Palo Alto Research Center

Data Types Are Values

James Donahue
Alan Demers

XEROX

Data Types Are Values

James Donahue
Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Alan Demers
Computer Science Department
Cornell University
Ithaca, New York 14853

CSL-83-5 March 1984 [P83-00005]

Abstract: An important goal of programming language research is to isolate the fundamental concepts of languages, those basic ideas that allow us to understand the relationship among various language features. This paper examines one of these underlying notions, *data type*, with particular attention to the treatment of *generic* or *polymorphic* procedures and static type-checking.

A version of this paper will appear in the ACM *Transactions on Programming Languages and Systems*.

CR Categories and Subject Descriptors: D.3 (Programming Languages), D.3.1 (Formal Definitions and Theory), D.3.3 (Language Constructs), F.3.2 (Semantics of Programming Languages)

Additional Keywords and Phrases: data types, polymorphism

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

An important goal of programming language research is to isolate the fundamental concepts of languages, those basic ideas that allow us to understand the relationship among various language features. This paper examines one of these underlying notions, *data type*, and presents a meaning for this term that allows us to:

> describe a simple treatment of *generic* or *polymorphic* procedures that preserves full static type-checking and allows unrestricted use of recursion; and
>
> give a precise meaning to the phrase *strong typing*, so that *Language X is strongly typed* can be interpreted as a critically important theorem about the semantics of the language.

This approach to the meaning of data types was used by the authors in the design of the programming language Russell [Boehm80, Demers80a,b] and Russell will be used to present examples below. One thesis of this paper is that the semantics of data types presented here has served us well in the Russell design.

The paper is organized as follows. In Section 2 we present the motivation leading to our search for a new meaning of data type. Section 3 describes our interpretation of data types as collections of operations. Sections 4 and 5 describe how this approach is used in Russell to allow unrestricted polymorphism and to provide a semantic justification for the syntactic type-checking rules of Russell. The final section closes with some thoughts on the general approach taken in this research.

2. The Problem

The particular problem that motivated our study of data types was the difficulty of writing *generic* or *polymorphic* procedures—procedures that can operate on variables and values of more than one type—in existing Algol-like languages. For example, consider the simple Pascal procedure

```
procedure Swap(var x,y: integer)
 var z: integer;
 begin z := x; x := y; y := z end
```

Even though Swap contains no code that depends on the particular properties of the type of its arguments, it cannot be used to swap variables of any type other than integer. This can make writing subroutine packages extremely tedious—the more general-purpose a subroutine is, the more copies of it are necessary. The introduction of user-defined data types makes this restriction even more telling. How can one apply existing operators to values of a newly-defined type?

One can, of course, evade this problem, as is done in dynamically-typed languages like LISP and SMALLTALK or in typeless languages like BCPL. However, these alternative approaches have their own sets of problems. In typeless languages, the common programming blunder of applying an operation to an operand of the wrong type is not a detectable error, as the notion of *wrong type* is absent. In dynamically-typed languages such type errors can be detected, but only when it is too late to do anything about them. Additionally, the obligation to check for such errors increases the execution cost of every program, even programs using only legal operator/operand combinations.

Milner [78] has suggested the use of *type determination* as a means of adding the desired flexibility. The basic idea is that there is sufficient information in a program for a compiler to determine the type constraints necessary for safe execution; the programmer need not supply any type information about the variables in his program. While this approach seems practicable in many cases, it too suffers a serious flaw. The degree of flexibility it achieves is less dependent on what the programmer writes than on how clever the compiler is. Milner's particular type-determination algorithm fails to allow passing generic procedures as arguments, because it assumes that all uses of an identifier have the same type. If the programmer could specify the types of parameters, it would be possible to write correct programs that violated this assumption. Conversely, there are instances in which a programmer wishes a program to have only very limited possibilities for use; this system provides no way to prohibit arbitrary uses that do not violate compiler-determined type constraints.

Our approach to allowing generic operations is to use y, allowing it to swap different variables. Why not parameterize `Swap` with respect to a type, allowing it to swap variables of different types?

This approach has the advantage of simplicity, but it also poses a basic problem in semantics: what does a data type argument mean? The answer to this question should be consistent with our present understanding of parameterization and application. We talk about the *value of the argument* when we pass an integer to a function; our meaning of data type should explain what is meant by the *value of the argument* when we pass a data type. Having such a meaning allows us to design a language in which data type parameters fit well—they are treated like any other parameters. It is also consistent with the dogma of denotational semantics [Tennent77, Scott77] which says that we should define the meanings of programs in terms of values in appropriately chosen *abstract value spaces.*

In the next Section we draw on work in denotational semantics to develop a notion of data types as values. As we shall see, this notion allows us to give a very simple interpretation of type parameterization and to give precise answers to several other language design questions involving data types. We shall discuss one of these points—the meaning of *strong typing*—in detail, and show how our interpretation of data types allows us to give a particularly attractive meaning to this phrase.

3. Data Types

Our goal is to give a meaning for the *value* of a data type parameter. For this purpose the common interpretation of data types as sets of values is less appropriate than the following meaning:

> A data type is a collection of named operations that provide an interpretation of values and variables of a single universal value space.

We first explain what we mean by *interpretation* and then discuss how this meaning of data type allows us to view data types as values. Our discussion will be informal, but we will suggest how, using the work of Scott, these ideas can be made mathematically precise.

3.1 Data Types as Interpretations of Values

We begin by assuming the underlying value space to be typeless. Informally that is what we mean by a *single value space*; the meaning of *universal* will be discussed below. No properties of the values themselves allow us to say, for example, that one value is an integer while another is a Boolean. This assumption accords well with the hardware of most machines: values are represented by (untyped) sequences of bits, and can be partitioned into disjoint sets only by introducing explicit tag fields, with the associated overhead in time and space.

Observe that values in a typeless value space have no inherent *meaning* as well as no inherent type. Continuing our analogy with hardware, the same sequence of bits can be used to represent logical values, integers, floating point values, and the programs that manipulate them. Given a particular bit string, we cannot say what it means any more than we can say what its type is. The most we can say is something of the form *if this value is used as an operand of the integer addition operator, it will behave as follows. . . .*

How then can we speak about the meaning of a typeless value? By considering the *interpretation* of the value by various operations. For example, a particular value may behave as the identity element under the integer addition operator. Certain collections of operations considered together may impose a consistent interpretation of the value space. For example, the value that behaves as the identity for integer addition also yields itself when used in integer multiplication. Thus, instead of saying that a (typed) value *is* the integer 0, we can say that for a particular choice of integer operations an (untyped) value *behaves like* the integer 0. The same untyped value may also behave like the Boolean True and the character 'a', when interpreted by the other collections of operations.

The notion that a set of operations can impose a consistent interpretation of the values of a typeless value space is the idea behind the treatment of data types in Russell. In Russell, values themselves have no inherent meaning; instead, meanings are imposed on them by collections of operations. We call these collections of operations *data types.*

3.2 Data Types as Values

The question still remains how we can regard the types *themselves* as values. To answer this, we draw on Scott's work in models of the untyped lambda calculus, which provided the inspiration for this approach.

