An Informal Description of Russell

H. Boehm
A. Demers
J. Donahue

TR 80-430
October 1980

Department of Computer Science
Cornell University
Ithaca, New York 14853

This work is supported in part by National Science Foundation grant MCS79-01048.
0. Introduction

0.1. The Russell Project

The Russell project was begun to explore the semantics of data types in programming languages. Work has proceeded along two major lines:

1. the development of a new semantics of data types, described in [Demers 78, Donahue 79] and
2. the design of a programming language incorporating this new semantics.

Some of the ideas developed in the course of the language design effort have been discussed in [Demers 80a, 80b]. A preliminary draft of the Russell language appeared as [Demers 79]. This report and the accompanying formal semantics [Demers 80c] specify a revised Russell language based on our previous report, but modified to reflect our continuing experimentation with the language.

The major changes of this "new" Russell from the language of [Demers 79] are simplifications and generalizations of constructs from the earlier language. For example, type modifications have been given a much simpler semantics, capsules have been replaced by a more straightforward record construction and the previous (and arbitrary) distinction between procedures and functions has been removed. The size of the resulting language is such that we expect no further shrinking to take place; thus, we have produced formal and informal descriptions of the current (and probably last) Russell.

0.2. Principles

The major goal of the design of Russell was to build a small language in which generality was achieved by regularity. Russell has only a very few primitives; the power of the language comes from the uniformity of the syntax and semantics, which allows these primitives to be composed to produce many of the features that must be "builtin" in most other languages. For example, the subrange types of Pascal [Jensen 75] are not part of the primitives of Russell, but can be constructed through the general means of defining type-producing functions. The semantic uniformity of Russell is made possible by the treatment of data types as values; the syntactic uniformity results from the "type-completeness" of the Russell design.

0.2.1. Data Types as Values

In most programming language descriptions, one finds the statement that types define sets of values, i.e., each value belongs to a particular set and these sets of values are called types. This is not the way types are treated in Russell. Values do not have type; we assume the value space over which Russell programs are interpreted to be typeless.

Instead, a data type in Russell is a set of operations that provides an interpretation of values (of some arbitrary value space). By operations, we mean the usual procedures and functions one finds in common programming languages. In Russell operations and sets of operations are values; they therefore are also legal arguments to other operations. The germ of the idea
that types can be treated as values appears in [Scott 76], in which the data types of a "universal" value space are regarded as functions that are elements in the universal space. In Russell, we have just applied this basic idea to a language with a far richer type composition mechanism.

Thus, values in Russell do not have "type" (we are not concerned about the set from which they are drawn). However, data types in Russell are values. The value of a type provides an interpretation (in the sense of "bringing out the meaning") of the values stored in variables or bound to identifiers.

Treating data types as values allows us to build a language with only a very few combining forms. For example, Russell uses a uniform semantics for declaration and parameter-passing: in each case, a value is bound to an identifier, regardless of the use that is made of the identifier. (The meaning of a type declaration is given by the same set of rules that hold for passing a simple value to a function.) Likewise, the composite forms of expressions in Russell (e.g., the conditional) can be used to produce values that are data types just as they can be used to produce values that are to be interpreted in other ways. (So, subranges are not needed, because they can be written as a function that produces a data type using the same construction that is used to build other sorts of functions in the language.)

Because the Russell value space is typeless, the meaning and importance of type-checking in Russell differs from that found in other languages. We do not view type-checking as a mechanism to detect statically the possibility of "type faults" during program execution. Indeed, the typelessness of the Russell value space allows any value to be used in any context; the semantics does not differentiate between an Integer value and a function value. Instead, the static type-checking in Russell guarantees "representation independence" -- no Russell program can be written that allows spurious results to be produced by "misinterpretation" of a value (e.g., treating a value meant to be used only by Boolean operations as a type-producing function.)

0.2.2. Signatures and Type Completeness

The semantic consistency of Russell is mirrored in its syntactic structure. Russell has been designed to be "type-complete":

1. Each expression (or denotation) in the language has a syntactic type (or signature). The signature of a denotation is determined solely by the signatures of its component denotations and does not depend on the context in which the denotation appears.

