THREE APPROACHES TO TYPE STRUCTURE[†]

John C. Reynolds

Syracuse University

Syracuse, New York 13210, U.S.A.

ABSTRACT We examine three disparate views of the type structure of programming languages: Milner's type deduction system and polymorphic let construct, the theory of subtypes and generic operators, and the polymorphic or second-order typed lambda calculus. These approaches are illustrated with a functional language including product, sum and list constructors. The syntactic behavior of types is formalized with type inference rules, but their semantics is treated intuitively.

1. INTRODUCTION

The division of programming languages into two species, typed and untyped, has engendered a long and rather acrimonious debate. One side claims that untyped languages preclude compile-time error checking and are succinct to the point of unintelligibility, while the other side claims that typed languages preclude a variety of powerful programming techniques and are verbose to the point of unintelligibility.

From the theorist's point of view, both sides are right, and their arguments are the motivation for seeking type systems that are more flexible and succinct than those of existing typed languages. This goal has inspired a substantial volume of theoretical work in the last few years. In this paper I will attempt to survey some of this work at a level that I hope will reveal its implications for future languages.

The main difficulty that I face is that type theory has moved in several directions that, as far as we presently know, are incompatible with one another. This situation dictates the organization of this paper: Section 2 lays a groundwork that is common to all directions, while each later section is devoted to a particular direction. Thus the later sections are largely independent of one another.

Primarily because of my limited knowledge, this survey will be far from comprehensive. Neither algebraic data types [1-3] nor conjunctive types [4,5] will be covered. Nor will I describe any of several fascinating systems [6-9] in which types and values are so intertwined that types become full-blown program specifications and programs become constructive proofs that such specifications are satisfiable.

Moreover, I will consider only functional languages, since the essential character of type structure is revealed more clearly without the added complexity of imperative features. (I believe that the proper type structure for Algol-like languages is obtained from the subtype discipline of Section 4 by taking "types" to be "phrase

[†]Work supported by National Science Foundation Grant MCS-8017577.

types", built from primitive types such as integer expression, real variable, and command [10-12].)

To be accessible to readers who are untrained in mathematical semantics, our exposition will be formal but not rigorous. In particular, we will often discuss semantics at an intuitive level, speaking of the set denoted by a type or the function denoted by an expression when rigorously we should speak of the domain denoted by a type or the continuous function denoted by an expression. Regrettably, this level of discourse will obscure some profound controversies about the semantics of types.

Finally, I must apologize for omissions and errors due to either ignorance or haste. This is a preliminary report, and I would welcome suggestions from readers for corrections or extensions.

2. THE BASE LANGUAGE

Although the approaches to type structure that we are going to survey are incompatible with one another, they are all built upon a common view of what types are all about. In this section, we will formalize this view as a "base" language, in terms of which the various approaches can be described as extensions or modifications. First we will introduce the expressions of the base language as though it were a typeless language. Then we will introduce types and give rules for inferring the types of expressions. Finally we will show how expressions can be augmented to contain enough type information that the inference of their types becomes trivial.

Because it is intended for illustrative purposes, the base language is more complicated than a well-designed functional language should be. In several instances, it contains similar constructs (e.g. numbered and named alternatives) that are both included only because they exhibit significant differences in some extension of the language.

2a. Expressions

To define expressions we will give their abstract syntax, avoiding any formalization of precedence or implicit parenthesization. In this definition we write K for the set of positive integers, Z for the set of all integers, I for the (countably infinite) set of identifiers, and E for the set of expressions. The simplest kinds of expressions are identifiers (used as variables):

E ::= I (S1)

constants denoting integers:

E ::= Z (S2)

boolean values:

E ::= true | false (S3)

and primitive operations on integers and boolean values, such as

E ::= add | equals (S4)

Less trivially, we have lambda expressions to denote functions, and a notation for function application:

E ::= λI. E | E E (S5)

Note that we do not require the operand of function application to be parenthesized, so that one can write f x instead of f(x). We will assume that application is left associative and that λ has a lower precedence than application, e.g. λx. λy. f y x means λx. (λy. ((f y) x)).

Informally, the meaning of functions is "explained" by the rule of beta-reduction

$$(\lambda i.\ e_1)\ e_2 = e_1\big|_{i \to e_2}\ , \qquad \text{(R5)}$$

where the right side denotes the result of substituting e_2 for i in e_1 (with renaming of bound identifiers in e_1 that occur free in e_2). For example, (λx. f x y x)(g a b) has the same meaning as f (g a b) y (g a b).

The rule of beta-reduction implies that our language has "call-by-name" semantics. For example, if i does not occur in e_1 then (λi. e_1) e_2 has the same meaning as e_1, even if e_2 is an expression whose evaluation never terminates. While much of what we are going to say is equally applicable to call by value, we prefer the greater elegance and generality of call by name (particularly since the development of lazy evaluation [13-14] has led to its efficient implementability).

To avoid any special notation for functions of several arguments, we will use the device of Currying, e.g. we will regard add as a function that accepts an integer and yields a function that accepts an integer and yields an integer, so that add 3 4 = 7. In general, where one might expect $\lambda(x_1, \ldots, x_n).\ e$ we will write $\lambda x_1.\ \ldots\ \lambda x_n.\ e$, and where one might expect $f(e_1, \ldots, e_n)$ we will write $(\ \ldots\ (f\ e_1)\ \ldots\ e_n)$ or, with implicit parenthesization, $f\ e_1\ \ldots\ e_n$.

Next, we introduce notation for the construction and analysis of records. Here there are two possible approaches, depending upon whether fields are numbered or named. For records with numbered fields, we use the syntax

E ::= <E, ... , E> | E.K (S6)

For example, <x, y> denotes a two-field record whose first field is x and second field is y, and if z denotes a two-field record then z.1 and z.2 denote its fields. In general, the meaning of these constructions is determined by the reduction rule

$$\langle e_1, \ldots, e_n\rangle.k = e_k \qquad \text{when } 1 \le k \le n\ . \qquad \text{(R6)}$$

Notice that this rule (as with beta reduction) implies a call-by-name semantics. For example, $\langle e_1, e_2\rangle.1 = \langle e_2, e_1\rangle.2 = e_1$, even when e_2 does not terminate.

For records with named fields, we use the syntax

$$E ::= \langle I_1: E, \ldots, I_n: E \rangle \mid E.I \qquad (S7)$$

with the restriction that the identifiers preceding the colons must be distinct. The reduction rule is

$$\langle i_1: e_1, \ldots, i_n: e_n \rangle.i_k = e_k \qquad \text{when } 1 \le k \le n . \qquad (R7)$$

With named fields, the value of a record is independent of the order of its fields, e.g. <real: x, imag: y> = <imag: y, real: x>.

We also introduce notation for alternatives (often called a sum, disjoint union, or variant-record construct):

$$E ::= \underline{\text{inject}}\ K\ E \mid \underline{\text{choose}}(E, \ldots, E) \qquad (S8)$$

The value of inject k e is the value of e "tagged" with k. If $f_1, \ldots, f_n$ are functions, then choose($f_1, \ldots, f_n$) is a function that, when applied to the value x tagged with k, yields f_k x. Thus the appropriate reduction rule is

$$\underline{\text{choose}}(f_1, \ldots, f_n)(\underline{\text{inject}}\ k\ e) = f_k\ e \qquad \text{when } 1 \le k \le n . \qquad (R8)$$

(Strictly speaking, "tagging" is pairing, so that inject k e is a pair like <k, e>. But we consider these to be different kinds of pairs so that, for example, (inject k e).1 is meaningless.)

In some languages, the analysis of alternatives is performed by some form of case construction that can be defined in terms of choose, e.g.

$$\underline{\text{altcase}}\ i: e\ \underline{\text{of}}\ (e_1, \ldots, e_n) \equiv \underline{\text{choose}}(\lambda i.e_1, \ldots, \lambda i.e_n)\ e .$$

However, the choose construction is conceptually simpler since it does not involve identifier binding.

The tags of alternatives, like the fields of records, can be named instead of numbered. For named alternatives, we will use the syntax

$$E ::= \underline{\text{inject}}\ I\ E \mid \underline{\text{choose}}(I_1: E, \ldots, I_n: E) \qquad (S9)$$

with the restriction that the identifiers preceding the colons must be distinct. The reduction rule is

$$\underline{\text{choose}}(i_1: f_1, \ldots, i_n: f_n)(\underline{\text{inject}}\ i_k\ e) = f_k\ e \qquad \text{when } 1 \le k \le n , \qquad (R9)$$

and the meaning of choose($i_1: f_1, \ldots, i_n: f_n$) is independent of the order of the components $i_k: f_k$.

For the construction of lists, we will use the primitives of LISP:

$$E ::= \underline{\text{nil}} \mid \underline{\text{cons}}\ E\ E \qquad (S10a)$$

where nil denotes the empty list and cons x y denotes the list whose first element is x and whose remainder is y. For the analysis of lists, however, we will deviate substantially from LISP:

$$E ::= \underline{\text{lchoose}}\ E\ E \qquad (S10b)$$

The value of lchoose e f is a function that, when applied to the empty list, yields e and, when applied to a list with first element x and remainder y, yields f x y. More formally, we have the reduction rules

$$\begin{aligned} &(\underline{\text{lchoose}}\ e\ f)\ \underline{\text{nil}} = e\ , \\ &(\underline{\text{lchoose}}\ e\ f)\ (\underline{\text{cons}}\ x\ y) = f\ x\ y\ . \end{aligned} \tag{R10}$$

The lchoose operation can be defined in terms of the conventional LISP primitives:

$$\underline{\text{lchoose}}\ e\ f = \lambda \ell .\ \underline{\text{if}}\ \text{null}\ \ell\ \underline{\text{then}}\ e\ \underline{\text{else}}\ f\ (\text{car}\ \ell)\ (\text{cdr}\ \ell)\ ,$$

and the LISP primitives can be defined in terms of lchoose and an error operation:

$$\begin{aligned} \text{null} &= \underline{\text{lchoose}}\ \underline{\text{true}}\ (\lambda x.\ \lambda y.\ \underline{\text{false}})\ , \\ \text{car} &= \underline{\text{lchoose}}\ \underline{\text{error}}\ (\lambda x.\ \lambda y.\ x)\ , \\ \text{cdr} &= \underline{\text{lchoose}}\ \underline{\text{error}}\ (\lambda x.\ \lambda y.\ y)\ . \end{aligned}$$

However, in a typed language lchoose is preferable to the LISP primitives since it converts the common error of applying car or cdr to the empty list into a type error.

For the definition of identifiers we use Landin's let construction [15] (albeit with a call-by-name rather than a call-by-value semantics). The syntax is

$$E ::= \underline{\text{let}}\ I = E\ \underline{\text{in}}\ E \tag{S11}$$

and the reduction rule is

$$\underline{\text{let}}\ i = e_2\ \underline{\text{in}}\ e_1 = e_1\Big|_{i \to e_2}\ .$$

Of course, as noted by Landin, let can be defined in terms of λ and application:

$$\underline{\text{let}}\ i = e_2\ \underline{\text{in}}\ e_1 \equiv (\lambda i.\ e_1)\ e_2\ . \tag{R11}$$

However, we will regard $\underline{\text{let}}\ i = e_2\ \underline{\text{in}}\ e_1$ as an independent construction in its own right, since its typing behavior is significantly different than that of $(\lambda i.\ e_1)\ e_2$.

Finally, we introduce a conditional expression

$$E ::= \underline{\text{if}}\ E\ \underline{\text{then}}\ E\ \underline{\text{else}}\ E \tag{S12}$$

with the reduction rules

$$\begin{aligned} &\underline{\text{if}}\ \underline{\text{true}}\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2 = e_1\ , \\ &\underline{\text{if}}\ \underline{\text{false}}\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2 = e_2\ , \end{aligned} \tag{R12}$$

a case (branch-on-integer) expression

$$E ::= \underline{\text{case}}\ E\ \underline{\text{of}}\ (E, \ \ldots\ , E) \tag{S13}$$

with the reduction rule

$$\underline{\text{case}}\ k\ \underline{\text{of}}\ (e_1, \ \ldots\ , e_n) = e_k \qquad \text{when } 1 \le k \le n\ , \tag{R13}$$

and a fixed-point expression

$$E ::= \underline{\text{fix}}\ E \tag{S14}$$

with the reduction rule

fix e = e(fix e) . (R14)

The last is the key to recursive definition. For example, one can use it to define McCarthy's [16]

label i: e ≡ fix(λi. e) ,

or Landin's

letrec i = e_2 in e_1 ≡ let i = fix(λi. e_2) in e_1 .

However, just as with choose, fix is conceptually simpler since it does not involve identifier binding. (We are purposely neglecting the complications of the multiple letrec, which is needed to define simultaneously recursive functions, and whose definition in terms of fix is rather messy.)

It should be noticed that our choice of call-by-name semantics implies that fix can be used to define infinite lists. For example,

fix(λℓ. cons 0 ℓ)

denotes the infinite list (0, 0, ...), and

fix(λf. λn. cons n (f (add 1 n)))

denotes the function mapping an integer n into the infinite list (n, n+1, ...).

2b. Types and their Inference Rules

We now introduce the types of our base language. Intuitively:

int denotes the set of integers.

bool denotes the set {true, false}.

$\omega \to \omega'$ denotes the set of functions that map values belonging to (the set denoted by) ω into values belonging to (the set denoted by) ω'.

prod(ω_1, ... , ω_n) denotes the set of n-field records in which, for $1 \le k \le n$, the kth field belongs to ω_k.

prod(i_1:ω_1, ... , i_n:ω_n) denotes the set of records with fields named i_1, ... , i_n, in which each i_k names a field belonging to ω_k.

sum(ω_1, ... , ω_n) denotes the set of tagged values such that the tag is an integer between 1 and n, and a value with tag k belongs to ω_k.

sum(i_1:ω_1, ... , i_n:ω_n) denotes the set of tagged values such that the tag belongs to {i_1, ... , i_n}, and a value with tag i_k belongs to ω_k.

list ω denotes the set of lists whose elements belong to ω.

More formally, the set Ω of types is defined by the abstract syntax

$$\begin{aligned}\Omega ::= &\ \underline{\text{int}} \mid \underline{\text{bool}} \\ &\mid \underline{\text{prod}}(\Omega, \ldots, \Omega) \mid \underline{\text{prod}}(I{:}\Omega, \ldots, I{:}\Omega) \\ &\mid \underline{\text{sum}}(\Omega, \ldots, \Omega) \mid \underline{\text{sum}}(I{:}\Omega, \ldots, I{:}\Omega) \\ &\mid \underline{\text{list}}\ \Omega\end{aligned}$$

with the proviso that the pairs $I{:}\Omega$ in $\underline{\text{prod}}(I{:}\Omega, \ldots, I{:}\Omega)$ or $\underline{\text{sum}}(I{:}\Omega, \ldots, I{:}\Omega)$ must begin with distinct identifiers, and that the ordering of these pairs is irrelevant.

Occasionally, we will need to speak of type expressions, which are defined by the same syntax with the added production

$$\Omega ::= T$$

where T is a countably infinite set of type variables.

We will assume that $\rightarrow$ is right associative and has a lower precedence than the other type operators. Thus for example, $\underline{\text{int}} \rightarrow \underline{\text{list}}\ \underline{\text{int}} \rightarrow \underline{\text{int}}$ stands for $\underline{\text{int}} \rightarrow ((\underline{\text{list}}\ \underline{\text{int}}) \rightarrow \underline{\text{int}})$.

Roughly speaking, an expression has a type if its value belongs to that type. But of course, just as the value of an expression depends upon the values of the identifiers occurring free within it, so the type of an expression will depend upon the types of the identifiers occurring free within it. To deal with this complication, we introduce the notion of a typing.

Let e be an expression, π (often called a type assignment) be a mapping of (at least) the identifiers occurring free in e into types, and ω be a type. Then

$$\pi \mid\!\!- e{:}\ \omega$$

is called a typing, and read "e has type ω under π". For example, the following are valid typings:

$$\begin{aligned}&x{:}\ \underline{\text{int}},\ f{:}\ \underline{\text{int}} \rightarrow \underline{\text{int}} \mid\!\!- f(f\ x){:}\ \underline{\text{int}} \\ &f{:}\ \underline{\text{int}} \rightarrow \underline{\text{int}} \mid\!\!- \lambda x.\ f(f\ x){:}\ \underline{\text{int}} \rightarrow \underline{\text{int}} \\ &\mid\!\!- \lambda f.\ \lambda x.\ f(f\ x){:}\ (\underline{\text{int}} \rightarrow \underline{\text{int}}) \rightarrow \underline{\text{int}} \rightarrow \underline{\text{int}} \\ &\mid\!\!- \lambda f.\ \lambda x.\ f(f\ x){:}\ (\underline{\text{bool}} \rightarrow \underline{\text{bool}}) \rightarrow \underline{\text{bool}} \rightarrow \underline{\text{bool}}\ .\end{aligned}$$

Notice that, as illustrated by the last two lines, a typing of a closed expression can have an empty type assignment, and two typings can give different types to the same expression, even under the same type assignment.