Abstractly, what are the characteristics of the value space we need? Firstly, as argued above, it must be typeless. In addition, it must be *large*—it must contain the (computable) operations over it. Thus types themselves, which are simply finite collections of operations, must be values in the space.

Note that the combination of these two requirements means that self-application must be allowable. The arguments or result of any operation may be any value (because of the typelessness of the value space), so indeed an operation could be used to interpret itself. Such a value space can reasonably be called *universal.* Does one exist, however?

Intuitively, yes. Returning to our hardware analogy, the store is typeless and (ignoring size limitations) allows the implementation of any computable operation. Moreover, the indistinguishability of program and data allows any operation to be performed on (the representation of) itself. A more abstract mathematical formulation of such spaces can be found in [Scott76, 77]. In these papers, Scott shows how to construct a value space D that is isomorphic to its own space of continuous functions, i.e., that satisfies (up to isomorphism) the equation $D = D \rightarrow D$. The immediate importance of such a domain is that it allows a simple definition of the untyped lambda calculus. For example, in the lambda expression `λx. x(x)`, the identifier x simultaneously stands for a function and its argument. This makes sense only for a value space satisfying the above isomorphism. More generally, the techniques introduced by Scott allow the construction of a variety of rich function spaces in which self-application is possible. In particular, one can build a reasonable mathematical model of a machine store, including the possibility of a program operating on itself. An example of this can be seen in the Russell semantics of [Demers80b].

It remains to be asked what sorts of operations form a data type in an Algol-like language: how are variables and values given interpretations? As a simple example, consider a language like Algol60 with only primitive (unstructured) types. In such a language, we may store values in variables, extract values from variables and compose values by applying certain primitive functions. The meaning of the "primitive" operations over such a space is taken as the meaning of a data type, since it is by these operations that the underlying value space of Algol60 programs is manipulated.

For example, consider the following Algol program fragment:

```
integer x,y; x := 0; y := x.
```

The meaning (or *denotation*) of **integer** must provide *at least* the following:

> The meaning of value extraction (so we know how to take the value of the variable x on the right-hand side of the second assignment),
>
> the meaning of assignment (so we know how to store an integer value in y), and
>
> the meaning of the constant (or nullary function) 0.

The set of operations which is needed to provide an *interpretation* is a language-dependent matter. In [Donahue79], we give a semantics for a polymorphic lambda calculus that uses a single function (a retraction) as the meaning of a data type. More operations are needed for data types in a language like Pascal, but the same basic approach works in both cases.

3.3 Comparison With Other Approaches

The meaning of data type presented above is similar to the idea of algebraic specifications described in [Guttag77] and [Goguen76] in its focus on the operations of a type. There is, however, a subtle but important difference between these approaches. In the algebraic approach, one assumes the existence of a collection of *carrier sets*, and most of the literature suggests that these carrier sets may be assumed to be disjoint. Our approach is to choose a single typeless carrier set that not only represents the values of every type, but also represents the operations of all types.

Our approach also has a (less obvious) connection with the common *types are sets of values* approach: we can find a way to treat our collection of operations as defining a set of values. In the same way that the integer operations must be related by simple algebraic laws, so must the other operations of a data type, including those of assignment and value extraction. We can make these properties precise by a straightforward use of standard denotational semantics (see [Milne76, Tennent77]).

To give a mathematical meaning to assignment, we first need a model of a *machine store*; we will use a function space S such that S = Loc → Val where `Loc` is some domain of *locations* and `Val` is the domain of *storable values* (in Russell, our universal value space). Now we can define value extraction by a function:

```
ValueOf: [ Loc X S ] → Val
```

that returns the value in the given location of the store. Similarly, assignment can be defined by a function:

```
Update: [ Loc X Val ] → S → S
```

that produces a new store by changing the contents of the given location to the new value. Now if we look at this pair of operations, we can see that for all `l` and `s`:

```
ValOfUpdate = λv. ValueOf(l, Update(l, v)(s))
```

(a function of type `Val→Val`) must be the same function as `ValOfUpdate ∘ ValOfUpdate`. In words, assigning a value to a variable and then taking the value of the variable must produce the same result as performing the assignment, taking the value of the variable and then performing the assignment and extraction again with the value produced. Note that this is a weaker condition than saying that `ValueOf` must always produce the value previously assigned, i.e.,

```
ValueOf(l, Update(l, s)(v)) = v
```

in that we leave open the possibility that at least some values will be altered by assignment.

In the parlance of denotational semantics, the function `ValOfUpdate` is a *retraction*, a function f such that f = f∘f. A retraction in D→D has the very important property that it "collapses" D onto the range of f and is the identity function on each element of its range. Thus, f can be seen as mapping every element of D into its image in the subspace given by the range of f. Elements of Val in this subspace will be unmodified by assignment and extraction, while the remaining elements will somehow be projected into the subspace.

The set of values of a type can now be found in the range of this retraction: it is the set of values that may legally be assigned to variables of the type. In fact, this is the definition of data type given in [Jensen75]. Note that for any data type, there may be many `ValueOf` and `Update` operations satisfying this retraction property. Below we will discuss the use of static type-checking as a means of hiding the implementation decision of which of the allowable `ValueOf` and `Update` operations are actually used.

We have now described what is meant by *universal space* and *collection of operations providing an interpretation*, i.e., we have said what we mean by *data type.* Moreover, this meaning of type allows us to give a straightforward semantics of type parameters. Consider the Pascal-like definition:

```
    Identity == func[T: type; x: val T] val T {(*return*) x} end
```

where we have parameterized `Identity` with respect to a type `T`. The meaning of the parameters can now be understood as follows:

> The value parameter `x` stands for a value from some universal value space, which will be interpreted in the body of the function by the operations of the type `T`; and
>
> The type parameter `T` stands for a set of operations used within the body of `Identity` to interpret values of the universal space. This set of operations can be treated as a natural extension of the procedure and function parameters allowed in many existing languages.

By viewing type parameters this way, we can give meaning to type-parameterized constructs independent of any particular arguments supplied to them. In the next section, we consider this point in more detail, giving the syntax and semantics of polymorphic constructs in Russell. In Section 5, we describe a meaning for *strong typing* that can be used with this approach to justify type-checking rules.

4. Type Checking and Polymorphism in Russell

We now show how the principles described above were applied in the treatment of polymorphism and type checking in Russell.

4.1 Signatures

In our view, types specify interpretations of data: no variable or value in a program has meaning until we have specified how it is to be interpreted. To define the interpretation of the value of an expression in a Russell program, we associate with each identifier or expression a syntactic type, or *signature*, similar to a program type of [Reynolds78]. The signature of an expression describes how the value of that expression should be interpreted by identifying the operations that may be performed on it. The criterion for *type correctness* of a Russell program is that signatures can be assigned uniquely according to the rules described below; thus, it is more proper to speak of the *signature correctness* of a Russell program.

Every value is interpreted by a Russell program in one of three ways:

as a function, which may be applied to arguments to yield a result;

as a type, which by the definition of the previous section consists of a finite set of named function values;

as a data item (variable or value) to be interpreted by the operations of some type.

