Abstract Types Have Existential Type

JOHN C. MITCHELL
Stanford University
AND
GORDON D. PLOTKIN
University of Edinburgh

Abstract data type declarations appear in typed programming languages like Ada, Alphard, CLU and ML. This form of declaration binds a list of identifiers to a type with associated operations, a composite "value" we call a *data algebra*. We use a second-order typed lambda calculus SOL to show how data algebras may be given types, passed as parameters, and returned as results of function calls. In the process, we discuss the semantics of abstract data type declarations and review a connection between typed programming languages and constructive logic.

Categories and Subject Descriptors: D.3 [**Software**]: Programming Languages; D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*; F.3 [**Theory of Conmputation**]: Logics and Meanings of Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics, operational semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Abstract data types, lambda calculus, polymorphism, programming languages, types

1. INTRODUCTION

Ada packages [17], Alphard forms [66, 71], CLU clusters [41, 42], and **abstype** declarations in ML [23] all bind identifiers to values. Although there are minor variations among these constructs, each allows a list of names to be bound to a composite value consisting of "private" type and one or more operations. For example, the ML declaration

```
abstype complex = real # real
  with create = ···
  and plus = ···
  and re = ···
  and im = ···
```

An earlier version of this paper appeared in the *Proceedings of the 12th ACM Symposium on Principles of Programming Languages* (New Orleans, La., Jan. 14–16). ACM, New York, 1985.
Authors' addresses: J. C. Mitchell, Department of Computer Science, Stanford University, Stanford, CA 94305; G. D. Plotkin, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland EH9 3J2.

binds the identifiers *complex*, *create*, *plus*, *re*, and *im* to the components of an implementation of complex numbers. The implementation consists of the collection defined by the ML expression *real # real*, meaning the type of pairs of reals, and the functions denoted by the code for *create*, *plus*, and so on. An important aspect of this construct is that access to the representation is limited. We cannot apply arbitrary operations on pairs of reals to elements of type *complex*; only the explicitly declared operations may be used.

We will call a composite value constructed from a set and one or more operations, packaged up in a way that limits access, a *data algebra*. We will discuss the typing rules associated with the formation and the use of data algebras and observe that data algebras themselves may be given types in a straightforward manner. This will allow us to devise a typed programming notation in which implementations of abstract data types may be passed as parameters or returned as the results of function calls.

The phrase "abstract data type" sometimes refers to a class of algebras (or perhaps an initial algebra) satisfying some specification. For example, the abstract type *stack* is sometimes regarded as the class of all algebras satisfying the familiar logical formulas axiomatizing *push* and *pop*. Associated with this view is the tenet that a program must rely only on the data type specification, as opposed to properties of a particular implementation. Although this is a valuable guiding principle, most programming languages do not contain assertions or their proofs, and without this information it is impossible for a compiler to guarantee that a program depends only on a data type specification. Since we are primarily concerned with properties of the abstract data type declarations used in common programming languages, we will focus on the limited form of information hiding or "abstraction" provided by conventional type checking rules.

We can be more specific about how data algebras are defined by considering the declaration of complex numbers in more detail. Using an explicitly typed ML-like notation, the declaration sketched earlier looks something like this:

```
abstype complex = real # real
  with create: real → real → complex = λx:real. λy:real.⟨x, y⟩
  and plus: complex → complex =
        λz:real # real. λw:real # real. ⟨ fst(z) + fst(w), snd(z) + snd(w)⟩
  and re: complex → real = λz:real # real.fst(z)
  and im: complex → real = λz:real # real. snd(z)
```

The identifiers *complex, create, plus, re,* and *im* are bound to a data algebra whose elements are represented as pairs of reals, as specified by the type expression *real # real.* The operations of the data algebra are given by the function expressions to the right of the equals signs.[1] Notice that the declared types of the operations differ from the types of the implementing functions. For example, *re* is declared to have type *complex → real,* but the implementing expression has type *real # real → real.* This is because operations are defined using the concrete representation of values, but the representation is hidden outside the declaration.

In the next section, we will discuss the type checking rules associated with abstract data type declarations, which are designed to make complex numbers

[1] In most programming languages, function definitions have the form "create(*x*:real, *y*:real) = ···." In the example above, we have used explicit lambda abstraction to move the formal parameters from the left- to the right-hand sides of the equals signs.

"abstract" outside the data algebra definition. In the process, we will give types to data algebras. These will be *existential types*, which were originally developed in constructive logic and are closely related to infinite sums (as in category theory, for example). In Section 3, we describe a statically typed language SOL. This language is a notational variant of Girard's system **F**, developed in the analysis of constructive logic [21, 22], and an extension of Reynolds' polymorphic lambda calculus [62]. An operational semantics of SOL, based on the work of Girard and Reynolds, is presented using reduction rules. However, we do not address a variety of practical implementation issues. Although the basic calculus we use has been known for some time, we believe that the analysis of data abstraction using existential types originates with this paper. (A preliminary version appeared as [56].)

The use of SOL as a proof-theoretic tool is based on an analogy between types and constructive logic. This analogy gives rise to a large family of typed languages and suggests that our analysis of abstract data types applies to more expressive languages involving specifications. Since the connection between constructive proofs and typed programs does not seem to be well known in the programming language community (at least at present), our brief discussion of specifications will follow a review of the general analogy in Section 4. Additional SOL programming examples are given in Section 5.

The design of SOL suggests new programming languages along the lines of Ada, Alphard, CLU, and ML but with richer and more flexible type structures. In addition, SOL seems to be a natural "kernel language" for studying the semantics of languages with polymorphic functions and abstract data type declarations. For this reason, we expect SOL to be useful in future studies of current languages. It is clear that SOL provides greater flexibility in the use of abstract data types than previous languages, since data algebras may be passed as parameters and returned as results. We believe that this is accomplished without any compromise in "type security." However, since we do not have a precise characterization of type security, we are unable to show rigorously that SOL is secure.[2]

Some languages that are similar to SOL in scope and intent are Pebble [7], designed to capture some essential features of Cedar (an extension of Mesa [57]), and Kernel Russell, KR, of [28], based on Russell [14, 15, 16]. Martin-Löf's constructive type theory [46] and the calculus of constructions [11] are farther from programming language syntax but share many properties of SOL. Some features of Martin-Löf's system have been incorporated into the Standard ML module design [44, 54], which was formulated after the work described here was completed. We will compare SOL with some of these languages in Section 3.8.

2. TYPING RULES FOR ABSTRACT DATA TYPE DECLARATIONS

The basic typing rules associated with abstract data type declarations do not differ much from language to language. To avoid the unnecessary complication of discussing a variety of syntactic forms, we describe abstract data types using the syntax we will adopt in SOL. Although this syntax was chosen to resemble

[2] Research begun after this paper was written has shed some light on the type security of SOL. See [52] and [55] for further discussion.

common languages, there is one novel aspect that leads to additional flexibility: We separate the names bound by a declaration from the data algebra they come to denote. For example, the complex number example is written as follows:

abstype complex **with**
  create: real → real → complex,
  plus: complex → complex,
  re: complex → real,
  im: complex → real
is
pack real ∧ real
  λx:real.λy:real.$\langle x, y \rangle$
  λz:real ∧ real.λw:real ∧ real.$\langle \mathrm{fst}(z) + \mathrm{fst}(w), \mathrm{snd}(z) + \mathrm{snd}(w) \rangle$
  λz:real ∧ real.fst(z)
  λz:real ∧ real.snd(z) **to** $\exists t.[(\mathrm{real} \to \mathrm{real} \to t) \wedge (t \to t) \wedge (t \to \mathrm{real}) \wedge (t \to \mathrm{real})]$,

where the expression beginning **pack** and running to the end of the example is considered to be the definition of the data algebra. (In SOL, we write *real* ∧ *real* for the type of pairs of reals. When parentheses are omitted, the connective ∧ has higher precedence than →.) This syntax is designed to allow implementations of abstract data types (data algebras) to be defined using expressions of any form and to emphasize the view that abstract data type declarations commonly combine two separable actions, defining a data algebra and binding identifiers to its components.

The SOL declaration of an abstract data type t with operations $x_1, \ldots, x_n$ has the general form

abstype t **with** x_1: σ_1, ..., x_n: σ_n **is** M **in** N,

where $\sigma_1, \ldots, \sigma_n$ are the types of the operations and M is a data algebra expression. As in the complex number example above, the type identifier t often appears in the types of operations $x_1, \ldots, x_n$. The scope of the declaration is N.

The simplest data algebra expressions in SOL are those of the form

pack $\tau M_1 \cdots M_n$ **to** $\exists t.\sigma$

where τ is a type expression, $M_1, \ldots, M_n$ are "ordinary" expressions (denoting values of type τ, or functions, for example) and $\exists t.\sigma$ is an "existential type" describing the way that the data algebra may be used. The language SOL also allows more general forms of data algebra expressions, which we will get to later on. There are three typing rules associated with **abstype**.

It is easy to see that a declaration

abstype t **with** x_1: σ_1, ..., x_n: σ_n
  is pack $\tau M_1 \cdots M_k$ **to** $\exists t.\sigma$
  in N

involving a basic data algebra expression only makes sense if $k = n$ (so that each operation gets an implementation) and the types of $M_1, \ldots, M_k$ match the declared types of the operations $x_1, \ldots, x_n$ in some appropriate way. The matching rule in SOL is that the type of M_i must be $[\tau/t]\sigma_i$, the result of substituting τ for t in σ_i (with appropriate renaming of bound type variables in σ_i). To see how this works in practice, look back at the complex number declaration. The declared

type of the first operation *create* is *real* → *real* → *complex*, whereas the type of the implementing function expression is *real* → *real* → (*real* ∧ *real*). The matching rule is satisfied in this case because the type of the implementing code may be obtained by substituting *real* ∧ *real* for *complex* in the declared type *real* → *real* → *complex*.

We can recast the matching rule using the existential types we have associated with data algebra expressions. An appropriate type for a data algebra is an expression that specifies how the operations may be used, without describing the type used to represent values. If each M_i has type $[\tau/t]\sigma_i$, then we say that

pack $\tau M_1 \cdots M_n$ **to** $\exists t.\sigma_1 \wedge \cdots \wedge \sigma_n$

has type $\exists t.\sigma_1 \wedge \cdots \wedge \sigma_n$. This type may be read "there exists a type t with operations of types σ_1 and $\cdots$ and σ_n." The operator $\exists$ binds the type variable t in $\exists t.\sigma$, so $\exists t.\sigma = \exists s.[s/t]\sigma$ when s does not occur in σ. Existential types provide just enough information to verify the matching condition stated above, without providing any information about the representation of the carrier or the algorithms used to implement the operations. The matching rule for **abstype** may now be stated.

(AB.1) In **abstype** t **with** x_1: σ_1, ..., x_n: σ_n **is** M **in** N, the data algebra expression M must have type $\exists t.\sigma_1 \wedge \cdots \wedge \sigma_n$.

Although it may seem unnecessarily verbose to write the type of **pack** $\cdots$ **to** $\cdots$ as part of the expression, this is needed to guarantee that the type is unique. Without the type designation, an expression like **pack** τM could have many types. For example, if the type of M is $\tau \to \tau$, then **pack** τM might have types $\exists t.t \to t$, $\exists t.t \to \tau$, $\exists t.\tau \to t$, and $\exists t.\tau \to \tau$. To avoid this, we have included the intended type of the whole expression as part of the syntax. Something equivalent to this is done in most other languages. In CLU, for example, types are determined using the keyword **cvt**, which specifies which occurrences of the representation type are to be viewed as abstract. ML, as documented in [23], uses keywords **abs** and **rep**, whereas later versions [50] use type constructors and pattern matching.

An important constraint in abstract type declarations is that only the explicitly declared operations may be applied to elements of the type [58]. In SOL, this constraint is formulated as follows:

(AB.2) In **abstype** t **with** x_1: $\sigma_1, \ldots, x_n$: σ_n **is** M **in** N, if y is any free identifier in N different from $x_1, \ldots, x_n$, then t must not appear free in the type of y.