We will now give rules for inferring valid typings of our base language. Each of these inference rules consists of zero or more premises separated by a long horizontal line from a conclusion, and contains various symbols called metavariables. An instance of a rule is obtained by replacing the metavariables by phrases of the appropriate kind as described by the following table (and occasionally subject to restrictions stated with the rule itself):

Metavariable	Kind of phrase
π	type assignments
$i\ i_1\ i_n\ i_k$	identifiers
$\omega\ \omega'\ \omega_1\ \omega_k\ \omega_n$	types
z	integers
$e\ e_1\ e_2\ e_n$	expressions
k	positive integers $(k > 0)$
n	nonnegative integers $(n \geq 0)$

Every instance of every rule has the property that, if the premises of the instance are all valid typings, then the conclusion of the instance is a valid typing. Thus a typing can be proved to be valid by giving a list of typings, ending with the typing in question, in which every typing is the conclusion of an instance of a rule whose premises all occur earlier in the list.

The following is a list of the inference rules, ordered to parallel the abstract syntax of the base language:

$$\frac{}{\pi \mid\!- i: \omega} \quad \text{when } i \text{ is in the domain of } \pi, \text{ and } \pi \text{ assigns } \omega \text{ to } i. \tag{I1}$$

$$\frac{}{\pi \mid\!- z: \underline{\text{int}}} \qquad \frac{}{\pi \mid\!- \underline{\text{true}}: \underline{\text{bool}}} \qquad \frac{}{\pi \mid\!- \underline{\text{false}}: \underline{\text{bool}}} \tag{I2,I3}$$

$$\frac{}{\pi \mid\!- \underline{\text{add}}: \underline{\text{int}} \to \underline{\text{int}} \to \underline{\text{int}}} \qquad \frac{}{\pi \mid\!- \underline{\text{equals}}: \underline{\text{int}} \to \underline{\text{int}} \to \underline{\text{bool}}} \tag{I4}$$

$$\frac{\pi, i: \omega \mid\!- e: \omega'}{\pi \mid\!- \lambda i.\ e: \omega \to \omega'} \qquad \frac{\pi \mid\!- e_1: \omega \to \omega' \quad \pi \mid\!- e_2: \omega}{\pi \mid\!- e_1\ e_2: \omega'} \tag{I5}$$

$$\frac{\pi \mid\!- e_1: \omega_1 \quad \vdots \quad \pi \mid\!- e_n: \omega_n}{\pi \mid\!- \langle e_1, \ldots, e_n \rangle: \underline{\text{prod}}(\omega_1, \ldots, \omega_n)} \tag{I6a}$$

$$\frac{\pi \mid\!- e: \underline{\text{prod}}(\omega_1, \ldots, \omega_n)}{\pi \mid\!- e.k: \omega_k} \quad \text{when } 1 \leq k \leq n \tag{I6b}$$

$$\frac{\pi \mid\!- e_1: \omega_1 \quad \vdots \quad \pi \mid\!- e_n: \omega_n}{\pi \mid\!- \langle i_1{:}e_1, \ldots, i_n{:}e_n \rangle: \underline{\text{prod}}(i_1{:}\omega_1, \ldots, i_n{:}\omega_n)} \tag{I7a}$$

$$\frac{\pi \vdash e: \underline{\text{prod}}(i_1{:}\omega_1, \ \ldots \ , i_n{:}\omega_n)}{\pi \vdash e.i_k: \omega_k} \quad \text{when } 1 \leq k \leq n \qquad \text{(I7b)}$$

$$\frac{\pi \vdash e: \omega_k}{\pi \vdash \underline{\text{inject}}\ k\ e: \underline{\text{sum}}(\omega_1, \ \ldots \ , \omega_n)} \quad \text{when } 1 \leq k \leq n \qquad \text{(I8a)}$$

$$\frac{\begin{array}{l}\pi \vdash e_1: \omega_1 \to \omega \\ \quad \vdots \\ \pi \vdash e_n: \omega_n \to \omega\end{array}}{\pi \vdash \underline{\text{choose}}(e_1, \ \ldots \ , e_n): \underline{\text{sum}}(\omega_1, \ \ldots \ , \omega_n) \to \omega} \qquad \text{(I8b)}$$

$$\frac{\pi \vdash e: \omega_k}{\pi \vdash \underline{\text{inject}}\ i_k\ e: \underline{\text{sum}}(i_1{:}\omega_1, \ \ldots \ , i_n{:}\omega_n)} \quad \text{when } 1 \leq k \leq n \qquad \text{(I9a)}$$

$$\frac{\begin{array}{l}\pi \vdash e_1: \omega_1 \to \omega \\ \quad \vdots \\ \pi \vdash e_n: \omega_n \to \omega\end{array}}{\pi \vdash \underline{\text{choose}}(i_1{:}e_1, \ \ldots \ , i_n{:}e_n): \underline{\text{sum}}(i_1{:}\omega_1, \ \ldots \ , i_n{:}\omega_n) \to \omega} \qquad \text{(I9b)}$$

$$\frac{}{\pi \vdash \underline{\text{nil}}: \underline{\text{list}}\ \omega} \qquad \frac{\begin{array}{l}\pi \vdash e_1: \omega \\ \pi \vdash e_2: \underline{\text{list}}\ \omega\end{array}}{\pi \vdash \underline{\text{cons}}\ e_1\ e_2: \underline{\text{list}}\ \omega} \qquad \text{(I10a,b)}$$

$$\frac{\begin{array}{l}\pi \vdash e_1: \omega' \\ \pi \vdash e_2: \omega \to \underline{\text{list}}\ \omega \to \omega'\end{array}}{\pi \vdash \underline{\text{lchoose}}\ e_1\ e_2: \underline{\text{list}}\ \omega \to \omega'} \qquad \text{(I10c)}$$

$$\frac{\begin{array}{l}\pi \vdash e_2: \omega \\ \pi, \ i: \omega \vdash e_1: \omega'\end{array}}{\pi \vdash \underline{\text{let}}\ i = e_2\ \underline{\text{in}}\ e_1: \omega'} \qquad \text{(I11)}$$

$$\frac{\begin{array}{l}\pi \vdash e: \underline{\text{bool}} \\ \pi \vdash e_1: \omega \\ \pi \vdash e_2: \omega\end{array}}{\pi \vdash \underline{\text{if}}\ e\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2: \omega} \qquad \frac{\begin{array}{l}\pi \vdash e: \underline{\text{int}} \\ \pi \vdash e_1: \omega \\ \quad \vdots \\ \pi \vdash e_n: \omega\end{array}}{\pi \vdash \underline{\text{case}}\ e\ \underline{\text{of}}\ (e_1, \ \ldots \ , e_n): \omega} \qquad \text{(I12,I13)}$$

$$\frac{\pi \vdash e: \omega \to \omega}{\pi \vdash \underline{fix}\ e: \omega} \qquad (I14)$$

In rules I5a and I11, the notation π, i:ω denotes the type assignment that assigns ω to i and assigns to all other identifiers in the domain of π the same type that is assigned by π.

To illustrate the use of these rules we give a proof of the validity of a typing of the expression

$$\underline{fix}(\lambda ap.\ \lambda x.\ \lambda y.\ \underline{lchoose}\ y\ (\lambda n.\ \lambda z.\ \underline{cons}\ n\ (ap\ z\ y))\ x)\ ,$$

which denotes a function for appending two lists. At the right of each typing in the proof we indicate the rule of which it is a conclusion. To save space, we write π_1 and π_2 to abbreviate the following type assignments:

π_1 = ap: $\underline{list}\ \underline{int} \to \underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$, x: $\underline{list}\ \underline{int}$, y: $\underline{list}\ \underline{int}$

$\pi_2 = \pi_1$, n: $\underline{int}$, z: $\underline{list}\ \underline{int}$

Then the proof is:

$\pi_2 \vdash$ ap: $\underline{list}\ \underline{int} \to \underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$ (I1)

$\pi_2 \vdash$ z: $\underline{list}\ \underline{int}$ (I1)

$\pi_2 \vdash$ ap z: $\underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$ (I5b)

$\pi_2 \vdash$ y: $\underline{list}\ \underline{int}$ (I1)

$\pi_2 \vdash$ ap z y: $\underline{list}\ \underline{int}$ (I5b)

$\pi_2 \vdash$ n: $\underline{int}$ (I1)

$\pi_2 \vdash \underline{cons}$ n (ap z y): $\underline{list}\ \underline{int}$ (I10b)

π_1, n: $\underline{int} \vdash \lambda z.\ \underline{cons}$ n (ap z y): $\underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$ (I5a)

$\pi_1 \vdash \lambda n.\ \lambda z.\ \underline{cons}$ n (ap z y): $\underline{int} \to \underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$ (I5a)

$\pi_1 \vdash$ y: $\underline{list}\ \underline{int}$ (I1)

$\pi_1 \vdash \underline{lchoose}$ y ($\lambda n.\ \lambda z.\ \underline{cons}$ n (ap z y)): $\underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$ (I10c)

$\pi_1 \vdash$ x: $\underline{list}\ \underline{int}$ (I1)

$\pi_1 \vdash \underline{lchoose}$ y ($\lambda n.\ \lambda z.\ \underline{cons}$ n (ap z y)) x: $\underline{list}\ \underline{int}$ (I5b)

ap: $\underline{list}\ \underline{int} \to \underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$, x: $\underline{list}\ \underline{int}$ (I5a)
$\vdash \lambda y.\ \underline{lchoose}$ y ($\lambda n.\ \lambda z.\ \underline{cons}$ n (ap z y)) x: $\underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$

ap: $\underline{list}\ \underline{int} \to \underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$ (I5a)
$\vdash \lambda x.\ \lambda y.\ \underline{lchoose}$ y ($\lambda n.\ \lambda z.\ \underline{cons}$ n (ap z y)) x:
$\underline{list}\ \underline{int} \to \underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$

$\vdash \lambda ap.\ \lambda x.\ \lambda y.\ \underline{lchoose}$ y ($\lambda n.\ \lambda z.\ \underline{cons}$ n (ap z y)) x: (I5a)
$(\underline{list}\ \underline{int} \to \underline{list}\ \underline{int} \to \underline{list}\ \underline{int}) \to \underline{list}\ \underline{int} \to \underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$

$\vdash \underline{fix}(\lambda ap.\ \lambda x.\ \lambda y.\ \underline{lchoose}$ y ($\lambda n.\ \lambda z.\ \underline{cons}$ n (ap z y)) x): (I14)
$\underline{list}\ \underline{int} \to \underline{list}\ \underline{int} \to \underline{list}\ \underline{int}$

As a second example, the reader might prove the typing

$$\vdash \underline{\text{fix}}(\lambda \text{red}.\ \lambda \ell.\ \lambda \text{f}.\ \lambda \text{a}.\ \underline{\text{lchoose}}\ \text{a}\ (\lambda \text{n}.\ \lambda \text{z}.\ \text{f n (red z f a)})\ \ell): \underline{\text{list}}\ \underline{\text{int}} \to (\underline{\text{int}} \to \underline{\text{bool}} \to \underline{\text{bool}}) \to \underline{\text{bool}} \to \underline{\text{bool}}$$

of the "reduce" function that, when applied to a list $(x_1, \ldots, x_n)$, a function f, and a value a, gives $f\ x_1\ (f\ x_2\ (\ \ldots\ (f\ x_n\ a)\ \ldots\ ))$.

2c. Explicit Typing

So far we have taken the attitude that types are properties of expressions that can be inferred but do not actually appear within expressions. This view is subject to several criticisms:

The problem of generating typings of an expression is an instance of proof generation, for which - in general - there may be no efficient algorithm or even no algorithm at all. (Although we will see in the next section that an efficient typing algorithm is known for a slight variation of our base language, such algorithms are not known for some of the language extensions that will be discussed later.)

An expression will often have more than one valid typing. For example, the typings shown in the previous subsection remain valid if int and bool are replaced by arbitrary types. Thus, if one takes the view that different typings lead to different meanings, then our base language is ambiguous.

Presumably the competent programmer knows the types of the programs he writes. By preventing him from communicating this knowledge in his programs, we are precluding valuable error checking and degrading the intelligibility of programs.

These criticisms suggest modifying our base language so that expressions contain type information. We will call such a modified language explicitly typed if it satisfies two criteria:

(1) Every expression, under a particular type assignment, has at most one type. (Thus there is a partial function, called a typing function, that maps an expression e and a type assignment π into the type of e under π.)

(2) The type of any expression, under a particular type assignment, is a function of the types of its immediate subexpressions under particular (though perhaps different) type assignments.

In the specific case of our base language, to obtain explicit typing we must require type information to appear in four contexts: lambda expressions, inject operations for numbered and named alternatives, and nil. (In each of these contexts, we will write the type information as a subscript.) Thus explicit typing requires

the modification of four pairs of abstract syntax and type-inference rules:

$$E ::= \lambda I_{\Omega}.\ E \qquad \frac{\pi,\ i:\ \omega \mid\!-\ e:\ \omega'}{\pi \mid\!-\ \lambda i_{\omega}.\ e:\ \omega \to \omega'} \qquad \text{(E5a)}$$

$$E ::= \underline{\text{inject}}_{\Omega,\ldots,\Omega}\ K\ E \qquad \frac{\pi \mid\!-\ e:\ \omega_k \qquad \text{when } 1 \le k \le n}{\pi \mid\!-\ \underline{\text{inject}}_{\omega_1,\ldots,\omega_n}\ k\ e:\ \underline{\text{sum}}(\omega_1,\ \ldots\ ,\ \omega_n)} \qquad \text{(E8a)}$$

$$E ::= \underline{\text{inject}}_{I:\Omega,\ldots,I:\Omega}\ I\ E \qquad \frac{\pi \mid\!-\ e:\ \omega_k \qquad \text{when } 1 \le k \le n}{\pi \mid\!-\ \underline{\text{inject}}_{i_1:\omega_1,\ldots i_n:\omega_n}\ i_k\ e:\ \underline{\text{sum}}(i_1:\omega_1,\ \ldots\ ,\ i_n:\omega_n)} \qquad \text{(E9a)}$$

$$E ::= \underline{\text{nil}}_{\Omega} \qquad \frac{}{\pi \mid\!-\ \underline{\text{nil}}_{\omega}:\ \underline{\text{list}}\ \omega} \qquad \text{(E10a)}$$

(Actually, one further restriction must be imposed on the base language to obtain explicit typing: we exclude the degenerate expressions $\underline{\text{case}}$ e $\underline{\text{of}}$ () and $\underline{\text{choose}}()$, which could have types ω and $\underline{\text{sum}}() \to \omega$ for arbitrary ω.)

Somewhat surprisingly, it is not necessary to modify the $\underline{\text{let}}$ construction, since the type of $\underline{\text{let}}$ i = e_2 $\underline{\text{in}}$ e_1 under π must be the type of e_1 under π, i:ω, where ω is the type of e_2 under π. (However, a similar argument does not work for the $\underline{\text{letrec}}$ construction.)

For example, the following are explicitly typed versions of the expressions whose typing was discussed in the previous subsection:

$\underline{\text{fix}}(\lambda \text{ap}_{\text{list int} \to \text{list int} \to \text{list int}}.\ \lambda x_{\text{list int}}.\ \lambda y_{\text{list int}}.$
$\quad \underline{\text{lchoose}}\ y\ (\lambda n_{\text{int}}.\ \lambda z_{\text{list int}}.\ \underline{\text{cons}}\ n\ (\text{ap}\ z\ y))\ x)\ ,$

$\underline{\text{fix}}(\lambda \text{red}_{\text{list int} \to (\text{int} \to \text{bool} \to \text{bool}) \to \text{bool} \to \text{bool}}.$
$\quad \lambda \ell_{\text{list int}}.\ \lambda f_{\text{int} \to \text{bool} \to \text{bool}}.\ \lambda a_{\text{bool}}.$
$\quad \underline{\text{lchoose}}\ a\ (\lambda n_{\text{int}}.\ \lambda z_{\text{list int}}.\ f\ n\ (\text{red}\ z\ f\ a))\ \ell)\ .$

Although the rigorous semantics of types is beyond the scope of this paper, it should be mentioned that the pragmatic arguments about implicit versus explicit typing reflect profoundly different views of the meaning of types. Consider, for example, the untyped expression $\lambda x.\ x$, and the explicitly typed expressions $\lambda x_{\text{int}}.\ x$ and $\lambda x_{\text{int} \to \text{int}}.\ x$. Three views can be taken of the meaning of these expressions:

(1) All three expressions have the same meaning. The types in the explicitly typed expressions are merely assertions about how these expressions will be used [17].