2. For each signature, it is possible to write a denotation in the language having that signature. Thus, the signature structure is both necessary and sufficient to describe all of Russell.

3. Any legal denotation can be the body of a function and any identifier that appears in the denotation may be made a parameter of the function. Thus, no constraints are imposed on the legal signatures of function parameters or function results.
The following classes of signatures are allowed:

1. **Value signatures**, which specify that the value associated with an identifier is to be interpreted by the (operations of) the type specified in the signature.

2. **Variable signatures**, which specify that the value associated with an identifier is to be interpreted as a "reference" to a value. The value referred to by the variable is to be interpreted by the (operations of) the type specified in the signature.

3. **Function signatures**, which specify that the value may be applied to a set of **arguments** to produce a value. A function signature specifies a fixed set of **parameters**, names that are local to the function and may be associated with different arguments at each application. (Because functions may take variables as arguments and yield variables as results, there is no separate syntactic category of "procedures" in Russell.)

4. **Type signatures**, which specify a set of operation names (i.e., functions, procedures or types) and their signatures.

The importance of type-completeness is discussed in detail in [Demers 80b]. In Russell, type-completeness allows us to:

1. build a consistent framework for the language. Every built-in data type and function and every composition of denotations has a signature. The legality of a Russell program is determined solely by the signatures of the denotations appearing in it.

2. use general forms of combination to include many special features of other languages. For example, call-by-name, variable allocation, parameterized data types and field selectors are all just special cases of functions in Russell.

The "type-checking" rules for Russell are defined in terms of a calculus of signatures. The signature of each argument to a function must "match" the signature of the corresponding parameter. That is, it must be possible to transform the signature of the argument expression to the parameter signature using the three rules of our signature calculus. The signature calculus for Russell is restrictive enough to guarantee that every legal Russell program must have a "representation-independent" semantics as described above. This property of representation independence of legal Russell programs is proven in a forthcoming report.

Russell allows the definition of data types for which self-application is legal (this can be done because of our use of a typeless value space). The type-checking rules imposed by the signature calculus do not rule out the possibility of meaningful self-application. They disallow only misinterpretations of values.

**0.3. Organization of This Report**

In the next section we describe the lexical conventions of Russell and the notation used in this Report. Sections 2 specifies signatures in Russell.
Section 3 gives the syntax and meaning of denotations, the means by which values are produced in a Russell program. Section 4 specifies the meaning of the predefined identifiers in Russell. Section 5 then gives the meaning of Russell programs.
1. **Notation and Syntactic Conventions**

1.1. **Notation**

The Russell syntax is presented in a modified form of BNF defined as follows:

A sequence of letters is a nonterminal symbol (i.e., a syntactic variable).

Any underlined symbol is a terminal symbol, which appears literally in the program.

The meta-symbols "::=" and "|" are used in the conventional way to mean "may be replaced by" and "or," respectively.

Braces ("{" and "}") are meta-symbols used for grouping.

Lists and optional elements are indicated by superscripts and subscripts in the manner described in [Wulf 78]. A superscript "#" indicates an optional element. A superscript "*" indicates a (possibly empty) list, and a superscript "+" indicates a (nonempty) list. The separator character for a list may be specified as a subscript. Thus,

\[
\text{export Id}^{\#} \{ \text{ExportElement}^+ \}
\]

denotes the keyword "export" followed by an optional identifier followed by 
"{" followed by one or more ExportElements separated by semicolons followed by "}".

1.2. **Vocabulary**

The primitive vocabulary of Russell consists of six sets of basic symbols: letters, digits, punctuation, operators, white space, and keywords. These sets include the following symbols:

- **Letter** ::= a | b | c | ... | z | A | B | ... | Z
- **Digit** ::= 0 | 1 | 2 | ... | 9
- **PunctuationChar** ::= $ | ( | ) | [ | ] | | | | ; | | | < | > | |
- **OpChar** ::= + | - | * | + | < | = | > | : | & | | | ~ | / | . | ^
- **WhiteSpace** ::= <blank>

**KeyWord** ::= **and** | **cor** | **do** | **else** | **enum** | **export** | **extend** | **field** | **fi**
| **func** | **hide** | **if** | **image** | **in** | **let** | **ni** | **od** | **record** | **type**
| **union** | **val** | **var** | **with** | == | =>