These three possible ways to interpret a value are mirrored in the syntax of signatures:

```
Sig       ::=  DataSig | OpSig
DataSig   ::=  var Exp | val Exp
OpSig     ::=  FuncSig | TypeSig
FuncSig   ::=  func[id1: Sig1; ...; idn: Sign] Sig0
TypeSig   ::=  type id0 [id1: OpSig1; ...; idn: OpSign]
```

A function signature includes formal parameter signatures and a result signature, as one might expect. It also includes formal parameter names; these are necessary for the description of polymorphic functions, as will become clear later.

A type signature specifies the names and signatures of the operations that make up the type. Since a Russell type must provide *all* the information necessary to interpret a variable or value, types in Russell provide operations such as assignment and value extraction that are considered primitive in other languages. Some operations, like assignment, are common to most types; but in general each Russell type comprises a different set of operations. Note that each type signature includes a bound variable (id_0 in the syntax above); the reason for this will be discussed later.

Finally, a data signature consists of a **var** or **val** indication together with an expression for a type whose operations should be used to interpret the value. An important property of the Russell signature correctness rules is that they will never assign a signature of the form **var** e or **val** e unless the expression e can be assigned a **type** signature.

The following example illustrates how a polymorphic version of the swap function of our earlier example would be written in Russell:

```
Swap == func [T: type t [New: func[] var t;
                        ValOf: func[var t] val t;
                        ← : func[var t; val t] val t ];
                        x, y: var T]
 {let z: var T == T$New[]
  in
  [z] T$← [T$ValOf[x]]; [x] T$← [T$ValOf[y]]; [y] T$← [T$ValOf[z]]
  ni}
```

The procedure heading of Swap (which is also the signature of Swap) specifies that the type parameter T provides operations named New, ValOf and ←. The parameters x and y are variables to be interpreted by the operations of T. The body of this function shows how the operations provided by T are used. The declaration:

```
    z: var T == T$New[]
```

declares z to be a "new T variable" by binding it to a value produced by applying a function named **New** selected from T. The existence of **var**-producing functions like **New** makes it possible for an operation of T to provide the meaning of "variable allocation" for the type. Similarly, the three assignments that swap the values of x and y are performed by a function named ← selected from T.

This simple example suggests how type checking of Russell programs is done using signatures. The signatures of the parameters x and y, **var** T, indicate that they are to be interpreted by the operations of the parameter T; the signature of T indicates that it is to be interpreted as a type providing all the operations required in the body of Swap. The signature correctness rules described below ensure that no *misinterpretation* takes place. As we show in the next section, signatures contain enough information to allow polymorphic functions—even recursive ones—to be type checked, without re-instantiating the function for each separate invocation of it.

Even more important than the ease of signature-checking polymorphic functions in Russell is the fact that the *meanings* of such functions are straightforward. Most treatments of polymorphic operations view them as "macros" to be expanded independently for each distinct type argument. This approach causes problems when combined with recursion, as the following example shows:

```
    R == func [n: val integer; T: type t[] ] val integer
            {if n > 0 ⇒ R[n-1, Array[1,10,T]] # n ≤= 0 ⇒ 17 fi}
```

This procedure is clearly signature-correct: the type parameter specifies no operations, so any type expression is a legal argument to R. Also, execution of the program clearly terminates. But simple textual macro expansion of the function R can *never* terminate, because the recursive calls of R use progressively more complicated type arguments. Using our treatment of data types as values, which gives a uniform semantics to all forms of procedure application, recursive procedures are no more difficult to type-check or to interpret than nonrecursive ones.

Although we have discussed only polymorphic functions that produce simple values, there is no reason to prohibit functions from accepting parameters and returning values of any signature whatsoever. Since all values exist in the same universal space, the semantics of parameters and function results is completely uniform. For example, there is no special form of "parameterized data type" in Russell; instead, one simply writes a function that returns a data type.

In Russell, one finds a degree of "type completeness" not common in programming languages—anything a programmer can write can be passed as an argument to something of even higher type. Guaranteeing that the combining forms of abstraction and application can be used in an unlimited fashion is one response to the recent arguments of Backus[78] about the weakness of combining forms in vonNeumann languages. (A further discussion of the importance of type- completeness can be found in [Demers80b].)

We now turn to a more careful description of Russell signature-checking and show how it allows us to give precise mathematical meaning to the phrase "strongly typed."

4.2 Structure of the Signature Correctness Rules

We now present the essentials of the Russell signature correctness rules in a style similar to [Bates82, Demers83, Martin-Lof79]. Roughly speaking, the system is a collection of logical inference rules that are used to prove formulas stating that a given expression has a given signature. There is a straightforward, efficient decision procedure for the system, making it suitable for use in a language implementation.

4.2.1 Formulas

The formulas of our system include:

Typings. These are formulas of the form $e \sim S$, which assert that e is a legal Russell expression whose signature is S.

Legality Assertions. These are formulas of the form **legal** S, which assert that S is a legal signature: if S has the form **var** e or **val** e, then e ~ **type** t[] will be deducible and the value produced by evaluating e must not depend on the contents of the store.

Signature Matchings. These are formulas of the form $S_1 \leq S_2$, which assert that S_1 is more restrictive than S_2: whenever $e \sim S_1$ is deducible, $e \sim S_2$ will be deducible as well.

4.2.2 Environments

An *environment* Γ is a set of typings in which only simple identifiers (not arbitrary expressions) may appear on the left hand sides. Environment Γ (*uniquely*) *defines* an identifier x if there is a (unique) signature S such that Γ contains the formula $x \sim S$. Γ is *functional* if every identifier it defines is uniquely defined, and *closed* if it defines every identifier that occurs free in any of its formulas. Intuitively, an environment contains the signatures of all identifiers that have been declared in a Russell program. Thus, every environment we use will be functional (every identifier must have a unique signature) and closed (the signature of an identifier may not contain undefined identifiers).

Given Γ and a set of program identifiers x_1 through x_n, we define $\Gamma / x_1, ..., x_n$, to be the maximal closed subset of Γ not defining any of x_1 through x_n. This construction simply deletes from Γ definitions of x_1 through x_n and any formulas that depend on them. We also define:

$(\Gamma, x_1 \sim S_1, ..., x_n \sim S_n) =_{def} \Gamma / x_1, ..., x_n \cup \{x_1 \sim S_1, ..., x_n \sim S_n\}$

which gives the effect of declaring new program identifiers in function constructions or **let**-binding—first all references to previous instances of the identifiers are deleted from the environment, and then new typings are added.

Finally, we define Γ_{NoVar} to be Γ with all variable identifiers (identifiers with **var** signatures) eliminated as above. The Russell signature correctness rules will assign a **func** or **type** signature to an expression only if that signature can be deduced from Γ_{NoVar}. The reasons for this restriction will be discussed in Section 5.

