Data Types, Parameters and Type Checking

Alan J. Demers
James E. Donahue

Department of Computer Science
Cornell University
Ithaca, New York 14853

1. Introduction

In statically typed programming languages, each variable and expression in a program is assigned a unique "type" and the program is checked to ensure that the arguments in each application are "type-compatible" with the corresponding parameters. The rules by which this "type-checking" is performed must be carefully considered for modern languages that allow the programmer to define his own data types and allow parameterized types or types as parameters. (Such languages include Alphard [Wulf78], CLU [Liskov77], Euclid [Lampson77] and Russell [Demers79].) These features increase the expressive power of the languages, but also increase the difficulty of type-checking them.

In this paper, we describe a treatment of type-checking that makes it possible to do completely static checking with a general parameterization mechanism allowing parameterized types, types as parameters, and even a disciplined form of self-application. Our method defines a calculus of "signatures," where signatures are similar to the "program types" of [Reynolds78]. Each identifier and expression is given a signature, and applications are type-correct when argument and parameter signatures are equivalent under a simple set of signature transformation rules. Below we present the signature calculus of Russell; we also present a semantic justification of this calculus and specify the language constraints necessary for us to justify our purely static approach to type-checking.

2. An Overview of Types in Russell

The treatment of type-checking described in this paper is used in the Russell programming language, described in [Demers79]. The type structure of Russell is based on a novel view of the meaning of a data type, presented in [Demers78], and succinctly described as follows: a data type is a set of named operations that provide an interpretation of the values of a single universal value space common to all types. For the purposes of this paper, the important consequence of this definition is that it allows us to

This work supported by National Science Foundation grant MCS-7901048.

treat data types as values, in complete analogy with procedure and function values in conventional languages.

The treatment of data types as values leads to a pleasing uniformity in the parameterization mechanism of Russell. The Russell semantics gives a meaning for the value of any identifier or expression, including types and variables (variables are treated essentially as reference values). Thus, any construction in the language may be parameterized with respect to any of its free identifiers (even type identifiers) using a straightforward call-by-value semantics. (The benefits of this "type completeness" as a language design principle are discussed in the other paper in this proceedings by the authors [Demers80a]; here we are concerned solely with its implications for type-checking.) In addition, following Landin's "Principle of Correspondence" [Landin66], we treat declaration and parameterization as semantically equivalent. Thus, the language has only a single, straightforward semantic mechanism (call-by-value parameter passing) for the introduction of new names. In particular, it is not necessary to treat type declarations and type parameters as special cases, as must be done done in Euclid, Alphard and CLU.

Finally, we note that every value in a Russell program either is treated as an operation (i.e. a procedure, function or type) or else is interpreted by the operations of some type. Thus, the set of primitive combining forms in Russell is extremely small, consisting of those forms needed to interpret operation values: selection of a component of a type, and application of a procedure or function to a list of argument values.

3. The Need for Static Type Checking

The purpose ascribed to type-checking depends on one's view of what a data type is. If types are viewed as sets of values (as in Pascal) or as sets of values and operations (as in [Morris73], or the "many-sorted algebras" of [Goguen76]), an attempt to apply an operation to a value of the wrong type produces an erroneous result (i.e., a "run-time error"). Thus, type-checking is considered an essentially redundant way of predicting at compile-time that a program will produce an error when executed.

In our view of data types, however, all types share a single universal value space. Any value may be interpreted as belonging to any type, or even as being a type. Thus, a type-erroneous application does not necessarily generate an error; it simply produces a spurious result by "misinterpreting" its argument. Note that this view reflects the situation in a typical language implementation on a vonNeumann machine -- values are represented as (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. In this framework, static type-checking takes on a more essential role: it is necessary in order to guarantee that values produced by one type will not be misinterpreted by operations of another type.

In practice there is not much difference between these points of view for simple languages. When static type-checking is sufficient to guarantee that no type-erroneous application will be

attempted, the representations of values of distinct types need not be distinct -- e.g., most Pascal implementations will represent both real and integer values in a single word. However, our view of types suggests that, no matter how elaborate the type structure of a language becomes, it must still be possible to do all type-checking statically, since no additional information will be available at run time to facilitate it.