Keywords are considered basic symbols, though in most implementations (as well as in this report) they will be represented by reserved sequences of characters.
1.3. **Lexical Conventions**

A Russell program is a sequence of **tokens** -- identifiers, keywords, strings, and punctuation characters -- possibly separated by white space. Keywords, punctuation characters and white space are part of the basic vocabulary of the language; identifiers and strings are defined as follows:

\[
\begin{align*}
\text{Id} & ::= \text{WordId} \mid \text{OperatorId} \mid \text{QuotedId} \\
\text{WordId} & ::= \text{Letter Alphameric}^* \\
\text{Alphameric} & ::= \text{Letter} \mid \text{Digit} \\
\text{OperatorId} & ::= \text{OpChar}^* \\
\text{QuotedId} & ::= ' \text{Character}^* ' \\
\text{String} & ::= \text{Number} \mid \text{QuotedString} \\
\text{Number} & ::= \text{Digit Alphameric}^* \\
\text{QuotedString} & ::= " \text{Character}^* " \\
\text{Character} & ::= \text{Alphameric} \mid \text{PunctuationChar} \mid \text{OpChar} \mid \text{WhiteSpace}
\end{align*}
\]

The construct (* Text *) is a **comment**. A comment may appear anywhere WhiteSpace is legal (except within character constants), and does not affect the meaning of the program. Comments may be nested; thus any comment delimiters appearing in the text of a comment must be balanced.

1.4. **Syntactic Kinds**

The Russell grammar is parameterized according to the following syntactic kinds:

\[
\begin{align*}
\text{Val} & \quad \text{(value)} \\
\text{Var} & \quad \text{(variable)} \\
\text{Func} & \quad \text{(function)} \\
\text{Type} & \quad \text{(datatype -- i.e., set of operations)}
\end{align*}
\]

Each rule of the grammar is really a production scheme, and is instantiated by replacing each of the **kind variables** occurring in it (which are identified by being enclosed in square brackets) by one of the above kinds. The legal replacements are:

\[
\begin{align*}
[\text{Operation}] & ::= \text{Func} \mid \text{Type} \\
[\text{Any}] & ::= \text{Var} \mid \text{Val} \mid \text{Func} \mid \text{Type}
\end{align*}
\]

Replacements must be done uniformly -- i.e., each occurrence of a given kind variable in a rule must be replaced by the same kind. Thus, for example, the
rule

\[\text{[Operation]Denotation ::= [Operation]Construction}\]

stands for the set of context-free productions

\[
\begin{align*}
\text{FuncDenotation ::= FuncConstruction} \\
\text{TypeDenotation ::= TypeConstruction}
\end{align*}
\]

but not for

\[
\begin{align*}
\text{FuncDenotation ::= TypeConstruction}
\end{align*}
\]
2. **Signatures**

Each denotation in a Russell program has a signature, which specifies the legal use of the denotation. The signature of an identifier is given when the identifier is introduced, either in a declaration that is part of a block or type modification or in a parameter list that is part of a function construction. The signature of a composite denotation is formed from the signatures of the components. (The syntax of denotations and the rules for composing their signatures appear in the next Section.) Below, we specify how signatures are formed; the rules by which signatures prescribe usage are given in Section 3.

2.1. **Variable and value signatures**

Variable and value signatures specify the type used to interpret the value of the identifier and also whether the name is to be used as a value or as a reference to a value.

\[
\begin{align*}
\text{VarSignature} &: \text{ var TypeDenotation} \\
\text{ValSignature} &: \text{ val TypeDenotation}
\end{align*}
\]

2.2. **Function signatures**

Function signatures specify the names and signatures of the parameters of the function and the signature of the result of the function.

\[
\begin{align*}
\text{FuncSignature} &: \text{ func } [ \text{ Parameter}_1^* ] [\text{Any}]\text{Signature} \\
\text{Parameter} &: [\text{Any}]\text{Parameter} \\
[\text{Any}]\text{Parameter} &: \{ \text{Id}^+ : \}^* [\text{Any}]\text{Signature}
\end{align*}
\]