4.2.3 Goals, Inference, and Signature Correctness

A *goal* G has the form $\Gamma \vdash$ F, with the meaning that the formula F is a consequence of the typings in Γ. An *inference rule* has the form:

$$\frac{G_1,\ \ldots,\ G_n}{G}$$

with the meaning that from G_1 through G_n we may conclude G. We call G the *conclusion* of the rule, and G_1 through G_n its *hypotheses* or *subgoals*. A *theorem* is a goal provable using the inference rules given below. A Russell program e will be said to be *signature correct* with respect to a given initial environment Γ_0 if, and only if, there is a theorem of the form $\Gamma_0 \vdash e \sim S$ for some signature S.

In the sections that follow, we adhere to the conventions that variables d, e, f, ... represent expressions, variables t, u, ..., z represent new identifiers, and an expression of the form $d[e_1/x_1, \ldots, e_n/x_n]$ (which represents the result of *simultaneously* substituting e_1 through e_n for x_1 through x_n in d), is legal only if no capture occurs.

4.3 Basic Rules

We begin with some simple rules illustrating how conventional Pascal-like type checking can be expressed in this system. First we consider sequential composition.

Sequential Composition

$$\frac{\Gamma \vdash e_1 \sim S_1, \ldots, \quad \Gamma \vdash e_n \sim S_n}{\Gamma \vdash (e_1; \ldots; e_n) \sim S_n}$$

This rule states that if each of e_1 through e_n is signature-correct in a given environment Γ, then their sequential composition $(e_1; \ \ldots; \ e_n)$ is signature-correct in that environment, and has the same signature as e_n. This corresponds to our intuition about the meaning of sequential composition: provided no type error occurs during evaluation of e_1 through e_{n-1}, their values are simply discarded; evaluating the compound expression produces the value (and thus has the signature) of e_n.

Russell includes conditional expressions, similar to guarded commands [Dijkstra75], with the following signature correctness inference rule.

Conditional

$$\frac{\Gamma \vdash c_1 \sim \textbf{val boolean},\ \Gamma \vdash e_1 \sim S, \ldots, \quad \Gamma \vdash c_n \sim \textbf{val boolean},\ \Gamma \vdash e_n \sim S}{\Gamma \vdash \textbf{if}\ c_1 \Rightarrow e_1\ \#\ \ldots\ \#\ c_n \Rightarrow e_n\ \textbf{fi} \sim S}$$

This rule states that each condition part c_i must have signature **val boolean**, and the expressions e_i must be signature correct and have identical signatures. Again, this rule corresponds to our intuition about the meaning of conditionals.

Equally straightforward rules apply to many of the other Russell constructs. The novelty of the Russell rules lies in their treatment of application (of a function to arguments) and selection (of a component of a type), which we describe below.

4.4 Application

A simple Pascal-like rule for signature correctness of (non-polymorphic) function applications is:

Simple Application

$$\frac{\Gamma \vdash \mathtt{e} \sim \mathbf{func}[\mathtt{x}_1\texttt{:}\ \mathtt{S}_1; \ldots; \mathtt{x}_n\texttt{:}\ \mathtt{S}_n]\ \mathtt{S}_0, \quad \Gamma \vdash \mathtt{e}_1 \sim \mathtt{S}_1, \ldots, \Gamma \vdash \mathtt{e}_n \sim \mathtt{S}_n}{\Gamma \vdash \mathtt{e}[\mathtt{e}_1, \ldots, \mathtt{e}_n] \sim \mathtt{S}_0}$$

The intuition behind this rule is clear: a function may legally be applied to a list of arguments provided each argument signature can be shown to match the corresponding formal parameter signature. However, this simple rule cannot handle application of a polymorphic function, in which some parameter signatures may contain occurrences of other parameters. For example, in the signature:

$$\mathbf{func}[\mathtt{T}\texttt{:}\ \mathbf{type}\ \mathtt{t}[];\ \mathtt{x}\texttt{:}\ \mathbf{val}\ \mathtt{T}]\ \mathbf{val}\ \mathtt{T}$$

of the polymorphic identity function, the signature **val** T of parameter x contains an occurrence of parameter T. A polymorphic function signature like this one can be thought of as specifying relationships that must exist among legal arguments to the function. To treat polymorphic functions requires a more powerful rule:

Russell Application

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{e} \sim \mathbf{func}\ [\ \mathtt{x}_1\texttt{:}\ \mathtt{S}_1; \ldots; \mathtt{x}_n\texttt{:}\ \mathtt{S}_n\ ]\ \mathtt{S}_0,\\ \Gamma \vdash \mathtt{e}_1 \sim \mathtt{S}_1[\mathtt{e}_1/\mathtt{x}_1, \ldots, \mathtt{e}_n/\mathtt{x}_n], \ldots,\\ \Gamma \vdash \mathtt{e}_n \sim \mathtt{S}_n[\mathtt{e}_1/\mathtt{x}_1, \ldots, \mathtt{e}_n/\mathtt{x}_n]\end{array}}{\Gamma \vdash \mathtt{e}[\ \mathtt{e}_1, \ldots, \mathtt{e}_n\ ] \sim \mathtt{S}_0[\mathtt{e}_1/\mathtt{x}_1, \ldots, \mathtt{e}_n/\mathtt{x}_n]}$$

The substitutions in this rule allow it to handle mutually dependent parameter and result signatures. For example, the application Identity[**integer**, 17] can be shown to be signature correct and to produce an **integer** result by instantiating the above rule as:

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{Identity} \sim \mathbf{func}[\mathtt{T}\texttt{:}\ \mathbf{type}\ \mathtt{t}[];\ \mathtt{x}\texttt{:}\ \mathbf{val}\ \mathtt{T}]\ \mathbf{val}\ \mathtt{T}\\ \Gamma \vdash \mathbf{integer} \sim \mathbf{type}\ \mathtt{t}[]\ [\mathbf{integer}/\mathtt{T},\ 17/\mathtt{x}]\\ \Gamma \vdash 17 \sim \mathbf{val}\ \mathtt{T}[\mathbf{integer}/\mathtt{T},\ 17/\mathtt{x}]\\ \Gamma \vdash \mathbf{legal\ val}\ \mathtt{T}[\mathbf{integer}/\mathtt{T},\ 17/\mathtt{x}]\end{array}}{\Gamma \vdash \mathtt{Identity}[\mathbf{integer},\ 17] \sim \mathbf{val}\ \mathtt{T}[\mathbf{integer}/\mathtt{T},\ 17/\mathtt{x}]}$$

Performing the indicated substitutions, we obtain:

$$\begin{array}{l}
\Gamma \vdash \texttt{Identity} \sim \textbf{func}[\texttt{T}\texttt{:}\ \textbf{type}\ \texttt{t[];}\ \texttt{x:}\ \textbf{val}\ \texttt{T}]\ \textbf{val}\ \texttt{T} \\
\Gamma \vdash \textbf{integer} \sim \textbf{type}\ \texttt{t[]} \\
\Gamma \vdash \texttt{17} \sim \textbf{val}\ \textbf{integer} \\
\Gamma \vdash \textbf{legal}\ \textbf{val}\ \textbf{integer} \\
\hline
\Gamma \vdash \texttt{Identity[}\textbf{integer}\ \texttt{,}\ \texttt{17]} \sim \textbf{val}\ \textbf{integer}
\end{array}$$

The conclusion of this inference is just what one would expect—the polymorphic identity function may be applied to an integer to produce an integer. The first three subgoals are also the natural ones showing that the function and its arguments are signature correct. The final subgoal, **legal val integer**, is more subtle. Intuitively, it guarantees that the result type of the application does not depend on the values of any variables in the store. We will return to this point when we discuss legality assertions and the substitution property below. (Note this is necessary only for the result signature; a property of the rules is that if whenever one can conclude e ~ S one can also conclude **legal** S.)