(2) The expressions have different meanings, but the meanings of the typed expressions are functions of the meanings of the untyped expression and the meanings of the types int and int → int [18, 19].

(3) The typed expressions have meanings, but the untyped expression does not. It is not sensible to speak of a function that is applicable to all conceivable values [20].

3. TYPE DEDUCTION AND THE POLYMORPHIC let

3a. Type Deduction

An efficient algorithm has been discovered (independently) by J. R. Hindley and R. Milner [21,22] that is capable of deducing the valid typings of expressions in our base language (with the partial exception of inject expressions). It is based upon the concept of a principal typing of an expression.

A typing scheme is a typing containing type variables; more precisely, it is like a typing except that type expressions may occur in place of types (including within type assignments). A principal typing of an expression is a typing scheme such that the valid typings of the expression are exactly those that can be obtained from the principal typing by substituting arbitrary types for the type variables (and perhaps extending the type assignment to irrelevant identifiers).

For example, the following are principal typings:

$x{:}\alpha,\ f{:}\ \alpha \to \alpha \ \vdash\ f(f\ x){:}\ \alpha$,

$f{:}\ \alpha \to \alpha \ \vdash\ \lambda x.\ f(f\ x){:}\ \alpha \to \alpha$,

$\vdash\ \lambda f.\ \lambda x.\ f(f\ x){:}\ (\alpha \to \alpha) \to \alpha \to \alpha$,

$\vdash\ \underline{\text{fix}}(\lambda ap.\ \lambda x.\ \lambda y.\ \underline{\text{lchoose}}\ y\ (\lambda n.\ \lambda z.\ \underline{\text{cons}}\ n\ (ap\ z\ y))\ x){:}$
$\text{list}\ \alpha \to \text{list}\ \alpha \to \text{list}\ \alpha$,

$\vdash\ \underline{\text{fix}}(\lambda red.\ \lambda \ell.\ \lambda f.\ \lambda a.\ \underline{\text{lchoose}}\ a\ (\lambda n.\ \lambda z.\ f\ n\ (red\ z\ f\ a))\ \ell){:}$
$\text{list}\ \alpha \to (\alpha \to \beta \to \beta) \to \beta \to \beta$.

Throughout this paper we will use lower case Greek letters as type variables. It is clear that the choice of variables in a principal typing is arbitrary.

Hindley and Milner showed that an expression has a principal typing if it has any typing, and that a principal typing (or nonexistence thereof) of an expression can be computed from principal typings of its subexpressions by using Robinson's unification algorithm [23]. Although we will not give a complete description of their algorithm, its essence can be seen by considering function applications.

When σ denotes a substitution, we write $\omega\sigma$ for the type expression obtained by applying σ to ω, and $\pi\sigma$ for the type assignment obtained by applying σ to each type expression assigned by π. Now suppose that e_1 and e_2 are expressions with principal typings $\pi_1 \vdash e_1{:}\ \omega_1$ and $\pi_2 \vdash e_2{:}\ \omega_2$. (For simplicity we assume that π_1 and π_2 assign to the same identifiers.) Then the set of conclusions of instances of

inference rule (I5b) whose premises are valid typings of e_1 and e_2 is

$$\{\pi \mid\!- e_1\ e_2\colon \omega' \mid (\exists \sigma_1, \sigma_2, \omega)\ \pi_1\sigma_1 = \pi_2\sigma_2 = \pi \text{ and } \omega_1\sigma_1 = \omega \to \omega' \text{ and } \omega_2\sigma_2 = \omega\}\ .$$

Let α be a type variable not occurring in π_2 or ω_2. Since we can extend σ_2 to substitute any type expression for α, the above set of conclusions is

$$\{\pi \mid\!- e_1\ e_2\colon \omega' \mid (\exists \sigma_1, \sigma_2, \omega)\ \pi_1\sigma_1 = \pi_2\sigma_2 = \pi \text{ and } \omega_1\sigma_1 = (\omega_2 \to \alpha)\sigma_2 = \omega \to \omega'\}\ .$$

Now we can use unification to determine whether there are any σ_1 and σ_2 such that $\pi_1\sigma_1 = \pi_2\sigma_2$ and $\omega_1\sigma_1 = (\omega_2 \to \alpha)\sigma_2$ and, if so, to produce "most general" substitutions $\hat{\sigma}_1$ and $\hat{\sigma}_2$ such that all such σ_1 and σ_2 can be obtained by composing $\hat{\sigma}_1$ and $\hat{\sigma}_2$ with some further substitution σ. In this case, the set of conclusions is

$$\{\pi \mid\!- e_1\ e_2\colon \omega' \mid (\exists \sigma)\ \pi = (\pi_1\hat{\sigma}_1)\sigma \text{ and } \omega' = (\alpha\hat{\sigma}_2)\sigma\}$$

This is the set of typings that can be obtained by substitution from

$$(\pi_1\hat{\sigma}_1) \mid\!- e_1\ e_2\colon (\alpha\hat{\sigma}_2)\ ,$$

which is therefore a principal typing of $e_1\ e_2$.

To illustrate the Hindley-Milner algorithm, the following is a list of principal typings of the subexpressions of the append function, as they would be generated by a naive version of the algorithm. Note that a proof of a particular typing, such as is given in Section 2b, can be obtained from the list of principal typings by substitution. (Also note that the choice of type variables in each line is arbitrary.)

ap: α |– ap: α

z: α |– z: α

ap: $\alpha \to \beta$, z: α |– ap z: β

y: α |– y: α

ap: $\alpha \to \beta \to \gamma$, z: α, y: β |– ap z y: γ

n: α |– n: α

n: γ, ap: $\alpha \to \beta \to$ list γ, z: α, y: β |– cons n (ap z y): list γ

n: γ, ap: $\alpha \to \beta \to$ list γ, y: β |– λz. cons n (ap z y): $\alpha \to$ list γ

ap: $\alpha \to \beta \to$ list γ, y: β |– λn. λz. cons n (ap z y): $\gamma \to \alpha \to$ list γ

y: α |– y: α

ap: list $\gamma \to$ list $\gamma \to$ list γ, y: list γ |–
lchoose y (λn. λz. cons n (ap z y)): list $\gamma \to$ list γ

x: α |– x: α

ap: list $\gamma \to$ list $\gamma \to$ list γ, y: list γ, x: list γ |–
lchoose y (λn. λz. cons n (ap z y)) x: list γ

ap: list $\gamma \to$ list $\gamma \to$ list γ, x: list γ |–
λy. lchoose y (λn. λz. cons n (ap z y)) x: list $\gamma \to$ list γ

ap: list $\gamma \to$ list $\gamma \to$ list γ |–
λx. λy. lchoose y (λn. λz. cons n (ap z y)) x: list $\gamma \to$ list $\gamma \to$ list γ

$\vdash \lambda ap.\ \lambda x.\ \lambda y.\ \underline{lchoose}\ y\ (\lambda n.\ \lambda z.\ \underline{cons}\ n\ (ap\ z\ y))\ x:$
$\quad (\underline{list}\ \gamma \to \underline{list}\ \gamma \to \underline{list}\ \gamma) \to \underline{list}\ \gamma \to \underline{list}\ \gamma \to \underline{list}\ \gamma$
$\vdash \underline{fix}(\lambda ap.\ \lambda x.\ \lambda y.\ \underline{lchoose}\ y\ (\lambda n.\ \lambda z.\ \underline{cons}\ n\ (ap\ z\ y))\ x):$
$\quad \underline{list}\ \gamma \to \underline{list}\ \gamma \to \underline{list}\ \gamma$

The Hindley-Milner algorithm requires a certain amount of auxilliary information in the inject operation. If $\pi \vdash e: \omega$ is a principal typing of e, then the principal typing of inject k e should be

$$\pi \vdash \underline{inject}\ k\ e: \underline{sum}(\alpha_1, \ \cdots\ , \alpha_{k-1}, \omega, \alpha_{k+1}, \ \cdots\ , \alpha_n)\ ,$$

where $\alpha_1, \ \cdots\ , \alpha_{k-1}, \alpha_{k+1}, \ \cdots\ , \alpha_n$ are distinct type variables that do not occur in π or ω. However, inject k e does not contain any information that determines the number n of alternatives. Thus we must alter syntax rule (S8a) and inference rule (I8a) to provide this information explicitly:

$$E ::= \underline{inject}_N\ K\ E \qquad \frac{\pi \vdash e: \omega_k}{\pi \vdash \underline{inject}_n\ k\ e: \underline{sum}(\omega_1, \ \cdots\ , \omega_n)} \quad \text{when } 1 \le k \le n\ . \qquad \text{(I8a')}$$

Similarly (but less pleasantly), we must require the inject operation for named alternatives to contain a list of the identifiers used as names. Rules (S9a) and (I9a) become:

$$E ::= \underline{inject}_{I,\ldots,I}\ I\ E$$

$$\frac{\pi \vdash e: \omega_k}{\pi \vdash \underline{inject}_{i_1,\ldots,i_n}\ i_k\ e: \underline{sum}(i_1{:}\omega_1, \ \cdots\ , i_n{:}\omega_n)} \quad \text{when } 1 \le k \le n\ . \qquad \text{(I9a')}$$

3b. The Polymorphic let

Suppose we use the reduce function to define the following:

$\underline{let}\ red = \underline{fix}(\lambda red.\ \lambda \ell.\ \lambda f.\ \lambda a.$
$\qquad \underline{lchoose}\ a\ (\lambda n.\ \lambda z.\ f\ n\ (red\ z\ f\ a))\ \ell)$
$\quad \underline{in}\ \lambda \ell\ell\ell.\ red\ \ell\ell\ (\lambda \ell.\ \lambda s.\ \underline{add}(\ red\ \ell\ \underline{add}\ 0)\ s)\ 0\ .$

Intuitively, if ℓ is a list of integers, then red ℓ add 0 is its sum, so that $\lambda \ell.\ \lambda s.$ add(red ℓ add 0) s is a function that accepts a list of integers and an integer and produces their sum. Thus the function defined above should sum a list of lists of integers.

But in fact this expression has no typing. From the principal typing of fix(λred. ...) and the inference rule (I11) for let, it is clear that the let expression can only have a typing if its body has a typing of the form

π, red: $\underline{list}\ \alpha \to (\alpha \to \beta \to \beta) \to \beta \to \beta \vdash$
$\qquad \lambda \ell\ell\ell.\ red\ \ell\ell\ (\lambda \ell.\ \lambda s.\ \underline{add}(red\ \ell\ \underline{add}\ 0)\ s)\ 0: \omega$

for some particular substitution of types for α, β, and ω. But the unique type

int → int → int of add forces α = β = int for red ℓ add 0 to make sense, and then λℓ. λs. add(red ℓ add 0) s must have type list int → int → int, so that the outer occurrence of red requires α = list int, contradicting α = int. The difficulty is that the above definition only makes sense if red is understood as a polymorphic function, i.e. a function that is applicable to a variety of types.

In designing the type structure of the language ML [22], Milner realized that his type deduction algorithm permitted the treatment of this kind of polymorphism. If e_2 has the principal typing $\pi \vdash e_2 : \omega$, and if $\alpha_1, \ldots, \alpha_n$ are type variables occurring in ω but not in π, then in typing let i = e_2 in e_1, different occurrences of i in e_1 can be assigned different types that are instances of ω obtain by different substitutions for $\alpha_1, \ldots, \alpha_n$. For instance, in the above example, one can assign the inner and outer occurrences of red the different types

list int → (int → int → int) → int → int

and

list list int → (list int → int → int) → int → int

to obtain the type list list int → int for the entire example.

In this scheme, however, (in contrast to Section 5b) polymorphic functions can be bound by let but not by λ, so that one cannot define higher-order functions that accept polymorphic functions. For instance, if we convert let i = e_2 in e_1 to $(\lambda i.\ e_1)\ e_2$ in the above example, we obtain

(λred. λℓℓ. red ℓℓ (λℓ. λs. add(red ℓ add 0) s) 0)
(fix(λred. ...)) ,

in which the first line has no typing.

The term "polymorphism" was coined by Christopher Strachey [24], who distinguished between "parametric" polymorphic functions, such as the reduce function, that treat all types in the same way, and "ad hoc" polymorphic functions that can behave differently for different types. In this paper, we shall reserve the word "polymorphism" for the parametric case, and call the ad hoc functions "generic". (Strachey's definition of "parametric" was intuitive; its precise semantic formulation is still controversial [20].)

3c. Infinite Types

Without explicitly mentioning it, we have assumed that types are finite phrases. However, nothing that we have done precludes infinite types. For example, the infinite type

prod(int, prod(int, prod(int, ...)))

is the type of infinite streams of integers, and list ω can be regarded as

sum(prod(), prod(ω, sum(prod(), prod(ω, sum(prod(), prod(ω, ...)))))).

(Note that prod() denotes a set with one element: the empty record <>.)

To make use of such infinite types, however, we must have finite type expressions to denote them, i.e. we must introduce the recursive definition of types. A simple approach is to introduce type expressions with the syntax

Ω ::= rectype T: Ω

where T denotes the set of type variables, and the intuitive semantics that rectype α: ω denotes an infinite type obtained by the endless sequence of substitutions:

$$\alpha \mid \alpha \to \omega \mid \alpha \to \omega \mid \alpha \to \omega \cdots .$$

For example, rectype α: prod(int, α) denotes the type of infinite streams, while rectype α: sum(prod(), prod(ω, α)) denotes the type list ω.

It should be emphasized that the rectype construction is a type expression rather than a type. For example,

rectype α: prod(int, α)
rectype β: prod(int, β)
rectype α: prod(int, prod(int, α))

all denote the same infinite type.

If list ω is regarded as an abbreviation for rectype α: sum(prod(), prod(ω, α)), then the various forms of expressions we have introduced for list manipulation can also be regarded as abbreviations:

nil $\equiv$ $\underline{\text{inject}}_2$ 1 <>
cons e_1 e_2 $\equiv$ $\underline{\text{inject}}_2$ 2 $\langle e_1, e_2 \rangle$
lchoose e_1 e_2 $\equiv$ choose(λx. e_1, λx. e_2 x.1 x.2) ,

where x is an identifier not occurring in e_1 or e_2.

An obvious question is whether the Hindley-Milner algorithm can be extended to encompass infinite, recursively defined types. But the answer is obscure. It is known that the unification algorithm can be extended to expressions involving rectype (by treating such expressions as cyclic structures and omitting the "occurs" check) [25,26], and I have heard researchers say that this extension can be applied to the Hindley-Milner algorithm without difficulty. But I have been unable to find anything in the literature about the question.

4. SUBTYPES AND GENERIC OPERATORS

In many programming languages there is a subtype relation between types such that, if ω is a subtype of ω', then there is an implicit conversion from values of type ω to values of type ω', so that any expression of type ω can be used in any context allowing expressions of type ω'. Experience with languages such as PL/I and Algol 68 has shown that a rich subtype structure, particularly in conjunction with generic operators, can produce a language with quirkish and counterintuitive behavior. This experience has led to research on subtypes and generic operators, using category theory as a tool, that has established design criteria for avoiding such behavior [27,28].

To see the problem, suppose int is a subtype of real, and add is a generic operator mapping pairs of integers into integers and pairs of reals into reals. Then an expression such as add 5 6, occurring in a context calling for a real expression, can be interpreted as either the integer-to-real conversion of the integer sum of 5 and 6, or as the real sum of the integer-to-real conversions of 5 and 6.

In this case, the laws of mathematics insure that the two interpretations are equivalent(except for the roundoff behavior of machines with unfortunate arithmetic). On the other hand, suppose digit string is a subtype of int with an implicit conversion that interprets digit strings as decimal representations, and equals is a generic operator applicable to either pairs of digit strings or pairs of integers. Then an expression such as equals "1" "01" is ambiguous, since the implicit conversion maps unequal digit strings into equal integers.

4a. Subtypes

We will write $\omega \leq \omega'$ to indicate that ω is a subtype of ω'. Then the idea that any expression of type ω can be used as an expression of type ω' is formalized by the inference rule

$$\frac{\pi \vdash e: \omega}{\pi \vdash e: \omega'} \qquad \text{when } \omega \leq \omega' \ . \tag{I15}$$

It is natural to assume that $\leq$ is a preorder, i.e. that it satisfies the laws

(a) (Reflexivity) $\omega \leq \omega$.

(b) (Transitivity) if $\omega \leq \omega'$ and $\omega' \leq \omega''$ then $\omega \leq \omega''$.

(This assumption can be justified by noting that the effect of (I15) remains unchanged if an arbitrary relation $\leq$ is replaced by its reflexive and transitive closure.)