If no name appears in a parameter, then a single anonymous parameter is introduced. The parameter specifier

\[
\text{Id}_1, \text{Id}_2, \ldots, \text{Id}_n : \text{Signature}
\]

is syntactic shorthand for the list of parameter specifiers

\[
\text{Id}_1 : \text{Signature} ; \text{Id}_2 : \text{Signature} ; \ldots ; \text{Id}_n : \text{Signature}
\]

Note that a function signature may specify any signature as the result of the function.
Examples:

\[ \text{func[]} \text{ var Integer} \]
\[ \text{func [ var T ] val T} \]
\[ \text{func [ x,y : val Integer ] val Boolean} \]
\[ \text{func[ f : func[ val T ] val T ] func [ val T ] val T} \]

The last example is the signature of a "functional", i.e., a function that takes a function as an argument and yields a function as a result.

2.2. Type signatures

A type is a set of operations, so a type signature lists the names and signatures of the operations that form the type. The signature also may include a local name for the type, to which the signatures of the operations may refer.

TypeSignature ::= type Id \# \{ TypeSignatureComponent \}

TypeSignatureComponent ::= Id \# [Operation]Signature

| Id \# field TypeDenotation

| New | := | = | < | > | ValueOf | alias

Examples:

\[ \text{type} \{ \text{New;=} ; := \} \]
\[ \text{type t \{ New:func[ var t ;}
\quad = :func[ x,y : val t ] val Boolean ;
\quad := :func[ var t ; val t ] val t \} \]
\[ \text{type Color} \{ \\
\quad \text{Red, Blue, Green : func[]} \text{ val Color ;}
\quad \text{New;=} ; := ;
\quad \text{Succ : func [] val Color } \text{ val Color } \}

A type signature component can be an identifier and [Operation] signature or can be one of the shorthand forms provided. These shorthand forms are used for the common operations provided by a type to interpret values and variables. The full signature of each of these shorthand forms is given the following table, assuming that the local name of the type signature is T.
<table>
<thead>
<tr>
<th>Shorthand Form</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>New : func[] var T</td>
</tr>
<tr>
<td>:=</td>
<td>:= : func[ var T; val T ] val T</td>
</tr>
<tr>
<td>=</td>
<td>= : func[ x,y : val T ] val Boolean</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt; : func[ x,y : val T ] val Boolean</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt; : func[ x,y : val T ] val Boolean</td>
</tr>
<tr>
<td>ValueOf</td>
<td>ValueOf : func[ var T ] val T</td>
</tr>
<tr>
<td>alias</td>
<td>alias : func[ x, y : var T ] val Boolean</td>
</tr>
<tr>
<td>A : field TypeDenotation</td>
<td>A : func[ var T ] var TypeDenotation ;</td>
</tr>
<tr>
<td></td>
<td>A : func[ val T ] val TypeDenotation</td>
</tr>
</tbody>
</table>

Thus, a signature for a type that in full form would be written as

```
type R {
  New : func[] var R ;
  := : func[ var R; val R ] val R ;
  ValueOf : func[ var R ] val R ;
  = : func[ x, y : val R ] val Boolean ;
  re : func[ var R ] var real ;
  re : func[ val R ] val real }
```

could be rewritten using the shorthand forms as

```
type { New ; := ; ValueOf ; = ; re : field real }
```

Notice that in the example above re is "overloaded;" two occurrences of the same identifier appear with different signatures in the type signature specification. In this case, the overloaded identifier "re" allows referencing components ("fields" in Russell) of both values and variables of the type (the variable field selector could not be used on a value of the type because of a signature mismatch). The rule that specifies the syntactic legality of overloading is given in the next Section.
3. **Denotations**

Values are produced in a Russell program by the evaluation of **denotations**. Below, we give the syntax, semantics and any syntactic constraints for each of the forms of denotations in Russell.

\[
\text{Denotation ::= [Any]Denotation} \\
\text{[Any]Denotation ::= [Any]Application | [Any]Primary} \\
\text{[Any]Primary ::= [Any]Id | [Any]Conditional | [Any]Sequence | [Any]Block} \\
\text{ValPrimary ::= Repetition} \\
\text{[Operation]Primary ::= [Operation]Id \{ << [Operation]Signature >> \}} \\
\quad | [Operation]Selection | [Operation]Construction \\
\text{TypePrimary ::= TypeModification}
\]

**Syntactic Constraints:**

[Operation] denotations must obey the following [import rule]: no [Operation] denotation may import from the surrounding scope any identifier having `var` signature. This restriction guarantees that syntactically identical [Operation] denotations always denote the same value (in particular, that syntactically identical type denotations always denote the same type).

