course a principal aim in compile-time type checking; another is the early detection of programming errors (many of which result in ill-typed programs). Our achievement is to extend type checking to embrace polymorphism. Moreover the type-checking algorithm in its final form (Algorithm $\mathscr{J}$ in Section 3) is remarkably simple, even though the proof of the Syntactic Soundness Theorem, which states that—if it succeeds—it produces a correct type assignment, is rather tedious.

We would like to give an independent characterization of the class of programs which can be well typed in our system, but we have no idea how to do this. However, we can give some pointers. At the suggestion of a referee we looked at Burge [1, Chapt. 3] concerning general functions for processing data structures. All of the functions there (with the exception of Section 3.11 which we did not examine) acquired the expected types from the ML type checker after they had been modified in two respects. First, Burge leaves implicit the coercion functions between a disjoint sum type and its summand types; we needed to make these explicit (this point was mentioned in our Introduction). Second, we used the ML abstract type construct (see Section 5 for an example) to formulate the recursive type definitions used by Burge. In this construct, the isomorphism between a defined abstract type and its representation is made explicit and must be used explicitly. To see the need for this requirement consider the case of an α-stream, which is defined to be a function which yields a pair consisting of an α and an α-stream. The type equation

$$\alpha\text{-stream} = \dashrightarrow (\alpha \times \alpha\text{-stream})$$

cannot be solved by unification (unless we allow infinite type expressions). But by

treating the equation as an isomorphism, and using two functions to convert back and forth between an abstract type and its representation, this difficulty is removed. We claim that this solution is in keeping with the notion of abstract type (see [8], for example).

On the negative side, there are certainly useful expressions which we cannot well type, though we are not clear how burdensome it is to do without them. An obvious example is Curry's Y combinator.

$$Y = \lambda f \cdot (\lambda x \cdot f(x(x)))(\lambda x \cdot f(x(x)))$$

since self-application is ill typed for us. But *letrec* avoids the need for Y. More practically, consider

$$\textit{let}\, F(f) = \lambda(a, b) \cdot (f(a), f(b))$$

which—it may be argued—should accept as argument a function which is polymorphic enough to operate upon a and b of different type. For example,

$$F(\text{reverse})(x, y)$$

produces a pair of reversed lists of different type if x and y are lists of different type. Our system rejects such a use of F (since it requires a and b to have the same type), but admits

$$\textit{let}\ \text{reversepair} = \lambda(x, y) \cdot (\text{reverse}(x), \text{reverse}(y))$$

or any other specialization of the function argument of F.

We feel that this example illustrates the main limitation of our system, but that we may have kept as much flexibility as is possible without the use of explicit type parameters. When these are introduced, the problem arises of the type of types; Reynolds [12] has made some progress in solving this problem, but we were anxious to see how much could be done while avoiding it.

3. A Simple Applicative Language and Its Types

3.1. *The Language* Exp

Let x range over identifiers, that is

$$x \in \text{Id}.$$

Then the expression language Exp is generated by the following grammar:

$$\begin{aligned} e ::= {} & x \mid (ee') \mid \textit{if } e \textit{ then } e' \textit{ else } e'' \mid \\ & \lambda x \cdot e \mid \textit{fix } x \cdot e \mid \textit{let } x = e \textit{ in } e'. \end{aligned}$$

Here (ee') means application, *fix* $x \cdot e$ stands for the least fixed point of $\lambda x \cdot e$, and the last clause binds x to the value of e throughout e'. We often use d, e, f—with primes and

suffixes—to range over Exp. Constants are omitted; we can imagine instead some standard bindings for certain identifiers in an initial environment.

We give an ordinary denotational semantics for Exp, in which we include a value "wrong," which corresponds to the detection of a failure at run-time. In this small language, the only failures are the occurrence of a non-Boolean value as the condition of a conditional, and the occurrence of a nonfunctional value as the operator of an application.

Our semantic domains may be taken to be complete partial orders (cpos); a cpo D (see [9]) is a partially ordered set such that (a) there exist a minimum element, $\perp_D$, (b) every directed subset of D has a least upper bound in D. Take as given a set $\{B_i\}$ of basic domains, with $B_0 = T$, the three element truth value domain

and we define recursively

$$
\begin{aligned}
V &= B_0 + B_1 + \cdots + F + W && \text{(disjoint sum of domains, with } \perp_V \text{ adjoined as minimum element),}\\
F &= V \to V && \text{(continuous functions from } V \text{ to } V\text{),}\\
W &= \{\cdot\} && \text{(error).}
\end{aligned}
$$

The solution (up to isomorphism) of such a set of domain equations is assured by Scott [15]. Although he worked with complete lattices, the solution also exists in cpos (see Plotkin [11]).

The semantic function is $\mathscr{E} \in \text{Exp} \to \text{Env} \to V$, where $\text{Env} = \text{Id} \to V$, the domain of environments. We use η to range over Env. In defining $\mathscr{E}$, and later, we use some familiar notation of Scott and Strachey [16], illustrated by these examples (where D is some summand of V):

(i) If $d \in D$, then d in V is the image of d under the injection of D into V.

(ii) If $v \in V$, then

$$
\begin{aligned}
v \mathsf{E} D &= \text{true} && \text{if } v = d \text{ in } V \text{ for some } d \in D,\\
&= \perp_T && \text{if } v = \perp_V,\\
&= \text{false} && \text{otherwise.}
\end{aligned}
$$

(iii) If $v \in V$, then

$$
\begin{aligned}
v \mid D &= d && \text{if } v = d \text{ in } V, \text{ for some } d \in D,\\
&= \perp_D && \text{otherwise.}
\end{aligned}
$$

The environment $\eta' = \eta\{v/x\}$ is identical with η except that $\eta'(x) = v$. The value $\cdot$ in V ($\cdot \in W$) is written "wrong." We require the conditional function COND $\in T \to V \to V \to V$, where COND tvv' is written $t \to v, v'$ and takes the value

$$\begin{array}{ll} v & \text{if } t = \text{true}, \\ v' & \text{if } t = \text{false}, \\ \perp_V & \text{if } t = \perp_T . \end{array}$$

3.2. *Semantic Equations for* Exp

In these equations, the open brackets $[\![\ ]\!]$ indicate syntactic arguments.

$$\mathscr{E}[\![x]\!]\eta = \eta[\![x]\!]$$

$$\mathscr{E}[\![(e_1 e_2)]\!]\eta = v_1 \mathsf{E} F \to (v_2 \mathsf{E} W \to \text{wrong}, (v_1 \mid F)v_2), \text{ wrong}$$
$$\text{where } v_i \text{ is } \mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2).$$

$$\mathscr{E}[\![\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3]\!]\eta = v_1 \mathsf{E} B_0 \to (v_1 \mid B_0 \to v_2 , v_3), \text{ wrong}$$
$$\text{where } v_i \text{ is } \mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2, 3)$$

$$\mathscr{E}[\![\lambda x \cdot e]\!]\eta = (\lambda v \cdot \mathscr{E}[\![e]\!]\, \eta\{v/x\}) \text{ in } V$$

$$\mathscr{E}[\![\textit{fix } x \cdot e]\!]\eta = Y(\lambda v \cdot \mathscr{E}[\![e]\!]\, \eta\{v/x\})$$

$$\mathscr{E}[\![\textit{let } x = e_1 \textit{ in } e_2]\!]\eta = v_1 \mathsf{E} W \to \text{wrong}, \mathscr{E}[\![e_2]\!]\, \eta\{v_1/x\}$$
$$\text{where } v_1 = \mathscr{E}[\![e_1]\!]\eta.$$

Notes. (i) Y is the least fixed-point operation. In many languages the e in $\textit{fix} f \cdot e$ would be restricted to be an abstraction $\lambda y \cdot e'$, and then

$$\textit{let } f = \textit{fix } f \cdot (\lambda y \cdot e')$$

might receive the syntax

$$\textit{let rec } f(y) = e'$$

(ii) It is easy to see that "*let* $x = e_1$ *in* e_2" has the same meaning under $\mathscr{E}$ as "$(\lambda x \cdot e_2)e_1$". But part of our aim is a type discipline which admits certain expressions in the first form and yet rejects their translations into the second form; this is because λ-abstractions may in general occur without an explicit operand, and need more careful treatment.