In addition to accomplishing the goals put forth in [58], this condition is easily seen to be a natural scoping rule for type identifiers. We can see why (AB.2) makes sense and what kind of expressions it prevents by considering the following example.

```
let f = λx: stack ··· in
  abstype stack with empty : stack,
      push : int ∧ stack → stack,
      pop : stack → int ∧ stack
    is ···
    in f(empty)
end
```

In this program fragment, the declaration of function *f* specifies a formal parameter *x:stack*, and so the domain of *f* is some type called *stack*. For this reason, the application of *f* to the empty stack might seem sensible at first glance. However, notice that since the name *stack* in the declaration of *f* is outside the scope of the *stack* declaration shown, the meaning of *stack* in the type of *f* is determined by some outer declaration in the full program. Therefore, the identifier *stack* in the type of *f* refers to a different type from the identifier *stack* in the type of *empty*. Semantically, we have a type mismatch. Rule (AB.2) prohibits exactly this kind of program since the identifier *f* is free in the scope of the **abstype** declaration, but *stack* occurs free (unbound) in the type of *f*. Note that rule (AB.2) mentions only free occurrences of type names. This is because SOL has types with bound variables, and the names of bound variables are unimportant.

Since SOL **abstype** declarations are local to a specific scope, rather than global, we also need to consider whether the representation of a data type should be accessible outside the scope of a declaration. The designers of ML, another language with local **abstype** declarations, decided that it should not (see [23], p. 56). In our notation and terminology, the ML restriction is

(AB.3) In **abstype** t **with** x_1: σ_1, ..., x_n: σ_n **is** M **in** N, the type variable t must not be free in the type of N.

One way for t to appear free in the type of N is for N to be one of the operations of the abstract type. For example, if t appears free in σ_1, then (AB.3) will prohibit the expression

abstype t **with** x_1: σ_1, ..., x_n: σ_n **is** M **in** x_1

which exports the first operation outside the scope of the declaration. (If t does not appear in the type of x_1, then this expression is allowed.) In designing modules for Standard ML, MacQueen has argued that this restriction is too strong [44, 45]. If programs are composed of sets of modules (instead of being block structured, like SOL terms) then it makes sense to use the constituents of a data algebra defined in one module to construct a related data algebra in another module. However, this really seems to be an objection to block-structured programs and not a criticism of **abstype** as a means of providing data abstraction. In fact, there are several good reasons to adopt rule (AB.3) in SOL.

One justification for (AB.3) is that SOL type checking becomes algorithmically intractable without it. In SOL, we consider any expression of the correct type a data algebra expression. One useful example not allowed in many conventional languages is the conditional expression. If both

pack $\tau M_1 \cdots M_n$ **to** $\exists t.\sigma$

and

pack $\rho P_1 \cdots P_n$ **to** $\exists t.\sigma$

are data algebra expressions of the same existential type, then

if B **then pack** $\tau M_1 \cdots M_n$ **to** $\exists t.\sigma$
else pack $\rho P_1 \cdots P_n$ **to** $\exists t.\sigma$

is a data algebra expression of SOL with type $\exists t.\sigma$. Conditional algebra expressions are useful for selecting between several alternative implementations of the same abstract type. For example, a program using matrices may choose between sparse or dense matrix implementations using a conditional data algebra expression inside an **abstype** declaration. Without (AB.3), the type of an **abstype** expression with a data algebra conditional such as

abstype t **with** x_1: $t, \ldots, x_n$: σ_n
  is if B **then** (**pack** $\tau M_1 \cdots M_n$ **to** $\exists t.\sigma$)
    else (**pack** $\rho P_1 \cdots P_n$ **to** $\exists t.\sigma$)
in x_1

may depend on whether the conditional test is true or false. (Specifically, the meaning of the expression above is either M_1 or P_1, depending on B). Thus, without (AB.3), we cannot type check expressions with conditional data algebra expressions at "compile time," that is, without computing the values of arbitrary tests.

Another way of describing this situation is to consider the form of type expression we would need if we wanted to give the expression above a type without evaluating B. Since the type of the expression actually depends on the value of B, we would have to mention B in the type. This approach is used in some languages (notably Martin–Löf's intuitionistic type theory), but it introduces ordinary value expressions into types. Consequently, type equality depends on equality of ordinary expressions. Some of the simplicity of SOL is due to the separation of type expressions from "ordinary" expressions, and considerable complication would arise from giving this up.

Finally, the termination of all recursion-free programs seems to fail if we drop (AB.3). In other words, there is a roundabout way of writing programs that do not halt on any input, without using any recursive declarations or iterative constructs. This is a complex issue whose full explanation is beyond the scope of this paper. The reader is referred to [10], [29], [49], and [54] for further discussion. Putting all of these reasons together, it seems that dropping (AB.3) would change the nature of SOL quite drastically. Therefore, we leave the study of **abstype** without (AB.3) to future research.

With rule (AB.3) in place, we can allow very general computation with data algebras. In addition to conditional data algebra expressions, SOL allows data algebra parameters. An example that illustrates their use is the general tree search routine given in Section 2.5. The usual algorithms for depth-first search and breadth-first search may be written so that they are virtually identical, except that depth-first search uses a stack and breadth-first search uses a queue. The general tree-search algorithm in Section 2.6 is based on this idea, using a formal parameter in place of a stack or queue. If a stack data algebra is supplied as an actual parameter, then the algorithm performs depth-first search. Similarly a queue parameter produces breadth-first search. Additional structures like priority queues may also be passed as actual parameters, resulting in "best-first" search algorithms.

Data algebra parameters are allowed in SOL simply because the typing rules do not prevent them. If z is a variable with type $\exists t.\sigma_1 \wedge \cdots \wedge \sigma_n$, then

abstype t **with** x_1: $\sigma_1, \ldots, x_n$: σ_n **is** z **in** N

is a well-typed expression of SOL. Since SOL allows parameters of all types, there is nothing to prevent the data algebra z from being a formal parameter. By typing data algebra expressions and treating all types in SOL in the same way, we allow conditional data algebra expressions, data algebra parameters, and many other useful kinds of computation on data algebras.

The next section presents the language SOL in full. To emphasize our belief that SOL **abstype** captures the "essence" of data abstraction, we have described the construct as if we had designed it for this purpose. However, we emphasize again that SOL is not our invention at all; SOL with existential types was invented by Girard as a proof-theoretic tool [22], and SOL without existential types was developed independently by Reynolds as a model of polymorphism [62]. The purpose of our paper is to explain that existential types provide a paradigm example of data type declarations and to suggest some advantages of this point of view.

3. THE TYPED LANGUAGE SOL

We show how implementations of abstract data types can be typed and passed as function parameters or results by describing the functional language SOL. Although SOL is an applicative language, we believe that this treatment of data algebras also pertains to imperative languages. This belief is based on the general similarity between binding constructs of functional and imperative languages and is supported by previous research linking lambda calculus and programming languages (e.g., [36, 37, 63]).

There are two classes of expressions in SOL: type expressions and terms. In contrast to more complicated languages such as Pebble [7], KR [28], Martin-Löf's constructive type theory [46], and the calculus of constructions [11], types may appear in terms, but terms do not appear in type expressions. The type expressions are defined by the following abstract syntax

$$\sigma ::= t \mid c \mid \sigma \to \tau \mid \sigma \wedge \tau \mid \sigma \vee \tau \mid \forall t.\sigma \mid \exists t.\sigma.$$

In our presentation of SOL, we use two sorts of variables, type variables r, s, t, ... and ordinary variables x, y, z, ... In the syntax above, t may be any type variable and c any type constant. Some possible type constants are *int* and *bool*, which we often use in examples. Intuitively, $\sigma \to \tau$ is the type of functions from σ to τ, an element of the product type $\sigma \wedge \tau$ is a pair with one component from σ and the other from τ, and an element of the disjoint union or tagged sum type $\sigma \vee \tau$ is an element of σ or τ.

The two remaining forms involve $\forall$ and $\exists$, which bind type variables in type expressions. The universal type $\forall t.\sigma$ is a type of polymorphic functions and elements of $\exists t.\sigma$ are data algebras. Free and bound variables in type expressions are determined precisely using a straightforward inductive definition, with $\forall$ binding t in $\forall t.\sigma$ and $\exists$ binding t in $\exists t.\sigma$. Since t is bound in $\forall t.\sigma$ and $\exists t.\sigma$, we consider $\forall t.\sigma = \forall s.[s/t]\sigma$ and $\exists t.\sigma = \exists s.[s/t]\sigma$, provided s does not occur free in σ. (Recall that $[s/t]\sigma$ is the result of substituting s for free occurrences of t in σ, with bound type variables renamed to avoid capture.)

In SOL, as in most typed programming languages, the type of an expression depends on the types given to its free variables. We incorporate "context" into

the typing rules using *type assignments*, which are functions from ordinary variables to type expressions. For each type assignment A, we define a partial function Type_A from expressions to types. Intuitively, $\text{Type}_A(M) = \sigma$ means that the type of M is σ, given the assignment A of types to variables that may appear free in M. Each partial function Type_A is defined by a set of deduction rules of the form

$$\frac{\text{Type}_A(M) = \sigma, \ldots}{\text{Type}_A(N) = \tau}$$

meaning that if the antecedents hold, then the value of Type_A at N is defined to be τ. The conditions on $\text{Type}_A(N)$ may mention other type assignments if N binds variables that occur in subterms.

A variable of any type is a term. Formally, we have the axiom

$$\text{Type}_A(x) = A(x)$$

saying that a variable x has whatever type it is given. We also allow term constants, provided that each constant is assigned a type that does not contain free type variables. One particularly useful constant is the polymorphic conditional **cond**, which will be discussed after $\forall$-types are introduced.

3.1 Functions and Let

In SOL, we take functions of a single argument as basic and introduce functions of several arguments as a derived form. A function expression explicitly declares the type of the formal parameter. Consequently, the type of the function body is determined in a typing context that incorporates the formal parameter type. If A is a type assignment, then $A[x:\sigma]$ is a type assignment with

$$(A[x:\sigma])(y) = \begin{cases} \sigma & \text{if } y \text{ is the same variable as } x \\ A(y) & \text{otherwise.} \end{cases}$$

The deduction rules for function abstraction and application are

$$\frac{\text{Type}_{A[x:\sigma]}(M) = \tau}{\text{Type}_A(\lambda x{:}\sigma.M) = \sigma \to \tau}$$

and

$$\frac{\text{Type}_A(M) = \sigma \to \tau,\ \text{Type}_A(N) = \sigma}{\text{Type}_A(MN) = \tau}$$

Thus a typed lambda expression has a functional type and may be applied to any argument of the correct type. An example function expression is the lambda expression

$$\lambda x{:}\text{int}.\ x + 1$$

for the successor function on integers.

The semantics of SOL is described using a set of operational reduction rules. The reduction rules use substitution, and, therefore, require the ability to rename bound variables. For functions, we rename bound variables according to the

equational axiom

$$\lambda x{:}\sigma.M = \lambda y{:}\sigma.[y/x]M, \qquad y \text{ not free in } M$$

The operational semantics of function definition and application are captured by the reduction rule

$$(\lambda x{:}\sigma.M)N \Rightarrow [N/x]M,$$

where we assume that substitution $[N/x]M$ includes renaming of bound variables to avoid capture. (Technically speaking, the collection of SOL reduction rules defines a relation on equivalence classes of SOL terms, where equivalence is defined by the collection of all SOL axioms for renaming bound variables. See, e.g., [2] for further discussion). Intuitively, the reduction rule above says that the expression $(\lambda x{:}\sigma.M)N$ may be evaluated by substituting the argument N for each free occurrence of the variable x in M. For example,

$$(\lambda x{:}\text{int}.\ x + 2)5 \Rightarrow 5 + 2.$$

Some readers may recognize this mechanism as the "copy rule" of ALGOL 60. We write $\Rightarrow\!\!>$ for the congruent and transitive closure of $\Rightarrow$.

We introduce **let** declarations by the abbreviation

$$\textbf{let } x = M \textbf{ in } N ::= (\lambda x{:}\sigma.N)M,$$

where $\sigma = \text{Type}_A(M)$. Note that since the assignment A of types to variables is determined by context, the definition of **let** depends on the context in which it is used. An alternative would be to write **let** $x{:}\sigma = M$ **in** N, but since σ is always uniquely determined, the more succinct **let** notation seems preferable.

The typing rules and operational semantics for **let** are inherited directly from λ. For example, we have

$$\textbf{let } f = \lambda x{:}\text{int}.\ x + 3 \textbf{ in } f(f(2)) \Rightarrow\!\!> (2 + 3) + 3.$$

A similar declaration is the ML recursive declaration

$$\textbf{letrec } f = M \textbf{ in } N$$

which declares f to be a recursive function with body M. (If f occurs in M, then this refers recursively to the function being defined; occurrences of f in M are bound by **letrec**.) Although we use **letrec** in programming examples, it is technically useful to define pure SOL as a language without recursion. This pure language has simpler theoretical properties, making it easier to study the type structure of SOL.