4. Problems in Traditional Type Checking Methods

In a programming language that provides only a finite set of builtin types, the type-checking rules can be specified exhaustively. Frequently these rules are laden with ad hoc coercion rules and run-time consistency checks. (The PL/I arithmetic rules are an outstanding example.) Flexible type definition and parameterized type mechanisms, however, make it essential that a programming language design be based on a clear understanding of the meaning of data types and type-checking. The use of data type definitions means that the set of types with which the type-checker must deal is no longer finite; thus, the type-checking rules cannot be an exhaustive list of special cases. Instead, the rules must show how the result of any type constructor in the language behaves with respect to existing (builtin or user-defined) types. That this is not straightforward is seen in the inconsistencies between the type-checking rules of "Algol-like" languages, in the complexity of the rules, and in the difficulty of providing semantic justifications for them.

Below we describe several aspects of type-checking in which this problem is apparent. Two basic problems arise: (a) types that are "obviously" semantically equivalent but have distinct denotations, and (b) type denotations whose meaning depends on (run-time) argument values. The treatment of types as values and the Principle of Correspondence suggest a solution to (a) which appears in [Demers78] and will be discussed only briefly here. Problem (b) greatly influenced our syntactic treatment of type-checking; it also motivated the scope and import rules of Russell, described and justified later in this paper.

Consider a program like the following in which a new type is declared to be "identical" to an existing type:

```
let
   T {type ... } == Integer
   T' {type ... } == Integer
in let
      x {var T} == ...
      y {var T'} == ...
   in
      ... x := y ...
```

(In Russell syntax, braces enclose "signatures," or syntactic types; thus T and T' are types, x is a variable of type T, and y is a variable of type T'.) Whether this program is considered legal depends on whether types T and T' with identical definitions are considered equivalent by the type-checking rules. Alphard and Euclid give different answers to this question, while the Pascal report says nothing at all.

By the Principle of Correspondence, the above program in Russell is equivalent to one in which the type declarations are replaced by parameters:

```
let
   P == proc[ T,T'{type}]
          let
             x {var T} == ...
             y {var T'} == ...
          in
             ... x := y ...
          end
in
   P[ Integer, Integer ]
```

A type-checking system based on "macro-expansion" semantics (as used in Alphard) may or may not declare the above program correct. However, if the call

P [Integer, Real]

is added, then the program as a whole must be considered type-incorrect, though it is unclear whether it is the body of P or the call which is invalid. Our desire to treat type parameters uniformly with other kinds of parameters leads us to conclude that the body of P is invalid. Since we can type-check and give the meaning of an ordinary procedure independent of any calls of the procedure, we must be able to do the same for a polymorphic procedure. This requirement precludes a macro-expansion interpretation of polymorphism and leads to a set of type-checking rules in which distinct type names are never treated as equivalent.

Parameterized types introduce a second kind of problem in which textually identical type expressions may denote different types because of a change in the value of some variable. The usual interpretation of equivalence for two applications of a parameterized type like array is that corresponding argument values must be equal. Thus, the assignment x := y, where the types of x and y are

array [1..n_a] of integer

and

array [1..n_b] of integer

is legal only when n_a and n_b have the same value. For arbitrary expressions, it is clearly undecidable whether this will always (or ever) be the case; thus, asking that the run-time values of type arguments be equal makes static type-checking of such an assignment impossible. Further complications arise if, as in Russell, the programmer can redefine the meaning of equality for any type. Conventional approaches to this problem either insist that arguments to parameterized types be manifest constants or defer type-checking to run time in such cases. Each approach has its drawbacks. Limiting type arguments to manifest constants severely restricts the programs that can be written (this is a major problem in Pascal). However, as was argued in Section 3, our view of types precludes run-time type-checking. Thus, we have been led to devise language restrictions (in the form of scope and import rules, described below) to ensure that identical type expressions denote equivalent types without insisting that all type arguments be manifest constants.