Semantically, the laws of reflexivity and transitivity imply the existence of implicit conversions that must "fit together" correctly if the meaning of our language is to be unambiguous. For each ω, $\omega \leq \omega$ implies that there is an implicit conversion

that can be applied to any value of type ω without changing its type; to avoid ambiguity this conversion must be an identity function. When $\omega \leq \omega'$ and $\omega' \leq \omega''$, one can convert from ω to ω'' either directly or through the intermediary of ω'; to avoid ambiguity the direct conversion from ω to ω'' must equal the functional composition of the conversions from ω to ω' and from ω' to ω''.

Mathematically, these restrictions on implicit conversions are tantamount to requiring that there be a functor from the preorder of types to the category of sets and functions (or some other appropriate category) that maps each type into the set that it denotes, and maps each pair ω, ω' such that $\omega \leq \omega'$ into the implicit conversion function from ω to ω'. This is the fundamental connection between subtypes and category theory.

If $\omega \leq \omega'$ and $\omega' \leq \omega$ then, under a given type assignment, an expression will have type ω if and only if it has type ω'. It is tempting to assume that such types are identical, i.e. to impose

(c) (Antisymmetry) if $\omega \leq \omega'$ and $\omega' \leq \omega$ then $\omega = \omega'$,

so that $\leq$ is a partial order. In fact, we will assume that $\leq$ is a partial order throughout this exposition, but it should be noted that most of the theory goes through in the more general case of preorders, and that one can conceive of applications that would use such generality (e.g., where ω and ω' denote abstractly equivalent types with different representations).

For each of the type constructors we have introduced, the existence of implicit conversions for the component types induces a natural conversion for the constructed type. For instance, suppose c is an implicit conversion from ω_2 to ω_1 and c' is an implicit conversion from ω_1' to ω_2'. Then it is natural to convert a function f of type $\omega_1 \to \omega_1'$ to the function λx_{ω_2}. c'(f(c x)) of type $\omega_2 \to \omega_2'$. If this conversion of functions is taken as an implicit conversion from $\omega_1 \to \omega_1'$ to $\omega_2 \to \omega_2'$, then the operator $\to$ will satisfy

$$\text{If } \omega_2 \leq \omega_1 \text{ and } \omega_1' \leq \omega_2' \text{ then } \omega_1 \to \omega_1' \leq \omega_2 \to \omega_2' \ , \qquad (\leq a)$$

i.e. $\to$ will be monotone in its second operand but antimonotone in its first operand.

If $c_1, \ldots, c_n$ are implicit conversions from ω_1 to $\omega_1', \ldots, \omega_n$ to ω_n', then it is natural to convert a record $\langle x_1, \ldots, x_n \rangle$ of type $\underline{\text{prod}}(\omega_1, \ldots, \omega_n)$ to the record $\langle c_1\ x_1, \ldots, c_n\ x_n \rangle$ of type $\underline{\text{prod}}(\omega_1', \ldots, \omega_n')$. Thus $\underline{\text{prod}}$ should be monotone in all of its operands:

$$\text{If } \omega_1 \leq \omega_1', \ldots, \omega_n \leq \omega_n' \text{ then } \underline{\text{prod}}(\omega_1, \ldots, \omega_n) \leq \underline{\text{prod}}(\omega_1', \ldots, \omega_n') \ . \qquad (\leq b)$$

Similarly, for products with named fields, we have

$$\text{If } \omega_1 \leq \omega_1', \ldots, \omega_n \leq \omega_n' \text{ then}$$
$$\underline{\text{prod}}(i_1{:}\omega_1, \ldots, i_n{:}\omega_n) \leq \underline{\text{prod}}(i_1{:}\omega_1', \ldots, i_n{:}\omega_n') \ . \qquad (\leq c)$$

In this case, however, there is another kind of conversion that is both well-behaved and useful: if $\langle i_1:x_1, \ldots, i_n:x_n \rangle$ is a record of type $\underline{\text{prod}}(i_1:\omega_1, \ldots, i_n:\omega_n)$, and if $\{i_1, \ldots, i_m\}$ is any subset of $\{i_1, \ldots, i_n\}$, then by forgetting fields it is natural to convert this record to $\langle i_1:x_1, \ldots, i_m:x_m \rangle$. This leads to

$$\underline{\text{prod}}(i_1:\omega_1, \ldots, i_n:\omega_n) \leq \underline{\text{prod}}(i_1:\omega_1, \ldots, i_m:\omega_m) \qquad \text{when } 0 \leq m \leq n . \qquad (\leq d)$$

(Theoretically, field-forgetting conversions are also applicable to records with numbered fields, but the constraint that the field numbers must form a consecutive sequence renders these conversions much more limited.)

If $c_1, \ldots, c_n$ are implicit conversions from ω_1 to ω_1', $\ldots$, ω_n to ω_n', then it is natural to convert $\underline{\text{inject}}$ k x of type $\underline{\text{sum}}(\omega_1, \ldots, \omega_n)$ to $\underline{\text{inject}}$ k $(c_k\ x)$ of type $\underline{\text{sum}}(\omega_1', \ldots, \omega_n')$. Thus $\underline{\text{sum}}$ is also monotone:

$$\text{If } \omega_1 \leq \omega_1', \ldots, \omega_n \leq \omega_n' \text{ then } \underline{\text{sum}}(\omega_1, \ldots, \omega_n) \leq \underline{\text{sum}}(\omega_1', \ldots, \omega_n') . \qquad (\leq e)$$

Similarly, for named alternatives we have

$$\text{If } \omega_1 \leq \omega_1', \ldots, \omega_n \leq \omega_n' \text{ then } \underline{\text{sum}}(i_1:\omega_1, \ldots, i_n:\omega_n) \leq \underline{\text{sum}}(i_1:\omega_1', \ldots, i_n:\omega_n') . \qquad (\leq f)$$

Just as with products, however, names give a richer subtype structure than numbers. Whenever $\{i_1, \ldots, i_m\}$ is a subset of $\{i_1, \ldots, i_n\}$, $\underline{\text{sum}}(i_1:\omega_1, \ldots, i_m:\omega_m)$ is a subset of $\underline{\text{sum}}(i_1:\omega_1, \ldots, i_n:\omega_n)$, and the identity injection is a natural implicit conversion. Thus

$$\underline{\text{sum}}(i_1:\omega_1, \ldots, i_m:\omega_m) \leq \underline{\text{sum}}(i_1:\omega_1, \ldots, i_n:\omega_n) \qquad \text{when } 0 \leq m \leq n . \qquad (\leq g)$$

The implicit conversions of forgetting named fields in products and adding named alternatives in sums have been investigated by L. Cardelli [29], who shows that they generalize the subclass concept of SIMULA 67 and also provide a suitable type structure for object-oriented languages such as SMALLTALK.

Finally, if c is an implicit conversion from ω to ω', then it is natural to convert $(x_1, \ldots, x_n)$ of type $\underline{\text{list}}\ \omega$ to $(c\ x_1, \ldots, c\ x_n)$ of type $\underline{\text{list}}\ \omega'$, so that $\underline{\text{list}}$ is also monotone:

$$\text{If } \omega \leq \omega' \text{ then } \underline{\text{list}}\ \omega \leq \underline{\text{list}}\ \omega' . \qquad (\leq h)$$

4b. Explicit Minimal Typing

With the introduction of subtypes, it is no longer possible to achieve explicit typing in the sense of Section 2c, since an expression that has type ω under some type assignment will also have type ω' whenever $\omega \leq \omega'$. However, we can still hope to arrange things so that, if an expression (under a given type assignment) has any type, then its set of types will have a least member (which must be unique since $\leq$ is a partial order).

We write

$$\pi \mid\!\!-_m e: \omega \ ,$$

called a minimal typing, to indicate that ω is the least type of e under π. In other words $\pi \mid\!\!-_m e: \omega$ means that $\pi \mid\!\!- e: \omega'$ holds for just those ω' such that $\omega \leq \omega'$.

If the expressions of our base language are to have minimal typings then the partial ordering of types must satify two dual properties:

(LUB) If ω_1 and ω_2 have an upper bound then ω_1 and ω_2 have a least upper bound.

(GLB) If ω_1 and ω_2 have a lower bound then ω_1 and ω_2 have a greatest lower bound.

Fortunately these properties are consistent with the ordering laws given in the previous subsection. It can be shown that the least partial ordering satisfying ($\leq$a) to ($\leq$h) and including some partial ordering of primitive types will satisfy (LUB) and (GLB) if the partial ordering of primitive types satisfies (LUB) and (GLB). (Moreover, this situation will remain true as additional ordering laws are introduced in Sections 4d and 4e.)

To see why these properties are needed for minimal typing, consider the conditional expression. Suppose $\pi \mid\!\!-_m e:$ bool, $\pi \mid\!\!-_m e_1: \omega_1$, and $\pi \mid\!\!-_m e_2: \omega_2$. Then the types ω such that $\pi \mid\!\!-$ if e then e_1 else $e_2: \omega$ will be the upper bounds of ω_1 and ω_2, so that $\pi \mid\!\!-_m$ if e then e_1 else $e_2: \omega$ will only hold if ω is a least upper bound of ω_1 and ω_2.

(By a similar argument, the minimal typing of case e of $(e_1, \ldots, e_n)$ requires that, if the finite set $\{\omega_1, \ldots, \omega_n\}$ has an upper bound, then it must have a least upper bound. Fortunately, for $n > 0$ this property is implied by (LUB).)

Assuming that (LUB) and (GLB) hold, we can give inference rules for the explicit minimal typing of our base language that effectively define a partial function mapping each e and π into the least type of e under π. As with the explicit typing in the absence of subtypes described in Section 2c, we must require type information to appear in lambda expressions, inject expressions for numbered alternatives, and the expression nil, and we must exclude the vacuous expressions choose() and case e of (). But now, type information is no longer needed in inject expressions for named alternatives, because of the implicit conversions that add alternatives. If ω is the least type of e then sum(i:ω) is the least type of inject i e.

(It is curious that numbered alternatives require less type information under the Hindley-Milner approach of Section 3a, while named alternatives require less type information under the present approach.)

The following is a list of the inference rules for minimal typing. In the provisos of some of the rules, we write lub for least upper bound and glb for greatest lower bound.

$$\frac{}{\pi \mid\!\!-_m i: \omega}$$ when i is in the domain of π, and π assigns ω to i. (M1)

$$\frac{}{\pi \mid\!\!-_m z: \underline{\text{int}}} \qquad \frac{}{\pi \mid\!\!-_m \underline{\text{true}}: \underline{\text{bool}}} \qquad \frac{}{\pi \mid\!\!-_m \underline{\text{false}}: \underline{\text{bool}}}$$ (M2,M3)

$$\frac{}{\pi \mid\!\!-_m \underline{\text{add}}: \underline{\text{int}} \to \underline{\text{int}} \to \underline{\text{int}}} \qquad \frac{}{\pi \mid\!\!-_m \underline{\text{equals}}: \underline{\text{int}} \to \underline{\text{int}} \to \underline{\text{bool}}}$$ (M4)

$$\frac{\pi, i: \omega \mid\!\!-_m e: \omega'}{\pi \mid\!\!-_m \lambda i_\omega .\ e: \omega \to \omega'} \qquad \frac{\pi \mid\!\!-_m e_1: \omega_1 \to \omega' \quad \pi \mid\!\!-_m e_2: \omega_2}{\pi \mid\!\!-_m e_1\ e_2: \omega'} \text{ when } \omega_2 \leq \omega_1$$ (M5)

$$\frac{\begin{array}{l}\pi \mid\!\!-_m e_1: \omega_1 \\ \vdots \\ \pi \mid\!\!-_m e_n: \omega_n\end{array}}{\pi \mid\!\!-_m \langle e_1, \ldots, e_n\rangle: \underline{\text{prod}}(\omega_1, \ldots, \omega_n)}$$ (M6a)

$$\frac{\pi \mid\!\!-_m e: \underline{\text{prod}}(\omega_1, \ldots, \omega_n)}{\pi \mid\!\!-_m e.k: \omega_k} \text{ when } 1 \leq k \leq n$$ (M6b)

$$\frac{\begin{array}{l}\pi \mid\!\!-_m e_1: \omega_1 \\ \vdots \\ \pi \mid\!\!-_m e_n: \omega_n\end{array}}{\pi \mid\!\!-_m \langle i_1{:}e_1, \ldots, i_n{:}e_n\rangle: \underline{\text{prod}}(i_1{:}\omega_1, \ldots, i_n{:}\omega_n)}$$ (M7a)

$$\frac{\pi \mid\!\!-_m e: \underline{\text{prod}}(i_1{:}\omega_1, \ldots, i_n{:}\omega_n)}{\pi \mid\!\!-_m e.i_k: \omega_k} \text{ when } 1 \leq k \leq n$$ (M7b)

$$\frac{\pi \mid\!\!-_m e: \omega_k}{\pi \mid\!\!-_m \underline{\text{inject}}_{\omega_1,\ldots,\omega_n}\ k\ e: \underline{\text{sum}}(\omega_1, \ldots, \omega_n)} \text{ when } 1 \leq k \leq n$$ (M8a)

$$\frac{\begin{array}{l}\pi \mid\!\!-_m e_1: \omega_1 \to \omega_1' \\ \vdots \\ \pi \mid\!\!-_m e_n: \omega_n \to \omega_n'\end{array}}{\pi \mid\!\!-_m \underline{\text{choose}}(e_1, \ldots, e_n): \underline{\text{sum}}(\omega_1, \ldots, \omega_n) \to \omega'} \text{ when } \omega' \text{ is the lub of } \{\omega_1', \ldots, \omega_n'\}$$ (M8b)

$$\frac{\pi \mid\!\!-_m e: \omega}{\pi \mid\!\!-_m \underline{\text{inject}}\ i\ e: \underline{\text{sum}}(i{:}\omega)}$$ (M9a)

$$\frac{\pi \mid\!\!-_m e_1 : \omega_1 \to \omega_1' \quad \vdots \quad \pi \mid\!\!-_m e_n : \omega_n \to \omega_n'}{\pi \mid\!\!-_m \underline{\text{choose}}(i_1{:}e_1, \ldots, i_n{:}e_n) : \underline{\text{sum}}(i_1{:}\omega_1, \ldots, i_n{:}\omega_n) \to \omega'} \quad \text{when } \omega' \text{ is the lub of } \{\omega_1', \ldots, \omega_n'\} \qquad \text{(M9b)}$$

$$\frac{}{\pi \mid\!\!-_m \underline{\text{nil}}_\omega : \underline{\text{list}}\ \omega} \qquad \frac{\pi \mid\!\!-_m e_1 : \omega_1 \quad \pi \mid\!\!-_m e_2 : \underline{\text{list}}\ \omega_2}{\pi \mid\!\!-_m \underline{\text{cons}}\ e_1\ e_2 : \underline{\text{list}}\ \omega} \quad \text{when } \omega \text{ is the lub of } \omega_1 \text{ and } \omega_2 \qquad \text{(M10a,b)}$$

$$\frac{\pi \mid\!\!-_m e_1 : \omega_1' \quad \pi \mid\!\!-_m e_2 : \omega_1 \to \underline{\text{list}}\ \omega_2 \to \omega_2'}{\pi \mid\!\!-_m \underline{\text{lchoose}}\ e_1\ e_2 : \underline{\text{list}}\ \omega \to \omega'} \quad \text{when } \omega' \text{ is the lub of } \omega_1' \text{ and } \omega_2' \text{ and } \omega \text{ is the glb of } \omega_1 \text{ and } \omega_2 \qquad \text{(M10c)}$$

$$\frac{\pi \mid\!\!-_m e_2 : \omega \quad \pi, i{:}\,\omega \mid\!\!-_m e_1 : \omega'}{\pi \mid\!\!-_m \underline{\text{let}}\ i = e_2\ \underline{\text{in}}\ e_1 : \omega'} \qquad \text{(M11)}$$

$$\frac{\pi \mid\!\!-_m e : \omega_0 \quad \pi \mid\!\!-_m e_1 : \omega_1 \quad \pi \mid\!\!-_m e_2 : \omega_2}{\pi \mid\!\!-_m \underline{\text{if}}\ e\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2 : \omega} \quad \text{when } \omega_0 \leq \underline{\text{bool}} \text{ and } \omega \text{ is the lub of } \omega_1 \text{ and } \omega_2 \qquad \text{(M12)}$$

$$\frac{\pi \mid\!\!-_m e : \omega_0 \quad \pi \mid\!\!-_m e_1 : \omega_1 \quad \vdots \quad \pi \mid\!\!-_m e_n : \omega_n}{\pi \mid\!\!-_m \underline{\text{case}}\ e\ \underline{\text{of}}\ (e_1, \ldots, e_n) : \omega} \quad \text{when } \omega_0 \leq \underline{\text{int}} \text{ and } \omega \text{ is the lub of } \{\omega_1, \ldots, \omega_n\} \qquad \text{(M13)}$$

$$\frac{\pi \mid\!\!-_m e : \omega_1 \to \omega_2}{\pi \mid\!\!-_m \underline{\text{fix}}\ e : \omega_2} \quad \text{when } \omega_2 \leq \omega_1 \qquad \text{(M14)}$$

4c. Generic Operators

We now consider extending our base language to include primitive operators that act upon values of several types, performing possibly different operations for different types. For example, if the partial ordering of primitive types is

we might want add to denote integer addition, real addition, and boolean disjunction, and equals to denote equality tests for integers, reals, and truth values. In this case we would replace inference rules (I4) by

$$\frac{\pi \vdash e_1 : \underline{\text{int}} \qquad \pi \vdash e_2 : \underline{\text{int}}}{\pi \vdash \underline{\text{add}}\ e_1\ e_2 : \underline{\text{int}}} \qquad \frac{\pi \vdash e_1 : \underline{\text{real}} \qquad \pi \vdash e_2 : \underline{\text{real}}}{\pi \vdash \underline{\text{add}}\ e_1\ e_2 : \underline{\text{real}}} \qquad \frac{\pi \vdash e_1 : \underline{\text{bool}} \qquad \pi \vdash e_2 : \underline{\text{bool}}}{\pi \vdash \underline{\text{add}}\ e_1\ e_2 : \underline{\text{bool}}}$$

$$\frac{\pi \vdash e_1 : \underline{\text{int}} \qquad \pi \vdash e_2 : \underline{\text{int}}}{\pi \vdash \underline{\text{equals}}\ e_1\ e_2 : \underline{\text{bool}}} \qquad \frac{\pi \vdash e_1 : \underline{\text{real}} \qquad \pi \vdash e_2 : \underline{\text{real}}}{\pi \vdash \underline{\text{equals}}\ e_1\ e_2 : \underline{\text{bool}}} \qquad \frac{\pi \vdash e_1 : \underline{\text{bool}} \qquad \pi \vdash e_2 : \underline{\text{bool}}}{\pi \vdash \underline{\text{equals}}\ e_1\ e_2 : \underline{\text{bool}}}$$

The general situation, for an n-ary operator op, can be described as follows: there will be an index set Γ and functions $\alpha_1, \ldots, \alpha_n, \rho$ from Γ to Ω such that the inference rules for op will be the instances of

$$\frac{\pi \vdash e_1 : \alpha_1\gamma \quad \vdots \quad \pi \vdash e_n : \alpha_n\gamma}{\pi \vdash \underline{\text{op}}\ e_1 \cdots e_n : \rho\gamma} \qquad \text{(I4')}$$

for each index γ in Γ.