3.2 Products and Sums

A simple kind of **record** type is the unlabeled pair. In SOL, we use $\wedge$ for pair or product types. Product types have associated pairing and projection functions as follows:

$$\frac{\text{Type}_A(M) = \sigma,\ \text{Type}_A(N) = \tau}{\text{Type}_A(\langle M, N\rangle) = \sigma \wedge \tau}$$

$$\frac{\text{Type}_A(M) = \sigma \wedge \tau}{\text{Type}_A(\textbf{fst } M) = \sigma,\ \text{Type}_A(\textbf{snd } M) = \tau}$$

The operational semantics of pairing and projection are given by the reduction rules

$$\mathbf{fst}\langle M, N\rangle \Rightarrow M, \qquad \mathbf{snd}\langle M, N\rangle \Rightarrow N.$$

For example,

$$\mathbf{let}\ p = \langle 1, 2\rangle\ \mathbf{in}\ \mathbf{fst}(p) \Rightarrow\!\!> 1.$$

We can introduce multivariable lambda abstraction as an abbreviation involving products. Some metanotation makes this easier to describe. We write $f^i y$ as an abbreviation for $f(f \cdots (fy) \cdots)$ with i occurrences of f. As a convenience, we consider $f^0 y = y$. We also write $y^{(i)_n}$ for the expression $\mathbf{fst}(\mathbf{snd}^{i-1}y)$ if $1 \le i < n$ and $\mathbf{snd}^{n-1}y$ if $i = n$. Thus, if $y = \langle x_1, \langle x_2, \ldots, \langle x_{n-1}, x_n\rangle \cdots\rangle\rangle$, we have $y^{(i)_n} \Rightarrow\!\!> x_i$. Using these abbreviations, we define multiargument lambda abstraction by

$$\lambda\langle x_1{:}\,\sigma_1, \ldots, x_n{:}\,\sigma_n\rangle.M ::= \lambda y{:}\sigma_1 \wedge (\cdots \wedge \sigma_n \cdots).[y^{(1)_n}, \ldots, y^{(n)_n}/x_1, \ldots, x_n]M$$

For example, $\lambda\langle x{:}\sigma, y{:}\tau\rangle.M$ is an abbreviation for

$$\lambda z{:}\sigma \wedge \tau.\ [\mathbf{fst}\ z, \mathbf{snd}\ z/x, y]M.$$

We will also use the abbreviation

$$\mathbf{let}\ f(x_1{:}\,\sigma_1, \ldots, x_n{:}\,\sigma_n) = M\ \mathbf{in}\ N ::= \mathbf{let}\ f = \lambda\langle x_1{:}\,\sigma_1, \ldots, x_n{:}\,\sigma_n\rangle.M\ \mathbf{in}\ N$$

which allows us to declare functions using a more familiar syntax.

Sum types $\vee$ have injection functions and a **case** expression. The SOL **case** statement is similar to the **tagcase** statement of CLU, for example [41].

$$\frac{\mathrm{Type}_A(M) = \sigma}{\mathrm{Type}_A(\mathbf{inleft}\ M\ \mathbf{to}\ \sigma \vee \tau) = \sigma \vee \tau,\ \mathrm{Type}_A(\mathbf{inright}\ M\ \mathbf{to}\ \tau \vee \sigma) = \tau \vee \sigma}$$

$$\frac{\mathrm{Type}_A(M) = \sigma \vee \tau,\ \mathrm{Type}_{A[x:\sigma]}(N) = \rho,\ \mathrm{Type}_{A[y:\tau]}(P) = \rho}{\mathrm{Type}_A(\mathbf{case}\ M\ \mathbf{left}\ x{:}\sigma.N\ \mathbf{right}\ y{:}\tau.P\ \mathbf{end}) = \rho}$$

In the expression above, **case** binds x in N and y in P. As with λ-binding, we equate **case** expressions that differ only in the names of bound variables.

$$\mathbf{case}\ M\ \mathbf{left}\ x{:}\sigma.N\ \mathbf{right}\ y{:}\tau.P\ \mathbf{end} = \mathbf{case}\ M\ \mathbf{left}\ u{:}\sigma.[u/x]N\ \mathbf{right}\ v{:}\tau.[v/y]P\ \mathbf{end},$$

provided u is not free in N and v is not free in P.

It is possible to replace the bindings in **case** expressions with λ-bindings as suggested in [63], making **case** a constant instead of a binding operator. However, the syntax above seems slightly easier to read.

The reduction rules for sums are

$$\mathbf{case}\ (\mathbf{inleft}\ M\ \mathbf{to}\ \sigma \vee \tau)\ \mathbf{left}\ x{:}\sigma.N\ \mathbf{right}\ y{:}\tau.P\ \mathbf{end} \Rightarrow [M/x]N$$
$$\mathbf{case}\ (\mathbf{inright}\ M\ \mathbf{to}\ \sigma \vee \tau)\ \mathbf{left}\ x{:}\sigma.N\ \mathbf{right}\ y{:}\tau.P\ \mathbf{end} \Rightarrow [M/y]P$$

For example,

$$\mathbf{let}\ z = \mathbf{inleft}\ 3\ \mathbf{to}\ \mathrm{int} \vee \mathrm{bool}\ \mathbf{in}\ \mathbf{case}\ z\ \mathbf{left}\ x{:}\mathrm{int}.x\ \mathbf{right}\ y{:}\mathrm{bool}.\mathbf{if}\ y\ \mathbf{then}\ 1\ \mathbf{else}\ 0\ \mathbf{end} \Rightarrow\!\!> 3$$

Note that the type of this **case** statement remains **int** if z is declared to be **inright** of a Boolean instead of **inleft** of an integer.

3.3 Polymorphism

Intuitively, $\lambda t.M$ is a polymorphic expression that can be "instantiated" to values of various types. In an Ada-like syntax, the term $\lambda t.M$ would be written

generic (type t**)**M

Polymorphic expressions are instantiated using *type application*, which we will write using braces $\{,\}$ to distinguish it from an ordinary function application. If M has type $\forall t.\sigma$, then the type of $M\{\tau\}$ is $[\tau/t]\sigma$. The Ada-like syntax for $M\{\tau\}$ is

new $M(\tau)$.

The formal definitions are

$$\frac{\text{Type}_A(M) = \tau}{\text{Type}_A(\lambda t.M) = \forall t.\tau}$$

t not free in $A(x)$ for any x free in M

$$\frac{\text{Type}_A(M) = \forall t.\sigma}{\text{Type}_A(M\{\tau\}) = [\tau/t]\sigma}$$

The restriction on the bound variable t in $\lambda t.M$ eliminates nonsensical expressions like $\lambda x{:}t.\lambda t.x$, where it is not clear whether t is free or bound. (See [20] for further discussion.) Note that unlike many programming languages, a SOL polymorphic function may be instantiated using any type expression whatsoever, regardless of whether its value could be determined at compile time.

One use of $\forall$-types is in the polymorphic conditional **cond**. The constant **cond** has type $\forall t.bool \to t \to t \to t$. We often use the abbreviation

if M **then** N **else** P ::= **cond** $\{\tau\}MNP$,

where $\text{Type}_A(N) = \text{Type}_A(P) = \tau$.

Polymorphic type binding may be used to define polymorphic functions such as the polymorphic maximum function. The type of a function *Max* which, given any type t and order relation $r{:}t \wedge t \to bool$, finds the maximum of a pair of t's is

$$Max\colon \forall t[(t \wedge t \to \text{bool}) \to (t \wedge t) \to t].$$

A SOL expression for the polymorphic maximum function is

Max ::= $\lambda t.\ \lambda r{:}t \wedge t \to \text{bool}.\ \lambda p{:}t \wedge t.$**if** $r(p)$ **then fst**(p) **else snd**(p)

If $r\colon \tau \wedge \tau \to bool$ is an order relation on type τ, then

$$Max\{\tau\}r\langle x, y\rangle$$

finds the maximum of a pair $\langle x, y\rangle$ of elements of type τ. While *Max* is written with the expectation that the actual parameter r will be an order relation, the SOL type checking rules cannot ensure this.

Intuitive Semantics and Reduction Rules for $\lambda t.M$. The intuitive meaning of $\lambda t.M$ is the infinite product of all meanings of M as t varies over all types. In the next section, we see that abstract data type declarations involve infinite sums. To see the similarity between $\forall$-types and infinite products, we review the general notion of product, as used in category theory [1, 27, 43]. There are two parts to the definition: product types (corresponding to product objects in categories) and product elements (corresponding to product arrows). Given a collection S of types, the product type $\prod S$ has the property that for each $s \in S$ there is a projection function **proj** s from $\prod S$ to s. Furthermore, given any family $F = \{f_s\}$ of elements indexed by S with $f_s \in s$, there is a unique product element $\prod F$ with the property that

$$\textbf{proj}\ s \prod F = f_s.$$

Uniqueness of products means that if **proj** $s \prod F = g_s$ for all $s \in S$, then $\prod F = \prod G$.

The correspondence with SOL is that we can think of a type expression σ and type variable t as defining a collection of types, namely the collection S of all substitution instances $[\tau/t]\sigma$ of σ. If M is a term with t not free in the type of any free ordinary variable, then M and t determine a collection of substitution instances $[\tau/t]M$. It is easy to show that if t is not free in the type of any free variable of M and $\text{Type}_A(M) = \sigma$, then $\text{Type}_A([\tau/t]M) = [\tau/t]\sigma$. By letting $f_{[\tau/t]\sigma} = [\tau/t]M$, we may view the collection of substitution instances of M as a family $F = \{f_s\}$ indexed by elements of S. Using this indexing of instances, we may regard $\forall t.\sigma$ as a product type $\prod S$ and $\lambda t.M$ as a product element $\prod F$, with projection accomplished by type application. The product axiom above leads to the reduction rule

$$(\lambda t.M)\{\tau\} \Rightarrow [\tau/t]M,$$

where we assume that bound variables are renamed in $[\tau/t]M$ to avoid capture of free type variables in τ. Since λ binds type variables, we also have the renaming rule

$$\lambda t.M = \lambda s.[s/t]M, \qquad s \text{ not free in } \lambda t.M.$$

There is a third "extensionality" rule for λ-abstraction over types, stemming from the uniqueness of products, but we are not concerned with it in this paper (primarily because it does not seem to be a part of ordinary programming language implementation and because it complicates the Static Typing Theorem in Section 3.7).

3.4 Data Abstraction and Existential Types

Data algebras, or concrete representations of abstract data types, are elements of existential types. The basic rule for data algebra expressions is this.

$$\frac{\text{Type}_A(M) = [\tau/t]\sigma}{\text{Type}_A(\textbf{pack}\ \tau M\ \textbf{to}\ \exists t.\sigma) = \exists t.\sigma}.$$

The more general form described in Section 2 may be introduced as the following abbreviation:

$$\textbf{pack}\ \tau M_1 \cdots M_n\ \textbf{to}\ \exists t.\sigma ::= \textbf{pack}\ \tau\langle M_1, \langle \ldots, M_n\rangle \cdots \rangle\rangle\ \textbf{to}\ \exists t.\sigma,$$

where

$$\sigma = \sigma_1 \wedge (\cdots \wedge \sigma_n \cdots).$$

Polymorphic data algebras may be written in Ada, Alphard, CLU, and ML. Since SOL has λ-binding of types, we can also write polymorphic representations in SOL. For example, let *t_stack* be a representation of stacks of elements of t, say,

$$t\_stack ::= \textbf{pack } (int \wedge \textbf{array of } t)\ empty\ push\ pop \\ \textbf{to } \exists s.s \wedge (t \wedge s \to s) \wedge (s \to t \wedge s),$$

where *empty* represents the empty stack, and *push* and *pop* are functions implementing the usual push and pop operations. Then the expression

$$stack ::= \lambda t.t\_stack$$

with type

$$stack\colon \forall t.\ \exists s.[s \wedge (t \wedge s \to s) \wedge (s \to t \wedge s)]$$

is a polymorphic implementation of stacks. We could also define a polymorphic implementation of queues

$$queue\colon \forall t.\ \exists q.[q \wedge (t \wedge q \to q) \wedge (q \to t \wedge q)]$$

similarly. Note that *stack* and *queue* have the same existential type, reflecting the fact that as algebras, they have the same signature.