5. Type-Checking in Russell

In this section we present the type-checking rules of Russell. To escape the undecidability problems described above, the Russell type-checking rules avoid the use of run-time values of expressions, without demanding that all type arguments be manifest constants. Instead, the rules

are applied to uninterpreted type expressions as purely syntactic forms. Each identifier in a Russell program is given a signature (similar to a "syntactic type" of [Reynolds78].) The signatures of expressions are determined from the signatures of their components by a set of purely syntactic composition rules. Finally, each procedure or function application is checked to ensure that corresponding argument and parameter signatures are equivalent under a simple calculus of signature transformation rules described below.

To justify the claim that our type-checking rules are sufficient to prevent misinterpretation of values, the syntactic transformations of the signature calculus must be provably "interpretation-preserving." Also, the language must be constrained to guarantee that syntactically identical type expressions in the same scope (which would be equivalent under the signature calculus) are semantically equivalent. In Russell, a simple import rule (discussed below) is sufficient to guarantee that identical type expressions are equivalent.

5.1. Signatures

In Russell, each identifier and expression in a program is given a syntactic type or "signature." A signature may describe a variable, a value, or an "operation" -- procedure, function or type.

```
Signature ::= var TypeDenotation
            | val TypeDenotation
            | OperationSignature
```

In this context a TypeDenotation is any expression that has type signature according to the signature composition rules described below. In particular, the TypeDenotation may contain free identifiers, subject to the import rule described below. For example,

```
var Stack[ N+M, Integer ]
    and
var Stack[ M+N, Integer ]
```

are valid (and distinct) signatures.

The signature of a procedure or function gives an identifier and signature for each parameter, plus a signature for the result, if there is one. (In Russell, functions may produce results having any signature, including variables, functions or data types.) Since we view a type as a set of operations, the signature of a type is simply a collection of operation signatures, with an identifier naming each component operation, and a bound identifier (or "local name"), which the components may use as a type denotation meaning "the type in which the component appears."

```
OperationSignature ::=
      proc[  Id{Signature}  +;  ]
    | func[  Id{Signature}  *;  ] Signature
    | type Id(  Id{OperationSignature}  +;  )
```

The need for an identifier associated with each parameter of a procedure or function arises because a parameter name may appear free in the signature of other parameters or the result. For example, a polymorphic function might have signature

```
func[
    T { type t( ... ) };
    x { val T }
] val T
```

where the type of the second parameter and of the result depend on the first argument. A similar technique is used in the polymorphic lambda calculus of [Reynolds74], where a "Curried" version of the above signature would be written

$$(\Delta T.\ T \to T)$$

5.2. Signature Composition

The rules for composing signatures of composite expressions are completely natural if one bears in mind that they deal with signatures as syntactic objects, and not as values. As mentioned above, there are two primitive forms of composition in Russell: selection of a component of a type, and application of a function to a list of argument values.

5.2.1. Selections

Consider selection of the "f" component of a type T, where T is some (arbitrarily complex) type denotation with signature

```
type t ( ... f { sig } ... )
```

The signature of the selection T $ f is obtained by textual substitution of the type denotation T for the local name t in the signature of the f component:

```
      | T
  sig |
      | t
```

For example, if Integer has signature

```
type I ( ...
    + {func[ x,y{val I} ] val I}
    ... )
```

then Integer$+ has signature

```
func[ x,y{val Integer} ] val Integer
```

Selection illustrates the importance of local names in type signatures: if another type T has the same signature as Integer, the signature of T$+ is

```
func[ x,y{val T} ] val T
```

This allows operations selected from T to be applied to T values rather than to Integers.