4.5 Component Selection

We have seen that the signature correctness rules handle applications of polymorphic functions by syntactic substitution of argument expressions for parameters. Similar substitutions occur in the rule for selecting a component of a type.

Selection

$$\begin{array}{l}
\Gamma \vdash \texttt{e} \sim \textbf{type}\ \texttt{t}\ [\ldots,\ \texttt{x:}\ \texttt{S}, \ldots],\ \Gamma \vdash \textbf{legal}\ \texttt{S[e/t]} \\
\hline
\Gamma \vdash \texttt{e\$x} \sim \texttt{S[e/t]}
\end{array}$$

The substitution that occurs in the conclusion of this rule allows the signature of a component of a type to refer to the type itself.

For example, consider a selection such as **integer**\$+. The signature of the built-in type **integer** is:

$$\textbf{type}\ \texttt{t}\ [\ldots\ ;\ \texttt{+:}\ \textbf{func}[\texttt{x,y:}\ \textbf{val}\ \texttt{t}]\ \textbf{val}\ \texttt{t};\ \ldots]$$

To produce the signature of **integer**\$+, the above rule can be instantiated as:

$$\begin{array}{l}
\Gamma \vdash \textbf{integer} \sim \textbf{type}\ \texttt{t}\ [\ldots;\ \texttt{+:}\ \textbf{func}[\texttt{x,y:}\ \textbf{val}\ \texttt{t}]\ \textbf{val}\ \texttt{t};\ \ldots] \\
\Gamma \vdash \textbf{legal}\ \textbf{func}[\texttt{x,y:}\ \textbf{val}\ \texttt{t}]\ \textbf{val}\ \texttt{t[}\textbf{integer}\texttt{/t]} \\
\hline
\Gamma \vdash \textbf{integer\$+} \sim \textbf{func}[\texttt{x,y:}\ \textbf{val}\ \texttt{t}]\ \textbf{val}\ \texttt{t[}\textbf{integer}\texttt{/t]}
\end{array}$$

Performing the indicated substitutions yields:

$$\begin{array}{l}
\Gamma \vdash \textbf{integer} \sim \textbf{type}\ \texttt{t}\ [\ldots\ ;\ \texttt{+}\ \texttt{:}\ \textbf{func}[\texttt{x,y:}\ \textbf{val}\ \texttt{t}]\ \textbf{val}\ \texttt{t};\ \ldots] \\
\Gamma \vdash \textbf{legal}\ \textbf{func}[\texttt{x,y:}\ \textbf{val}\ \textbf{integer}]\ \textbf{val}\ \textbf{integer} \\
\hline
\Gamma \vdash \textbf{integer\$+} \sim \textbf{func}[\texttt{x,y:}\ \textbf{val}\ \textbf{integer}]\ \textbf{val}\ \textbf{integer}
\end{array}$$

It is clear that the two subgoals will be deducible for any reasonable choice of environment Γ, so the signature of **integer**$+ is:

```
func[x,y: val integer] val integer
```

as one would expect. If **real** were a type with the same signature as **integer**, the signature of **real**$+ would be:

```
func[x,y: val real] val real
```

again, just as one would expect. It is the substitution of a type expression (**real** or **integer** in this case) into the signatures of components selected from it that allows identically-named components selected from types with identical signatures to have distinct signatures.

Like the function application rule discussed in the previous section, this rule includes a subgoal of the form **legal** S. We now turn to a discussion of why these goals are necessary.

4.6 Signature Legality

The greatest strength of the Russell signature system is its generality. For example, it is simple to construct a Russell function T whose signature is given by:

```
T: func [i: val integer] type t []
```

Such functions provide the only *parameterized type* mechanism that is needed in Russell. However, combining such generality with variables and a modifiable store requires great care. As an illustration of this, consider the following program fragment:

```
let x: var T[integer$ValOf[i]] == ...
    in ...
        [i] integer$← [integer$ValOf[i] integer$+ [1]]
        let y: var T[integer$ValOf[i]] == ...
        in ...
            [x] T[integer$ValOf[i]]$←
             [T[integer$ValOf[i]]$ValOf[y]]
```

where i has signature **var integer** and T is the type-returning function described above. The signatures of x and y in this program fragment are textually identical—both variables have signature:

```
var T[integer$ValOf[i]].
```

Nevertheless, the assignment of y to x in the last line clearly should not be considered signature-correct, since the value of i, which occurs free in the signature, changes between the introduction of x and y.

It was the need to detect and prohibit errors like this that led us to introduce legality assertions into the Russell signature rules. The basic rule for inferring that a data signature is legal is the following.

Data Signature Legality

$$\begin{array}{l} \Gamma_{\mathrm{NoVar}} \vdash \mathtt{e} \sim \mathbf{type}\ \mathtt{t}\ [] \\ \hline \Gamma \vdash \mathbf{legal}\ \mathbf{var}\ \mathtt{e},\ \Gamma \vdash \mathbf{legal}\ \mathbf{val}\ \mathtt{e} \end{array}$$

The intuition behind this rule is that if one can deduce e ~ **type** t [ ] without referring to any **var** identifiers in the environment, then the result of evaluating e is independent of the contents of memory—in particular, if e is evaluated several times the result will always be the same. Thus, the value of an expression with signature **var** e or **val** e can be interpreted by operations selected from the value of e; identically named operations obtained in this way will always be equivalent since the result of evaluating e itself is always the same.

For completeness, we require rules for legality of function and type signatures as well as data signatures.

Function Signature Legality

$$\begin{array}{l} (\Gamma,\ x_1 \sim S_1, \dots,\ x_n \sim S_n) \vdash \mathbf{legal}\ S_1 \\ (\Gamma,\ x_1 \sim S_1, \dots,\ x_n \sim S_n) \vdash \mathbf{legal}\ S_n \\ (\Gamma,\ x_1 \sim S_1, \dots,\ x_n \sim S_n) \vdash \mathbf{legal}\ S \\ \hline \Gamma \vdash \mathbf{legal}\ \mathbf{func}[x_1{:}\ S_1 \dots x_n{:}\ S_n]\ \ S \end{array}$$