Abstract data type declarations are formed according to the rule

$$\frac{\text{Type}_A(M) = \exists t.\sigma,\ \text{Type}_{A[x:\sigma]}(N) = \rho}{\text{Type}_A(\textbf{abstype } s \textbf{ with } x{:}\sigma \textbf{ is } M \textbf{ in } N) = \rho},$$

provided t is not free in ρ or the type $A(y)$ of any free $y \neq x$ occurring in N.

This definition of **abstype** provides all the type constraints discussed in Section 2. Condition (AB.1) is included in the assumption $\text{Type}_A(M) = \exists t.\sigma$, whereas (AB.2) and (AB.3) follow from the restrictions on free occurrences of t. As mentioned earlier, the only restriction on data algebras is that they have the correct type. The more general form is defined by the abbreviation

$$\textbf{abstype } t \textbf{ with } x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n \textbf{ is } M \textbf{ in } N \\ ::= \textbf{abstype } t \textbf{ with } y{:}\sigma_1 \wedge (\cdots \wedge \sigma_n \cdots) \\ \textbf{is } M \textbf{ in } [y^{(1)_n}, \ldots, y^{(n)_n}/x_1, \ldots, x_n]N,$$

where $y^{(1)_n}$ is as defined in Section 3.2.

One advantage of combining polymorphism with data abstraction is that we can use the polymorphic representation of stacks to declare integer stacks. The expression

```
abstype int_stk with empty: int_stk,
    push: int ∧ int_stk → int_stk,
    pop: int_stk → int ∧ int_stk
  is stack {int}
  in N
```

declares a type of integer stacks with three operations. Note that the names for the stack operations are local to N, rather than defined globally by *stack*.

3.5 Programming with Data Algebras

One feature of SOL is that a program may select one of several data type implementations at run time. For example, a parser that uses a symbol table could be parameterized by the symbol table implementation and passed either a hash table or binary tree implementation according to conditions. This ability to manipulate data algebras makes a common feature of file systems and linkage editors an explicit part of SOL. For example, many of the functions of the CLU library, a design for handling multiple implementations [41], may be accomplished directly by programs.

In allowing programs to select representations, we also allow programs to choose among data types that have the same signature. This flexibility accrues from the fact that SOL types are signatures, rather than complete data type specifications: Since we only check signature information, data types that have the same signature have implementations of the same existential type. This is used to advantage in the tree-search algorithm of Figure 1. It may also be argued that this points out a deficiency in the SOL typing discipline. In a language with specifications as types, type checking could guarantee that every actual parameter to a function is an implementation of a stack, rather than just an implementation with a designated element and two binary operations. Languages with this capability will be discussed briefly in Section 4.4.

The common algorithm for depth-first search uses a stack, whereas the usual approach to breadth-first search uses a queue. Since *stack* and *queue* implementations have the same SOL type, the program fragment in Figure 1 declares a tree-search function with a data algebra parameter instead of a stack or queue. If a stack is passed as a parameter, the function does depth-first search, while a queue parameter produces breadth-first. In addition, other data algebras, such as priority queues, could be passed as parameters. A priority queue produces a "best-first" search; the search proceeds along paths that the priority queue deems "best."

The three arguments to the function *search* are a node *start* in a labeled tree, a label *goal* to search for, and the data algebra parameter *struct*. We assume that one tree node is labeled with the goal, so there is no error test. The result of a call to *search* is the first node reached, starting from *start*, whose label matches *goal*. The tree structure is declared at the top of the program fragment to make the types of the tree functions explicit. The tree has a root, each node has a label and is either a leaf or has two descendants. The function *is_leaf*? tests whether a node is a leaf, while *left* and *right* return the left and right descendants of any nonleaf.

3.6 Reduction Rules and Intuitive Semantics of Existential Types

Intuitively, the meaning of the **abstype** expression

$$\textbf{abstype } t \textbf{ with } x{:}\sigma \textbf{ is (pack } \tau M \textbf{ to } \exists t.\sigma) \textbf{ in } N$$

is the meaning of N in an environment where t is bound to τ, and x to M. Operationally, we can evaluate **abstype** expressions using the reduction rule

$$\textbf{abstype } t \textbf{ with } x{:}\sigma \textbf{ is (pack } \tau M \textbf{ to } \exists t.\sigma) \textbf{ in } N \Rightarrow [M/x][\tau/t]N,$$

```
/* Tree data type declaration */
abstype t with root:t, label:t→string, is_leaf?:t→bool,
    left:t→t, right:t→t  is tree
 in
 ...
/* Search returns first node reached from start with label(node) = goal.
The structure parameter may be a stack, queue, priority queue, etc. */
 let  search(start:t, goal:string, struct: ∀t∃s[s∧(t∧s→s)∧(s→t∧s)]) =
   abstype s with empty:s, insert:t∧s→s, delete:s→t∧s
   is struct {t}
   in
      /* function to select next node; also returns updated structure */
      let next(node:t, st:s) =
        if is_leaf?(node) then delete(st)
          else delete(insert(left(node), insert(right(node),st)))
      in
       /* recursive function find calls next until goal reached */
        letrec find(node:t, st:s) =
          if label(node)=goal then node else find(next(node, st))
        in
         /* call find to reach node with label(node)=goal.*/
          find(start, empty)
        end
      end
   end
 in
       ...      /* program using search function */
 end
end
```

Fig. 1. Program with search function directed by data algebra parameter.

where substitution includes renaming of bound variables as usual. (It is not too hard to prove that the typing rules of SOL guarantee that $[M/x][\tau/t]N$ is well-typed.) Since **abstype** binds variables, we also have the renaming equivalence

$$\textbf{abstype } t \textbf{ with } x{:}\sigma \textbf{ is } M \textbf{ in } N = \textbf{abstype } s \textbf{ with } y{:}\ [s/t]\sigma \textbf{ is } M \textbf{ in } [y/x][s/t]N,$$

provided s is not free in σ, and neither s nor y is free in N.

Existential types are closely related to infinite sums. We can see the relationship by reviewing the categorical definition of infinite sum [1, 27, 43]. The general definition of sum includes sum types (corresponding to sum objects in categories) and sum functions (corresponding to sum arrows). Given a collection S of types, the sum type $\sum S$ has the property that for each $s \in S$ there is an injection function **inj** s from s to $\sum S$. Furthermore, given any fixed type r and family $F = \{f_s\}$ of functions indexed by S, with $f_s\colon s \to r$, there is a unique sum function $\sum F{:}\sum S \to r$ with the property that

$$\sum F(\textbf{inj } s\, x) = f_s x.$$

Uniqueness of sums means that if $\sum F(\mathbf{inj}\ s\,x) = g_s$ for all $s \in S$, then $\sum F = \sum G$.

The correspondence with sums is similar to the correspondence between polymorphism and products. It will be easier to see how **abstype** gives us sums if, for any term, M with $\mathrm{Type}_A(M) = \sigma \to \rho$ and t not free in ρ, we adopt the abbreviation

$$\sum t.M ::= \lambda z{:}(\exists t.\sigma).\ \mathbf{abstype}\ t\ \mathbf{with}\ x{:}\sigma\ \mathbf{is}\ z\ \mathbf{in}\ Mx$$

where x and z are fresh variables. To see how $\sum t.M$ is a sum function, recall from the discussion of $\forall$-types that a type expression σ and type variable t define a collection of types, namely, the collection of substitution instances $[\tau/t]\sigma$. Similarly, a term M and type variable t define a family F of substitution instances $[\tau/t]M$. As before, we index elements of F by types in S by associating $[\tau/t]M$ with $[\tau/t]\sigma$. If M has type $\sigma \to \rho$ for some ρ that does not have t free, then F is a family of functions from types in S to a fixed ρ. We may now regard the type $\exists t.\sigma$ as the sum type $\sum S$, the term $\sum t.M$ as the sum element $\sum F$, and $\lambda y{:}s.(\mathbf{pack}\ s\,y\ \mathbf{to}\ \exists t.\sigma)$ as the injection function **inj** s. The sum axiom holds in SOL, since

$$\begin{aligned}&(\textstyle\sum t.M)(\mathbf{pack}\ \tau y\ \mathbf{to}\ \exists t.\sigma)\\ &\quad ::= [\lambda z{:}\exists t\sigma.\ \mathbf{abstype}\ t\ \mathbf{with}\ x{:}\sigma\ \mathbf{is}\ z\ \mathbf{in}\ Mx)]\mathbf{pack}\ \tau y\ \mathbf{to}\ \exists t.\sigma\\ &\qquad \Rrightarrow \mathbf{abstype}\ t\ \mathbf{with}\ x{:}\sigma\ \mathbf{is}\ (\mathbf{pack}\ \tau y\ \mathbf{to}\ \exists t.\sigma)\ \mathbf{in}\ Mx)\\ &\qquad \Rrightarrow [\tau/t]My.\end{aligned}$$

It is interesting to compare **abstype** with **case** since $\vee$-types with **inleft**, **inright**, and **case** correspond to finite categorical sums. Essentially, **abstype** is an infinitary version of **case**.

As an aside, we note that the binding construct **abstype** may be replaced by a constant **sum**. This treatment of **abstype** points out that the binding aspects of **abstype** are essentially λ binding. If N is a term with type $\sigma \to \rho$, and t is not free in ρ, then both $\lambda t.N$ and $\sum t.N$ are well typed. Therefore, it suffices to have a function **sum** $\exists t.\sigma\ \rho$ that maps $\lambda t.N{:}\forall t.[\sigma \to \rho]$ to $\sum t.N{:}(\exists t.\sigma) \to \rho$. Essentially, this means **sum** $\exists t.\sigma\ \rho$ must satisfy the equation

$$(\mathbf{sum}\ \exists t.\sigma\ \rho x)(\mathbf{pack}\ \tau y\ \mathbf{to}\ \exists t.\sigma) = x\{\tau\}y$$

for any x, y of the appropriate types. In the version of SOL with **sum** as basic, we use this equation, read from left to right, as the defining reduction rule for **sum**. Given **sum**, both $\sum$ and **abstype** may be defined by

$$\begin{aligned}&\textstyle\sum t.M ::= \mathbf{sum}\ \exists t.\sigma\ \rho\ \lambda t.M,\\ &\quad \mathbf{abstype}\ t\ \mathbf{with}\ x{:}\sigma\ \mathbf{is}\ N\ \mathbf{in}\ M ::= (\textstyle\sum t.\lambda x{:}\sigma.M)N.\end{aligned}$$

The reduction rules for $\sum$ and **abstype** follow the reduction rules for **sum**. From a theoretical point of view, it would probably be simpler to define SOL using **sum** instead of $\sum$ or **abstype**, since this reduces the number of binding operators in the language. However, for expository purposes, it makes sense to take **abstype** as primitive, since this makes the connection with data abstraction more readily apparent. The difference is really inessential since any one of $\sum$, **abstype**, and **sum** may be used to define the other two (using other constructs of the language).

3.7 Properties of SOL

Two important typing properties of SOL can be proved as theorems. The first theorem may be interpreted as saying that SOL typing prevents run-time type errors. Technically, the Type Preservation Theorem says that if we begin with a well-typed term (expression or program) and evaluate or "run" it using the reduction rules given, then at every step of the way we have a well-typed term of the same type. This implies that if a term M contains a function f of type $int \to int$, say, then evaluation will never produce an expression containing f applied to a Boolean argument, since this would not be well typed. Therefore, although evaluating a term M may rearrange it dramatically, evaluation will only produce terms in which f is applied to integer arguments.

TYPE PRESERVATION THEOREM. *Let M be a term of SOL with $Type_A(M) = \sigma$. If $M \Rightarrow\!\!\!> N$, then $Type_A(N) = \sigma$.*

A similar theorem for a simpler language without polymorphism appears in [12], where it is called the Subject Reduction Theorem. The proof uses induction on reduction paths and is essentially straightforward.

Another important theorem is a formal statement of the fact that type information may be discarded at run time. More specifically, it is clear from the language definition that SOL type checking can be done efficiently without executing programs (i.e., without referring to the operational semantics of the language). The Static Typing Theorem shows that once the type of a term has been calculated, the term may be evaluated (or "run") without further examining types. This is stated formally by comparing the operational reduction rules given in the language definition with a similar set of reduction rules on untyped terms.