5.2.2. Applications

The composition rule for function application also involves textual substitution of denotations for bound identifiers. Consider an application

$$F(a_1, \ldots, a_k),$$

where F is an (arbitrarily complex) denotation with signature

```
func [
    p1{psig1}; ... pk{psigk}
] rsig
```

The result signature is obtained by textual substitution of argument denotations for the corresponding parameter identifiers in the result part of the signature of F:

```
       | a1, ..., ak
       |
  rsig |
       |
       | p1, ..., pk
```

Thus, an application of the form f[Integer,10] where f has signature

```
func[ T{type t(...)}; n{val T} ] val T
```

would have signature val Integer. No "type-

checking" is involved here; the result signature of a type-erroneous application is a perfectly well-defined syntactic object (which may itself contain type-erroneous applications, of course).

5.3. Type-Checking of Applications: The Signature Calculus

To type-check a Russell program, we must guarantee that the signatures of the arguments and parameters match in function or procedure applications. Every parameter in a Russell program has an explicit signature; and every argument can be given a signature using the composition rules described above. Thus, the Russell type-checking rules operate by transforming and comparing signatures. Note that since types themselves have values and signatures, type parameters require no special treatment.

Type-checking an application proceeds in two steps: expansion of parameter signatures by argument denotations, and matching of argument and parameter signatures. The application is legal iff each argument signature matches the corresponding expanded parameter signature.

Consider an application

$$P[a_1, \ldots, a_k],$$

where P has signature

$$\underline{\text{proc}}\ [\ p_1\{psig_1\};\ \ldots;\ p_k\{psig_k\}\ ]$$

and the arguments have signatures

$$asig_1, \ldots, asig_k$$

First the parameter signatures are expanded by textual substitution of arguments for the corresponding parameter identifiers; thus, the i^{th} expanded parameter signature is

$$psig_i \Big|_{p_1, \ldots, p_k}^{a_1, \ldots, a_k}$$

Signature expansion is used so that the parameter signatures may be modified to reflect the interrelationships among the arguments of a particular call of an operation. For example, if g has signature

$$\underline{\text{proc}}[\ n\{\underline{\text{val}}\ \text{Integer}\};\ x\{\underline{\text{var}}\ T[n]\}\ ]$$

then the expanded parameter signatures used in matching the call g[a+b,y] would be

$$\underline{\text{val}}\ \text{Integer}\ \ \text{and}\ \ \underline{\text{var}}\ T[a+b]$$

An argument and parameter are said to match iff the argument signature can be transformed to the parameter signature using a small set of syntactic transformations, the signature calculus. The signature calculus rules are:

a) Renaming. The local name on a type signature, or the parameter names in a procedure or function signature, may be uniformly replaced by any new identifier. Thus, for example, the signature

$$\underline{\text{func}}\ [x,y\{sig_1\}]\ sig_2$$

matches the signature

$$\underline{\text{func}}[\ a,b\{sig_1 \Big|_{x,y}^{a,b}\}]\ sig_2\Big|_{x,y}^{a,b}$$

The substitutions are necessary because x and y may appear free in the signatures sig_1 and sig_2.

b) Reordering. Two type signatures match if the signatures include all the same component names and identically-named components have matching signatures. For example, the signa-

ture

$$\underline{\text{type}}\ T\ (\ a_1\{sig_1\};\ a_2\{sig_2\}\ )$$

matches

$$\underline{\text{type}}\ T\ (\ a_2\{sig_2'\};\ a_1\{sig_1'\}\ )$$

if sig_1 matches sig_1' and sig_2 matches sig_2'.

c) Forgetting. An argument type signature may be simplified by eliminating ("forgetting") some of its operations. Thus, the argument signature

$$\underline{\text{type}}\ T\ (a_1\{sig_1\};\ a_2\{sig_2\}\ )$$

can be reduced by "forgetting" a_2 to

$$\underline{\text{type}}\ T\ (a_1\{sig_1\}\ )$$

This rule is the sole means of "encapsulating" data type definitions required in Russell.

All the above rules are straightforward, easily explained and can obviously be shown to be interpretation-preserving. It is important to note the absence of any transformation rule for variable or value signatures -- to match, two variable or value signatures must be textually identical. There are no ad hoc transformations based on knowledge of the behavior of particular types. This rule has the advantage of simplicity, especially when contrasted with the "type compatibility" rules of Euclid [Lampson77, pp. 31-32] or the "type subsumption, syntactic satisfaction and implicit binding" rules of Alphard [Wulf78, pp. 11-13].