(iii) The semantics for $(e_1 e_2)$ corresponds to call-by-value, since the test "$v_2 \mathsf{E} W$" ensures that the meaning of $(e_1 e_2)$ is $\perp_V$ if the meaning of e_2 is $\perp_V$. The omission of this test gives a call-by-name semantics (a similar test may be omitted in the semantics of the *let* construct), and the Semantic Soundness Theorem goes through equally in this case.

3.3. *Discussion of Types*

We now proceed, in outline, as follows. We define a new class of expressions which we shall call types; then we say what is meant by a value *possessing* a type. Some values have many types, and some have no type at all. In fact "wrong" has no type. But if a functional value has a type, then as long as it is applied to the right kind (type) of argument it will produce the right kind (type) of result—which cannot be "wrong"!

Now we wish to be able to show that—roughly speaking—an Exp expression evaluates (in an appropriate environment) to a value which has a type, and so cannot be wrong. In fact, we can give a sufficient syntactic condition that an expression has this robust quality; the condition is just that the expression has a "well-typing" with respect to the environment, which means that we can assign types to it and all its subexpressions in a way which satisfies certain laws.

So there are two main tasks, once the laws of type assignment are given. The first—to show that an expression (or program) with a legal type assignment cannot "go wrong"—is tackled in this section; surprisingly enough, it is the easier task (at least for an applicative language). The second task is to *discover* a legal type assignment, given a program with incomplete type information. This task is often called *type checking*. Of course, this term can also mean just verifying that a given type assignment is legal; in a practical situation we probably require something between the two, since one cannot expect a programmer to attach a type to every subexpression. In Section 4 we look at the class of legal type assignments for a given program (the class is infinite in general, since we admit polymorphism), and we give an algorithm which, if it succeeds, produces a legal type assignment. We conjecture that if the latter exists then the algorithm finds one which is most general, in the sense that any other legal type assignment is a substitution instance of it (substituting types for type variables).

3.4. *Types and their Semantics*

The syntax of types is as follows.

(1) $\iota_0, \iota_1, \ldots$ are (basic) types; one for each B_i.

(2) There is a denumerable set of type variables, which are types. We use $\alpha, \beta, \gamma, \ldots$ to range over type variables.

(3) If ρ and σ are types, so is $\rho \to \sigma$.

A *monotype* is a type containing no type variables. We use $\mu, \nu, \pi, \ldots$ to range over monotypes. We use the word *polytype* when we wish to imply that a type may, or does, contain a variable.

We first give the semantics of monotypes; that is, we give the conditions under which a value $v \in V$ *possesses* a monotype μ, which we write $v : \mu$.

(i) $v : \iota_i$ iff $v = \perp_V$ or $v \mathsf{E} B_i$

(ii) $v : \mu \to \nu$ iff *either* $v = \perp_V$, *or* $v \mathsf{E} F$ and $(v \mid F)u : \nu$ whenever $u : \mu$.

It is clear then that many values have no type. Examples are

$$\text{wrong}, (\lambda v \in V \cdot \text{wrong}) \text{ in } V,$$
$$(\lambda v \in V \cdot v \mathsf{E} B_0 \rightarrow (v \mid B_0 \rightarrow x \text{ in } V, y \text{ in } V), \text{wrong}) \text{ in } V$$
$$(\text{where } x \in B_1, y \in B_2 \text{ for example}).$$

But in the last example if $y \in B_1$ instead, then the function has type $\iota_0 \rightarrow \iota_1$ (and no other).

Some values have many types; the identity function

$$(\lambda v \in V \cdot v) \text{ in } V$$

for example has type $\mu \rightarrow \mu$ for every μ. And of course $\perp_V$ has every type (it is the only value which has every type).

This notion of type is derived from Scott [17]. In fact, it is what Scott calls *functionality* (after Curry), and is distinct from the notion of a *retract*. If we temporarily identify a type with the set of values which possess it, then it is easy to show that types are *downward closed* and *directed complete*, that is

(i) $\forall v, v' \in V \cdot (v : \mu \text{ and } v' \sqsubseteq v) \Rightarrow v' : \mu$,

(ii) For each directed subset X of V, $(\forall v \in X \cdot v : \mu) \Rightarrow \sqcup X : \mu$.

Retracts share the second property, but not the first. Recently Shamir and Wadge [18] have defined a type to be *any* set with these two properties, and they investigate the consequences of identifying a value v with the type $\{v' \mid v' \sqsubseteq v\}$.

The semantics of polytypes is as follows. First, we use $\rho \leqslant \sigma$ to mean that ρ may be obtained from σ by substituting types for type variables ($\leqslant$ is clearly reflexive and transitive). For example,

$$\mu \rightarrow \mu \leqslant \alpha \rightarrow \alpha \leqslant \beta \rightarrow \beta \leqslant \alpha \rightarrow \beta,$$

but

$$\alpha \rightarrow \beta \nleqslant \beta \rightarrow \beta \qquad (\text{unless } \alpha = \beta).$$

Then we define

$$v : \sigma \text{ iff } \forall \mu \leqslant \sigma \cdot v : \mu.$$

For example,

$$(\lambda v \cdot v) \text{ in } V : \alpha \rightarrow \alpha.$$

Polytypes thereby also stand for subsets of V, and these are also directed complete. The reader may like to think of each type variable in a polytype as universally quantified at the outermost; for example,

$$\alpha \rightarrow \alpha \text{ "means" } \forall \alpha \cdot \alpha \rightarrow \alpha,$$

where the bound α ranges over monotypes. In fact, it is because α here ranges over monotypes (not all types) and because we do not admit expressions like

$$(\forall \alpha \cdot \alpha \rightarrow \alpha) \rightarrow (\forall \alpha \cdot \alpha \rightarrow \alpha)$$

as types—though we can see they "mean" if the bound variables are taken to range over monotypes—that we avoid the difficulties (and also some of the interest) of Reynolds [12] in his richer notion of type.

We need the following simple properties, which are immediate from our definitions.

PROPOSITION 1. *If* $v : \sigma$ *and* $\tau \leqslant \sigma$ *then* $v : \tau$.

PROPOSITION 2. *If* $v : \sigma \to \tau$ *and* $v' : \sigma$, *then* $(v \mid F)v' : \tau$.

In each case, a property of monotypes is lifted to polytypes.

3.5. *Type Assignments*

To prepare the ground for the theorem that well-typed expressions cannot "go wrong," we need to define what is meant by *typing* an expression. We need first some notion of a type environment to give types to the free variables in an expression.

A *prefix* p is a finite sequence whose members have the form *let* x, *fix* x, or λx, where x is a variable. A prefixed expression (pe) has the form $p \mid e$, where every variable free in e occurs in a member of p. We separate the members of a prefix by a period $(\cdot)$.

Every pe has sub-pe's given by the following, together with transitive reflexive closure:

(i) $p \mid x$ has no sub-pe's except itself,

(ii) $p \mid (ee')$ has sub-pe's $p \mid e$ and $p \mid e'$,

(iii) $p \mid (\textit{if } e \textit{ then } e' \textit{ else } e'')$ has sub-pe's $p \mid e$, $p \mid e'$ and $p \mid e''$,

(iv) $p \mid (\lambda x \cdot e)$ has sub-pe $p \cdot \lambda x \mid e$,

(v) $p \mid (\textit{fix } x \cdot e)$ has sub-pe $p \cdot \textit{fix } x \mid e$,

(vi) $p \mid (\textit{let } x = e \text{ in } e')$ has sub-pe's $p \mid e$ and $p \cdot \textit{let } x \mid e'$.