Given a term M, we let $Erase(M)$ denote the untyped expression produced by erasing all type information from M. The function *Erase* has the simple inductive definition

$$
\begin{aligned}
&Erase(x) = x\\
&Erase(c) = c\\
&Erase(\lambda x{:}\sigma.M) = \lambda x.Erase(M)\\
&Erase(MN) = Erase(M)Erase(N)\\
&Erase(\langle M, N\rangle) = \langle Erase(M), Erase(N)\rangle\\
&Erase(\textbf{fst } M) = \textbf{fst } Erase(M)\\
&Erase(\textbf{snd } M) = \textbf{snd } Erase(M)\\
&Erase(\textbf{inleft } M \textbf{ to } \sigma \vee \tau) = \textbf{inleft } Erase(M)\\
&Erase(\textbf{inright } M \textbf{ to } \sigma \vee \tau) = \textbf{inright } Erase(M)\\
&Erase(\textbf{case } M \textbf{ left } x{:}\sigma.N \textbf{ right } y{:}\tau.P \textbf{ end})\\
&\quad = \textbf{case } Erase(M) \textbf{ left } x.Erase(N) \textbf{ right } y.Erase(P) \textbf{ end}\\
&Erase(\lambda t.M) = Erase(M)\\
&Erase(M\{\tau\}) = Erase(M)\\
&Erase(\textbf{pack } \rho M \textbf{ to } \exists t.\sigma) = Erase(M)\\
&Erase(\textbf{abstype } t \textbf{ with } x{:}\sigma \textbf{ is } M \textbf{ in } N) = \textbf{let } x = Erase(M) \textbf{ in } Erase(N)
\end{aligned}
$$

We define the untyped reduction relation $\Rightarrow^E$ by erasing types from terms in each reduction rule, for example,

$$(\lambda x.M)N \Rightarrow^E [N/x]M.$$

Let $\Rightarrow\!\!\gg^E$ be the congruent and transitive closure of $\Rightarrow^E$. Then we have the following theorem:

STATIC TYPING THEOREM. *Let M, N be two terms of SOL with $Type_A(M) = Type_A(N)$. Then $M \Rightarrow\!\!\gg N$ iff $Erase(M) \Rightarrow\!\!\gg^E Erase(N)$.*

Since the theorem shows that two sets of reduction rules have essentially equivalent results, it follows that programs may be executed using any interpreter or compiler on the basis of untyped reduction rules. Like the Type Preservation Theorem, the proof uses induction on the length of reduction paths and is essentially straightforward. Although easily proved, these theorems are important since they confirm our expectations about the relationship between typing and program execution.

It is worth mentioning the relationship between the Static Typing Theorem and the seemingly contradictory "folk theorem" that tagged sums (in SOL notation, $\sigma \vee \tau$ types) require run-time type information. Both are correct but based on different notions of "untyped" evaluation. The Static Typing Theorem says that if a term M is well typed, then M can be evaluated using untyped reduction $\Rightarrow\!\!\gg^E$. However, notice that *Erase* does not remove **inleft** and **inright**, only the type designations on these constructs. Therefore, in evaluating a **case** statement

case M **left** $\cdots$ **right** $\cdots$ **end**

the untyped evaluation rules can depend on whether M is of the form **inleft** M_1 or **inright** M_1. In the "folk theorem," this is considered type information, hence the apparent contradiction.

The SOL reduction rules have several other significant properties. For example, the reduction rules have the Church–Rosser property [22, 61].

CHURCH–ROSSER THEOREM. *Suppose M is a term of SOL which reduces to M_1 and M_2. Then there is a term N such that both M_1 and M_2 reduce to N.*

In contrast to the untyped lambda calculus, no term of SOL can be reduced infinitely many times.

STRONG NORMALIZATION THEOREM. *There are no infinite reduction sequences.*

The strong normalization theorem was first proved by Girard [22]. In light of the strong normalization theorem, the Church–Rosser theorem follows from a simple check of the weak Church–Rosser property (see Proposition 3.1.25 of [2]). A consequence of Church–Rosser and Strong Normalization is that all maximal reduction sequences (from a given term) end in the same normal form.[3] As proved in Girard's thesis [22] and discussed in [20] and [59], the proof of the strong normalization theorem cannot be carried out formally in either Peano arithmetic or second-order Peano arithmetic (second-order Peano is also called "analysis"). Furthermore, the class of number-theoretic functions that are

[3] A normal form M is a term that cannot be reduced. Our use of the phrase *strong normalization* follows [2]. Some authors use strong normalization for the property that all maximal reduction sequences from a given term end in the same normal form.

representable in pure SOL without base types are precisely the recursive functions that may be proved total in second-order Peano arithmetic [22, 68]. These and related results are discussed in [20] at greater length.

3.8 Alternative Views of Abstract Data Type Declarations

As noted in the introduction, several language design efforts are similar in spirit to ours. The language SOL is based on Reynolds' polymorphic lambda calculus [62] and Girard's proof-theoretic language [22]. Some similar languages are Pebble [7], Kernel Russel, KR, [28], ML with modules as proposed by MacQueen [44], and Martin-Löf's constructive type theory [46]. We compare **abstype** in SOL with an early proposal of Reynolds [62] and, briefly, with the constructs of Pebble and KR.

In defining the polymorphic lambda calculus, Reynolds proposed a kind of **abstype** declaration based on λ-binding [62]. As Reynolds notes, the expression

abstype t **with** $x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n$ **is** M **in** N

has the same meaning as

$$(\lambda t.\lambda x_1\colon \sigma_1 \ldots \lambda x_n\colon \sigma_n.N)\{\tau\}M_1, \ldots, M_n$$

If M is of the form **pack** $\tau M_1 \ldots M_n$ **to** $\exists t.\sigma$. However, **abstype** should not be considered an abbreviation for this kind of expression for two reasons. First, it is not clear what to do if M is not of the form **pack** $\tau M_1 \ldots M_n$ **to** $\exists t.\sigma$. Therefore, we can only simulate a restricted version of SOL by this means; much flexibility is lost. A lesser drawback of using λ to define **abstype** in this way is that the expression

$$(\lambda t.\lambda\langle x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n\rangle.N)\{\tau\}M_1 \ldots M_n$$

is well typed in cases in which the corresponding **abstype** expression fails to satisfy (AB.3). As noted in Section 2, rule (AB.3) keeps the "abstract" type from being exported outside the scope of a declaration. However, other justifications for (AB.3) discussed in Section 2 do not apply here, since Reynolds' suggestion cannot be used to construct conditional data algebra expressions, for example.

While the above definition of **abstype** using λ has some drawbacks, a more suitable definition using λ is described in the final section of the later paper [64].

Pebble and KR take a view of data algebras that appears to differ from SOL. An intuitively appealing view of **pack** $\tau M_1 \cdots M_n$ is simply as a record whose first component is a type. This seems to lead one to introduce a "type of types," a path followed by [7] and [28]. We would expect a product type for **pack** $\cdots$ to be something like

$$Type \wedge \sigma_1 \wedge \cdots \wedge \sigma_n.$$

However, this does not link the value of the first component to the types of the remaining components. To solve this problem, Pebble and KR associate abstract data types with "dependent product" types of the form

$$t\colon \mathrm{Type} \wedge \sigma_1 \wedge \cdots \wedge \sigma_n,$$

where t is considered bound in $\sigma_1 \wedge \cdots \wedge \sigma_n$.

Since Pebble does not supply projection functions for dependent products, the dependent product of Pebble actually seems to be a sum (in the sense of category theory), like SOL ∃-types. KR dependent products do have something that looks like a projection function: If A is a data algebra, then $Carrier(A)$ is a type expression of KR. However, since $Carrier(\textbf{pack } \tau M \textbf{ to } \exists t.\sigma)$ is not considered equal to τ, it seems that KR dependent products are not truly products. Perhaps further analysis will show that KR dependent products are also sums and closer to SOL existential types than might appear at first glance.

As pointed out in [30], there are actually two reasonable notions of sum type, "weak" and "strong" sums. The SOL existential type is a typical example of weak sums, whereas strong sums appear as the $\sum$-types of Martin Löf's type theory [46]. The main difference lies in rule (AB.3), which holds for weak sums, but not for strong. Thus, while Martin-Löf's product types over universes give a form of polymorphism that is similar to SOL polymorphism, Martin-Löf's sum types differ from our existential types. For this reason, the languages are actually quite different. In addition, the restrictions imposed by universes simplify the semantics of Martin-Löf's language, at the cost of a slightly more complicated syntax. (Some relatively natural programming examples, such as the Sieve of Eratosthenes program given in Section 5.2 of this paper, are prohibited by the universe restrictions of Martin-Löf type theory.) For further discussion of sum and product types over universes, the reader is referred to [9], [10], [31], [45], [46], [49], and [54].

4. FORMULAS AS TYPES

4.1 Introduction

The language SOL exhibits an analogy between logical formulas and types that has been used extensively in proof theory [12, 13, 22, 30, 35, 38, 39, 46, 67, 69]. The programming significance of the analogy has been stressed by Martin-Löf [46]. We review the basic idea using propositional logic and then discuss quantification briefly. In addition to giving some intuition into the connection between computer science and constructive logic, the formulas-as-types analogy also suggests other languages with existential types. One such language, involving specifications as types, is discussed briefly at the end of this section. In general, our analysis of **abstype** suggests that any constructive proof rules for existential formulas provide data type declarations. For this reason, the formulas-as-types languages provide a general framework for studying many aspects of data abstraction.

4.2 Propositional Logic

Implicational propositional logic uses formulas that contain only propositional variables and $\rightarrow$, implication. The formulas of implicational propositional logic are defined by the grammar

$$\sigma ::= t \mid \sigma \rightarrow \tau,$$

where we understand that t is a propositional variable. We are concerned with an *intuitionistic* interpretation of formulas, so it is best not to think of formulas

as simply being true or false whenever we assign truth values to each variable. While various forms of intuitionistic semantics have been developed [19, 33, 34, 70], we will not go into this topic. Instead, we will characterize intuitionistic validity by means of a proof system.

Natural deduction is a style of proof system that is intended to mimic the common blackboard-style argument

> Assume σ.
> By $\cdots$ we conclude τ.
> Therefore $\sigma \to \tau$.

We make an assumption in the first line of this argument. In the second line, this assumption is combined with other reasoning to derive τ. At this point, we have proved τ, but the proof depends on the assumption of σ. In the third step, we observe that since σ leads to a proof of τ, the implication $\sigma \to \tau$ follows. Since the proof of $\sigma \to \tau$ is sound without proviso, we have "discharged" the assumption of σ in proceeding from τ to $\sigma \to \tau$. In a natural deduction proof, each proposition may depend on one or more assumptions. A proposition is considered *proved* only when all assumptions have been discharged.

The natural deduction proof system for implicational propositional logic consists of three rules, given below. For technical reasons, we use labeled assumptions. (This is useful from a proof-theoretic point of view as a means of distinguishing between different assumptions of the same formula.) Let V be a set, intended to be the set of labels, and let A be a mapping from labels to formulas. We will use the notation $\text{Conseq}_A(M) = \sigma$ to mean that M is a proof with consequence σ, given the association A of labels to assumptions. Proofs and their consequences are defined as follows:

$$\text{Conseq}_A(x) = A(x)$$

$$\frac{\text{Conseq}_A(M) = \sigma \to \tau,\ \text{Conseq}_A(N) = \sigma}{\text{Conseq}_A(MN) = \tau},$$

$$\frac{\text{Conseq}_{A[x:\sigma]}(M) = \tau}{\text{Conseq}_A(\lambda x{:}\sigma.M) = \sigma \to \tau}.$$

The set $\text{Assume}(M)$ of undischarged assumptions of M is defined by

$$\text{Assume}(x) = \{x\}$$
$$\text{Assume}(MN) = \text{Assume}(M) \cup \text{Assume}(N)$$
$$\text{Assume}(\lambda x{:}\sigma.M) = \text{Assume}(M) - \{x\}$$

In English, we may summarize these two definitions as follows:

A label x is a proof of $A(x)$ with assumption labeled x.

If M is a proof of $\sigma \to \tau$ and N is a proof of σ, then MN is a proof of τ (depending on all assumptions used in either proof).

If M is a proof of τ with assumption σ labeled x, then $\lambda x{:}\sigma.M$ is a proof of $\sigma \to \tau$ with the assumption x discharged.