3.1. **Applications**

A denotation may be an application of a function to a list of arguments:

\[
\text{[Any]Application ::= \{ [ Denotation* ] \}} \quad \text{FuncPrimary \{ [ Denotation* ] \}}\]

Brackets surrounding single arguments may be omitted and binary functions may be used in infix form. Each implementation must provide rules for bracketing such applications.

**Meaning:**

The meaning of an application is to evaluate the function primary and the operand denotations and to then apply the function to the values of the operands. The order of evaluation of the arguments is unspecified; however, all occurrences of the same application must be evaluated in the same order.

**Signature:**

The signature of a function application is produced by textual substitution of argument denotations for parameters. Consider an application

\[
F \{ \text{arg}_1, \ldots, \text{arg}_k \}
\]

where the argument signatures are
asig₁, ..., asigₖ

For this application to be legal, F must be a function denotation, and its
signature must have the form

```
func [ x₁ : psig₁ ; ... ; xₖ : psigₖ ] rsig
```

The result signature is produced by textual substitution of argument denota-
tions for parameters:

```
<table>
<thead>
<tr>
<th>arg₁, ..., argₖ</th>
</tr>
</thead>
</table>
rsig               |
| x₁, ..., xₖ      |
```

Any operation denotations appearing in the result signature must satisfy the
import rule given above.

In addition, in a legal Russell program each argument signature must
match the corresponding parameter signature according to the rules given
below. However, as the result signature depends only on argument denotations,
not argument signatures, the result signature is defined even for an illegal
application.

**Examples:**

```
f[x,y]
x + y
g[]
(stack [ Integer ]) $ Top [ S ]
A + f[x, y, (Stack[Integer])] $ Top [S ]
```

**Syntactic Constraints:**

In a legal Russell program, the signature of each argument in a function
application must match the signature of the corresponding parameter according
to the rules given below. Like the rule for determining the result signature
for a function application, the matching rules do not require the actual
values of arguments; instead, they make use of textual substitution of argu-
ment denotations for parameters.

The first step in argument-parameter matching is expansion of the param-
eter signatures according to the particular set of arguments supplied. In this
way, the parameter signatures are modified to reflect the relationships among
the arguments of a particular application. As above, we consider an applica-
tion
\[ F[ \arg_1, \ldots, \arg_k ] \]

where the argument signatures are
\[ \text{asig}_1, \ldots, \text{asig}_k \]

and the signature of \( F \) has the form
\[ \text{func} [ x_1 : \text{psig}_1; \ldots; x_k : \text{psig}_k ] \text{rsig} \]

The parameter signatures are expanded by textual substitution of argument denotations for the corresponding parameter identifiers. Thus, the expanded parameter signature is
\[
\begin{array}{c}
| \arg_1, \ldots, \arg_k \\
| \text{psig}_i \\
| x_1, \ldots, x_k \\
\end{array}
\]

where \( \arg_i \) is the \( i \)th argument denotation. For example, if \( F \) has signature
\[ \text{func} [ n : \text{val Integer} ; A : \text{var} \ T[n] ] \text{val Integer} \]

then the expanded parameter signatures for the call
\[ F[10, x] \]

would be
\[ \text{val} \ \text{Integer} \ \text{and} \ \text{var} \ T[10] \]

These expanded parameter signatures are considered to be in the scope of the caller and any [Operation]Denotations contained within them must obey the import rule with respect to this scope.

The expanded parameter signatures and the argument signatures are then compared using the three signature calculus rules given below. An argument signature and an expanded parameter signature are said to match iff they can be transformed to identical signatures using these rules. An application is valid iff no mismatch is found for each pair of expanded parameter and argument signatures.