This rule states that a function signature is legal if the parameter and result signatures are legal. Since polymorphic function signatures allow parameter names to occur in the signatures of other parameters and in the signature of the result, we cannot in general deduce that the parameter and signatures are legal without first adding the parameters to the environment. For example, to deduce that the signature:

$$\mathbf{func}[\mathtt{T}{:}\ \mathbf{type}\ \mathtt{t}\ [];\ \mathtt{x}{:}\ \mathbf{val}\ \mathtt{T}]\ \mathbf{val}\ \mathtt{T}$$

of the polymorphic identity function is legal, the above rule would be instantiated as:

$$\begin{array}{l} (\Gamma,\ \mathtt{T} \sim \mathbf{type}\ \mathtt{t}\ [],\ \mathtt{x} \sim \mathbf{val}\ \mathtt{T}) \vdash \mathbf{legal}\ \mathbf{type}\ \mathtt{t}\ [] \\ (\Gamma,\ \mathtt{T} \sim \mathbf{type}\ \mathtt{t}\ [],\ \mathtt{x} \sim \mathbf{val}\ \mathtt{T}) \vdash \mathbf{legal}\ \mathbf{val}\ \mathtt{T} \\ (\Gamma,\ \mathtt{T} \sim \mathbf{type}\ \mathtt{t}\ [],\ \mathtt{x} \sim \mathbf{val}\ \mathtt{T}) \vdash \mathbf{legal}\ \mathbf{val}\ \mathtt{T} \\ \hline \Gamma \vdash \mathbf{legal}\ \mathbf{func}[\mathtt{T}{:}\ \mathbf{type}\ \mathtt{t}\ [];\ \mathtt{x}{:}\ \mathbf{val}\ \mathtt{T}]\ \mathbf{val}\ \mathtt{T}. \end{array}$$

Note that the second and third subgoals, showing legality of the signatures of the parameter x and of the result, are identical.

The rule for legality of type signatures is similar.

Type Signature Legality

$$\begin{array}{l} (\Gamma,\ \mathtt{t} \sim \mathbf{type}\ \mathtt{t}[x_1{:}\ S_1 \dots x_n{:}\ S_n]) \vdash \mathbf{legal}\ S_1, \dots, \\ (\Gamma,\ \mathtt{t} \sim \mathbf{type}\ \mathtt{t}[x_1{:}\ S_1 \dots x_n{:}\ S_n]) \vdash \mathbf{legal}\ S_n \\ \hline \Gamma \vdash \mathbf{legal}\ \mathbf{type}\ \mathtt{t}[x_1{:}\ S_1 \dots x_n{:}\ S_n] \end{array}$$

Here the local name t is added to the environment to allow us to deduce that the component signatures are legal.

4.7 Matching Function and Type Signatures

The function application rule given above is quite restrictive—it requires that argument and parameter signatures in an application be textually identical. In view of the many arbitrarily-chosen bound identifiers that occur in signatures, it is desirable to relax this restriction. The following rules, which embody the *signature calculus* of [Demers80a], accomplish this.

Signature Weakening

$$\begin{array}{l} \Gamma \vdash \mathtt{S}_2 \leq \mathtt{S}_1 \\ \Gamma \vdash \mathtt{e} \sim \mathtt{S}_2 \\ \hline \Gamma \vdash \mathtt{e} \sim \mathtt{S}_1 \end{array}$$

This rule states an obvious property: if $\mathtt{S}_2$ is more restrictive than $\mathtt{S}_1$ and e can be given signature $\mathtt{S}_2$, then it can also be given signature $\mathtt{S}_1$.

Renaming

$$\begin{array}{l} \Gamma \vdash \textbf{legal func } [\mathtt{x}_1\!: \mathtt{S}_1 ; \ldots; \mathtt{x}_k\!: \mathtt{S}_k]\ \mathtt{S}_0 \\ \hline \Gamma \vdash \textbf{func}[\mathtt{x}_1\!:\!\mathtt{S}_1; \ldots; \mathtt{x}_k\!:\!\mathtt{S}_k]\ \mathtt{S}_0 \leq \textbf{func}[\ldots; \mathtt{y}_i\!:\!\mathtt{S}_i[\ldots, \mathtt{y}_j/\mathtt{x}_j, \ldots]; \ldots]\ \mathtt{S}_0[\ldots, \mathtt{y}_j/\mathtt{x}_j, \ldots] \end{array}$$

where the identifiers $\mathtt{y}_1$ through $\mathtt{y}_k$ are new.

Parameter names may be replaced uniformly in function signatures.

Reordering

$$\begin{array}{l} \Gamma \vdash \textbf{legal type } \mathtt{t}\ [\mathtt{x}_1\!: \mathtt{R}_1; \ldots; \mathtt{x}_n\!: \mathtt{R}_n] \\ \hline \Gamma \vdash \textbf{type } \mathtt{t}\ [\mathtt{x}_1\!: \mathtt{R}_1; \ldots; \mathtt{x}_n\!: \mathtt{R}_n] \leq \textbf{type } \mathtt{t}\ [\mathtt{y}_1\!: \mathtt{S}_1 ; \ldots; \mathtt{y}_n\!: \mathtt{S}_n] \end{array}$$

whenever the (unordered) set of $\langle \mathtt{x}_i , \mathtt{R}_i \rangle$ pairs is identical to the (unordered) set of $\langle \mathtt{y}_j , \mathtt{S}_j \rangle$ pairs.

The order of presentation of components in a type signature is irrelevant.

Forgetting

$$\begin{array}{l} \Gamma \vdash \textbf{legal type } \mathtt{t}\ \ \mathtt{x}_1\!: \mathtt{R}_1; \ldots; \mathtt{x}_n\!: \mathtt{R}_n] \\ \hline \Gamma \vdash \textbf{type } \mathtt{t}\ [\mathtt{x}_1\!: \mathtt{R}_1; \ldots; \mathtt{x}_n\!: \mathtt{R}_n] \leq \textbf{type } \mathtt{t}\ [\mathtt{y}_1\!: \mathtt{S}_1 ; \ldots; \mathtt{y}_m\!: \mathtt{S}_m] \end{array}$$

whenever the (unordered) set of $\langle \mathtt{x}_i , \mathtt{R}_i \rangle$ pairs is a superset of the (unordered) set of $\langle \mathtt{y}_j , \mathtt{S}_j \rangle$ pairs.

A type signature can be made weaker (less restrictive) by eliminating (or *forgetting*) some of its operations.

The above rules do not greatly increase the complexity of signature checking but make the language much easier to use. For example, the signature of the builtin type `Integer` as given in [Boehm80] is:

```
type t [ New: func[] var t;
           ←: func[x: var t; y: val t] val t;
           ValOf: func[ x: var t] val t;
           + : func[x,y: val t] val t;
           - : func[x,y: val t] val t;
           * : func[x,y: val t] val t;
           / : func[x,y: val t] val t; ... ]
```

This can easily be shown to match the simpler signature:

```
type t [ New: func[] var t;
           ValOf: func[x: var t] val t;
           ←: func[x: var t; y: val t] val t ]
```

which occurs as a parameter signature in the Russell Swap function given earlier. Thus, the built-in type `Integer` is a legal argument to `Swap`, so that `Swap` can be used to exchange the values of two `Integer` variables. (Adding the forgetting rule is truly a convenience. The Russell language provides general facilities for producing new type values by modifying the set of operations provided by some existing type; this ability to perform type modification can be used to achieve the effect of using the forgetting rule, although it is somewhat cumbersome.)