5.4. Scope and Import Rules

Above we described an approach to static type-checking based on the manipulation of type expressions as syntactic objects. Clearly, such an approach can only work if language constraints exist to ensure that textually identical type expressions always denote semantically equivalent type values. At the same time, these constraints must not be so restrictive as to prevent e.g. the computation of arguments to parameterized types. A formal definition of "semantically equivalent" type values requires a formal semantics for the language; this is currently in preparation [Demers 80b]. However, we can ensure the correctness of our type checking rules in a way that is largely independent of the details of the formal semantics by insisting that the meanings of Russell programs be invariant under certain syntactic transformations. Our approach to this is described below.

The scope and import rules of Russell have been designed to guarantee that denotations have the substitution property, described as follows. Let D be any denotation appearing in a legal Russell program, and let D be variable-free (i.e., neither D nor any of the free identifiers of D has var signature). Define the scope of D to be the smallest enclosing scope in which all free identifiers of D are bound; this has the form

$$\begin{array}{l}\underline{\text{let}}\\ \quad \alpha\\ \underline{\text{in}}\\ \quad \beta\end{array}$$

The substitution property demands that an equivalent program results if all occurrences of D are replaced by a new identifier bound to the value of D:

```
let
    α;
    x == D
in
      |x
      |
    β |
      |
      |D
```

Informally, this rule requires that evaluations of identical variable-free denotations must produce semantically identical values and must be free of observable side-effects. Note that a 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.

In Russell, the substitution property is achieved by enforcing the following rules.

1. The builtin types have the substitution property. This constraint affects the signatures as well as the meanings of certain builtin types. For example, the "obvious" signature for the dereferencing operation Ref[Integer]$↑ is

```
func [{val Ref[Integer]}] var Integer
```

This signature would be unacceptable, as it can easily be used to write variable-free denotations that do not have substitution property (using any plausible semantics); for example:

```
... ValueOf[ p↑ ] ...
p↑ := ValueOf[ p↑ ] + 1
... ValueOf[ p↑ ] ...
```

The remedy in this case is to introduce a type Heap, analogous to an untyped Euclid collection, and require a Heap variable as the first argument of a dereferencing operation. With this change, Ref[Integer]$↑ has signature

```
func [
    {var Heap}; {val Integer}
] var Integer
```

and cannot be embedded in a variable-free denotation as was done in the example above.

2. No identifier may be redeclared in a scope in which it is accessible. This "unique visibility" rule is necessary to avoid capture of free identifiers of the signature of an argument when that argument is passed to an operation declared in an outer scope. For example, it prohibits such (clearly incorrect) programs as

```
let
    T == ...
    P == proc[x{var T}]  ...
in  let
        T == ...
        y {var T} == ...
    in begin
        ... P [ y ] ...
```

which without the unique visibility rule would be considered legal.

3. No free identifier in an operation (i.e., procedure, function or type) denotation may have var signature. This rule simply ensures that all operation denotations are variable-free. This is the most restrictive of the three rules, as it prevents procedures 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 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 ] . . .
```

5.5. Higher Order Variables and Self-Application

In our discussion of type-checking we have made a careful distinction between signatures and data types. Data types are sets of operations, and have values; signatures are purely syntactic constructs used by the type-checker, and do not have values. In particular, an operation signature like

```
func [{val T}] val T
```

is not a type; thus,

```
var func [{val T}] val T
```

is not a legal signature, and there are no operation-valued variables.

It is difficult to see how operation-valued variables could be introduced directly into the Russell type-checking framework. For example, suppose we somehow managed to produce a type F that interpreted values as T-to-T functions. The operation of "taking the value of an F variable" would have signature

```
func[{var F}]
    func[{val T}] val T
```

Any nontrivial application of this function would be an operation denotation (since the result is a function) that imported a variable (the argument), and thus would violate the import rule.

It is possible to add operation-valued variables to Russell using the image constructor. Let sig be any operation signature; then