For example, $\lambda y \mid (\textit{let } f = \lambda x \cdot (xy) \textit{ in } (fy))$ has sub-pe's (besides itself)

$$\lambda y \mid \lambda x \cdot (xy), \qquad \lambda y \cdot \lambda x \mid (xy), \qquad \lambda y \cdot \lambda x \mid x, \qquad \lambda y \cdot \lambda x \mid y,$$
$$\lambda y \cdot \textit{let } f \mid (fy), \qquad \lambda y \cdot \textit{let } f \mid f, \qquad \lambda y \cdot \textit{let } f \mid y$$

so a sub-pe is just a subexpression prefixed with all the variable bindings which enclose it. Note that $\lambda x \cdot (xy)$ in the above is not enclosed by *let* f—it is not in the *scope* of this binding.

We say that a member *let* x or *fix* x or λx of p is *active* in p just if no x occurs to the right of it in p.

Now a *typing* of a pe $p \mid e$ is an assignment of a type to each element of p, and to each subexpression and each λx, *fix* x, or *let* x in e, with the constraint that in a subexpression $(\textit{let } x = e' \textit{ in } e'')$ the same type is assigned to *let* x and e'. Thus one typing of the illustrated pe (it is nearly, but not quite, a well typing in the sense later defined) is as follows:

$$\lambda y_\alpha \mid (\textit{let } f_{(\alpha\to\beta)\to\beta} = (\lambda x_{\alpha\to\beta}(x_{\alpha\to\beta} y_\alpha)_\beta)_{(\alpha\to\beta)\to\beta} \textit{ in } (f_{(\alpha\to\gamma)\to\gamma} y_\alpha)_\gamma)_\gamma .$$

We denote a typing of $p \mid e$ by $\bar{p} \mid \bar{e}$, or by $\bar{p} \mid \bar{e}_\sigma$ when we want to indicate the type σ assigned to e itself.

In any $\bar{p} \mid \bar{e}$, and any binding *let* x_σ in either $\bar{p}$ or $\bar{e}$, a type variable in σ which does not occur in (the type of) any enclosing λy_τ or *fix* y_τ binding is called a *generic* type variable for the binding *let* x_σ. In the example above, β is generic, but α is not, for the binding *let* $f_{(\alpha\to\beta)\to\beta}$. Intuitively, the generic type variables in a binding *let* x_σ represent degrees of freedom for the types at which x may be used; they represent the local polymorphism of x. Notice that if no λ or *fix* bindings enclose *let* x_σ, then all the type variables in σ are generic. A *generic instance* of σ is an instance of σ in which only generic type variables are instantiated.

For technical reasons we require that generic type variables occur in a controlled manner. We say that $\bar{p} \mid \bar{d}$ is *standard* if for every typed sub-pe $\bar{p}' \mid \bar{d}'$ (with induced typing) the generic type variables of each member *let* x_σ of $\bar{p}'$ occur nowhere else in $\bar{p}' \mid \bar{d}'$. Thus in particular, if *let* $x_\rho = \bar{e}_\rho$ *in* $\bar{e}'_\sigma$ is a subexpression of $\bar{d}$, with induced typing, then the generic type variables in ρ may not occur in $\bar{e}'_\sigma$ (though they must of course occur in $\bar{e}_\rho$).

We now define the notion of a *well-typed* (wt) pe as follows:

(i) $\bar{p} \mid x_\tau$ is wt iff it is standard, and *either*

(a) λx_τ or *fix* x_τ is active in $\bar{p}$, *or*

(b) *let* x_σ is active in $\bar{p}$, and τ is a generic instance of σ.

(ii) $\bar{p} \mid (\bar{e}_\rho \bar{e}'_\sigma)_\tau$ is wt iff $\bar{p} \mid \bar{e}$ and $\bar{p} \mid \bar{e}'$ are both wt, and $\rho = \sigma \to \tau$.

(iii) $\bar{p} \mid (\textit{if } \bar{e}_\rho \textit{ then } \bar{e}'_\sigma \textit{ else } \bar{e}''_\sigma)_{\tau'}$ is wt iff $\bar{p} \mid \bar{e}$, $\bar{p} \mid \bar{e}'$ and $\bar{p} \mid \bar{e}''$ are all wt, $\rho = \iota_0$, and $\sigma = \tau = \tau'$.

(iv) $\bar{p} \mid (\lambda x_\rho \cdot \bar{e}_\sigma)_\tau$ is wt iff $\bar{p} \cdot \lambda x_\rho \mid \bar{e}$ is wt and $\tau = \rho \to \sigma$.

(v) $\bar{p} \mid (\textit{fix } x_\rho \cdot \bar{e}_\sigma)_\tau$ is wt iff $\bar{p} \cdot \textit{fix } x_\rho \mid \bar{e}$ is wt and $\rho = \sigma = \tau$.

(vi) $\bar{p} \mid (\textit{let } x_\rho = \bar{e}_\rho \textit{ in } \bar{e}'_\sigma)_\tau$ is wt iff $\bar{p} \mid \bar{e}$ and $\bar{p} \cdot \textit{let } x_\rho \mid \bar{e}'$ are both wt, and $\sigma = \tau$.

Although this recursive definition is useful for some proofs, an alternative characterization of wt is sometimes useful. The proof of the next proposition is fairly straightforward, and we omit it. Note that a wt $\bar{p} \mid \bar{d}$ is necessarily standard, by an easy structural induction.

PROPOSITION 3. *$\bar{p} \mid \bar{d}$ is wt iff the following conditions hold:*

(A) *It is standard.*

(B) *For every (bound) occurrence x_σ, the corresponding binding occurrence is either λx_σ, or* fix x_σ, *or* let x_τ, *where σ is a generic instance of τ.*

(C) *The following conditions hold for all subexpressions (with induced typing) of $\bar{d}$*

$(\bar{e}_\rho \bar{e}'_\sigma)_\tau$	$\rho = \sigma \to \tau$,
$(\text{if } \bar{e}_\rho \text{ then } \bar{e}'_\sigma \text{ else } \bar{e}''_\tau)_{\tau'}$	$\rho = \tau_0$ *and* $\sigma = \tau = \tau'$,
$(\lambda x_\rho \cdot \bar{e}_\sigma)_\tau$	$\tau = \rho \to \sigma$,
$(\text{fix } x_\rho \cdot \bar{e}_\sigma)_\tau$	$\rho = \sigma = \tau$,
$(\text{let } x_\rho = \bar{e}_\rho \textit{ in } \bar{e}'_\sigma)_\tau$	$\sigma = \tau$. ∎

The typing which we illustrated above therefore fails to be wt for only one reason: The subexpression $(f_{(\alpha\to\gamma)\to\gamma} y_\alpha)_\gamma$ violates the first of conditions (C) in Proposition 3.

Consider another example. The following (with $\bar{p}$ empty) is a well typing:

$$\begin{array}{l} \mathit{let}\ I_{\alpha\to\alpha} = (\lambda x_\alpha \cdot x_\alpha)_{\alpha\to\alpha}\ \mathit{in} \\ \quad (I_{(\iota\to\iota)\to(\iota\to\iota)}\, I_{\iota\to\iota})_{\iota\to\iota}\,. \end{array}$$

Note that α is generic in the type $\alpha \to \alpha$ of the declaration of I, so may be instantiated (possibly differently) in the types of bound occurrences of I.

To illustrate the need to instantiate only generic type variables, for variables declared by *let*, notice first that in $\lambda x_\alpha \cdot x_\beta$ we must have $\alpha = \beta$, by condition (B) of Proposition 3. Indeed, we can argue intuitively for this as follows: if we declare

$$\mathit{let}\ I = \lambda x \cdot x\ \mathit{in}\ \ldots,$$

then we wish to have that any expression (Ie) in the scope of this declaration receives the same type as the subexpression e. But now suppose we write (with assigned types)

$$\mathit{let}\ I_{\alpha\to\beta} = (\lambda x_\alpha \cdot (\mathit{let}\ y_\alpha = x_\alpha\ \mathit{in}\ y_\beta)_\beta)_{\alpha\to\beta}\ \mathit{in}\ \ldots$$

then—since this is semantically equivalent to the simpler declaration—we should again demand that $\alpha = \beta$. But this is imposed in our definition of well typing, just because α is not generic for the binding *let* y_α, so may not be instantiated in a bound occurrence of y.