4.8 Constructions

There is one final class of signature correctness rules—rules for the construction of types and functions (whose values include non-local references). Here we present only the rule for function constructions:

Function Construction

$$\frac{\Gamma \vdash \mathbf{legal}\ \mathbf{func}[\ x_1\!: S_1 \ldots x_n\!: S_n\ ]\ S_0 \qquad (\ \Gamma_{\mathrm{NoVar}},\ x_1 \sim S_1, \ldots, x_n \sim S_n) \vdash e \sim S_0}{\Gamma \vdash \mathbf{func}[x_1\!: S_1 \ldots x_n\!: S_n]\ \{\ e\ \} \sim \mathbf{func}[x_1\!: S_1 \ldots x_n\!: S_n]\quad S_0}$$

The important thing to note about this rule is that the signature of the function body e must be deducible from Γ_{NoVar} and the signatures of the parameters; that is, the function body may not import any **var** identifiers. This rule, called the *import rule* in [Demers80a], ensures that the value returned by a function application depends only on its arguments—there are no hidden dependencies on the contents of memory.

5. Strong Typing

The Russell signature checking rules make the language "strongly typed" in the usual sense of the phrase: each identifier is given an explicit type and programs with mismatched argument/

parameter types are disallowed. An important question, though, is the relation between the syntactic signature-matching rules of the previous section and the semantics of data types discussed in Section 3. The claim that we would like to make is that signature-matching rules given above are not arbitrary; they are sufficient to preserve important properties of the semantics of Russell. Below we will argue that the signature-matching rules given above satisfy a correctness criterion based on the semantics of data types that we have developed in the previous sections of this paper. This form of argument gives a much stronger meaning to the claim *Russell is strongly typed.*

The easiest way to understand the effect of the Russell signature-matching rules is to consider the effect of evaluating a program that violated them. For instance, consider the following very simple example:

```
    let x == Integer$New[]
     in [x] Integer$← [Boolean$True[]]; if x=0 ⇒ 3 # x ≠ 0 ⇒ 17 fi ni
```

(we have abbreviated the program by omitting obvious component selections from the type `Integer` in the final conditional statement; instead of `[x] Integer$= [Integer$0[]]`, we have simply written `x=0`). The assignment statement `x ← Boolean$True[]`, which uses `Integer` assignment to assign a `Boolean` value is clearly a violation of the signature-matching rules of the previous section. But, if we just ignore the signature violation and consider the evaluation of this program, our typeless semantics argues that it will terminate and produce either 3 or 17. Which value is produced, however, depends on the encodings of the value that represents the `Boolean True` and the procedure used by `Integer` to test for 0; in other words, the result depends on how the `Integer` operations *misinterpret* a value that was intended only to be used by the `Boolean` operations. Note that the example of the section on signature legality shows that guaranteeing that no misinterpretation occurs is not as simple as checking the syntactic identity of signatures in Russell.

Another way to look at the possibility of misinterpretation is in terms of "representation independence." If it is impossible for `Integer` (or any other type) to interpret values intended solely for the Boolean operations, then the `Boolean` operations can be implemented using any choice of representation that is convenient (as long as the abstract properties of the operations are satisfied). This representation choice is safely encapsulated inside the collection of `Boolean` operations—should it later be advantageous to change it, the semantics of any legal Russell program will not change. This idea of justifying type-checking rules in terms of representation independence appears in [Reynolds74] (which was influential in the formulation of our ideas); since then it has also been discussed in [Donahue79, McCracken79, Haynes82].

Obviously representation independence can be destroyed by primitives like the Mesa LOOPHOLE operation which allows any value to masquerade as one to be interpreted by any type. Assuming that Russell has no such operations, we would like to know that the signature-checking rules preserve representation independence. Looking at the signature-checking rules of the previous section, we see that there are three things to be checked:

1. that the rules for the combining forms are consistent with their semantics. If one had been foolish enough to define the signature-checking rule for conditionals as:

$$\frac{\Gamma \vdash c_1 \sim \textbf{val boolean},\ \Gamma \vdash e_1 \sim S, \ldots,\quad \Gamma \vdash c_n \sim \textbf{val boolean},\ \Gamma \vdash e_n \sim S}{\Gamma \vdash \textbf{if } c_1 \Rightarrow e_1 \# \ldots \# c_n \Rightarrow e_n \textbf{ fi} \sim \textbf{val } \texttt{Integer}}$$

then representation independence would clearly be destroyed. Such a verification is generally a straightforward task.

2. that the rules for matching function and type signatures given in Section 4.7 preserve representation independence. It is clear that renaming and reordering preserve the semantic intent of the signature. Forgetting, on the other hand, produces a signature with more limited opportunities for use and the signature weakening rule goes the right way (stronger signatures can be weakened), so the coercion suggested by forgetting (throwing away some of the operations of a type) is consistent with the signature-matching rules.

3. that the rules are sufficient to rule out the possible misinterpretations that might arise should it be possible for the same type expression to denote different types in the same scope (as discussed in Section 4.7). The Russell rules, which manipulate type expressions as syntactic objects, can only work if the language is constrained to ensure that textually identical type expressions in the same scope always denote semantically equivalent type values.

We can informally argue that the signature-checking rules ensure that syntactic identity guarantees semantic equivalence for type expression based on the invariance of the meanings of Russell programs under certain syntactic transformations. These transformations can be viewed as equations or axioms constraining the behavior of programs. A collection of such equations for Russell can be found in [Demers83]. Their relevance to the type checking rules is described below.

Russell has been designed to guarantee that expressions in the language have the *substitution property*, described as follows. A program fragment of the form:

$$\textbf{let } x == \alpha \textbf{ in } \beta[x/y] \textbf{ ni}$$

is semantically equivalent (i.e., produces the same value and has the same effect on the state) to the fragment:

$$\textbf{let } x == \alpha \textbf{ in } \beta[\alpha/y] \textbf{ ni}$$

whenever the expression α is *variable-free* (i.e., where none of the free identifiers of α has **var** signature; equivalently if α can be given a signature using an environment Γ_{NoVar} for some Γ). Informally, this rule allows "β-reduction" of variable-free declarations—within the scope of a declaration, any or all occurrences of its left hand side (x) may be replaced by its right hand side (α) without affecting the meaning of the program fragment. Thus, evaluations of identical variable-free denotations must produce semantically identical values and must be free of observable side-effects. And since all type expressions that appear in signatures must be variable-free by the signature legality rule, the Russell rules preserve representation independence. Another consequence of this rule is that variable-free Russell programs, like the lambda calculus, have the Church-Rosser property; in particular, terminating programs cannot distinguish between call-by-value and call-by-name semantics.

The substitution property is the reason for the *import restriction* used in the signature-checking rule for function constructions: no free identifier in the body of a function may have **var** signature (the rules also applies to type constructions). By making operation expressions (functions and types) variable-free, we can have the rather strong substitution property defined above. The only variables that an expression may reference are those that appear explicitly within it; there can be no hidden state accessed through functions that manipulate global variables.