For example, our generic add operator would be described by $\Gamma = \{\underline{\text{int}}, \underline{\text{real}}, \underline{\text{bool}}\}$, with α_1, α_2, and ρ all being the identity injection from Γ to Ω. The equals operator would be described similarly, except that ρ would be a constant function giving bool.

An obvious question is under what conditions this kind of inference rule scheme provides minimal typing. A sufficient condition is that Γ must possess a partial ordering such that

The function $\alpha_1, \ldots, \alpha_n$, and ρ are all monotone and, for all $\omega_1, \ldots, \omega_n$, if the set

$$\{\gamma \mid \omega_1 \leq \alpha_1\gamma, \ldots, \omega_n \leq \alpha_n\gamma\} \qquad (*)$$

is nonempty, then this set possesses a least member.

If this condition is satisfied (as is the case for add and equals) then minimal typing is given by the rule

$$\frac{\pi \vdash_m e_1 : \omega_1 \quad \vdots \quad \pi \vdash_m e_n : \omega_n}{\pi \vdash_m \underline{\text{op}}\ e_1 \cdots e_n : \rho\gamma_0} \qquad \text{when } \gamma_0 \text{ is the least member of } \{\gamma \mid \omega_1 \leq \alpha_1\gamma, \ldots, \omega_n \leq \alpha_n\gamma\} \qquad \text{(M4')}$$

Semantically, however, there is a further issue. If the meaning of our language is to be unambiguous, then the meanings of op for various indices must satisfy the following relationship with the implicit conversion functions:

For each index γ, let $\overline{op}_\gamma$ be the function of type $\alpha_1\gamma \to \ldots \to \alpha_n\gamma \to \rho\gamma$ that is the meaning of op for γ. For types ω and ω' such that $\omega \leq \omega'$, let $c_{\omega\leq\omega'}$ be the conversion function from ω to ω'. Then for all indices γ and γ' such that $\gamma \leq \gamma'$, and all values $x_1, \ldots, x_n$ of types $\alpha_1\gamma, \ldots, \alpha_n\gamma$, $\overline{op}_\gamma$ and $\overline{op}_{\gamma'}$, must satisfy

$$c_{\rho\gamma\leq\rho\gamma'}(\overline{op}_\gamma\ x_1\ \ldots\ x_n) = \overline{op}_{\gamma'}(c_{\alpha_1\gamma\leq\alpha_1\gamma'}\ x_1)\ \ldots\ (c_{\alpha_n\gamma\leq\alpha_n\gamma'}\ x_n).$$

In conjunction with (*), this relationship (which can be expressed category-theoretically by saying $\overline{op}$ is a certain kind of natural transformation) is sufficient to preclude ambiguous meanings.

For our examples of add and equals, the relationship becomes

$$c_{int\leq real}(\overline{add}_{int}\ x_1\ x_2) = \overline{add}_{real}(c_{int\leq real}\ x_1)(c_{int\leq real}\ x_2)$$

and

$$\overline{equals}_{int}\ x_1\ x_2 = \overline{equals}_{real}(c_{int\leq real}\ x_1)(c_{int\leq real}\ x_2)\ .$$

The condition for add is the classical homomorphic relation between integer and real addition, while the condition for equals (assuming the meanings of equals are equality predicates) is equivalent to requiring $c_{int\leq real}$ to map distinct integers into distinct reals. Note that there is no constraint on $\overline{add}_{bool}$ or $\overline{equals}_{bool}$.

4d. The Universal Type

Suppose we expand the partially ordered set of types by introducing a new type that is a subtype of every type:

For all ω, univ $\leq \omega$. (≤i)

Several pleasantries occur. We can give minimal typings for the vacuous case and choose operations:

$$\frac{\pi \vdash_m e:\omega_0}{\pi \vdash_m \underline{case}\ e\ \underline{of}\ (): univ} \quad \text{when } \omega_0 \leq int \qquad \text{(M13')}$$

$$\frac{}{\pi \vdash_m \underline{choose}(): \underline{sum}() \to univ} \qquad \text{(M8b')}$$

for an inject operation for numbered alternatives with less explicit type information:

$$\frac{\pi \vdash_m e: \omega_k}{\pi \vdash_m \underline{inject}_n\ k\ e: \underline{sum}(univ, \ldots, univ, \omega_k, univ, \ldots, univ)} \quad \text{when } 1 \leq k \leq n \qquad \text{(M8a')}$$

and for a nil expression without type information:

$$\frac{}{\pi \vdash_m \underline{nil}: \underline{list}\ univ} \qquad \text{(M10a')}$$

Less happily, however, the introduction of univ complicates the nature of generic operators by making (*) in the previous subsection harder to satisfy. For instance, taking $\omega_1 = \ldots = \omega_n =$ univ, (*) implies that Γ, if nonempty, must have a least element.

For our example of add and equals, the introduction of univ forces us to add a least index (which we will also call univ) to Γ, with α_1 univ = α_2 univ = ρ univ = univ (except that we could take ρ univ = bool for equals). In effect, we must introduce vacuous versions of add and equals corresponding to the inference rules

$$\frac{\pi \vdash e_1\text{: univ} \qquad \pi \vdash e_2\text{: univ}}{\pi \vdash \text{add } e_1\ e_2\text{: univ}} \qquad \frac{\pi \vdash e_1\text{: univ} \qquad \pi \vdash e_2\text{: univ}}{\pi \vdash \text{equals } e_1\ e_2\text{: univ (or bool)}}$$

Semantically, one can interpret univ as denoting a domain with a single element $\perp_{univ}$, with an implicit conversion function to any other domain that maps $\perp_{univ}$ into the least element of that domain. A more intuitive understanding can be obtained from the following expressions of type univ:

```
case 1 of ()
fix(λx_univ. x)
```

If evaluated, the first expression gives an error stop and the second never terminates. These are both meanings that make sense for any type.

4e. The Nonsense Type

We have developed a system in which, for a given type assignment, every expression with a type has a least type, but there are still nonsensical expressions with no type at all. Thus the function that maps e and π into the least type of e under π is only partial.

To avoid the mathematical complications of partial functions, we can introduce a new type of which every type is a subtype,

$$\text{For all } \omega,\ \omega \leq \text{ns} \qquad (\leq j)$$

and make ns a type of every expression by adding the inference rule

$$\frac{}{\pi \vdash e\text{: ns}} \qquad (I16)$$

Now (under a given type assignment), since even nonsensical expressions have the type ns, every expression has at least one type, and therefore a least type. The inference rules for minimal typing remain correct if one adds a "metarule" that $\pi \vdash_m e\text{: ns}$ holds whenever $\pi \vdash_m e\text{: } \omega$ cannot be inferred for any other ω.

This idea was introduced by the author in [27]. Today, I remain bemused by its elegance, but much less sanguine about its practical utility. The difficulty is that it permits nonsensical expressions to occur within sensible ones. For example,

$(\lambda x_{ns}.\ 3)(\underline{add}\ 1\ \underline{nil})$
$<3, \underline{add}\ 1\ \underline{nil}>.1$

are both integer expressions containing the nonsensical subexpression $\underline{add}\ 1\ \underline{nil}$. Abstractly, perhaps, they have the value 3, but pragmatically they should cause compile-time errors.

4f. Open Questions

As far as I know, no one has dealt successfully with the following questions: Can the above treatment of generic operators be extended to the definition of such operators by some kind of *let* construction? Can the Hindley-Milner algorithm be extended to deal with subtypes? Can the theory of subtypes be extended to encompass infinite, recursively defined types?

5. TYPE DEFINITIONS AND EXPLICIT POLYMORPHISM

5a. Type Definitions

In this section we will extend the explicitly typed language of Section 2c to permit the definition of types. We begin by adding $\Omega ::= T$ to the definition of Ω, so that Ω becomes the set of type expressions built from the type variables in T. Then we introduce the new expressions

$$E ::= \underline{lettype}\ T = \Omega\ \underline{in}\ E \mid \underline{lettran}\ T = \Omega\ \underline{in}\ E \qquad \text{(S17,S18)}$$

which satisfy similar reduction rules

$$\underline{lettype}\ \alpha = \omega\ \underline{in}\ e = e\big|_{\alpha \to \omega} \qquad \underline{lettran}\ \alpha = \omega\ \underline{in}\ e = e\big|_{\alpha \to \omega}, \qquad \text{(R17,R18)}$$

where the right sides denote the result of substituting ω for α in the type expressions embedded within the explicitly typed expression e. (Both *lettype* and *lettran* bind the occurrences of α in the type expressions embedded in e.)

The difference between *lettype* and *lettran* lies in their inference rules:

$$\frac{\pi \vdash e: \omega'}{\pi \vdash \underline{lettype}\ \alpha = \omega\ \underline{in}\ e: (\omega'\big|_{\alpha \to \omega})} \quad \text{when } \alpha \text{ does not occur (free) in any type expression assigned by } \pi \qquad \text{(E17)}$$

$$\frac{\pi \vdash (e\big|_{\alpha \to \omega}): \omega'}{\pi \vdash \underline{lettran}\ \alpha = \omega\ \underline{in}\ e: \omega'} \qquad \text{(E18)}$$

The second rule shows that $\underline{lettran}\ \alpha = \omega\ \underline{in}$ e is a *transparent* type definition that simply permits α to be used as a synonym for ω within e. It makes sense whenever $e\big|_{\alpha \to \omega}$ makes sense (and has the same meaning).

On the other hand, $\underline{lettype}\ \alpha = \omega\ \underline{in}$ e is an *opaque* or *abstract* type definition, which only makes sense if e makes sense when α is regarded as an arbitrary type, independent of ω.

The proviso that α must not occur free in any type expression assigned by π is necessary since α has independent meanings inside and outside the scope of lettype. For example, without this proviso we could infer

$$f\colon \alpha \to \alpha \vdash \lambda x_{\alpha}.\ f\ x\colon \alpha \to \alpha$$
$$f\colon \alpha \to \alpha \vdash \underline{\text{lettype}}\ \alpha = \underline{\text{int}}\ \underline{\text{in}}\ \lambda x_{\alpha}.\ f\ x\colon \underline{\text{int}} \to \underline{\text{int}}$$
$$\vdash \lambda f_{\alpha \to \alpha}.\ \underline{\text{lettype}}\ \alpha = \underline{\text{int}}\ \underline{\text{in}}\ \lambda x_{\alpha}.\ f\ x\colon (\alpha \to \alpha) \to \underline{\text{int}} \to \underline{\text{int}}$$
$$\vdash \underline{\text{lettype}}\ \alpha = \underline{\text{bool}}\ \underline{\text{in}}\ \lambda f_{\alpha \to \alpha}.\ \underline{\text{lettype}}\ \alpha = \underline{\text{int}}\ \underline{\text{in}}\ \lambda x_{\alpha}.\ f\ x\colon (\underline{\text{bool}} \to \underline{\text{bool}}) \to \underline{\text{int}} \to \underline{\text{int}}$$

But two applications of (R17) reduce the expression in the last line to

$$\lambda f_{\text{bool} \to \text{bool}}.\ \lambda x_{\text{int}}\ f\ x\ ,$$

which has no type.

Our main interest is in abstract definitions, in which the "abstract type" α is defined by a "representation" ω. However, for such a definition to be useful, one must be able to include definitions of primitive functions (or constants) on the abstract type in terms of its representation. This seems to require a more complex definitional expression such as

$$E ::= \underline{\text{lettype}}\ T = \Omega\ \underline{\text{with}}\ I_{\Omega} = E,\ \ldots\ ,\ I_{\Omega} = E\ \underline{\text{in}}\ E \qquad \text{(S19)}$$

where each triplet $I_{\Omega} = E$ specifies an identifier denoting a primitive function, its type in terms of the abstract type, and its definition in terms of the representation. For example,

$$\underline{\text{lettype}}\ \text{complex} = \underline{\text{prod}}(\underline{\text{real}},\ \underline{\text{real}})$$
$$\underline{\text{with}}\ i_{\text{complex}} = \langle 0,\ 1\rangle,$$
$$\text{addc}_{\text{complex} \to \text{complex} \to \text{complex}} =$$
$$\lambda x_{\text{prod(real, real)}}.\ \lambda y_{\text{prod(real, real)}}.\ \langle \underline{\text{add}}\ x.1\ y.1,\ \underline{\text{add}}\ x.2\ y.2\rangle$$
$$\underline{\text{in}}\ \ldots$$

However, because our language provides higher-order functions, this more complex definitional form can be defined as an abbreviation in terms of the original lettype construct:

$$\underline{\text{lettype}}\ \alpha = \omega\ \underline{\text{with}}\ i_{1\omega_1} = e_1,\ \ldots\ ,\ i_{n\omega_n} = e_n\ \underline{\text{in}}\ e$$
$$\equiv (\underline{\text{lettype}}\ \alpha = \omega\ \underline{\text{in}}\ \lambda i_{1\omega_1}.\ \ldots\ \lambda i_{n\omega_n}.\ e)\ e_1\ \ldots\ e_n\ .$$

This definition reduces to

$$e\big|_{\alpha \to \omega}\big|_{i_1 \to e_1,\ \ldots\ ,\ i_n \to e_n}$$

(where because of the effect of bound-identifier renaming, the substitution of the e_k's for i_k's is simultaneous), which embodies the right semantics. More critically, one can derive the inference rule

$$\begin{array}{ll} \pi,\ i_1{:}\omega_1,\ \ldots\ ,\ i_n{:}\omega_n \ \vdash e{:}\ \omega_0 & \text{when } \alpha \text{ does not occur (free)} \\ \pi \ \vdash e_1{:}\ \omega_1\big|_{\alpha \to \omega} & \text{in any type expression} \\ \quad\vdots & \text{assigned by } \pi \\ \pi \ \vdash e_n{:}\ \omega_n\big|_{\alpha \to \omega} & \\ \pi \ \vdash \underline{\text{lettype}}\ \alpha = \omega\ \underline{\text{with}}\ i_{1\omega_1} = e_1,\ \ldots\ ,\ i_{n\omega_n} = e_n\ \underline{\text{in}}\ e{:}\ \omega_0\big|_{\alpha \to \omega} & \end{array} \qquad \text{(E19)}$$

which captures the notion of abstraction since ω does not occur in the typing of e. (Some authors [30] have suggested that the principle of abstraction necessitates a restriction on this rule that α should not occur (free) in ω_0, so that $\omega_0\big|_{\alpha \to \omega} = \omega_0$ and the type of the lettype ... with ... expression is the same as the type of its body. This restriction is also justified by the alternative definition of the lettype ... with ... expression that will be given in Section 5e.