A formula σ is *intuitionistically provable* if there is a proof M with $\text{Conseq}_A(M) = \sigma$ and $\text{Assume}(M) = \emptyset$. (It is easy to show that if $\text{Assume}(M) = \emptyset$, then

$\text{Conseq}_A(M)$ does not depend on A.) Even when $\rightarrow$ is the only propositional connective, there are classical tautologies that are not intuitionistically provable. For example, it is easy to check that the formula $((s \rightarrow t) \rightarrow s) \rightarrow s$ is a classical tautology just by trying all possible assignments of *true* and *false* to s and t. However, this formula is not intuitionistically provable.

Of course, we have just defined the typed lambda calculus: The terms of typed lambda calculus are precisely the proofs defined above and their types are the formulas given. In fact, Conseq_A and Type_A are precisely the same function, and $\text{Assume}(M)$ is precisely the set of free variables of M. The similarity between natural deduction proofs and terms extends to the other connectives and quantifiers. The proof rules for $\wedge$, $\vee$, $\forall$, and $\exists$ are precisely the formation rules given earlier for terms of these types.

One interesting feature of the proof rule for $\vee$ of [60] is that it is the discriminating case statement of CLU [42], rather than the problematic **outleft** and **outright** functions of ML [23]. The "out" functions of ML are undesirable since they rely on run-time exceptions (cf. [41], p. 569). Specifically, if $x:\tau$ in ML, then (**inright** x): $\sigma \vee \tau$ and **outleft**(**inright** x): σ. However, we cannot actually compute a value of type σ from $x:\tau$, so this is not semantically sensible. The ML solution to this problem is to raise a run-time exception when **outleft**(**inright** x) is evaluated, which introduces a form of run-time type checking. Since the $\vee$ rule leads us directly to a case statement that requires no run-time type checking, it seems that the formulas-as-types analogy may be a useful guide in designing programming languages.

4.3 Universal and Existensial Quantifiers

The intuitionistic proof rules for universal and existential types are repeated below for emphasis. It is a worthwhile exercise for the reader to become convinced that these make logical sense.

$$\frac{\text{Conseq}_A(M) = \forall t.\sigma}{\text{Conseq}_A(M\{\tau\}) = [\tau/t]\sigma},$$

$$\frac{\text{Conseq}_A(M) = \tau}{\text{Conseq}_A(\lambda t.M) = \forall t.\tau} \quad t \text{ not free in } A(x) \text{ for } x \text{ free in } M,$$

$$\frac{\text{Conseq}_A(M) = [\tau/t]\sigma}{\text{Conseq}_A(\textbf{pack } \tau M \textbf{ to } \exists t.\sigma) = \exists t.\sigma},$$

$$\frac{\text{Conseq}_A(M) = \exists t.\sigma,\ \text{Conseq}_{A[x:\sigma]}(N) = \rho}{\text{Conseq}_A(\textbf{abstype } s \textbf{ with } x:\sigma \textbf{ is } M \textbf{ in } N) = \rho},$$

provided t is not free in ρ or the type $A(y)$ of any free $y \neq x$ occurring in N.

The rules for $\forall$ are the usual universal instantiation and generalization. The third is an existential generalization rule, and the fourth a form of existential instantiation. Except for the explicit proof notation chosen to suggest programming language syntax, these proof rules are exactly those found in [60]. Although a full discussion would take us beyond the scope of this paper, it is worth remarking that reduction rules may also be derived using the formulas-as-types analogy: The reduction rules of SOL are precisely the proof-simplification rules given in [61].

4.4 Other Languages With Existential Types

The formulas-as-types analogy can be applied to other natural deduction proof systems. Two particularly relevant logics are the second-order logics of [60], Chapter V. The simpler of these amounts to adding first-order terms to the second-order logic of SOL. In this language, types are formulas that describe the behavior of terms.

In an ideal programming language, we would like to use specifications to describe abstract data types. The ideal or "intended" type of stack is the specification

$$\forall t.\ \exists s.\ \exists \text{empty}: s.\ \exists \text{push}: t \wedge s \to s.$$
$$\exists \text{pop}: s \to t \wedge s: \forall x{:}s.\ \forall y{:}t.\ \{\text{pop}(\text{push}(x, y)) = \langle x, y \rangle\},$$

or, perhaps more properly, a similar specification with an induction axiom:

$$\forall t.\ \exists s.\ \exists \text{empty}: s.\ \exists \text{push}: t \wedge s \to s.\ \exists \text{pop}: s \to t \wedge s.$$
$$\forall x{:}s.\forall y{:}t.\ \{\text{pop}(\text{push}(x, y)) = \langle x, y \rangle \wedge \textit{induction axiom}\}.$$

Both specifications are, in fact, type expressions in the language based on first- and second-order logic. We expect the meaning of each type expression to correspond to a class of algebras satisfying the specification (see, e.g., [24] for a discussion of universal algebra). However, the language based on first- and second-order logic is cumbersome for programming since constructing an element of one of these existential types involves proving that an implementation meets its specification. Some interesting research into providing environments for programming with specifications as types is provided in [8] and [9]. Induction rules, used for proofs by "data type induction" [25], are easily included in specifications since induction is expressible in second-order logic.

A richer "ramified second-order" system in Chapter V of [60] includes λ-abstraction in the language of types. Via formulas-as-types, this leads to the richer languages of [47] and [51].

5. MORE PROGRAMMING EXAMPLES

5.1 Universal and Existential Parameterization

Some useful constructions involving abstract data types are to pass representations as parameters, parameterize the data types themselves, and return implementations as results of procedures. In SOL, we can distinguish between two kinds of type parameterization. Suppose M uses operations $x: \sigma$ on type t, and t is not free in the type of any other free variable of M. Then the terms

$$M_1 = \lambda t.\lambda x{:}\sigma.\ M$$
$$M_2 = \sum t.\lambda x{:}\sigma.\ M$$

are both parameterized by a type and operations. However, there are significant differences between these two terms. To begin with, M_1 is well typed even if t appears free in the type of M, where M_2 is not. Furthermore, the two terms have different types. If the type of M is ρ, then their types are

$$M_1: \forall t(\sigma \to \rho)$$

and

$$M_2\colon (\exists t.\sigma) \to \rho.$$

We will say that M_1 is *universally parameterized* and M_2 is *existentially parameterized.*

Generic packages are universally parameterized data algebras. For example, given any type t with operations

$$\begin{aligned} &plus\colon t \wedge t \to t \\ &times\colon t \wedge t \to t, \end{aligned}$$

we can write a data algebra *t_matrix* implementing matrix operations over t. Four operations we might choose to include are

$$\begin{aligned} &create\colon t \wedge \cdots \wedge t \to \text{mat} \\ &mplus\colon \text{mat} \wedge \text{mat} \to \text{mat}, \\ &mtimes\colon \text{mat} \wedge \text{mat} \to \text{mat}, \\ &det\colon \text{mat} \to t. \end{aligned}$$

If *mbody* is an expression of the form

$$\begin{aligned} mbody ::= \textbf{pack } \tau M_1 \cdots M_4 \textbf{ to } \exists s[&(t \wedge \cdots \wedge t \to s) \\ &\wedge (s \wedge s \to s) \wedge (s \wedge s \to s) \wedge (s \to t)] \end{aligned}$$

implementing *create*, *mplus*, *mtimes*, and *det* using *plus* and *times*, then

$$matrix ::= \lambda t.\ \lambda plus\colon t \wedge t \to t.\ \lambda times\colon t \wedge t \to t.mbody$$

is a universally parameterized data algebra. The type of *matrix* is

$$\begin{aligned} \forall t.(t \wedge t \to t) \to (t \wedge t \to t) \to \exists s[&(t \wedge \cdots \wedge \\ &t \to s) \wedge (s \wedge s \to s) \wedge (s \wedge s \to s) \wedge (s \to t)]. \end{aligned}$$

Note that *mbody* could not be existentially parameterized by t since t appears free in the type of *mbody*.

Functions from data algebras to data algebras are existentially parameterized. One simple manipulation of data algebras is to remove operations from the signature. For example, a doubly ended queue, or dequeue, has two insert and two remove operations. The type of an implementation *dq* of dequeues with *empty*, *insert1*, *insert2*, *remove1*, and *remove2*, is

$$\begin{aligned} dq\text{-type} ::= \forall t.\exists d.[&d \wedge (t \wedge d \to d) \wedge \\ &(t \wedge d \to d) \wedge (d \to t \wedge d) \wedge (d \to t \wedge d)] \end{aligned}$$

A function that converts dequeue implementations to queue implementations is a simple example of an existentially parameterized structure. Given *dq*, we can implement queues using the form

$$\begin{aligned} Q(x, t) ::= &\textbf{abstype } d \textbf{ with } \text{empty: } \ldots, \text{insert1: } \ldots, \text{insert2: } \ldots, \\ &\text{remove1: } \ldots, \text{remove2: } \cdots \\ &\quad \textbf{is } x\{t\} \\ &\quad \textbf{in pack } d \text{ empty insert1 remove2 } \textbf{to } \exists t.\sigma \end{aligned}$$

with *dq* substituted for x. Thus the term

$$dq\text{-to-}q ::= \lambda x\colon dq\text{-type}.\lambda t.Q(x, t)$$

with type

$$dq\text{-type} \rightarrow \forall t.\ \exists s.[s \wedge (t \wedge s \rightarrow s) \wedge (s \rightarrow t \wedge s)]$$

is a function from data algebras to data algebras. Suppose that *queue* is the data algebra produced by applying *dq-to-q* to *dq*. Since the type of queue is a closed type expression, the fact that queue uses the same representation type as *dq* seems effectively hidden. Generally, universal parameterization may be used to effect some kind of sharing of types, whereas existential parameterization obscures the identity of representations. (See [45], which was written later, for related discussion.)

Some other useful transformations on data algebras are the analogs of the theory building operations *combine*, *enrich*, and *derive* of CLEAR [5, 6]. Although a general *combine* operation as in CLEAR, for example, cannot be written in SOL because of type constraints, we can write a combine operation for any pair of existential types. For example, we can write a procedure to combine data algebras of types $\exists s.\sigma$ and $\exists t.\rho$ into a single data algebra. The type of this function

$\text{Combine}_1 = \lambda x{:}\exists t.\sigma\ \lambda y{:}\exists t.\rho.$

abstype s **with** $z{:}\sigma$ **is** x **in**
abstype t **with** $w{:}\rho$ **is** y **in**
pack s [**pack** $t\langle z, w\rangle$ **to** $\exists t(\sigma \wedge \rho)$] **to** $\exists s \exists t(\sigma \wedge \rho)$

is

$$\text{Combine}_1{:}\ \exists s.\sigma \rightarrow \exists t.\rho \rightarrow \exists s \exists t(\sigma \wedge \rho).$$

For universally parameterized data algebras of types $\forall r \exists s.\sigma$ and $\forall r \exists t.\rho$, we can write combine so that in the combined data algebra, the type parameter will be shared. The combine function with sharing

$\text{Combine}_2 = \lambda x{:}\forall r \exists s.\sigma\ \lambda y{:}\forall r \exists t.\rho.$

$\lambda r.$**abstype** s **with** $z{:}\sigma$ **is** $x\{r\}$ **in**
abstype t **with** $w{:}\rho$ **is** $y\{r\}$ **in**
pack s [**pack** $t\langle z, w\rangle$ **to** $\exists t(\sigma \wedge \rho)$] **to** $\exists s \exists t(\sigma \wedge \rho)$

has type

$$\text{Combine}_2{:}\ \forall r \exists s.\sigma \rightarrow \forall r \exists t.\rho \rightarrow \forall r \exists s \exists t(\sigma \wedge \rho).$$

A similar, but slightly more complicated, combine function can be written for the case in which the two parameters are both universally parameterized by a type and several operations on the type. For example, a polymorphic matrix package could be combined with a polymorphic polynomial package to give a combined package parameterized by a type t and two binary operations *plus* and *times* providing both matrices and polynomial over t. Furthermore, the combine function could be written to enrich the combined package by adding a function that finds the characteristic polynomial of a matrix.

5.2 Data Structures Using Existential Types

Throughout this paper, we have viewed data algebras as implementations of abstract data types. An alternative view is that data algebras are simply records tagged with types. This view leads us to consider using data algebras as parts of data structures. In many cases, these data structures do not seem directly related

to any kind of abstract data type. The following example uses existentially typed data structures to represent streams.