3.6. *Substitutions*

A *substitution* S is a map from type variables to types. S may be extended in a natural way to yield a map from types to types, from typed pe's to typed pe's, etc.

We say that S *involves* a type variable α if *either* $S\alpha \neq \alpha$, *or* for some $\beta \neq \alpha$, $\alpha \in S\beta$. ($\alpha \in \tau$ means α occurs in τ.)

We need substitutions extensively in the second part of this paper, but for the present we need only one property relating substitutions and wt.

PROPOSITION 4. *If S involves no generic variables of a wt $\bar{p} \mid \bar{d}$, then $S(\bar{p} \mid \bar{d})$ is also a wt.*

Proof. We use Proposition 3. First, observe that the assumption on S yields that the generic variables for each binding in $S(\bar{p} \mid \bar{d})$ are exactly those for the corresponding binding in $\bar{p} \mid \bar{d}$. Since $S\beta$ contains no generic variable when β is not generic, $S(\bar{p} \mid \bar{d})$ is standard.

Second, if x_σ is bound by λx_σ or *fix* x_σ in $\bar{p} \mid \bar{d}$, then $x_{S\sigma}$ is bound by $\lambda x_{S\sigma}$ or *fix* $x_{S\sigma}$ in $S(\bar{p} \mid \bar{d})$. If x_σ is bound by *let* x_τ, and $\sigma = [\rho_1/\alpha_1, \ldots, \rho_n/\alpha_n]_\tau$, where α_i are the generic variables of τ, then in $S(\bar{p} \mid \bar{d})$ $x_{S\sigma}$ is bound by *let* $x_{S\tau}$, and $S\sigma = [S\rho_1/\alpha_1, \ldots, S\rho_n/\alpha_n](S\tau)$ is a generic instance of $S\tau$.

Third, conditions (C) of Proposition 3 are easily verified for $S(\bar{p} \mid \bar{d})$, using identities like $S(\sigma \to \tau) = S\sigma \to S\tau$. ∎

3.7. *Well-Typed Expressions Do Not Go Wrong*

First we need a simple relation between semantic environments η and our type environments—which are typed prefixed $\bar{p}$. We say

η *respects* $\bar{p}$ iff, whenever *let* x_ρ or λx_ρ or *fix* x_ρ is active in $\bar{p}$, $\eta[\![x]\!] : \rho$.

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

Proof. A fairly simple structural induction. Take the six cases for $\bar{d}_\tau$.

(i) x_τ. Then *either* λx_τ or *fix* x_τ is active in $\bar{p}$, and $\eta[\![x]\!] : \tau$, so $\mathscr{E}[\![x]\!]\eta : \tau$, *or let* x_σ is active in $\bar{p}$, and $\eta[\![x]\!] : \sigma$; but then $\tau \leqslant \sigma$, so $\mathscr{E}[\![x]\!]\eta = \eta[\![x]\!] : \tau$, by Proposition 1.

(ii) $(\bar{e}_{\sigma\to\tau}\bar{e}_\sigma)_\tau$. Then $\bar{p} \mid \bar{e}_{\sigma\to\tau}$ is wt, so $\mathscr{E}[\![e]\!]\eta : \sigma \to \tau$, and similarly $\mathscr{E}[\![e']\!]\eta : \sigma$. Then from the semantic equation (remembering that wrong has no type) and by Proposition 2 we get $\mathscr{E}[\![d]\!]\eta : \tau$.

(iii) (if $\bar{e}_{\iota_0}$ *then* $\bar{e}'_\sigma$ *else* $\bar{e}''_\sigma$). Straightforward; the only extra detail needed here is that $\perp_V$ has every type.

(iv) $(\lambda x_\rho \cdot \bar{e}_\sigma)_{\rho\to\sigma}$. Then $\bar{p} \cdot \lambda x_\rho \mid \bar{e}_\sigma$ is wt. Now we require $(\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\})$ in $V : \rho \to \sigma$. Denote this function by f in V. The inverse of Proposition 2 does not hold, that is, to show f in $V : \rho \to \sigma$ it is not sufficient (though it is necessary) that whenever $v : \rho, fv : \sigma$. What is required is that for every $\eta \to \nu \leqslant \rho \to \sigma, f$ in $V : \mu \to \nu$.

Suppose then that $\mu \to \nu \leqslant \rho \to \sigma$. Then there is a substitution S, involving only the type variables in ρ and σ, such that $\mu \to \nu = S(\rho \to \sigma)$. Then, since none of these type variables is generic in $\bar{p} \cdot \lambda x_\rho \mid \bar{e}_\sigma$, it follows that $S(\bar{p}) \cdot \lambda x_\mu \mid S(\bar{e})_\nu$ is wt by Proposition 4. Moreover η respects $S(\bar{p})$ (since by Proposition 1 whenever $\eta[\![x]\!] : \sigma'$ and $\tau' \leqslant \sigma'$, $\eta[\![x]\!] : \tau'$) so for any $v : \mu$ we also have $\eta\{v/x\}$ respects $S(\bar{p}) \cdot \lambda x_\mu$.

It then follows by induction that $\mathscr{E}[\![e]\!]\,\eta\{v/x\} : \nu$, so we have shown that $v : \mu$ implies $fv : \nu$, and this yields f in $V : \mu \to \nu$ as required.

(v) $(\mathit{fix}\ x_\rho \cdot \bar{e}_\rho)_\rho$. Then $\bar{p} \cdot \mathit{fix}\ x_\rho \mid \bar{e}_\rho$ is wt. Now we require that $v : \rho$, where

$$v = Y(\lambda v' \cdot \mathscr{E}[\![e]\!]\,\eta\{v'/x\}).$$

Now $v = \sqcup_i v_i$, where $v_0 = \perp_V$, $v_{i+1} = \mathscr{E}[\![e]\!]\,\eta\{v_i/x\}$, and by the directed completeness of types we only have to show $v_i : \rho$ for each i.

Clearly $v_0 : \rho$. Assume $v_i : \rho$. Since η respect $\bar{p}$, we have that $\eta\{v_i/x\}$ respects $\bar{p} \cdot \mathit{fix}\ x_\rho$, so by the main induction hypothesis $v_{i+1} : \rho$ also, and we are done.

(vi) $(\mathit{let}\ x = \bar{e}_\rho\ \mathit{in}\ \bar{e}'_\sigma)_\sigma$. Then $\bar{p} \mid \bar{e}_\rho$ is wt, so we immediately have $v : \rho$, where $v = \mathscr{E}[\![e]\!]\eta$. We require $\mathscr{E}[\![e']\!]\,\eta\{v/x\} : \sigma$.

Now $\bar{p} \cdot \mathit{let}\ x_\rho \mid \bar{e}'_\sigma$ is also wt, and because $v : \rho$ we have that $\eta\{\sigma/x\}$ respects $\bar{p} \cdot \mathit{let}\ x_\rho$; the rest follows by the induction hypothesis. ∎

As a corollary, under the conditions of the theorem we have

$$\mathscr{E}[\![d]\!]\eta \neq \text{wrong},$$

since wrong has no type.

4. A Well-Typing Algorithm and Its Correctness

4.1. *The Algorithm $\mathscr{W}$*

In this section we tackle the question of finding a well typing for a prefixed expression. We present an algorithm $\mathscr{W}$ for this. We would like to prove that $\mathscr{W}$ is both syntactically sound and (in some sense) complete. By syntactic soundness, we mean that whenever $\mathscr{W}$ succeeds it produces a wt; by completeness, we mean that whenever a wt exists, $\mathscr{W}$ succeeds in finding one which is (in some sense) at least as general.