The signature calculus rules are:

1. **Renaming**, or substitution of local identifiers. When matching two operation signatures, the names of the parameters may be uniformly replaced with any identifier not occurring in either signature. Thus, the signature
\[ \text{func} [ x, y : \text{sig}_1 ] \text{sig}_2 \]

matches the signature
\[
\begin{array}{c}
|a, b \\
\text{func} [ a, b : \text{sig}_1 ] \text{sig}_2 \\
| x, y \\
\end{array}
\]
The substitutions are necessary because x and y may appear free in the signatures \( \text{sig}_1 \) and \( \text{sig}_2 \).

Similarly, the local name in a type signature may be replaced by any other identifier not appearing in the signature. Thus, the signatures

\[
\text{type } T \{ \text{New : func[]} \ \text{var} \ T \}
\]

and

\[
\text{type } X \{ \text{New : func[]} \ \text{var} \ X \}
\]

match.

2. **Reordering.** Two type signatures match if the signatures include all the same component names and identically-named components have matching signatures, independent of the order in which they appear in the two signatures. Thus, the signature

\[
\text{type } T \{ \ a_1 : \text{sig}_1 \ ; \ a_2 : \text{sig}_2 \ \}
\]

matches

\[
\text{type } T \{ \ a_2 : \text{sig}_2' \ ; \ a_1 : \text{sig}_1' \ \}
\]

if \( \text{sig}_1 \) matches \( \text{sig}_1' \) and \( \text{sig}_2 \) matches \( \text{sig}_2' \).

3. **Forgetting.** In matching two type signatures, the argument signature can be simplified by "forgetting" operations that appear in the signature. Thus, the argument signature

\[
\text{type } T \{ \ a_1 : \text{sig}_1 \ ; \ a_2 : \text{sig}_2 \ ; \ a_3 : \text{sig}_3 \ \}
\]

can be reduced by "forgetting" \( a_2 \) to

\[
\text{type } T \{ \ a_1 : \text{sig}_1 \ ; \ a_3 : \text{sig}_3 \ \}
\]

This notion of matching of signatures subsumes the usual notions of type-checking found in Algol-like languages. The rule for "type equivalence" in Russell is found in the absence of any transformation rule for variable and value signatures -- to match, two variable or value signatures must be syntactically identical.

3.2. **Identifiers**

Each identifier appearing in a Russell program is a denotation for the value bound to the identifier. The optional signature specification in an [Operation]Denotation is used to disambiguate overloaded function identifiers; all declarations of an overloaded function identifier must have distinct signatures. A value may be bound to an identifier through argument-parameter or declaration binding or because the identifier is "builtin," i.e., its meaning is predefined. There are eight such predefined identifiers in Russell: they are the builtin data types Void, Boolean, Integer, Character, String, Heap and
FileSystem and the builtin type-producing function Array. Their signatures and meanings are given in Section 4.

3.2. Conditionals

\[
\text{[Any]}\text{Conditional ::= if } \text{Guard } \implies \text{Denotation}^*\ \text{[Any]}\text{Denotation }\Box\ \text{fi} \\
\quad | \text{ValDenotation \texttt{cand} ValDenotation} \\
\quad | \text{ValDenotation \texttt{cor} ValDenotation}
\]

Guard ::= ValDenotation | \texttt{else}

The following syntactic sugar is provided:

1. A \texttt{cand} B may be used as shorthand for
\[
\text{if}\ A \implies B \Box\ \texttt{else} \implies \text{Boolean}\$\text{False}[] \text{fi}
\]

2. A \texttt{cor} B may be used as shorthand for
\[
\text{if}\ A \implies \text{Boolean}\$\text{True}[][] \Box\ \texttt{else} \implies B \text{fi}
\]

3. The guard \texttt{else} may be used as shorthand for the negation of the disjunction of all of the other guards.

Signature:

All of the guarded denotation sequences must have the same signature; the signature of the conditional denotation is the signature of any of its guarded denotation sequences.