```
image( sig )
```

is a type with signature

```
type I (
    New{ func[] var I }
    :={ proc[{var I},{val I}] }
    ValueOf{ func[{var I}] val I }
    In{ func[{sig}] val I }
    Out{ func[{val I}] sig }
)
```

For example, the type

```
F' == image( func[{val T}] val T )
```

is similar to the type F shown above to violate the import rule. The key difference is that the ValueOf operation of F' yields a val F' rather than a function; to obtain a function value it is necessary to apply the Out function. Direct conversion of an F' variable to a function by composition of ValueOf and Out, e.g.

```
F'$Out[ F'$ValueOf[ x ] ]
```

still violates the import rule; however, in most cases the same effect can be achieved by first binding the value of the variable to a new identifier and then applying the Out function:

```
let
    vx == F'$ValueOf[ x ]
in
    ... F'$Out[ vx ] ...
```

Since vx has signature val F', the import rule is satisfied and type-checking is unaffected.

The image constructor allows self-application

in Russell programs. For example, the (recursive) type declaration

```
T == image( func[{val T}] val T )
```

is perfectly well-behaved. An example of an expression of this type is the following T-to-T identity function:

```
f == T$In [
   func[ x{val} T] val T
      result x end
]
```

Clearly f can be applied to itself in a type-correct way, e.g.

```
( T$Out[f] )[ f ]
```

From this example we can conclude that, with our view of data types, type-correctness and self-application are unrelated; the fundamental goal of typechecking -- to prevent misinterpretation of values -- can be achieved even if self-application is allowed.

6. Conclusion

The ability to define new data types and to produce objects of arbitrarily complex kinds by parameterization greatly increase the difficulty of type-checking. In this paper we investigate the problems of type-checking in the presence of these features, and present an approach that allows static type-checking in the presence of completely general type definition and parameterization mechanisms. The type-checking rules we present are simple and, most importantly, are based on the firm ground of a semantic characterization of data types. Thus, a basic test of the "correctness" of the Russell design has been to guarantee that the type-checking rules are sufficient to prevent any combination of language features from being used to "misinterpret" a value. The obligation to prove this property of the language makes it advantageous to look for a few general mechanisms rather than a large number of unrelated features.

7. References

[Demers78]
Demers, A., J. Donahue and G. Skinner. Data Types as Values: Polymorphism, Type-Checking, Encapsulation. Proceedings Fifth Annual Principles of Programming Languages Symposium, 1978, pp. 23-30.

[Demers79]
Demers, A. and J. Donahue. Revised Report on Russell. Report TR79-389, Computer Science Department, Cornell University, September 1979.

[Demers80a]
Demers, A. and J. Donahue. Type-Completeness as a Language Principle. Proceedings Seventh Annual Principles of Programming Languages Symposium, 1980.

[Demers80b]
Demers, A. and J. Donahue. A Formal Semantics for Russell. (in preparation)

[Goguen76]
Goguen, J.A., J.W. Thatcher and E.G. Wagner. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. Report RC6487, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1976.

[Landin66]
Landin, P.J. The Next 700 Programming Languages. Comm. ACM 9:3 (1966).

[Liskov77]
Liskov, Barbara, Alan Snyder, Russell Atkinson and Craig Scaffert. Abstraction Mechanisms in CLU. Comm. ACM 20:8 (August 1977).

[Morris73]
Morris, James H. Types are Not Sets. Proceedings First Annual Principles of Programming Languages Symposium, 1973, pp. 120-124.

[Reynolds74]
Reynolds, John. Towards a Theory of Type Structure. Colloquium on Programming, Paris, 1974.

[Reynolds78]
Reynolds, John. Syntactic Control of Interference. Proceedings Fifth Annual Principles of Programming Languages Symposium, 1978, pp. 39-46.

[Wulf78]
Wulf, W. A. (ed.) An Informal Definition of Alphard. CMU-CS-78-105, Department of Computer Science, Carnegie-Mellon University, 1978.