For full generality, we should further extend lettype to permit the simultaneous definition of several abstract types with primitive functions, e.g.

```
lettype point = ... , line = ...
with intersect line → line → point = ... ,
     connect point → point → line = ...
in ... .
```

Such a generalization can still be defined in terms of our simple lettype, but the details are messy.

It is an open question how this kind of type definition can be combined with the subtype discipline of Section 4. There are clearly problems if the programmer can define his own implicit conversions. But even if he can only select a subset of the conversions that are induced by existing conversions between representations, it is not clear how to preserve the conditions (LUB) and (GLB) of Section 4b.

5b. Explicit Polymorphism

In [31], I defined a language that has come to be known as the polymorphic, or second-order typed lambda calculus, in which polymorphic functions can be defined by abstraction on type variables, and such functions can be applied to type expressions. (Somewhat to my chagrin, it was only much later that I learned that a similar, somewhat more general language had been invented earlier by J.-Y. Girard [32].)

This facility can be added to our explicitly typed base language (with type variables permitted in type expressions) by introducing the new expressions

$$E ::= \Lambda T.\ E \mid E[\Omega] \qquad \text{(S20)}$$

in which Λ binds type variables just as λ binds identifiers, with the rule of type beta reduction:

$$(\Lambda\alpha.\ e)[\omega] = e\big|_{\alpha \to \omega}\ . \qquad \text{(R20)}$$

For example,

$$\Lambda\alpha.\ \lambda f_{\alpha \to \alpha}.\ \lambda x_{\alpha}.\ f(f\ x)$$

is a polymorphic "doubling" function that can be applied to any type to obtain a doubling function for that type, e.g.

$$(\Lambda\alpha.\ \lambda f_{\alpha \to \alpha}.\ \lambda x_{\alpha}.\ f(f\ x))[\underline{\text{int}}] = \lambda f_{\text{int} \to \text{int}}.\ \lambda x_{\text{int}}.\ f(f\ x)$$

$$(\Lambda\alpha.\ \lambda f_{\alpha \to \alpha}.\ \lambda x_{\alpha}.\ f(f\ x))[\underline{\text{real}} \to \underline{\text{bool}}] = \lambda f_{(\text{real} \to \text{bool}) \to (\text{real} \to \text{bool})}.\ \lambda x_{\text{real} \to \text{bool}}.\ f(f\ x)\ .$$

Less trivially,

$$\Lambda\alpha.\ \underline{\text{fix}}(\lambda ap_{\text{list}\ \alpha \to \text{list}\ \alpha \to \text{list}\ \alpha}.\ \lambda x_{\text{list}\ \alpha}.\ \lambda y_{\text{list}\ \alpha}.\ \underline{\text{lchoose}}\ y\ (\lambda n_{\alpha}.\ \lambda z_{\text{list}\ \alpha}.\ \underline{\text{cons}}\ n\ (ap\ z\ y))\ x)$$

is a polymorphic function that, when applied to a type α, yields a function for appending lists of type $\underline{\text{list}}\ \alpha$. Similarly,

$$\Lambda\alpha.\ \Lambda\beta.\ \underline{\text{fix}}(\lambda red_{\text{list}\ \alpha \to (\alpha \to \beta \to \beta) \to \beta \to \beta}.\ \lambda \ell_{\text{list}\ \alpha}.\ \lambda f_{\alpha \to \beta \to \beta}.\ \lambda a_{\beta}.\ \underline{\text{lchoose}}\ a\ (\lambda n_{\alpha}.\ \lambda z_{\text{list}\ \alpha}.\ f\ n\ (red\ z\ f\ a))\ \ell)$$

is a polymorphic function that, when applied to types α and β, yields a function for reducing lists of type $\underline{\text{list}}\ \alpha$ to values of type β.

Even in 1974, the idea of passing types as arguments to functions was fairly widespread. The novelty was to extend the set of type expressions to provide types for polymorphic functions that were sufficiently refined to permit explicit typing:

$$\Omega ::= \Delta T.\ \Omega$$

Here Δ is an operator that binds type variables within type expressions. (Note that this implies that bound type variables can be renamed. We will regard renamed type expressions such as $\Delta\alpha.\ \alpha \to \alpha$ and $\Delta\beta.\ \beta \to \beta$ to be identical.) The idea is that if e has type ω' then $\Lambda\alpha.$ e has type $\Delta\alpha.\omega'$, and if e has type $\Delta\alpha.\ \omega'$ then e[ω] has type $\omega'\big|_{\alpha \to \omega}$. Thus the polymorphic functions of type $\Delta\alpha.\ \omega'$ can be thought of as functions that, when applied to a type α, give a value of type ω'.

For example, the types of the polymorphic doubling, append, and reduce functions given above (under the empty type assignment) are:

$$\Delta\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha\ ,$$

$$\Delta\alpha.\ \underline{\text{list}}\ \alpha \to \underline{\text{list}}\ \alpha \to \underline{\text{list}}\ \alpha\ ,$$

$$\Delta\alpha.\ \Delta\beta.\ \underline{\text{list}}\ \alpha \to (\alpha \to \beta \to \beta) \to \beta \to \beta\ .$$

More precisely, explicit typing is provided by the inference rules

$$\frac{\pi \mid\!\!- e: \omega'}{\pi \mid\!\!- \Lambda\alpha.\ e: \Delta\alpha.\ \omega'} \qquad \begin{array}{l}\text{when } \alpha \text{ does not occur free} \\ \text{in any type expression} \\ \text{assigned by } \pi\end{array} \qquad \text{(E20a)}$$

and

$$\frac{\pi \vdash e: \Delta\alpha.\ \omega'}{\pi \vdash e[\omega]:(\omega'\big|_{\alpha \to \omega})} \qquad \text{(E20b)}$$

Once type abstraction and application have been introduced, the lettype expression of the previous subsection can be defined as an abbreviation (just as the ordinary let can be defined in terms of ordinary abstraction and application):

$$\underline{\text{lettype}}\ \alpha = \omega\ \underline{\text{in}}\ e \equiv (\Lambda\alpha.\ e)[\omega] \ .$$

From this definition, one can use reduction rule (R20) to derive (R17), and inference rules (E20a) and (E20b) to derive (E17).

More important, we have a notion of explicit polymorphism that includes that of Section 3b, but goes further to permit higher-order functions that accept polymorphic functions, albeit at the expense of a good deal of explicit type information. For instance, we can mirror the example of a polymorphic let in Section 3b by

$$\begin{aligned}&\underline{\text{let}}\ \text{red} = \text{"polymorphic reduce function"}\ \underline{\text{in}}\\ &\quad \lambda\ell\ell\ell_{\text{list list int}}.\ \text{red}[\underline{\text{list}}\ \underline{\text{int}}, \underline{\text{int}}]\ \ell\ell\\ &\quad (\lambda\ell_{\text{list int}}.\ \lambda s_{\text{int}}.\ \underline{\text{add}}(\text{red}[\underline{\text{int}}, \underline{\text{int}}]\ \ell\ \underline{\text{add}}\ 0)\ s)\ 0\ .\end{aligned}$$

But now we can go further and rewrite this let as the application of a higher-order function to the polymorphic reduce function:

$$\begin{aligned}&(\lambda\text{red}_{\Delta\alpha.\ \Delta\beta.\ \text{list}\ \alpha \to (\alpha \to \beta \to \beta) \to \beta \to \beta}.\\ &\quad \lambda\ell\ell\ell_{\text{list list int}}.\ \text{red}[\underline{\text{list}}\ \underline{\text{int}}, \underline{\text{int}}]\ \ell\ell\\ &\quad (\lambda\ell_{\text{list int}}.\ \lambda s_{\text{int}}.\ \underline{\text{add}}(\text{red}[\underline{\text{int}}, \underline{\text{int}}]\ \ell\ \underline{\text{add}}\ 0)\ s)\ 0)\\ &\quad (\text{"polymorphic reduce function"}) \ .\end{aligned}$$

5c. Higher-Order Polymorphic Programming

At first sight, functions that accept polymorphic functions seem exotic beasts of dubious utility. But the work of a number of researchers sugests that such functions may be the key to a novel programming style. They have studied an austere subset of the language we are considering in which the fixpoint operator fix is excluded, and have shown that this restricted language has extraordinary properties [32,33]. On the one hand, all expressions have normal forms, i.e. their evaluation always terminates (though the proof of this fact requires "second-order arithmetic" and cannot be obtained from Peano's axioms). On the other hand, the variety of functions that can be expressed goes far beyond the class of primitive recursive functions. (Indeed, one can express any program whose termination can be proved in second-order arithmetic.) Beyond this, they have shown that certain types of polymorphic functions provide rich structures akin to data algebras. [20, 34, 35]. (In particular, every many-sorted, anarchic, free algebra is "mimicked" by some

polymorphic type.)

Our purpose here is not to give the details of this theoretical work, but to illustrated the unusual programming style that underlies it. The starting point is a variant of the way in which early investigators of the (untyped) lambda calculus encoded truth values and natural numbers.

Suppose we regard bool as an abbreviation for a polymorphic type, and true and false as abbreviations for certain functions of this type, as follows:

$$\underline{\text{bool}} \equiv \Delta\alpha.\ \alpha \to (\alpha \to \alpha) \quad ,$$
$$\underline{\text{true}} \equiv \Lambda\alpha.\ \lambda x_\alpha.\ \lambda y_\alpha.\ x \quad ,$$
$$\underline{\text{false}} \equiv \Lambda\alpha.\ \lambda x_\alpha.\ \lambda y_\alpha.\ y \quad .$$

Then, when e_1 and e_2 have type ω, we can define the conditional expression by

$$\underline{\text{if}}\ b\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2 \equiv b[\omega]\ e_1\ e_2 \quad .$$

Moreover, we can define such functions as

$$\text{not} \equiv \lambda b_{\text{bool}}.\ \Lambda\alpha.\ \lambda x_\alpha.\ \lambda y_\alpha.\ b[\alpha]\ y\ x \quad ,$$
$$\text{and} \equiv \lambda b_{\text{bool}}.\ \lambda c_{\text{bool}}.\ \Lambda\alpha.\ \lambda x_\alpha.\ \lambda y_\alpha.\ b[\alpha](c[\alpha]\ x\ y)\ y \quad .$$

Similarly (ignoring negative numbers), we can define int and the natural numbers by

$$\underline{\text{int}} \equiv \Delta\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha) \quad ,$$
$$0 \equiv \Lambda\alpha.\ \lambda f_{\alpha \to \alpha}.\ \lambda x_\alpha.\ x \quad ,$$
$$1 \equiv \Lambda\alpha.\ \lambda f_{\alpha \to \alpha}.\ \lambda x_\alpha.\ f\ x \quad ,$$
$$n \equiv \Lambda\alpha.\ \lambda f_{\alpha \to \alpha}.\ \lambda x_\alpha.\ f^n\ x \quad ,$$

where $f^n\ x$ denotes

$$\underbrace{f(\ \ldots\ (f}_{n\ \text{times}}\ x)\ \ldots\ ) \quad ,$$

so that the number n becomes a polymorphic function accepting a function and giving its n-fold composition. (For example, 2 is the polymorphic doubling function.) Then we can define the function

$$\text{succ} \equiv \lambda n_{\text{int}}.\ \Lambda\alpha.\ \lambda f_{\alpha \to \alpha}.\ \lambda x_\alpha.\ f(n[\alpha]\ f\ x)$$

that increases its argument by one.

Now suppose g of type $\underline{\text{int}} \to \omega$, c of type ω, and h of type $\omega \to \omega$ satisfy the equations

$$g\ 0 = c \quad ,$$
$$g\ (n + 1) = h\ (g\ n) \quad .$$

Then $g\ n = h^n\ c$, so that

$$g = \lambda n_{\text{int}}.\ n[\omega]\ h\ c \quad .$$

For example (taking ω = $\underline{\text{int}}$),

$$\text{add m 0} = \text{m} \ , \qquad \text{add m (n + 1)} = \text{succ(add m n)} \ ,$$

so that

$$\text{add m} = \lambda n_{\underline{\text{int}}}.\ \text{n}[\underline{\text{int}}]\ \text{succ m} \ ,$$

or more abstractly,

$$\text{add} \equiv \lambda m_{\underline{\text{int}}}.\ \lambda n_{\underline{\text{int}}}.\ \text{n}[\underline{\text{int}}]\ \text{succ m} \ .$$

Similarly, we can define

$$\text{mult} \equiv \lambda m_{\underline{\text{int}}}.\ \lambda n_{\underline{\text{int}}}.\ \text{n}[\underline{\text{int}}]\ (\text{add m})\ 0 \ ,$$

$$\text{exp} \equiv \lambda m_{\underline{\text{int}}}.\ \lambda n_{\underline{\text{int}}}.\ \text{n}[\underline{\text{int}}]\ (\text{mult m})\ 1 \ .$$

More generally, suppose f of type $\underline{\text{int}} \to \omega$, c of type ω, and h of type $\underline{\text{int}} \to \omega \to \omega$ satisfy the equations

$$\text{f } 0 = \text{c} \ , \qquad \text{f } (n + 1) = \text{h n (f n)} \ .$$

Let g of type $\underline{\text{int}} \to \underline{\text{int}} \times \omega$ be such that g n = <n, f n>, (where $\omega \times \omega'$ is $\underline{\text{prod}}(\omega,\omega')$).

$$\text{g } 0 = \langle 0, \text{c} \rangle \ ,$$

$$\begin{aligned} \text{g } (n + 1) &= \langle n + 1, \text{f } (n + 1) \rangle = \langle \text{succ n, h n (f n)} \rangle \\ &= (\lambda z_{\underline{\text{int}} \times \omega}.\ \langle \text{succ z.1, h z.1 z.2} \rangle)\ \langle \text{n, f n} \rangle \\ &= (\lambda z_{\underline{\text{int}} \times \omega}.\ \langle \text{succ z.1, h z.1 z.2} \rangle)\ (\text{g n}) \ . \end{aligned}$$

Thus by the argument given above,

$$\text{g} = \lambda n_{\underline{\text{int}}}.\ \text{n}[\underline{\text{int}} \times \omega]\ (\lambda z_{\underline{\text{int}} \times \omega}.\ \langle \text{succ z.1, h z.1 z.2} \rangle)\ \langle 0, \text{c} \rangle \ ,$$

so that

$$\text{f} = \lambda n_{\underline{\text{int}}}.\ (\text{n}[\underline{\text{int}} \times \omega]\ (\lambda z_{\underline{\text{int}} \times \omega}.\ \langle \text{succ z.1, h z.1 z.2} \rangle)\ \langle 0, \text{c} \rangle).2 \ .$$

Thus if we define primrec of type $\Delta\alpha.\ (\underline{\text{int}} \to \alpha \to \alpha) \to \alpha \to \underline{\text{int}} \to \alpha$ by

$$\begin{aligned} \text{primrec} \equiv\ & \Lambda\alpha.\ \lambda h_{\underline{\text{int}} \to \alpha \to \alpha}.\ \lambda c_{\alpha}. \\ & \lambda n_{\underline{\text{int}}}.\ (\text{n}[\underline{\text{int}} \times \alpha]\ (\lambda z_{\underline{\text{int}} \times \alpha}.\ \langle \text{succ z.1, h z.1 z.2} \rangle)\ \langle 0, \text{c} \rangle).2 \ , \end{aligned}$$

then f = primrec[ω] h c.

For example, since the predecessor function satisfies

$$\text{pred } 0 = 0 \ , \qquad \text{pred}(n + 1) = n = (\lambda n_{\underline{\text{int}}}.\ \lambda m_{\underline{\text{int}}}.\ \text{n})\ \text{n (pred n)} \ ,$$

we can define

$$\text{pred} \equiv \text{primrec}[\underline{\text{int}}](\lambda n_{\underline{\text{int}}}.\ \lambda m_{\underline{\text{int}}}.\ \text{n})\ 0 \ ,$$

and since the factorial function satisfies

$$\text{fact } 0 = 1 \ , \qquad \text{fact}(n + 1) = \text{mult n (fact n)} \ ,$$

we can define

$$\text{fact} \equiv \text{primrec}[\underline{\text{int}}]\ \text{mult}\ 1 \ .$$

Moreover, we can define functions from integers to other types. For example, primrec[list int] cons $(\mathrm{nil}_{\mathrm{int}})$ is the function from int to list int that maps n into (n - 1, ... , 0).