Meaning:

Each denotation appearing as a guard must have signature \texttt{val} Boolean. The meaning of a conditional denotation is to evaluate one of the denotation sequences for which the evaluation of its guard produces the result Boolean\$\text{True}[]]. The order of evaluation of the guards is unspecified but deterministic. All evaluations of a denotation evaluate the guards in the same order and textually identical denotations evaluate the guards in the same order.
Examples:

\[
\text{if } g[x] \Rightarrow p[x,y] \quad \text{if } g[x] \Rightarrow h[y] \\
\text{if } x = \text{Boolean$True[]} \Rightarrow \text{Boolean$False[]} \\
\text{else } \Rightarrow \text{Boolean$True[]} \\
\text{fi}
\]

\[
\text{if } \quad \\
f[x] < 3 \Rightarrow g[x,y] \\
f[x] = 3 \Rightarrow h[x,y,z] \\
f[x] = 4 \Rightarrow h[x,z,y] \\
f[x] \geq 4 \Rightarrow h[x,x,x] \\
\text{fi}
\]

2.4. Selections

A type in Russell is a set of operation values, i.e., procedures, functions and types. A particular element of this set is chosen by applying selection to the type denotation. Additionally, syntactic sugar is provided to produce values by compositions of selections.

2.4.1. Operation Selections

An operation is selected from a data type by applying a selection of the form:

\[
[\text{Operation}]\text{Selection} ::= \{ \text{TypePrimary$ } \} \# \text{Id} \{ \ll [\text{Operation}]\text{Signature} \gg \} \#
\]

An implementation may allow the type primary to be omitted. For example, "Integer$+" may be abbreviated as "$+". The optional signature specification allows disambiguation in the case of overloaded component names of a type.

Signature:

A selection T$f is legal iff the signature of T is

\[
\text{type } t \{ \ldots f : \text{sig} \ldots \}
\]

i.e., T denotes a type with a component named f. The signature of T$f is obtained by substituting the type denotation T for all occurrences of the local name t in the f component of the signature of T:

\[
| T \\
| \text{sig} \\
| t
\]
**Meaning:**

The meaning of a selection is to evaluate the type primary and then to produce as a result the value of the selected component.

3.4.2. **Value Selections**

The other form of selections is used to produce values by compositions of evaluations of constants (i.e., nullary functions) of a data type.

\[
\text{[Any]Selection ::= \{ TypePrimary \}}^\# \text{ String}
\]

The selection

\[
T \$\ "a_1 a_2 \cdots a_n" \text{ for } n \geq 0
\]

is taken as a syntactic abbreviation for

1. \(T^\#[]\) if \(n=0\),

2. \(T^\#'a_1[]\) if \(n=1\) or

3. \((T^\#'a_1[])(T^\#'a_2[])\cdots(T^\#'a_n[])\) for \(n>1\). Thus, we compose the results of the constants given in the string using the \(^\#\) operator provided by the type.

Numbers are treated in the same way: If \(d_1\) is a digit and the remaining \(d_i\) are either digits or letters, then the selection

\[
T^\#d_1d_2\cdots d_n
\]

is interpreted as

\[
T^\#"d_1d_2\cdots d_n"
\]

Obviously, for a selection of this form to be valid, the TypePrimary must provide the necessary constants (the elements of the String) and a composition operator with the appropriate signature. The signature of the result is determined by the rules given above for producing the signatures of [Operation]Selections and applications.

**Examples:**

Integer $+$

(\text{Stack [ Boolean ]}) $\text{Push}$

Integer$12345$

String$"\text{Boehm, Demers and Donahue}"$
3.5. **Sequences**

\[
[\text{Any}]\text{Sequence} ::= ( \text{Denotation}^* [\text{Any}]\text{Denotation} )
\]

**Signature:**

The signature of a sequence is the signature of the last denotation.

**Meaning:**

The meaning of a sequence is to evaluate the denotations in order and to produce the value of the last denotation as the result.