Although $\mathscr{W}$ is probably complete, it is difficult to find a simple proof. So we concentrate on soundness, and then comment on implementation of $\mathscr{W}$ and on extending it to deal with richer languages. Since a type-checking algorithm which simulates $\mathscr{W}$ has been working successfully for nearly 2 years in the context of the LCF metalanguage ML [2], we have evidence for its usefulness and even—to some extent—for its completeness.

$\mathscr{W}$ is based on the unification algorithm of Robinson [14]. Indeed, the only feature of well typing which does not fall directly within the framework of unification is the condition that τ should be a generic instance of σ whenever x_τ is bound by *let* x_σ. The completeness (in some sense) of $\mathscr{W}$ should follow from the second part of the following proposition concerning unification, but we need only the first half for our proof that $\mathscr{W}$ is sound.

Proposition 5 (Robinson). *There is an algorithm $\mathscr{U}$, taking a pair of expressions (over some alphabet of variables) and yielding a substitution, such that for any pair of expressions σ and τ*

(A) *If $\mathscr{U}(\sigma, \tau)$ succeeds, yielding U, then U* unifies *σ and τ (i.e., $U\sigma = U\tau$).*

(B) *If R unifies σ and τ, then $\mathscr{U}(\sigma, \tau)$ succeeds yielding a U such that for some substitution S, $R = SU$.*

Moreover, U involves only variables in σ and τ.

To find a well typing of a complete program f, we would expect to supply also a typed prefix $\bar{p}$, containing only *let* bindings, giving the types of values bound to predefined identifiers. We would then expect $\mathscr{W}$ to yield $\bar{f}$ such that $\bar{p} \mid \bar{f}$ is a wt.

To state (and prove) $\mathscr{W}$ recursively however, prefixes containing all types of binding occur, and $\mathscr{W}$ in general needs to modify the nongeneric type variables in the prefix to meet constraints imposed on the program. We therefore make $\mathscr{W}$ return also a substitution T, indicating the necessary transformation.

To be precise, we show that if $\mathscr{W}(\bar{p}, f)$ succeeds and returns $(T, \bar{f})$, then $(T\bar{p}) \mid \bar{f}$ is a wt.

We first state $\mathscr{W}$. At certain points $\mathscr{W}$ requires type variables which have not previously occurred; such new type variables are denoted by β or β_i. $\mathscr{W}(\bar{p}, f)$ is defined by induction on the structure of f; the algorithm is expressed in a purely applicative programming style, in contrast with the more efficient algorithm $\mathscr{J}$ presented later, which is expressed more in the style of imperative programming.

Algorithm $\mathscr{W}$

$$\mathscr{W}(\bar{p}, f) = (T, \bar{f}), \text{ where}$$

(i) If f is x, then:

if λx_σ or *fix* x_σ is active in $\bar{p}$ then
$\quad T = I, \bar{f} = x_\sigma$;
if *let* x_σ is active in $\bar{p}$ then
$\quad T = I, \bar{f} = x_\tau$
where $\tau = [\beta_i/\alpha_i]\sigma$, α_i are the generic variables of σ,
and β_i are new variables.

(ii) If f is (de), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$;
let $U = \mathscr{U}(S\rho, \sigma \to \beta)$, β new;
then $T = USR$, and $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

(iii) If f is (*if d then e else e'*), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$ and $U_0 = \mathscr{U}(\rho, \iota_0)$;
let $(S, \bar{e}_\sigma) = \mathscr{W}(U_0R\bar{p}, e)$, and $(S', \bar{e}'_{\sigma'}) = \mathscr{W}(SU_0R\bar{p}, e')$;
let $U = \mathscr{U}(S'\sigma, \sigma')$;
then $T = US'SU_0R$, and
$\quad \bar{f} = U((\textit{if } S'SU_0\bar{d} \textit{ then } S'\bar{e} \textit{ else } \bar{e}')_\sigma)$.

(iv) If f is $(\lambda x \cdot d)$, then:

let $(R, \bar{d}) = \mathscr{W}(\bar{p} \cdot \lambda x_\beta, d)$, where β is new;
then $T = R$, and $\bar{f} = (\lambda x_{R\beta} \cdot \bar{d}_\rho)_{R\beta \to \rho}$.

(v) If f is (*fix* $x \cdot d$), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p} \cdot \textit{fix } x_\beta, d)$, β new;
let $U = \mathscr{U}(R\beta, \rho)$;
then $T = UR$, and $\bar{f} = (\textit{fix } x_{UR\beta} \cdot U\bar{d})_{UR\beta}$.

(vi) If f is (*let* $x = d$ *in* e), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$;
let $(S, e_\sigma) = \mathscr{W}(R\bar{p} \cdot \textit{let } x_\rho, e)$;
then $T = SR$, and $\bar{f} = (\textit{let } x_{S\rho} = S\bar{d} \text{ in } \bar{e})_\sigma$. ∎

4.2. *The Soundness of $\mathscr{W}$*

To show that $\mathscr{W}$ is sound it is convenient to have a few simple definitions. If A is a type, a typed prefix, or a typed pe then

$$\mathrm{Vars}(A) \stackrel{\text{def}}{=} \{\alpha \mid \alpha \in A,\ \alpha \text{ a type variable}\}.$$

If A is a typed prefix or a typed pe then

$$\mathrm{Gen}(A) \stackrel{\text{def}}{=} \{\alpha \mid \alpha \in A,\ \alpha \text{ a generic type variable}\},$$
$$\mathrm{Spec}(A) \stackrel{\text{def}}{=} \mathrm{Vars}(A) - \mathrm{Gen}(A).$$

If S is a substitution, then

$$\begin{aligned}\mathrm{Inv}(S) &\stackrel{\text{def}}{=} \{\alpha \mid S \text{ involves } \alpha\}\\ &= \{\alpha \mid \exists\beta \cdot S\beta \neq \beta \text{ and } \alpha \in \{\beta\} \cup \mathrm{Vars}(S\beta)\}.\end{aligned}$$

We need the following simple properties, whose proof we omit:

PROPOSITION 6. (A) $\mathrm{Inv}(RS) \subseteq \mathrm{Inv}(R) \cup \mathrm{Inv}(S)$

(B) $\mathrm{Vars}(S\tau) \subseteq \mathrm{Vars}(\tau) \cup \mathrm{Inv}(S)$.

THEOREM 2 (Syntactic Soundness). *Let $\bar{p}$ be a standard prefix, and $p \mid f$ a (closed) pe. Then, if $\mathscr{W}(\bar{p}, f) = (T, \bar{f}_\tau)$,*

(A) *$T\bar{p} \mid \bar{f}$ is wt,*

(B) $\mathrm{Inv}(T) \subseteq \mathrm{Spec}(\bar{p}) \cup \mathit{New}$,

and

(C) $\mathrm{Vars}(\tau) \subseteq \mathrm{Spec}(\bar{p}) \cup \mathit{New}$,

where New is the set of new type variables used by $\mathscr{W}$.

Proof. By induction on the structure of f, using the recursive definition of wt. We omit the cases of conditional and *fix* expressions, since nothing new arises there, and we treat the easier of the other cases first

(i) f is x. Then $T = I$, so (B) is immediate. If λx_σ or *fix* x_σ is active in $\bar{p}$ then $\bar{f} = x_\sigma$ and (A), (C) are immediate.

If *let* x_σ is active, then $\bar{f} = x_\tau$, where $\tau = [\beta_i/\alpha_i]\sigma$, $\{\alpha_i\}$ generic in σ, $\mathit{New} = \{\beta_i\}$. Then $T\bar{p} \mid \bar{f} = \bar{p} \mid x_\tau$ is standard and (A), (C) follow easily.