However, our ability to define numerical functions is not limited to the scheme of primitive recursion. For example, the exponentiation laws $f^m \cdot f^n = f^{m+n}$ and $(f^m)^n = f^{m \times n}$ lead to the definitions

$$\mathrm{add} \equiv \lambda m_{\mathrm{int}}.\ \lambda n_{\mathrm{int}}.\ \Lambda\alpha.\ \lambda f_{\alpha \to \alpha}.\ \lambda x_{\alpha}.\ n[\alpha]\ f\ (m[\alpha]\ f\ x) \quad ,$$

$$\mathrm{mult} \equiv \lambda m_{\mathrm{int}}.\ \lambda n_{\mathrm{int}}.\ \Lambda\alpha.\ \lambda f_{\alpha \to \alpha}.\ n[\alpha](m[\alpha]\ f) \quad ,$$

and the law $(\lambda f.\ f^m)^n = \lambda f.\ f^{(m^n)}$ (which can be proved by induction on n) leads to

$$\mathrm{exp} \equiv \lambda m_{\mathrm{int}}.\ \lambda n_{\mathrm{int}}.\ \Lambda\alpha.\ n[\alpha \to \alpha](m[\alpha]) \quad .$$

More spectacularly, suppose we define

$$\mathrm{aug} \equiv \lambda f_{\mathrm{int} \to \mathrm{int}}.\ \lambda n_{\mathrm{int}}.\ \mathrm{succ}\ n\ [\underline{\mathrm{int}}]\ f\ 1 \quad ,$$

of type (int → int) → int → int. Then

$$\mathrm{aug}\ f\ 0 = f\ 1\ , \qquad \mathrm{aug}\ f\ (n+1) = f(\mathrm{aug}\ f\ n)\ .$$

Thus, if we define

$$\mathrm{ack} \equiv \lambda m_{\mathrm{int}}.\ m[\underline{\mathrm{int}} \to \underline{\mathrm{int}}]\ \mathrm{aug}\ \mathrm{succ} \quad ,$$

of type int → int → int, we get

$$\mathrm{ack}\ 0 = \mathrm{succ}\ , \qquad \mathrm{ack}\ (m+1) = \mathrm{aug}\ (\mathrm{ack}\ m) \quad ,$$

so that

$$\mathrm{ack}\ 0\ n = n + 1 \quad ,$$

$$\mathrm{ack}\ (m+1)\ 0 = \mathrm{aug}\ (\mathrm{ack}\ m)\ 0 = \mathrm{ack}\ m\ 1 \quad ,$$

$$\mathrm{ack}\ (m+1)\ (n+1) = \mathrm{aug}\ (\mathrm{ack}\ m)\ (n+1) = \mathrm{ack}\ m\ (\mathrm{aug}\ (\mathrm{ack}\ m)\ n)$$
$$= \mathrm{ack}\ m\ (\mathrm{ack}\ (m+1)\ n) \quad ,$$

i.e., $\lambda n_{\mathrm{int}}.\ \mathrm{ack}\ n\ n$ is Ackermann's function.

In addition to the primitive types int and bool, various compound types can be defined in terms of polymorphic functions. For numbered products we can define

$$\underline{\mathrm{prod}}(\omega_1, \ \ldots\ , \omega_n) \equiv \Delta\alpha.\ (\omega_1 \to \ldots \to \omega_n \to \alpha) \to \alpha$$

and, when $e_1, \ \ldots\ , e_n$ have types $\omega_1, \ \ldots\ , \omega_n$ and r has type $\underline{\mathrm{prod}}(\omega_1, \ \ldots\ , \omega_n)$,

$$\langle e_1, \ \ldots\ , e_n \rangle \equiv \Lambda\alpha.\ \lambda f_{\omega_1 \to \ldots \to \omega_n \to \alpha}.\ f\ e_1 \ \ldots\ e_n \quad ,$$

$$r.k \equiv r[\omega_k]\ (\lambda x_{1\omega_1}.\ \ldots\ \lambda x_{n\omega_n}.\ x_k) \quad ,$$

since we have the reduction

$$\langle e_1, \ \ldots\ , e_n \rangle.k$$
$$= (\Lambda\alpha.\ \lambda f_{\omega_1 \to \ldots \to \omega_n \to \alpha}.\ f\ e_1 \ \ldots\ e_n)\ [\omega_k](\lambda x_{1\omega_1}.\ \ldots\ \lambda x_{n\omega_n}.\ x_k)$$
$$= (\lambda f_{\omega_1 \to \ldots \to \omega_n \to \omega_k}.\ f\ e_1 \ \ldots\ e_n)(\lambda x_{1\omega_1}.\ \ldots\ \lambda x_{n\omega_n}.\ x_k)$$

$$= (\lambda x_{1\omega_1}.\ \ldots\ \lambda x_{n\omega_n}.\ x_k)\ e_1\ \ldots\ e_n = e_k \quad .$$

Similarly, for numbered sums, we can define

$$\underline{\text{sum}}(\omega_1, \ \ldots\ , \omega_n) \equiv \Delta\alpha.\ (\omega_1 \to \alpha) \to \ldots \to (\omega_n \to \alpha) \to \alpha$$

and, when e has type ω_k and $f_1, \ \ldots\ , f_n$ have types $\omega_1 \to \omega, \ \ldots\ , \omega_n \to \omega$,

$$\underline{\text{inject}}_{\omega_1,\ldots,\omega_n}\ k\ e \equiv \Lambda\alpha.\ \lambda h_{1\omega_1 \to \alpha}.\ \ldots\ \lambda h_{n\omega_n \to \alpha}.\ h_k\ e \quad ,$$

$$\underline{\text{choose}}(f_1, \ \ldots\ , f_n) \equiv \lambda s_{\text{sum}(\omega_1,\ldots,\omega_n)}.\ s[\omega]\ f_1\ \ldots\ f_n \quad ,$$

since we have the reduction

$$\begin{aligned}
&\underline{\text{choose}}(f_1, \ \ldots\ , f_n)(\underline{\text{inject}}_{\omega_1,\ldots,\omega_n}\ k\ e) \\
&\quad = (\lambda s_{\text{sum}(\omega_1,\ldots,\omega_n)}.\ s\ [\omega]\ f_1\ \ldots\ f_n)(\Lambda\alpha.\ \lambda h_{1\omega_1 \to \alpha}.\ \ldots\ \lambda h_{n\omega_n \to \alpha}.\ h_k\ e) \\
&\quad = (\Lambda\alpha.\ \lambda h_{1\omega_1 \to \alpha}.\ \ldots\ \lambda h_{n\omega_n \to \alpha}.\ h_k\ e)\ [\omega]\ f_1\ \ldots\ f_n \\
&\quad = (\lambda h_{1\omega_1 \to \omega}.\ \ldots\ \lambda h_{n\omega_n \to \omega}.\ h_k\ e)\ f_1\ \ldots\ f_n = f_k\ e \quad .
\end{aligned}$$

Finally, we consider lists, drawing upon a discovery by C. Böhm [35]. We define

$$\underline{\text{list}}\ \omega \equiv \Delta\beta.\ (\omega \to \beta \to \beta) \to \beta \to \beta \quad ,$$

with the idea that, if $\ell = (x_1, \ \ldots\ , x_n)$, f has type $\omega \to \omega' \to \omega'$, and a has type ω',

$$\ell[\omega']\ f\ a = f\ x_1\ (f\ x_2\ \ldots\ (f\ x_n\ a) \quad \ldots\ ) \quad .$$

In other words, we regard a list as its own reduce function. Then we define

$$\underline{\text{nil}}_\omega \equiv \Lambda\beta.\ \lambda f_{\omega \to \beta \to \beta}.\ \lambda a_\beta.\ a$$

and, when e_1 has type ω and e_2 has type $\underline{\text{list}}\ \omega$,

$$\underline{\text{cons}}\ e_1\ e_2 \equiv \Lambda\beta.\ \lambda f_{\omega \to \beta \to \beta}.\ \lambda a_\beta.\ f\ e_1\ (e_2\ f\ a) \quad .$$

Now we can carry out a development analogous to that for natural numbers. If g of type $\underline{\text{list}}\ \omega \to \omega'$, c of type ω', and h of type $\omega \to \omega' \to \omega'$ satisfy

$$g\ \underline{\text{nil}}_\omega = c \ , \qquad g(\underline{\text{cons}}\ x\ \ell) = h\ x\ (g\ \ell) \ ,$$

then

$$g = \lambda\ell_{\underline{\text{list}}\ \omega}.\ \ell[\omega']\ h\ c \ .$$

For example, consider the function rappend, of type $\underline{\text{list}}\ \omega \to \underline{\text{list}}\ \omega \to \underline{\text{list}}\ \omega$, that is similar to the append function but with interchanged arguments. Since this function satisfies

$$\text{rappend}\ m\ \underline{\text{nil}}_\omega = m \ , \qquad \text{rappend}\ m\ (\underline{\text{cons}}\ x\ \ell) = \underline{\text{cons}}\ x\ (\text{rappend}\ m\ \ell) \quad ,$$

we have

$$\text{rappend} = \lambda\ell_{\underline{\text{list}}\ \omega}.\ \ell[\underline{\text{list}}\ \omega]\ (\lambda x_\omega.\ \lambda y_{\underline{\text{list}}\ \omega}.\ \underline{\text{cons}}\ x\ y)\ m \quad ,$$

and thus

$$\text{append} \equiv \lambda\ell_{\underline{\text{list}}\ \omega}.\ \lambda m_{\underline{\text{list}}\ \omega}.\ \ell[\underline{\text{list}}\ \omega]\ (\lambda x_\omega.\ \lambda y_{\underline{\text{list}}\ \omega}.\ \underline{\text{cons}}\ x\ y)\ m \quad .$$

Next, consider the function flatten, of type $\underline{\text{list}}\ \underline{\text{list}}\ \omega \to \underline{\text{list}}\ \omega$, that appends together the elements of its arguments. Since this function satisfies

$$\text{flatten}\ \underline{\text{nil}}_{\underline{\text{list}}\ \omega} = \underline{\text{nil}}_{\omega} \quad , \qquad \text{flatten}\ (\underline{\text{cons}}\ x\ \ell) = \text{append}\ x\ (\text{flatten}\ \ell) \quad ,$$

we have

$$\text{flatten} \equiv \lambda\ell_{\underline{\text{list}}\ \underline{\text{list}}\ \omega}.\ \ell[\underline{\text{list}}\ \omega]\ \text{append}\ \underline{\text{nil}}_{\omega} \quad .$$

By similar arguments,

$$\text{length} \equiv \lambda\ell_{\underline{\text{list}}\ \omega}.\ \ell[\underline{\text{int}}]\ (\lambda x_{\omega}.\ \lambda y_{\underline{\text{int}}}.\ \text{succ}\ y)\ 0 \quad ,$$

$$\text{sumlist} \equiv \lambda\ell_{\underline{\text{list}}\ \underline{\text{int}}}.\ \ell[\underline{\text{int}}]\ \underline{\text{add}}\ 0 \quad ,$$

$$\text{mapcar} \equiv \lambda\ell_{\underline{\text{list}}\ \omega}.\ \lambda f_{\omega \to \omega'}.\ \ell[\underline{\text{list}}\ \omega']\ (\lambda x_{\omega}.\ \lambda y_{\underline{\text{list}}\ \omega'}.\ \underline{\text{cons}}\ (f\ x)\ y)\ \underline{\text{nil}}_{\omega'} \quad ,$$

$$\text{null} \equiv \lambda\ell_{\underline{\text{list}}\ \omega}.\ \ell[\underline{\text{bool}}]\ (\lambda x_{\omega}.\ \lambda y_{\underline{\text{bool}}}.\ \underline{\text{false}})\ \underline{\text{true}} \quad ,$$

$$\text{car} \equiv \lambda\ell_{\underline{\text{list}}\ \omega}.\ \ell[\omega]\ (\lambda x_{\omega}.\ \lambda y_{\omega}.\ x)\ \underline{\text{error}} \quad .$$

More generally, suppose f of type $\underline{\text{list}}\ \omega \to \omega'$, c of type ω' and h of type $\omega \to \underline{\text{list}}\ \omega \to \omega' \to \omega'$ satisfy

$$f\ \underline{\text{nil}}_{\omega} = c \quad , \qquad f(\underline{\text{cons}}\ x\ \ell) = h\ x\ \ell\ (f\ \ell) \quad .$$

Let g, of type $\underline{\text{list}}\ \omega \to (\underline{\text{list}}\ \omega) \times \omega'$, be such that $g\ \ell = \langle \ell, f\ \ell \rangle$. Then

$$g\ \underline{\text{nil}}_{\omega} = \langle \underline{\text{nil}}_{\omega}, c \rangle \quad ,$$

$$g\ (\underline{\text{cons}}\ x\ \ell) = \langle \underline{\text{cons}}\ x\ \ell, f\ (\underline{\text{cons}}\ x\ \ell) \rangle = \langle \underline{\text{cons}}\ x\ \ell, h\ x\ \ell\ (f\ \ell) \rangle$$

$$= (\lambda x_{\omega}.\ \lambda z_{(\underline{\text{list}}\ \omega) \times \omega'}.\ \langle \underline{\text{cons}}\ x\ z.1, h\ x\ z.1\ z.2 \rangle)\ x\ (g\ \ell) \quad ,$$

so that

$$g = \lambda\ell_{\underline{\text{list}}\ \omega}.\ \ell[(\underline{\text{list}}\ \omega) \times \omega']$$
$$(\lambda x_{\omega}.\ \lambda z_{(\underline{\text{list}}\ \omega) \times \omega'}.\ \langle \underline{\text{cons}}\ x\ z.1, h\ x\ z.1\ z.2 \rangle)\ \langle \underline{\text{nil}}_{\omega}, c \rangle \quad ,$$

and $f\ \ell = (g\ \ell).2$. Thus, if we define

$$\text{listrec} \equiv \Lambda\alpha.\ \Lambda\beta.\ \lambda h_{\alpha \to \underline{\text{list}}\ \alpha \to \beta \to \beta}.\ \lambda c_{\beta}.\ \lambda\ell_{\underline{\text{list}}\ \alpha}.$$
$$(\ell[(\underline{\text{list}}\ \alpha) \times \beta](\lambda x_{\alpha}.\ \lambda z_{(\underline{\text{list}}\ \alpha) \times \beta}.\ \langle \underline{\text{cons}}\ x\ z.1, h\ x\ z.1\ z.2 \rangle)\langle \underline{\text{nil}}_{\omega}, c \rangle).2$$

we have $f = \text{listrec}[\omega][\omega']\ h\ c$.

For example, since cdr of type $\underline{\text{list}}\ \omega \to \underline{\text{list}}\ \omega$ satisfies

$$\text{cdr}\ \underline{\text{nil}}_{\omega} = \underline{\text{error}} \quad , \qquad \text{cdr}(\underline{\text{cons}}\ x\ \ell) = \ell \quad ,$$

we can define

$$\text{cdr} \equiv \text{listrec}[\omega][\underline{\text{list}}\ \omega](\lambda x_{\omega}.\ \lambda y_{\underline{\text{list}}\ \omega}.\ \lambda z_{\underline{\text{list}}\ \omega}.\ y)\ \underline{\text{error}} \quad ,$$

and since, when a has type ω' and f has type $\omega \to \underline{\text{list}}\ \omega \to \omega'$,

$$\underline{\text{lchoose}}\ a\ f\ \underline{\text{nil}}_{\omega} = a \quad , \qquad \underline{\text{lchoose}}\ a\ f\ (\underline{\text{cons}}\ x\ \ell) = f\ x\ \ell \quad ,$$

we can define

$$\underline{\text{lchoose}}\ a\ f \equiv \text{listrec}[\omega][\omega'](\lambda x_{\omega}.\ \lambda y_{\underline{\text{list}}\ \omega}.\ \lambda z_{\omega'}.\ f\ x\ y)\ a \quad .$$

Less trivially, consider the function insertandapply, of type $\underline{\text{int}} \to (\underline{\text{list}}\ \underline{\text{int}} \to \underline{\text{list}}\ \underline{\text{int}}) \to \underline{\text{list}}\ \underline{\text{int}} \to \underline{\text{list}}\ \underline{\text{int}}$, such that when ℓ is an ordered list of integers, insertandapply n f ℓ inserts n into the proper position of ℓ and applies f to the portion of ℓ following this position. This function satisfies

$$\text{insertandapply n f } \underline{\text{nil}}_{\text{int}} = \underline{\text{cons}}\ \text{n (f } \underline{\text{nil}}_{\text{int}}) \ .$$

$$\text{insertandapply n f } (\underline{\text{cons}}\ \text{x } \ell) =$$
$$\underline{\text{if}}\ \text{n} \leq \text{x } \underline{\text{then}}\ \underline{\text{cons}}\ \text{n (f } (\underline{\text{cons}}\ \text{x } \ell)) \ \underline{\text{else}}\ \underline{\text{cons}}\ \text{x (insertandapply n f } \ell) \ ,$$

so that we can define

$$\text{insertandapply} \equiv \lambda \text{n}_{\text{int}} \cdot\ \lambda \text{f}_{\text{list int} \to \text{list int}} \cdot\ \text{listrec}[\underline{\text{int}}][\underline{\text{list}}\ \underline{\text{int}}]$$
$$(\lambda \text{x}_{\text{int}} \cdot\ \lambda \ell_{\text{list int}} \cdot\ \lambda \text{m}_{\text{list int}} \cdot$$
$$\underline{\text{if}}\ \text{n} \leq \text{x } \underline{\text{then}}\ \underline{\text{cons}}\ \text{n (f } (\underline{\text{cons}}\ \text{x } \ell)) \ \underline{\text{else}}\ \underline{\text{cons}}\ \text{x m})$$
$$(\underline{\text{cons}}\ \text{n (f } \underline{\text{nil}}_{\text{int}})) \ .$$

Then let merge, of type $\underline{\text{list}}\ \underline{\text{int}} \to \underline{\text{list}}\ \underline{\text{int}} \to \underline{\text{list}}\ \underline{\text{int}}$, be the function that merges two ordered lists. This function satisfies

$$\text{merge } \underline{\text{nil}}_{\text{int}} = \lambda \text{m}_{\text{list int}} \cdot\ \text{m} \ ,$$

$$\text{merge } (\underline{\text{cons}}\ \text{x } \ell) = \text{insertandapply x (merge } \ell) \ .$$

Thus we can define

$$\text{merge} \equiv \lambda \ell_{\text{list int}} \cdot\ \ell[\underline{\text{list}}\ \underline{\text{int}} \to \underline{\text{list}}\ \underline{\text{int}}]\ \text{insertandapply } (\lambda \text{m}_{\text{list int}} \cdot\ \text{m}) \ .$$

Semantically, the introduction of type abstraction and application raises thorny problems. Since, in the absence of fix, all programs terminate, one might expect to obtain a semantics in which types denote ordinary sets and $\omega \to \omega'$ denotes the set of all functions from ω to ω'. But it can be shown that no such set is possible [36]. (Specifically, one can show that if the polymorphic type

$$\Delta\alpha. \ (((\alpha \to \underline{\text{bool}}) \to \underline{\text{bool}}) \to \alpha) \to \alpha$$

denoted some set, then from this set one could construct a set P that is isomorphic to $(\text{P} \to \underline{\text{bool}}) \to \underline{\text{bool}}$, which is impossible.)