Intuitively, streams are infinite lists. In an applicative language, it is convenient to think of a stream as a kind of "process" that has a set of possible internal states and a specific value associated with each state. Since the process implements a list, there is a designated initial state and a deterministic state transition function. Therefore, a stream consists of a type s (of states) with a designated individual (start state) of type s, a next-state function of type $s \to s$, and a value function of type $s \to t$, for some t. An integer stream, for example, will have a value function of type $s \to int$, and so the type of integer streams will be $\exists s[s \wedge (s \to s) \wedge (s \to int)]$.

The Sieve of Eratosthenes can be used to produce an integer stream enumerating all prime numbers. This stream is constructed using a sift operation on streams. Given an integer stream s_1, Sift(s_1) is a stream of integers that are not divisible by the first value of s_1. If Num is the stream 2, 3, . . . , then the sequence formed by taking the first value of each stream

$$\text{Num, Sift(Num), Sift(Sift(Num)), } \ldots$$

will be the sequence of all primes.

With streams represented using existential types, *Sift* may be written as the following function over existential types.

```
Sift =
  λ stream: ∃s[s ∧ (s → s) ∧ (s → int)].
    abstype s with start: s, next: s → s, value: s → int is stream
      in let n = value(start)
        in letrec f = λ state: s.
            if n divides value(state) then f(next(state))
                                      else state
          in
            pack s f(start) λx: s.f(next(x)) value to ∃s[s ∧ (s → s) ∧ (s → int)]
          end
        end
    end
```

Sieve will be the stream with states represented by integer streams, start state the stream of all integers greater than 1, and *Sift* the successor function on states. The value associated with each *Sieve* state is the first value of the integer stream, so that the values of *Sieve* enumerate all primes.

```
Sieve =
  abstype s with start: s, next: s → s, value: s → int
    is pack ∃t[t ∧ (t → t) ∧ (t → int)]
        pack int 2 Successor λx: int.x to ∃t[t ∧ (t → t) ∧ (t → int)]
        Sift
        λ state: ∃t[t ∧ (t → t) ∧ (t → int)].
          abstype r with r_start, r_next, r_val is state
            in r_val(r_start)
      to ∃t[t ∧ (t → t) ∧ (t → int)]
```

Expressed in terms of *Sieve*, the ith prime number is

```
abstype s with start: s, next: s → s, value: s → int
  is Sieve
  in value(next^i start),
```

where "next^i start" is the expression next(next(···(next start)···)) with i occurrences of next.

It is worth noticing that Sieve is "circular" in the sense that the representation type $\exists t[t \wedge (t \rightarrow t) \wedge (t \rightarrow int)]$ used to define Sieve is also the type of Sieve itself. For this reason, this example could not have been written in a predicative system like Martin-Löf's intuitionistic type theory [9, 46]. The typing rules of that theory require that elements of one type be composed only of elements of simpler types.

6. CONCLUSION AND DIRECTIONS FOR FURTHER INVESTIGATION

We have used language SOL, a syntactic variant of Girard's system F and an extension of Reynolds' polymorphic lambda calculus [22, 62], to discuss abstract data type declarations in programming languages. SOL is easily defined and has straightforward operational semantics. The language also allows us to decompose abstract data type declarations into two parts: defining a data algebra and binding names to its components. For this reason, SOL allows implementations of abstract data types to be passed as function parameters or returned as results. This makes the language more flexible than many contemporary typed languages, without sacrificing efficient compile-time type checking.

The flexibility of SOL comes about primarily because we treat data algebras as values that have types themselves. The types of data algebras in SOL are existential types, a type motivated by an analogy between programming languages and constructive logic and closely related to infinite sums. We believe that although the design of SOL does not address certain practical objectives, the language demonstrates useful extensions to current programming languages. SOL also seems very useful for studying the mathematical semantics of data type declarations.

One promising research direction is to use SOL to formalize and prove some natural properties of abstract data types. For example, if M and N implement two data algebras with the same observable behavior (see, e.g., [32]), then the meaning of a program using M should correspond appropriately to the meaning of the same program using N. However, SOL is sufficiently complicated that it is not clear how to define "observable behavior." Among other difficulties, data algebras are heterogeneous structures whose operations may be polymorphic or involve existential types. Reynolds, Donahue, and Haynes have examined various related "representation independence" properties of SOL-like languages without existential types [18, 26, 55, 64]. Some of these ideas have been applied to SOL in [52], which was written after the work described here was completed. However, there is still much to be done in this direction.

There are a number of technical questions about SOL that merit further study. The semantics of various fragments of SOL are studied in [3], [4], [18], [26], [47], [51], [62], and [65], but many questions remain. Some open problems are listed in [3], [4], and [51]. In addition, there are a number of questions related to automatic insertion of type information into partially typed expressions of SOL. For example, it would be useful to find an algorithm which, given a term M of the untyped lambda calculus, could determine whether type expressions

and type binding can be added to M to produce a well-typed term of SOL. Some questions of this nature are discussed in [40], [48], and [53].

A general problem in the study of types is a formal characterization of type security. We have given two theorems about typing in SOL: Expressions may be evaluated without considering type information, and the syntactic type of an expression is not affected by reducing the expression to simpler forms. These theorems imply that types may be ignored when evaluating SOL expressions and that SOL type checking is sufficient to prevent run-time type errors. The study of representation independence (mentioned above) leads to another notion of type security, but further research seems necessary to show that SOL programs are "type-safe" in other ways.

One interesting aspect of SOL is that it may be derived from quantified propositional (second-order) logic using the formulas-as-types analogy discussed in Section 4. Our analysis of **abstype** demonstrates that the proof rules for existential formulas in a variety of logical systems all correspond to declaring and using abstract data types. Thus, the formulas-as-types languages provide a general framework for studying abstract data types. In particular, the language derived from first- and second-order logic seems to incorporate specifications into programs in a very natural way. The semantics and programming properties of this language seem worth investigating and relating to other studies of data abstraction based on specification.

APPENDIX. COLLECTED DEFINITION OF SOL

The type expressions of SOL are defined by the following abstract syntax:

$$\sigma ::= t \mid c \mid \sigma \to \tau \mid \sigma \wedge \tau \mid \sigma \vee \tau \mid \forall t.\sigma \mid \exists t.\sigma,$$

where t is any type variable and c is any type constant. (We use two sorts of variables, type variables $r, s, t, \ldots$ and ordinary variables $x, y, z, \ldots$)

A type assignment A is a function from ordinary variables to type expressions. We use $A[x:\sigma]$ to denote the type assignment A_1 with $A_1(y) = A(y)$ for y different from x, and $A_1(x) = \sigma$. The partial functions Type_A, for all type assignments A, and the operational semantics of SOL are defined as follows:

Constants and Variables

$$\text{Type}_A(c^\tau) = \tau \quad \text{for constant } c^\tau \text{ of type } \tau$$
$$\text{Type}_A(x) = A(x)$$

Functions and Application

$$\frac{\text{Type}_{A[x:\sigma]}(M) = \tau}{\text{Type}_A(\lambda x:\sigma.M) = \sigma \to \tau}$$

$$\frac{\text{Type}_A(M) = \sigma \to \tau,\ \text{Type}_A(N) = \sigma}{\text{Type}_A(MN) = \tau}$$

$$\lambda x:\sigma.M = \lambda y:\sigma.[y/x]M, \qquad y \text{ not free in } M$$
$$(\lambda x:\sigma.M)N \Rightarrow [N/x]M,$$

Products

$$\frac{\mathrm{Type}_A(M) = \sigma,\ \mathrm{Type}_A(N) = \tau}{\mathrm{Type}_A(\langle M, N\rangle) = \sigma \wedge \tau}$$

$$\frac{\mathrm{Type}_A(M) = \sigma \wedge \tau}{\mathrm{Type}_A(\mathbf{fst}\ M) = \sigma,\ \mathrm{Type}_A(\mathbf{snd}\ M) = \tau}$$

$$\mathbf{fst}\langle M, N\rangle \Rightarrow M, \qquad \mathbf{snd}\langle M, N\rangle \Rightarrow N$$

Sums

$$\frac{\mathrm{Type}_A(M) = \sigma}{\mathrm{Type}_A(\mathbf{inleft}\ M\ \mathbf{to}\ \sigma \vee \tau) = \sigma \vee \tau,\ \mathrm{Type}_A(\mathbf{inright}\ M\ \mathbf{to}\ \tau \vee \sigma) = \tau \vee \sigma}$$

$$\frac{\mathrm{Type}_A(M) = \sigma \vee \tau,\ \mathrm{Type}_{A[x:\sigma]}(N) = \rho,\ \mathrm{Type}_{A[y:\tau]}(P) = \rho}{\mathrm{Type}_A(\mathbf{case}\ M\ \mathbf{left}\ x{:}\sigma.N\ \mathbf{right}\ y{:}\tau.P\ \mathbf{end}) = \rho}$$

$$\mathbf{case}\ M\ \mathbf{left}\ x{:}\sigma.N\ \mathbf{right}\ y{:}\tau.P\ \mathbf{end} = \mathbf{case}\ M\ \mathbf{left}\ u{:}\sigma.[u/x]N\ \mathbf{right}\ v{:}\tau.[v/y]P\ \mathbf{end},$$

provided u is not free in N and v is not free in P.

$$\mathbf{case}(\mathbf{inleft}\ M\ \mathbf{to}\ \sigma \vee \tau)\ \mathbf{left}\ x{:}\sigma.N\ \mathbf{right}\ y{:}\tau.P\ \mathbf{end} \Rightarrow [M/x]N$$
$$\mathbf{case}(\mathbf{inright}\ M\ \mathbf{to}\ \sigma \vee \tau)\ \mathbf{left}\ x{:}\sigma.N\ \mathbf{right}\ y{:}\tau.P\ \mathbf{end} \Rightarrow [M/y]P$$

Polymorphism

$$\frac{\mathrm{Type}_A(M) = \tau}{\mathrm{Type}_A(\lambda t.M) = \forall t.\tau} \qquad t \text{ not free in } A(x) \text{ for any } x \text{ free in } M$$

$$\frac{\mathrm{Type}_A(M) = \forall t.\sigma}{\mathrm{Type}_A(M\{\tau\}) = [\tau/t]\sigma}$$

$$\lambda t.M = \lambda s.[s/t]M, \qquad s \text{ not free in } \lambda t.M.$$
$$(\lambda t.M)\{\tau\} \Rightarrow [\tau/t]M$$

Abstract Data Types

$$\frac{\mathrm{Type}_A(M) = [\tau/t]\sigma}{\mathrm{Type}_A(\mathbf{pack}\ \tau M\ \mathbf{to}\ \exists t.\sigma) = \exists t.\sigma}$$

$$\frac{\mathrm{Type}_A(M) = \exists t.\sigma,\ \mathrm{Type}_{A[x:\sigma]}(N) = \rho}{\mathrm{Type}_A(\mathbf{abstype}\ s\ \mathbf{with}\ x{:}\sigma\ \mathbf{is}\ M\ \mathbf{in}\ N) = \rho},$$

provided t is not free in ρ or the type $A(y)$ of any free $y \neq x$ occurring in N

$$\mathbf{abstype}\ t\ \mathbf{with}\ x{:}\sigma\ \mathbf{is}\ M\ \mathbf{in}\ N = \mathbf{abstype}\ s\ \mathbf{with}\ y{:}[s/t]\sigma\ \mathbf{is}\ M\ \mathbf{in}\ [y/x][s/t]N,$$

provided s is not free in σ, and neither s nor y is free in N.

$$\mathbf{abstype}\ t\ \mathbf{with}\ x{:}\sigma\ \mathbf{is}\ (\mathbf{pack}\ \tau M\ \mathbf{to}\ \exists t.\sigma)\ \mathbf{in}\ N \Rightarrow [M/x][\tau/t]N.$$

ACKNOWLEDGMENTS

Thanks to John Guttag and Albert Meyer for helpful discussions. Mitchell thanks IBM for a graduate fellowship while at MIT, and Plotkin acknowledges the support of the BP Venture Research Unit.