(iv) f is $(\lambda x \cdot d)$. Let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p} \cdot \lambda x_\beta, d)$, using new variables New_1, say.

By induction, $R(\bar{p} \cdot \lambda x_\beta) \mid \bar{d}_\rho$ is wt, so for (A) $\bar{f} = R\bar{p} \mid (\lambda x_{R\beta} \cdot \bar{d})_{R\beta \to \rho}$ is wt (def^{n} of wt). Also by induction,

$$\left.\begin{matrix}\mathrm{Inv}(R)\\ \mathrm{Vars}(\rho)\end{matrix}\right\} \subseteq \mathrm{Spec}(\bar{p} \cdot \lambda x_\beta) \cup \mathrm{New}_1$$
$$= \mathrm{Spec}(\bar{p}) \cup \{\beta\} \cup \mathrm{New}_1$$
$$= \mathrm{Spec}(\bar{p}) \cup \mathrm{New} \qquad (\text{since } \mathrm{New} = \mathrm{New}_1 \cup \{\beta\})$$

and (B) follows since $T = R$. For (C)

$$\mathrm{Vars}(R\beta \to \rho) \subseteq \mathrm{Inv}(R) \cup \{\beta\} \cup \mathrm{Vars}(\rho) \qquad \text{by Proposition 6,}$$
$$\subseteq \mathrm{Spec}(\bar{p}) \cup \mathit{New}$$

as required.

(vi) f is (*let* $x = d$ *in* e). Then let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, using new variables New_1, say. Then by induction

$$R\bar{p} \mid \bar{d} \text{ is wt} \tag{1}$$

$$\left.\begin{matrix}\mathrm{Inv}(R)\\ \mathrm{Vars}(\rho)\end{matrix}\right\} \subseteq \mathrm{Spec}(\bar{p}) \cup \mathrm{New}_1 \tag{2}$$

Now from (2)

$$\mathrm{Spec}(R\bar{p}) \subseteq \mathrm{Inv}(R) \cup \mathrm{Spec}(\bar{p})$$
$$\subseteq \mathrm{Spec}(\bar{p}) \cup \mathrm{New}_1 \tag{3}$$

and from (1) by standardness

$$\mathrm{Gen}(R\bar{p} \mid \bar{d}) \cap \mathrm{Spec}(R\bar{p}) = \varnothing. \tag{4}$$

We also have that $\mathrm{Gen}(R\bar{p}) = \mathrm{Gen}(\bar{p})$, which is disjoint from $\mathrm{Vars}(\rho)$ by (2), hence $R\bar{p} \cdot \mathit{let}\ x_\rho$ is a standard prefix.

So let $S, \bar{e}_\sigma = \mathscr{W}(R\bar{p} \cdot \mathit{let}\ x_\rho, e)$ using new variables New_2. Then by induction

$$S(R\bar{p} \cdot \mathit{let}\ x_\rho) \mid \bar{e} \text{ is wt} \tag{5}$$

$$\left.\begin{matrix}\mathrm{Inv}(S)\\ \mathrm{Vars}(\sigma)\end{matrix}\right\} \subseteq \mathrm{Spec}(R\bar{p} \cdot \mathit{let}\ x_\rho) \cup \mathrm{New}_2 \tag{6}$$

But $\mathrm{Spec}(R\bar{p} \cdot \mathit{let}\ x_\rho) = \mathrm{Spec}(R\bar{p})$, so putting (6) and (4) together yields (since New_2 are new variables)

$$\mathrm{Inv}(S) \cap \mathrm{Gen}(R\bar{p} \mid \bar{d}) = \varnothing$$

and it follows by Proposition 4 that $S(R\bar{p} \mid \bar{d})$ is wt, and using (5) we have by definition of wt that

$$SR\bar{p} \mid (\mathit{let}\ x_{S\rho} = S\bar{d}\ \mathit{in}\ \bar{e})_\sigma$$

is wt; but this is just $T\bar{p} \mid \bar{f}$, so we have proved (A). For (B), we have

$$\begin{aligned}\mathrm{Inv}(T) &\subseteq \mathrm{Inv}(S) \cup \mathrm{Inv}(R), \text{ by Proposition 6,}\\ &\subseteq \mathrm{Spec}(\bar{p}) \cup New_1 \cup New_2\,, \text{ using (6), (3), and (2)}\end{aligned}$$

and for (C), by similar reasoning,

$$\mathrm{Vars}(\sigma) \subseteq \mathrm{Spec}(\bar{p}) \cup New_1 \cup New_2\,.$$

This is all we need, since $New = New_1 \cup New_2$ in this case.

(ii) f is (de). Let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, using new variables New_1, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$, using new variables New_2; then by similar reasoning to case (vi) we find that

$$SR\bar{p} \mid S(\bar{d}_\rho) \qquad \text{is wt} \tag{7}$$

$$SR\bar{p} \mid \bar{e}_\sigma \qquad \text{is wt} \tag{8}$$

$$\left.\begin{matrix}\mathrm{Inv}(SR)\\ \mathrm{Vars}(S\rho),\ \mathrm{Vars}(\sigma)\end{matrix}\right\} \subseteq \mathrm{Spec}(\bar{p}) \cup New_1 \cup New_2\,. \tag{9}$$

Now if $U = \mathscr{U}(S\rho, \sigma \to \beta)$, where β is new, we have by Proposition 5 that

$$US\rho = U\sigma \to U\beta, \tag{10}$$

$$\mathrm{Inv}(U) \subseteq \mathrm{Vars}(S\rho) \cup \mathrm{Vars}(\sigma) \cup \{\beta\}. \tag{11}$$

It follows that U involves no generic variables of the wt's (7) and (8), and that

$$USR\bar{p} \mid U(((S\bar{d})\bar{e})_8)$$

is wt; but this is just $T\bar{p} \mid \bar{f}$, so we have proved (A). For (B), we have first that

$$New = New_1 \cup New_2 \cup \{\beta\},$$

so

$$\begin{aligned}\mathrm{Inv}(T) &\subseteq \mathrm{Inv}(U) \cup \mathrm{Inv}(SR), && \text{by Proposition 6,}\\ &\subseteq \mathrm{Spec}(\bar{p}) \cup New, && \text{from (9) and (11),}\end{aligned}$$

and for (C),

$$\begin{aligned}\mathrm{Vars}(\tau) &= \mathrm{Vars}(U\beta)\\ &\subseteq \mathrm{Inv}(U) \cup \{\beta\}, && \text{by Proposition 6,}\\ &\subseteq \mathrm{Spec}(\bar{p}) \cup New, && \text{again from (9) and (11).}\end{aligned}$$ ∎

4.3. *Implementation of $\mathscr{W}$; a Simplified Algorithm $\mathscr{J}$*

As it stands, $\mathscr{W}$ is hardly an efficient algorithm; substitutions are applied too often. It was formulated to aid the proof of soundness. We now present a simpler algorithm $\mathscr{J}$ which simulates $\mathscr{W}$ in a precise sense.

$\mathscr{J}$ differs from $\mathscr{W}$ in two ways. First, we adopt an idea familiar in the literature on resolution-based theorem-proving systems, in which substitutions are *composed*, but only *applied* when it is essential to do so. Second, we take advantage of the fact that what is often needed in practice from a well-typing algorithm is not the whole type assignment f, but only the type assigned to f itself.