The only known models [18,19] are domain-theoretic ones that give a semantics to the language with fix as well as without. Moreover, polymorphic types in these models denote huge domains that include generic functions that are not parametric in the sense of Strachey [24].

It should be mentioned that [18] and [19] provide models for a more general language than is described here, in which one can define functions from types to types and (in a somewhat different sense than in Section 3c, recursively defined types.

5d. Type Deduction Revisited

As the examples in the last subsection have made all too clear, explicit typing requires a dismaying amount of type information. This raises the problem of type deduction.

Consider taking an explicitly typed expression and erasing all type information by carrying out the replacements

$$\lambda i_{\omega}.\ e \Rightarrow \lambda i.\ e \qquad \Lambda\alpha.\ e \Rightarrow e \qquad e[\omega] \Rightarrow e$$

(For simplicity, we are only considering the pure polymorphic typed lambda calculus here, as defined by syntax equations (S1), (S5a,b) and (S20a,b).) This erasure maps our language back into the original (lambda-calculus subset of the) base language. But now the base language has a richer type structure than in Section 2b, since $\Delta\alpha.\ \omega$ remains a type expression and inference rules (E20a,b) become

$$\frac{\pi \mid\!\!- e: \omega'}{\pi \mid\!\!- e: \Delta\alpha.\ \omega'} \qquad \begin{array}{l}\text{when } \alpha \text{ does not occur free}\\ \text{in any type expression}\\ \text{assigned by } \pi\end{array} \qquad \text{(I20a)}$$

$$\frac{\pi \mid\!\!- e: \Delta\alpha.\ \omega'}{\pi \mid\!\!- e: (\omega'\big|_{\alpha \to \omega})} \qquad \text{(I20b)}$$

An obvious question is whether one can give an algorithm for type deduction (in the sense of Section 3a) with these additional rules, i.e. (roughly speaking) an algorithm for recovering the erased type information. The answer, however, is far from obvious; despite strenuous efforts the question remains open. However, it has proved possible to devise useful but more limited algorithms that allow the programmer to omit some, though not all, type information [37,38].

5e. Existential Types

In [32], Girard introduced, in addition to Δ, another binding operator $\exists$ that produces types called existential types, with the syntax

$$\Omega ::= \exists T.\ \Omega$$

Recently, Mitchell and Plotkin [30] suggested that an existential type can be thought of as the signature of an abstract data type (in roughly the sense of algebraic data types), and that values of the existential type can be thought of as representations of the abstract data type.

For example, in a type definition of the form

$$\underline{\text{lettype}}\ \alpha = \omega\ \underline{\text{with}}\ i_{1\omega_1} = e_1, \ \ldots\ , i_{n\omega_n} = e_n\ \underline{\text{in}}\ e\ ,$$

the type ω and the expression $\langle e_1, \ \ldots\ , e_n\rangle$ together constitute a data-type representation that is a value of the existential type $\exists\alpha.\ \underline{\text{prod}}(\omega_1, \ \ldots\ , \omega_n)$. Note that this existential type provides exactly the information to ascertain that the

e_k's bear the right relationship to ω, i.e. that each must have the type $\omega_k\big|_{\alpha \to \omega}$.

To "package" such representations, we introduce the operator rep, with the syntax

$$E ::= \underline{rep}_{\exists T.\Omega}\ \Omega\ E \tag{S21a}$$

For example, the above representation would be the value of

$$\underline{rep}_{\exists\alpha.prod(\omega_1,\ldots,\omega_n)}\ \omega\ \langle e_1, \ldots, e_n\rangle \ .$$

To unpackage representations, we introduce the operator abstype, with the syntax

$$E ::= \underline{abstype}\ T\ \underline{with}\ I = E\ \underline{in}\ E \ . \tag{S21b}$$

These operators are related by the reduction rule

$$\underline{abstype}\ \alpha\ \underline{with}\ i = (\underline{rep}_{\exists\alpha.\omega'}\ \omega\ e_2)\ \underline{in}\ e_1 = e_1\big|_{\alpha \to \omega}\big|_{i \to e_2} \ . \tag{R21}$$

(Here we can assume the two occurrence of α are the same type variable since the occurrence in the subscript is a bound variable that can be renamed.)

Using these new constructs, the lettype expression given above can be defined as an abbreviation for

$$\underline{abstype}\ \alpha\ \underline{with}\ i = (\underline{rep}_{\exists\alpha.prod(\omega_1,\ldots,\omega_n)}\ \omega\ \langle e_1, \ldots, e_n\rangle)$$
$$\underline{in}\ \underline{let}\ i_1 = i.1\ \underline{in}\ \ \ldots\ \ \underline{let}\ i_n = i.n\ \underline{in}\ e \ ,$$

where i is an identifier not occurring free in e. Notice that this definition reduces to

$$(\underline{let}\ i_1 = i.1\ \underline{in}\ \ \ldots\ \ \underline{let}\ i_n = i.n\ \underline{in}\ e)\big|_{\alpha \to \omega}\big|_{i \to \langle e_1, \ldots, e_n\rangle} \ ,$$

and then to

$$e\big|_{\alpha \to \omega}\big|_{i_1 \to e_1, \ldots, i_n \to e_n} \ ,$$

which coincides with the reduction of the alternative definition of lettype ... with given in Section 5a.

The new operations satisfy the following inference rules:

$$\frac{\pi \ \vdash e: (\omega'\big|_{\alpha \to \omega})}{\pi \ \vdash \underline{rep}_{\exists\alpha.\omega'}\ \omega\ e: \exists\alpha.\ \omega'} \tag{E21a}$$

$$\frac{\pi,\ i: \omega' \ \vdash e_1: \omega_0 \qquad \pi \ \vdash e_2: \exists\alpha.\ \omega'}{\pi \ \vdash \underline{abstype}\ \alpha\ \underline{with}\ i = e_2\ \underline{in}\ e_1: \omega_0} \qquad \begin{array}{l}\text{when } \alpha \text{ does not occur free in}\\ \omega_0 \text{ or in any type expression}\\ \text{assigned by } \pi\end{array} \tag{E21b}$$

(Note that the presence of the subscript $\exists\alpha.\ \omega'$ is necessary to make the first rule explicit.) From these rules and the definition of lettype ... with ... in terms of abstype and rep, one can derive an inference rule that is similar to (E19) except for the addition of a proviso that α must not occur free in ω_0 (which Mitchell and Plotkin believe is necessary to make lettype ... with ... truly abstract.)

Of course, existential types would be uninteresting if their only purpose was to provide an alternative definition of lettype ... with ... . The real point of the matter is that they permit abstract data type representations to be "first class" values that can be bound to identifiers and given as arguments to functions. For instance, the example in Section 5a can now be rewritten as

```
(λr∃complex. prod(complex, complex → complex → complex)·
    abstype complex with prims = r in
        let i = prims.1 in let addc = prim.2 in ... )
(rep∃complex. prod(complex, complex → complex → complex)
    prod(real, real)
    <<0, 1>, λx prod(real,real)· λy prod(real,real)·
                        <add x.1 y.1, add x.2 y.2>>) .
```

Here the first three lines denote a function that could be defined separately and applied to a variety of representations of type

$$\exists\ \text{complex.}\ \underline{\text{prod}}(\text{complex, complex} \to \text{complex} \to \text{complex})\ ,$$

such as representations of complex numbers using different coordinate systems.

The analogy with algebraic data types is that $\exists\alpha.\ \omega$ is a signature and that values of the type are algebras of the signature. However, the analogy breaks down in two ways. On the one hand, $\exists\alpha.\ \omega$ cannot contain equations constraining the algebra, i.e. one can only mirror anarchic algebras this way. On the other hand, existential types go beyond algebra in permitting the primitive operations to be higher-order functions.

ACKNOWLEDGEMENT

This survey has been inspired by conversations with numerous researchers in the area of type theory. I am particularly indebted to Gordon Plotkin, Nancy McCracken, and Lockwood Morris.

REFERENCES

1. Guttag, J.V., and Horning, J.J., "The Algebraic Specification of Abstract Data Types", Acta Informatica 10 (1), 1978, pp. 27-52.

2. Goguen, J.A., Thatcher, J.W., and Wagner, E.G., "Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", Current Trends in Programming Methodology, 4, Data Structuring, ed. R.T. Yeh, Prentice-Hall, Englewood Cliffs, 1978.

3. Kapur, D., "Towards a Theory for Abstract Data Types", Tech. Rep. TR-237, Laboratory for Computer Science, M.I.T., May 1980.

4. Coppo, M., and Dezani-Ciancaglini, M., "A New Type Assignment for λ-terms", Archive f. math. Logik u. Grundlagenforschung 19 (1979) 139-156.

5. Barendregt, H., Coppo, M., and Dezani-Ciancaglini, M., "A Filter Lambda Model and the Completeness of Type Assignment", to appear in the Journal of Symbolic Logic.

6. Martin-Lof, P., "Constructive Mathematics and Computer Programming", Proceedings of the Sixth (1979) International Congress for Logic, Methodology and Philosophy of Science, North-Holland, Amsterdam, 1979.

7. Constable, Robert L. and Zlatin, D.R., "The Type Theory of PL/CV3", ACM Transactions on Programming Languages and Systems, 6 (1984), 94-117.

8. Coquand, T., and Huet, G., "A Theory of Constructions", unpublished.

9. Burstall, R., and Lampson, B., "A Kernel Language for Abstract Data Types and Modules", Semantics of Data Types, eds. G. Kahn, D.B. MacQueen, and G. Plotkin, Lecture Notes in Computer Science 173, Springer-Verlag, Berlin (1984), pp. 1-50.

10. Reynolds, J.C., "The Essence of Algol", Algorithmic Languages, eds. J.W. de Bakker and J.C. van Vliet, North-Holland, 1981, pp. 345-372.

11. Oles, F.J., "A Category-Theoretic Approach to the Semantics of Programming Languages", Ph. D. dissertation, Syracuse University, August 1982.

12. Oles, F.J., "Type Algebras, Functor Categories, and Block Structure", Algebraic Methods in Semantics, eds. M. Nivat and J.C. Reynolds, Cambridge University Press (1985).

13. Henderson, P., and Morris, J.H., "A Lazy Evaluator". Proc. 3rd annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Atlanta, 1976, 95-103.

14. Friedman, D.P., and Wise, D.S., "CONS Should not Evaluate its Arguments". In Automata, Languages and Programing, eds. Michaelson and Milner, Edinburgh University Press, 1976, 257-284.

15. Landin, P.J., "A Correspondence Between Algol 60 and Church's Lambda-Notation", Comm. ACM 8, (February-March 1965), pp. 89-101 and 158-165.

16. McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", Comm. ACM 3 (April 1960), pp. 184-195.

17. MacQueen, D.B., and Sethi, R., "A Semantic Model of Types for Applicative Languages", ACM Symposium on LISP and Functional Programming, 1982, pp. 243-252.

18. McCracken, N., "An Investigation of a Programming Language with a Polymorphic Type Structure", Ph. D. dissertation, Syracuse University, June 1979.

19. McCracken, N.J., "A Finitary Retract Model for the Polymorphic Lambda-Calculus", submitted to Information and Control.

20. Reynolds, J.C., "Types, Abstraction and Parametric Polymorphism", Information Processing 83, ed. R.E.A. Mason, Elsevier Science Publishers B.V. (North-Holland) 1983, pp. 513-523.

21. Hindley, R., "The Principal Type-scheme of an Object in Combinatory Logic", Trans. Amer. Math. Society 146 (1969) 29-60.

22. Milner, R., "A Theory of Type Polymorphism in Programming", Journal of Computer and System Sciences 17 (1978), 348-375.

23. Robinson, J.A., "A Machine-oriented Logic Based on the Resolution Principle", JACM 12, 1 (1965), 23-41.

24. Strachey, C., "Fundamental Concepts in Programming Languages", Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967.

25. Huet, G., "Resolution d'Équations dans des Langages d'Ordre 1,2,..., ω", doctoral thesis, University of Paris VII (September 1976).

26. Morris, F.L., "On List Structures and Their Use in the Programming of Unification", School of Computer and Information Science, Report 4-78, Syracuse University, 1978.

27. Reynolds, J.C., "Using Category Theory to Design Implicit Conversions and Generic Operators", Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark, January 14-18, 1980, ed. N.D. Jones, Lecture Notes in Computer Science 94, Springer-Verlag, New York, pp. 211-258.

28. Goguen, J.A., "Order Sorted Algebras: Exceptions and Error Sorts, Coercions and Overloaded Operators", Semantics and Theory of Computation Report 14, Computer Science Department, UCLA (December 1978). To appear in Journal of Computer and Systems Science.

29. Cardelli, L., "A Semantics of Multiple Inheritance", Semantics of Data Types, eds. G. Kahn, D.B. MacQueen and G. Plotkin, Lecture Notes in Computer Science 173, Springer-Verlag, Berlin (1984), pp. 51-67.

30. Mitchell, J.C., and Plotkin, G.D., "Abstract Types Have Existential Type", Proc. 12th Annual ACM Symposium on Principles of Programming Languages, New Orleans, 1985.

31. Reynolds, J.C., "Towards a Theory of Type Structure, Proc. Colloque sur la Programmation, Lecture Notes in Computer Science 19, Springer-Verlag, New York, 1974, pp. 408-425.

32. Girard, J.-Y., "Interprétation Fonctionelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur", Thèse de Doctorat d'État, Paris, 1972.

33. Fortune, S., Leivant, D., and O'Donnell, M., "The Expressiveness of Simple and Second-Order Type Structures", Journal of the ACM 30, 1 (January 1983), pp. 151-185.

34. Leivant, D., "Reasoning About Functional Programs and Complexity Classes Associated with Type Disciplines", Twenty-fourth Annual Symposium on Foundation of Computer Science (1983) 460-469.

35. Böhm, C., and Berarducci, A., "Automatic Synthesis of Typed Λ-Programs on Term Algebras", submitted for publication.

36. Reynolds, J.C., "Polymorphism is not Set-Theoretic", Semantics of Data Types, eds. G. Kahn, D.B. MacQueen, and G. Plotkin, Lecture Notes in Computer Science 173, Springer-Verlag, Berlin (1984), pp. 145-156.

37. Leivant, D., "Polymorphic Type Inference", Proc. 10th Annual ACM Symposium on Principles of Programming Languages, Austin, 1983.

38. McCracken, N.J., "The Typechecking of Programs with Implicit Type Structure", Semantics of Data Types, eds. G. Kahn, D.B. MacQueen, and G. Plotkin, Lecture Notes in Computer Science 173, Springer-Verlag, Berlin (1984), pp. 301-315.