REFERENCES

1. ARBIB, M. A., AND MANES, E. G. *Arrows, Structures, and Functors: The Categorical Imperative.* Academic Press, Orlando, Fla., 1975.
2. BARENDREGT, H. P. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, The Netherlands, 1984 (revised edition).
3. BRUCE, K. B., AND MEYER, A. A completeness theorem for second-order polymorphic lambda calculus. In *Proceedings of the International Symposium on Semantics of Data Types. Lecture Notes in Computer Science 173*, Springer-Verlag, New York, 1984, pp. 131–144.
4. BRUCE, K. B., MEYER, A. R., AND MITCHELL, J. C. The semantics of second-order lambda calculus. In *Information and Computation* (to be published).
5. BURSTALL, R. M., AND GOGUEN, J. Putting theories together to make specifications. In *Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 1045–1958.
6. BURSTALL, R. M., AND GOGUEN, J. An informal introduction to specification using CLEAR. In *The Correctness Problem in Computer Science*, Boyer and Moore, Eds. Academic Press, Orlando, Fla., 1981, pp. 185–213.
7. BURSTALL, R., AND LAMPSON, B. A kernel language for abstract data types and modules. In *Proceedings of International Symposium on Semantics of Data Types. Lecture Notes in Computer Science 173*, Springer-Verlag, New York, 1984, pp. 1–50.
8. CONSTABLE, R. L. Programs and types. In *21st IEEE Symposium on Foundations of Computer Science* (Syracuse, N.Y., Oct. 1980). IEEE, New York, 1980, pp. 118–128.
9. CONSTABLE, R. L., ET AL. *Implementing Mathematics With The Nuprl Proof Development System.* Graduate Texts in Mathematics, vol. 37, Prentice-Hall, Englewood Cliffs, N.J., 1986.
10. COQUAND, T. An analysis of Girard's paradox. In *Proceedings of the IEEE Symposium on Logic in Computer Science* (June 1986). IEEE, New York, 1986, pp. 227–236.
11. COQUAND, T., AND HUET, G. The calculus of constructions. *Inf. Comput. 76*, 2/3 (Feb./Mar. 1988), 95–120.
12. CURRY, H. B., AND FEYS, R. *Combinatory Logic I.* North-Holland, Amsterdam, 1958.
13. DEBRUIJN, N. G. A survey of the project Automath. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, Orlando, Fla., 1980, pp. 579–607.
14. DEMERS, A. J., AND DONAHUE, J. E. Data types, parameters and type checking. In *7th ACM Symposium on Principles of Programming Languages* (Las Vegas, Nev., Jan. 28–30, 1980). ACM, New York, 1980, pp. 12–23.
15. DEMERS, A. J., AND DONAHUE, J. E. 'Type-completeness' as a language principle. In *7th ACM Symposium on Principles of Programming Languages* (Las Vegas, Nev., Jan. 28–30, 1980). ACM, New York, 1980, pp. 234–244.
16. DEMERS, A. J., DONAHUE, J. E., AND SKINNER, G. Data types as values: polymorphism, type-checking, encapsulation. In *5th ACM Symposium on Principles of Programming Languages* (Tucson, Ariz., Jan. 23–25, 1978). ACM, New York, 1978, pp. 23–30.
17. U.S. DEPARTMENT OF DEFENSE *Reference Manual for the Ada Programming Language.* GPO 008-000-00354-8, 1980.
18. DONAHUE, J. On the semantics of data type. *SIAM J. Comput. 8* (1979), 546–560.
19. FITTING, M. C. *Intuitionistic Logic, Model Theory and Forcing.* North-Holland, Amsterdam, 1969.
20. FORTUNE, S., LEIVANT, D., AND O'DONNELL, M. The expressiveness of simple and second order type structures. *J. ACM 30*, 1 (1983), 151–185.
21. GIRARD, J.-Y. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *2nd Scandinavian Logic Symposium*, J. E. Fenstad, Ed. North-Holland, Amsterdam, 1971, pp. 63–92.
22. GIRARD, J.-Y. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These D'Etat, Univ. Paris VII, Paris, 1972.
23. GORDON, M. J., MILNER, R., AND WADSWORTH, C. P. *Edinburgh Lecture Notes in Computer Science 78*, Springer-Verlag, New York, 1979.
24. GRÄTZER G. *Universal Algebra.* Van Nostrand, New York, 1968.
25. GUTTAG, J. V., HOROWITZ, E., AND MUSSER, D. R. Abstract data types and software validation. *Commun. ACM 21*, 12 (Dec. 1978), 1048–1064.

26. HAYNES, C. T. A theory of data type representation independence. In *Proceedings of International Symposium on Semantics of Data Types. Lecture Notes in Computer Science 173*, Springer-Verlag, New York, 1984, pp. 157–176.
27. HERRLICH, H., AND STRECKER, G. E. *Category Theory.* Allyn and Bacon, Newton, Mass., 1973.
28. HOOK, J. G. Understanding Russell—A first attempt. In *Proceedings of International Symposium on Semantics of Data Types. Lecture Notes in Computer Science 173*, Springer-Verlag, New York, 1984, pp. 69–85.
29. HOOK, J., AND HOWE, D. Impredicative strong existential equivalent to type:type. Tech. Rep. TR 86-760, Cornell Univ., Ithaca, N.Y., 1986.
30. HOWARD, W. The formulas-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism.* Academic Press, Orlando, Fla., 1980, pp. 479–490.
31. HOWE, D. J. The computational behavior of Girard's paradox. In *IEEE Symposium on Logic in Computer Science* (June 1987). IEEE, New York, 1987, pp. 205–214.
32. KAPUR, D. Towards a theory for abstract data types. Tech. Rep. MIT/LCS/TM-237, MIT, Cambridge, Mass., 1980.
33. KLEENE, S. C. Realizability: A retrospective survey. In *Cambridge Summer School in Mathematical Logic. Lecture Notes in Mathematics 337*, Springer-Verlag, New York, 1971, pp. 95–112.
34. KRIPKE, S. A. Semantical analysis of intuitionistic logic I. In *Formal Systems and Recursive Functions. Proceedings of the 8th Logic Colloquium* (Oxford, 1963). North-Holland, Amsterdam, 1965, pp. 92–130.
35. LAMBEK, J. From lambda calculus to Cartesian closed categories. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, Orlando, Fla., 1980, pp. 375–402.
36. LANDIN, P. J. A correspondence between Algol 60 and Church's Lambda-notation. *Commun. ACM 8*, 2, 3 (Feb.–Mar. 1965), 89–101; 158–165.
37. LANDIN, P. J. The next 700 programming languages. *Commun. ACM 9*, 3 (Mar. 1966), 157–166.
38. LÄUCHLI, H. Intuitionistic propositional calculus and definably non-empty terms. *J. Symbolic Logic 30* (1965), 263.
39. LÄUCHLI, H. An abstract notion of realizability for which intuitionistic predicate calculus is complete. In *Intuitionism and Proof Theory: Proceedings of the Summer Conference at Buffalo N.Y.* (1968). North-Holland, Amsterdam, 1970, pp. 227–234.
40. LEIVANT, D. Polymorphic type inference. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages* (Austin, Tex., Jan. 24–26, 1983). ACM, New York, 1983, pp. 88–98.
41. LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanism in CLU. *Commun. ACM 20*, 8 (Aug. 1977), 564–576.
42. LISKOV, B. ET AL. *CLU Reference Manual. Lecture Notes in Computer Science 114*, Springer-Verlag, New York, 1981.
43. MAC LANE, S. *Categories for the Working Mathematician. Graduate Texts in Mathematics 5*, Springer-Verlag, New York, 1971.
44. MACQUEEN, D. B. Modules for standard ML. In *Polymorphism 2*, 2 (1985), 35 pages. An earlier version appeared in *Proceedings of 1984 ACM Symposium on Lisp and Functional Programming.*
45. MACQUEEN, D. B. Using dependent types to express modular structure. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg Beach, Fla, Jan. 13–15, 1986). ACM, New York, 1986, pp. 277–286.
46. MARTIN-LÖF, P. Constructive mathematics and computer programming. Paper presented at *The 6th International Congress for Logic, Methodology and Philosophy of Science.* Preprint, Univ. of Stockholm, Dept. of Mathematics, Stockholm, 1979.
47. MCCRACKEN, N. An investigation of a programming language with a polymorphic type structure. Ph.D. dissertation, Syracuse Univ., Syracuse, N.Y., 1979.
48. MCCRACKEN, N. The typechecking of programs with implicit type structure. In *Proceedings of International Symposium on Semantics of Data Types. Lecture Notes in Computer Science* 173, 1984. Springer-Verlag, New York, pp. 301–316.
49. MEYER, A. R., AND REINHOLD, M. B. Type is not a type. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg Beach, Fla., Jan. 13–15, 1986). ACM, New York, 1986. pp. 287–295.

50. MILNER, R. The standard ML core language. *Polymorphism 2*, 2 (1985), 28 pages. An earlier version appeared in *Proceedings of 1984 ACM Symposium on Lisp and Functional Programming.*
51. MITCHELL, J. C. Semantic models for second-order Lambda calculus. In *Proceedings of the 25th IEEE Symposium on Foundations of Computer Science* (1984). IEEE, New York, 1984, pp. 289–299.
52. MITCHELL, J. C. Representation independence and data abstraction. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg Beach, Fla., Jan. 13–15, 1986). ACM, New York, 1986, pp. 263–276.
53. MITCHELL, J. C. Polymorphic type inference and containment. *Inf. Comput. 76*, 2/3 (Feb./Mar. 1988), 211–249.
54. MITCHELL, J. C., AND HARPER, R. The essence of ML. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages* (San Diego, Calif., Jan. 13–15, 1988). ACM, New York, 1988, pp. 28–46.
55. MITCHELL, J. C., AND MEYER, A. R. Second-order logical relations. In *Logics of Programs. Lecture Notes in Computer Science 193*, Springer-Verlag, New York, 1985, pp. 225–236.
56. MITCHELL, J. C., AND PLOTKIN, G. D. Abstract types have existential types. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages* (New Orleans, La., Jan. 14–16, 1985). ACM, New York, 1985, pp. 37–51.
57. MITCHELL, J. G., MAYBERRY, W., AND SWEET, R. Mesa language manual. Tech. Rep. CSL-79-3, Xerox PARC, Palo Alto, Calif., 1979.
58. MORRIS, J. H. Types are not sets. In *1st ACM Symposium on Principles of Programming Languages* (Boston, Mass., Oct. 1–3, 1973). ACM, New York, 1973, pp. 120–124.
59. O'DONNELL, M. A practical programming theorem which is independent of Peano arithmetic. In *11th ACM Symposium on the Theory of Computation* (Atlanta, Ga., Apr. 30–May 2, 1979). ACM, New York, 1979, pp. 176–188.
60. PRAWITZ, D. *Natural Deduction.* Almquist and Wiksell, Stockholm, 1965.
61. PRAWITZ, D. Ideas and results in proof theory. In *2nd Scandinavian Logic Symposium.* North-Holland, Amsterdam, 1971, pp. 235–308.
62. REYNOLDS, J. C. Towards a theory of type structure. In *Paris Colloquium on Programming. Lecture Notes in Computer Science 19*, Springer-Verlag, New York, 1974, pp. 408–425.
63. REYNOLDS, J. C. The essence of Algol. In *Algorithmic Languages*, J. W. de Bakker and J. C. van Vliet, Eds. IFIP, North-Holland, Amsterdam, 1981, pp. 345–372.
64. REYNOLDS, J. C. Types, abstraction, and parametric polymorphism. In *IFIP Congress* (Paris, Sept. 1983).
65. REYNOLDS, J. C. Polymorphism is not set-theoretic. In *Proceedings of International Symposium on Semantics of Data Types. Lecture Notes in Computer Science 173*, Springer-Verlag, New York, 1984, pp. 145–156.
66. SHAW, M. (Ed.) *ALPHARD: Form and Content.* Springer-Verlag, New York, 1981.
67. STATMAN, R. Intuitionistic propositional logic is polynomial-space complete. *Theor. Comput. Sci. 9* (1979), 67–72.
68. STATMAN, R. Number theoretic functions computable by polymorphic programs. In *22nd IEEE Symposium on Foundations of Computer Science.* IEEE, New York, 1981, pp. 279–282.
69. STENLUND, S. *Combinators, λ-terms and Proof Theory.* Reidel, Dordrecht, Holland, 1972.
70. TROELSTRA, A. S. *Mathematical Investigation of Intuitionistic Arithmetic and Analysis. Lecture Notes in Mathematics 344*, Springer-Verlag, New York, 1973.
71. WULF, W. W., LONDON, R., AND SHAW, M. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Softw. Eng. SE-2* (1976), 253–264.

Received June 1986; revised March 1988; accepted March 1988