In fact $\mathscr{J}$ builds only one substitution, called E, which is idempotent—that is, $EE = E$—which is to say that if $\beta \in E\alpha$, then $E\beta = \beta$. This substitution is held in a program variable (called E) global to $\mathscr{J}$, and $\mathscr{J}$ works by transforming E. In place of the unification function $\mathscr{U}$, $\mathscr{J}$ calls a unification procedure UNIFY which delivers no result but side-effects the variable E. We assume that $\mathscr{U}$ and UNIFY are related as follows: of E and E' are the values of E before and after the command

$$\text{UNIFY}(\sigma, \tau)$$

and if

$$\mathscr{U}(E\sigma, E\tau) = U$$

then

$$E' = UE.$$

Thus, applying UNIFY to types σ and τ in the presence of E corresponds to applying $\mathscr{U}$ to types $E\sigma$ and $E\tau$; σ and τ may be thought of as *implicit* types standing for the *explicit* types which would be gained by applying the "explicating" substitution E (the idempotency of E means that further explication is unnecessary). The effect of UNIFY (if successful) is to generate the new explicating substitution E'. $\mathscr{J}$ will similarly handle an implicit typed prefix $\bar{p}$, which can be explicated when necessary by applying E. We assume that $\mathscr{J}$ has local variables ρ, σ, and σ', whose values are implicit types, and generates its result in a fourth variable τ.

Assuming an initial idempotent E, $\mathscr{J}$ is given a typed prefix $\bar{p}$ and an expression f such that $E\bar{p}$ is standard and $p \mid f$ is a pe (i.e., all free identifiers in f are bound in p). Here is the algorithm:

Algorithm $\mathscr{J}$

$$\mathscr{J}(\bar{p}, f) = \tau, \text{ where}$$

(i) If f is x, then:

If λx_σ or *fix* x_σ is active in $\bar{p}$, $\tau := \sigma$.
If *let* x_σ is active in $\bar{p}$, $\tau := [\beta_i/\alpha_i]E\sigma$, where
α_i are the generic type variables of *let* $x_{E\sigma}$ in $E\bar{p}$,
and β_i are new variables.

(ii) If f is (de) then:

$\rho := \mathscr{J}(\bar{p}, d)$; $\sigma := \mathscr{J}(\bar{p}, e)$;
UNIFY $(\rho, \sigma \rightarrow \beta)$; ($\beta$ new)
$\tau := \beta$

(iii) If f is (*if d then e else e'*), then:

$$\rho := \mathscr{J}(\bar{p}, d);\ \text{UNIFY}\ (\rho, \iota_0);$$
$$\sigma := \mathscr{J}(\bar{p}, e);\ \sigma' := \mathscr{J}(\bar{p}, e');$$
$$\text{UNIFY}(\sigma, \sigma');\ \tau := \sigma$$

(iv) If f is $(\lambda x \cdot d)$, then:

$$\rho := \mathscr{J}(\bar{p} \cdot \lambda x_\beta, d);\ (\beta \text{ new})$$
$$\tau := \beta \rightarrow \rho$$

(v) If f is (*fix x · d*), then:

$$\rho := \mathscr{J}(\bar{p} \cdot \mathit{fix}\ x_\beta, d);\ (\beta \text{ new})$$
$$\text{UNIFY}(\beta, \rho);\ \tau := \beta$$

(vi) If f is (*let x = d in e*) then:

$$\rho := \mathscr{J}(\bar{p}, d);\ \sigma := \mathscr{J}(\bar{p} \cdot \text{let}\ x_\rho, e);$$
$$\tau := \sigma. \ \blacksquare$$

What is the simulation relation between $\mathscr{J}$ and $\mathscr{W}$? It is simply expressed by the following proposition, whose proof we omit, since it is an easy structural induction involving few of the subtleties which we encountered in the soundness theorem.

PROPOSITION 7. *Let $p \mid f$ be a pe, E be idempotent, and $E\bar{p}$ be standard. Then $\mathscr{J}(\bar{p}, f)$ succeeds (producing τ' and a new value E' for E) iff $\mathscr{W}(E\bar{p}, f)$ succeeds (producing T and f_τ), and moreover if both succeed*

$$\text{(A)}\quad E' = TE,$$

$$\text{(B)}\quad E'\tau' = \tau.$$

Thus the type produced by $\mathscr{J}$, when explicated, is exactly that ascribed to f by $\mathscr{W}$.

In practice, E may be efficiently represented by a table INST of variable-type pairs, representing those variables which have hitherto been instantiated. The effect of UNIFY is merely to add some entries to INST, representing instantiation of previously uninstantiated variables. The substitution E represented by INST is given recursively as follows

$$\begin{aligned} E(\iota) &= \iota && \text{(basic types)},\\ E(\alpha) &= E(\rho) && \text{if } (\alpha, \rho) \in \text{INST for some } \rho,\\ &= \alpha && \text{otherwise},\\ E(\sigma \rightarrow \tau) &= E\sigma \rightarrow E\tau. \end{aligned}$$

In fact, in the extended version of $\mathscr{J}$ implemented for ML, which is written in LISP, INST itself is represented by a property (in the LISP sense) INSTANCE of type variables. Each type variable α has ρ as its INSTANCE property value, if $(\alpha, \rho) \in$ INST for some ρ; otherwise the property value is NIL.

5. Types in Extended Languages

We now consider some extensions of our language, and how our results may be strengthened to apply to them.

(1) As we said in the introduction, the addition of extra (primitive) type operators such as $\times$ (Cartesian product), $+$ (disjoint sum) and *list* (list forming), causes no difficulty. Together with $\rightarrow$, these are the primitive type operators in the language ML. For $\times$ one has the standard polymorphic functions

pair: $\alpha \rightarrow \beta \rightarrow (\alpha \times \beta)$ (one could add the syntax (e, e') for pair $(e)(e')$),
fst: $\alpha \times \beta \rightarrow \alpha$,
snd: $\alpha \times \beta \rightarrow \beta$.

For $+$, one has

inl: $\alpha \rightarrow \alpha + \beta$,	inr: $\beta \rightarrow \alpha + \beta$	(left and right injections),
outl: $\alpha + \beta \rightarrow \alpha$,	outr: $\alpha + \beta \rightarrow \beta$	(left and right projections),
isl: $\alpha + \beta \rightarrow$ *bool*,	isr: $\alpha + \beta \rightarrow$ *bool*	(left and right discriminators)

with natural interpretations. For *list*, one has the standard list-processing functions mentioned in Section 2. Notice that all members of a list must have the same type.

With appropriate adjustment to the semantic domains, the Semantic Soundness Theorem extends naturally; the Syntactic Soundness Theorem goes through virtually without change.

(2) Next, we consider assignable variables and assignments. One way (used in ML) of adding these is to allow the assignment expression form "$x := e$" (whose value is the value of e), and the expression form "*letref* $x = e$ *in* e'" to declare an assignable variable, initialized to the value of e. The first effect of these additions is a major change in the semantic domains, since now expressions may have side effects. Although we believe that a Semantic Soundness Theorem may be proved, it appears to be a cumbersome task. The reason for the difficulty is illustrated by considering

$$\lambda x \cdot (y := x)$$

which is an identity function with a side effect. To say that it has type $\alpha \rightarrow \alpha$, meaning that whenever it is given an argument of type μ it gives a result of type μ, takes no account of the side effect on y. What is required is a more careful definition (in terms of the semantic domains) of what such functional types mean, which takes side effects into account.

By contrast, it is easy enough to give well-typing rules for the two new kinds of expression. We would extend the definition of wt by the following clauses:

(i) (c) If *letref* x_τ is active in a standard prefix $\bar{p}$, then $\bar{p} \mid x_\tau$ is wt. (Thus, all *letref*-bound occurrences of x must have the same type).

(vii) $\bar{p} \mid (x_\rho := \bar{e}_\sigma)_\tau$ is wt iff $\bar{p} \mid \bar{e}$ is wt, and *letref* x_ρ is active in $\bar{p}$, and $\rho = \sigma = \tau$.

(viii) $\bar{p} \mid (\textit{letref}\ x_\rho = \bar{e}_\rho\ \textit{in}\ \bar{e}'_\sigma)_\tau$ is wt iff $\bar{p} \mid \bar{e}$ and $\bar{p} \cdot \textit{letref}\ x_\rho \mid \bar{e}'$ are wt, and $\sigma = \tau$.