This is admittedly a restrictive rule, as it prevents functions from inspecting or modifying global variables and prohibits applications like:

```
Array [1, N, integer]
```

where `N` is a variable. The rule does not, however, prevent obtaining the intended *effect* of the above application; it is simply necessary to introduce a new identifier and bind it to the current value of `N`:

```
let VN == ValueOf N] in... Array[1, VN, integer] ...
```

Less obviously, this restriction also rules out certain primitives in the language. For example, if we added a "pointer dereferencing" operation in the language with the natural signature:

```
↑: func[p: val REF Integer] var Integer
```

then one could write the following function, which would be perfectly legal according to the signature rules given above but would destroy the substitution property for variable free expressions:

```
func[p: val REF Integer] val Integer
 { Integer$ValOf[p↑] }
```
*Note that this function only traffics in **vals**.*

Providing a less restrictive treatment of variables while preserving the ability to do purely syntactic signature-matching remains a hard problem.

6. Conclusions

In attempting to define terms like "data type," language designers should keep in mind the spirit in which such definitions should be given. We quote from Polya [73, pg. 86]:

> The mathematician [or the language designer] is not concerned with the current meaning of his technical terms, at least not primarily concerned with that. The mathematical definition *creates* the mathematical meaning.

Thus, when we define a term, we should seek a definition that creates a useful meaning, a meaning that gives insight into language features or language design decisions. On the basis of the preceding discussion, we feel our definition of data type as a set of operations providing an interpretation of values satisfies this criterion of utility:

> This treatment of types allows us to give a straightforward meaning to polymorphism. Types are values and are legal arguments to procedures, functions or types. Moreover, as we argued

in Section 3, our meaning of types subsumes the usual meaning of types as sets of values; it all depends on whether one looks at the retraction or at its range.

As this definition explicitly rules out type errors as semantic notions (values do not *have* types), the role of syntactic type-checking must be carefully explained. In Section 5, we presented a justification of the Russell signature matching rules in terms of a fundamental property of the semantics—*representation independence.*

Our study of data types has also convinced us that ignoring the possibility of polymorphism is a false simplification in language design. If a designer has a clear conception of what "data type" means to him, then the meaning of polymorphic constructs should be obvious. Indeed, the question of the meaning of type parameters seems to be a perfect touchstone for one's understanding of types.

References

[Backus78]

Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM. 1978 August; 21(8): 613-641.

[Bates82]

Bates, J. and Constable, R. Proofs as programs. Ithaca, NY: Cornell University, Computer Science Department; 1982; Technical Report TR82-530.

[Boehm80]

Boehm, H., Demers, A, and Donahue, J. An informal description of Russell. Ithaca, NY: Cornell University, Computer Science Department; 1980; Technical Report TR80-430.

[Demers80a]

Demers, A. and Donahue, J. Data types, parameters and type checking. Proceedings of the Seventh Symposium on Principles of Programming Languages; 1980 January 28-30; Las Vegas. ACM: 21-23.

[Demers80b]

Demers, A. and Donahue, J. Type-completeness as a language principle. Proceedings of the Seventh Symposium on Principles of Programming Languages; 1980 January 28-30; Las Vegas, NV. ACM: 234-244.

[Demers83]

Demers, A. and Donahue, J. Making variables abstract: An equational theory for Russell. Proceedings of the Tenth Symposium on Principles of Programming Languages; 1983 January; Austin, TX. ACM: 59-72.

[Dijkstra75]

Dijkstra, E. W. Guarded commands, non-determinancy and the formal derivation of programs. Communications of the ACM. 1975 August; 18(8): 453-457.

[Dijkstra78]

Dijkstra, E. W. On the BLUE language submitted to the DoD. SIGPLAN Notices. 1978 October; 13(10): 10-15.

[Donahue79]

Donahue, J. On the semantics of "data type." SIAM Journal of Computing. 1979 November; 8(4): 546-560.

[Goguen76]

Goguen, J. A., Thatcher, J. W., and Wagner, E. G. An initial algebra approach to the specification, correctness and implementation of abstract data types. Yorktown Heights, NY: IBM Thomas J. Watson Research Center; 1976; Technical Report RC6487.

[Gries77]

Gries, D. and Gehani, N. Some ideas on data types in high-level languages. Communications of the ACM. 1977 June; 20(6).

[Guttag75]

Guttag, J. V. The specification and application to programming of abstract data types. Toronto, ONT: Computer Systems Research Group, University of Toronto; 1975; Technical Report CSRG-59.

[Guttag77]

Guttag, J. Abstract data types and the development of data structures. Communications of the ACM. 1977 June; 20(6): 396-404.

[Haynes82]

Haynes, C. Theory of data type representation independence. Ph.D. Dissertation; Department of Computer Science, University of Iowa; 1982.

[Hoare72]

Hoare, C. A. R. Notes on data structuring. In: Dahl, Dijkstra and Hoare, Structured programming. New York: Academic Press; 1972: 83-174.

[Jensen75]

Jensen, K. and Wirth, N. Pascal user's manual and report. New York: Springer-Verlag, 1975.

[Lampson77]

Lampson, B., Horning, J., London, R., Mitchell, J., and Popek, G. Report on the programming language Euclid. SIGPLAN Notices. 1977 February; 12(3).

[Landin66]

Landin, P. J. A formal description of Algol60. In: Steele, ed. Formal language description languages. Amsterdam: North-Holland Publishing; 1966.

[Liskov77]

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction mechanisms in CLU. Communications of the ACM. 1977 August; 20(8): 564-576.

[Martin-Lof79]

Martin-Lof, P. Constructive mathematics and computer programming. Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science; Hanover; 1979 August.

[McCracken79]

McCracken, N. An investigation of a programming language with polymorphic type structure. Syracuse, NY: Ph.D. Dissertation, School of Computer and Information Science, Syracuse University; 1979.

[Milne76]

Milne, R. and Strachey, C. A theory of programming language semantics. New York: Halstead Press; 1976.

[Milner78]

Milner, R. A theory of type polymorphism in programming. Journal of Computer and System Sciences 17. 1978; 348-375.

[Polya73]

Polya, G. How to solve it. Princeton, NJ: Princeton University Press; 1973.

[Reynolds74]

Reynolds, J. Towards a theory of type structure. Paris: Colloquium on Programming; 1974.

[Reynolds78]

Reynolds, J. Syntactic control of interference. Proceedings of Fifth Symposium on Principles of Programming; Tucson, AZ; 1978 January 23-25; ACM: 39-46.

[Scott76]

Scott, D. Data types as lattices. SIAM Journal on Computing. 1976 September; 5(3): 522-587.

[Scott77]

Scott, D. Logic and programming languages. Communications of the ACM. 1977 September; 20(9): 634-641.

[Tennent77]

Tennent, R. Language design methods based on semantic principles. Acta Informatica. 1977; 8(2): 97-112.

[Wulf76]

Wulf, W., London, R., and Shaw, M. Abstraction and verification in Alphard: Introduction to language and methodology. In: Shaw, M., ed. ALPHARD: Form and content, New York: Springer-Verlag; 1981.

[Wulf77]

Wulf, W. Private communication.

[Wulf78]

Wulf, W. (ed.) An informal definition of Alphard. In: Shaw, M., ed. ALPHARD: Form and content. New York: Springer-Verlag; 1981.

XEROX

Data Types Are Values

James Donahue and Alan Demers

CSL-83-5