It is a routine matter to extend the algorithm $\mathscr{W}$ and the Syntactic Soundness Theorem to handle these clauses. But returning again to semantic considerations, there is a problem concerned with nonlocal assignments within λ-abstractions (procedures). Consider

$$\textit{let}\ g = (\textit{letref}\ y = \text{nil}\ \textit{in}\ \lambda z \cdot y := \text{cons}(z, y))$$

which sets up y as a hidden (own) variable for the function g. With our rules, g obtains generic type $\alpha \rightarrow \alpha$ *list*. Remembering that the value of an assignment is the value assigned, each call of g augments the hidden y, returning the augmented list as value. Now since α is generic, the expressions

$$g(2), \qquad g(\textit{true})$$

are both admissible within the scope of g, with types *int list* and *bool list*, respectively. But if they are evaluated in this order, the value of the second expression is a list whose two members are *true* and 2; it is *not* a list of Boolean values!

There are at least two possible solutions to this dilemma. One is to decree that in such a situation α is not a generic type variable. Another, which we adopt in ML, is to forbid nonlocal assignments to *polymorphic* assignable variables within λ-abstractions (procedures). Polymorphic assignable variables are still useful, for example, in programming iterations (*while* statements) which themselves can be given simple wt rules.

I believe that the second solution, even slightly relaxed, admits a Semantic Soundness Theorem. The details, however, are unattractive, and I have been discouraged (particularly after a useful discussion with John Reynolds) from attempting to complete the proof. What is rather needed is a language design which pays more respect to side effects; one approach might be to modify PASCAL by requiring that all variables assigned in a procedure be listed as output parameters of the procedure. But how to combine this with the rather useful properties of *own* variables is, as far as I know, an open problem in language design, and a good solution would be a valuable step forward. For a recent promising attempt to control side effects see Reynolds [13].

(3) To complete the list of nontrivial extensions which we have included in ML, consider the declaration (possibly recursive) of a new type operator in terms of old ones. Such a declaration may have nonglobal scope. If it is also accompanied by the declaration of a set of functions over the new type operator, and the explicit definition of the type operator is only available in defining the set of functions (not within the whole scope of the new type operator), one has a version of what is currently called *abstract* type. In ML, we would define the class of binary trees whose tips are labeled by objects of arbitrary type as follows, using the type variable α to stand for the type of tip labels:

$$\begin{array}{rl} \textit{absrectype} & \alpha\ \text{bitree} = \alpha + (\alpha\ \text{bitree} \times \alpha\ \text{bitree}) \\ \textit{with} & \text{sons}(t) = \cdots \\ \textit{and} & \text{maketree}(t, t') = \cdots \\ \textit{and} & \text{tiptree}(a) = \cdots \end{array}$$

in which only the omitted defining expressions are given access to the representation of bitrees. The defined functions are polymorphic, with generic types

$$\alpha \text{ bitree} \rightarrow (\alpha \text{ bitree})^2, \quad (\alpha \text{ bitree})^2 \rightarrow \alpha \text{ bitree}, \quad \alpha \rightarrow \alpha \text{ bitree}.$$

For full details of this construct, see [2]; we owe the construct partly to Lockwood Morris and to discussion with Jerry Schwarz. In this case the wt rules are also rather easy; we have not checked syntactic and semantic soundness, but suspect that there should be no great difficulty.

(4) Two features which contribute a kind of polymorphism have been completely ignored so far. The first is *coercions*. The expression $x := 42$, where x is a *real* assignable variable, is ill typed for us. However, there is no barrier to having the type checker report such instances of ill typing and allowing the compiler to receive the report and insert a coercion to rectify it.

The second feature is to allow certain procedures—either standard or user defined—to possess more than one type. We may wish "+" to possess $int^2 \rightarrow int$ and $real^2 \rightarrow real$, without of course possessing $\alpha^2 \rightarrow \alpha$ (which is the least general polytype having the two given types as instances). While we have not investigated the question, there appears to be a good possibility of superposing this feature upon our discipline.

6. Conclusion

We have presented a discipline of polymorphic programming which appears to be practically useful, and have given a rather simple type-checking algorithm. In a restricted language we have shown that this algorithm can be proved correct (the proof was factored into two Soundness Theorems). Though much work remains to be done, we hope to have made the point that the practice of type checking can and should by supported by semantic theory and proof.

Acknowledgments

I am indebted to the referees for their apposite comments on the first version of this paper, which drew my attention to some inaccuracies and also led me to a clearer exposition in several places. My thanks also go to Dorothy McKie for her careful preparation of two versions of the paper.

References

1. W. H. Burge, "Recursive Programming Techniques," Addison-Wesley, Reading, Mass., 1975.
2. M. Gordon, R. Milner, and C. Wadsworth, "Edinburgh LCF," CSR-11-77, Computer Science Dept., Edinburgh University, 1977.
3. M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth, A metalanguage for interactive proof in LCF, *in* "Proc. 5th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Tucson, Arizona, 1978."

4. D. GRIES AND N. GEHANI, Some ideas on data type in high-level lcnguages, *Comm. ACM.* **20** (1977), 414–420.
5. R. HINDLEY, The principal type-scheme of an object in combinatory logic, *Trans. Amer. Math. Soc.* **146** (1969), 29–60.
6. B. W. LAMPSON, J. J. HORNING, R. L. LONDON, J. G. MITCHELL, AND G. L. POPEK, Report on the programming language Euclid, *SIGPLAN Notices (ACM)* **12**, 2 (1977).
7. P. J. LANDIN, The next 700 programming languages, *Comm. ACM* **9** (1966), 157–164.
8. B. H. LISKOV AND S. ZILLES, Programming with abstract data types, *in* "Proc. of ACM SIGPLAN conference on Very High Level Languages," *SIGPLAN Notices (ACM)* **9** (1974), 50–59.
9. R. MILNER, "Models of LCF," Mathematical Centre, Amsterdam, Tracts, Vol. 82, pp. 49–63, 1976.
10. J. H. MORRIS, "Lambda-Calculus Models of Programming Languages," Ph.D. Thesis, MAC-TR-57, MIT, 1968.
11. G. PLOTKIN, A power-domain construction, *SIAM J. Comput.* **5** (1976), 452–487.
12. J. C. REYNOLDS, "Towards a Theory of Type Structure," Systems and Inform. Sci., Syracuse University, 1974.
13. J. C. REYNOLDS, Syntactic control of interference, *in* "Proc. 5th ACM Symposium on Principles of Programming Languages, Tucson, Arizona, 1978," pp. 39–46.
14. J. A. ROBINSON, A machine-oriented logic based on the resolution principle, *J. Assoc. Comput. Mach.* **12** (1965), 23–41.
15. D. SCOTT, Lattice theoretic models for various type-free calculi, *in* "Proc. 4th International Congress for Logic, Methodology and Philosophy of Science, Bucharest, Rumania, 1972."
16. D. SCOTT AND C. STRACHEY, Towards a mathematical semantics for computer languages, *in* "Proc. Symposium on Computers and Automata," Vol. 21, Microwave Res. Inst. Symposia Series, Polytech. Inst. of Brooklyn, 1971.
17. D. SCOTT, Data types as lattices, *SIAM J. Comput.* **5** (1976), 522–587.
18. A. SHAMIR AND W. WADGE, Data types as objects, *in* "Proc. 4th ICALP Conference, Turku, Finland, 1977."
19. C. STRACHEY, Fundamental concepts in programming languages, Notes for the International Summer School in Computer Programming, Copenhagen, 1967.
20. R. D. TENNENT, On a new approach to representation-independent data classes, *Acta Inform.* **8** (1977), 315–324.
21. B. WEGBREIT, The treatment of types in EL1, *Comm. ACM* **17** (1974), 251–264.
22. A. VAN WIJNGAARDEN ET AL., Revised report on the algorithmic language ALGOL 68, *Acta Informatica* **5** (1975), 1–236.
23. R. A. WULF, R. L. LONDON, AND M. SHAW, "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology," ISI/RR-76-46, Univ. of California, 1976.