**Examples:**

\[
( x + 1 )
\]
\[
( y := \text{if } x = 3 \Rightarrow 7 \text{ else } \Rightarrow 5 \text{ fi} ; \text{ValueOf[ y ]} )
\]

3.6. **Blocks**

A block allows locally declared names to be used in the evaluation of a denotation.

\[
[\text{Any}]\text{Block} ::= \text{let Declaration}^* \text{ in Denotation}^* [\text{Any}]\text{Denotation ni}
\]

**Declaration ::= [Any]Declaration**

\[
[\text{Any}]\text{Declaration} ::= \text{Id}^* \{ : \text{Signature} \}^* = [\text{Any}]\text{Denotation}
\]

**Signature:**

The signature of a block is the signature of the final denotation in the denotation sequence. This signature must obey the **export rule** of Russell: no denotation may have a signature containing free occurrences of locally declared identifiers.

**Meaning:**

The evaluation of a block proceeds by:

1. Evaluating the denotations in the right-hand side of the declarations and binding the values produced to the left-hand side identifiers. In evaluating these denotations, the identifiers appearing in [Operation]Declarations are known and may be used. (This allows recursive declaration of functions and data types.)
2. Evaluating the denotation sequence that is the body of the block. In evaluating this sequence, all identifiers introduced in the declarations of the block are known (as are any other globally declared identifiers).

Examples:

```plaintext
let x := 3 in x + 5 ni

let V : var Integer == Integer$New[ ] in V := 3; ValueOf[ V ] ni

let
  Fact == func[ N : val Integer ]
    { if N == 0 then 1 else N * Fact[ N - 1 ] fi }

in
  Fact[ 6 ]

ni
```

Syntactic Constraints:

For a block to be syntactically valid, three properties are necessary:

1. None of the declarations in a block may redeclare any identifier that is globally known to the block. However, function identifiers may be overloaded, i.e., two or more distinct functions may have the same name. The signatures of all occurrences of an overloaded function identifier must be distinct; the signature is used to disambiguate uses of the identifier.

2. The declarations in the block must be syntactically valid. The signatures of left-hand sides of the declarations must match the signatures of the right-hand sides, using the signature matching rules for applications (the left-hand side identifiers play the role of the parameters, the right-hand side denotations the role of arguments). Also, the right-hand side denotations must be syntactically valid assuming the [Operation] identifiers in the set of declarations have their specified signatures.

3. The body of the block must be syntactically valid using the signatures of the declared identifiers. If no signature is explicitly specified for a newly declared identifier, the signature of the denotation given in its declaration is used.

The prohibition of redeclaration, the semantics of the denotations provided in the Russell language and the import rule described above guarantee that the following substitution property holds for all legal Russell programs. Any legal program containing an [Operation] declaration can be modified by (possibly repeatedly) substituting the right-hand side [Operation] denotation for all free occurrences of the left-hand side [Operation] identifier (including in the declaration itself) to yield another legal Russell program with the same meaning. For instance, the final example of blocks given above is semantically equivalent to:
let  
Fact == func[ N : val Integer ]  
   { if N = 0 ==> 1  [] else ==> N * Fact[ N - 1 ] fi }  
in  
func[ N : val Integer ]  
   { if N = 0 ==> 1  [] else ==> N * Fact[ N - 1 ] fi }  
ni

where we have substituted the right-hand side of the declaration of Fact for its occurrence in the body of the block. This substitution could be repeated indefinitely without changing the meaning of the block.

3.7. Repetitions

Repetition ::= do { Guard ==> Denotation⁺ }* od

Signature:
The signature of a repetition is val Void.

Meaning:
The meaning of a repetition is to repeatedly evaluate the body until none of the guards has value Boolean$True[]$. In each repetition, one of the denotation sequences having a true guard is evaluated. The result of a repetition is Void$Null[]$.

3.8. Constructions

[Operation]Primary ::= [Operation]Construction

A construction introduces an operation value and associated signature into a Russell program. Sections 3.7.1 and 3.7.2 describe the values and signatures of the constructions provided in Russell.

3.8.1. Function Constructions

FuncConstruction ::= func[ Parameter⁺ ] Signature# { Denotation⁺ Denotation }
matching rules given above.

**Meaning:**

The result of a function construction is a value that can be applied to a set of arguments. When applied, the function value produces the result of the evaluation of the denotation sequence that is the body of the function. The bindings of values to identifiers used when evaluating the body in an application is the binding that existed when the function value was produced, extended by binding the argument values to the parameter identifiers.

**Examples:**

```plaintext
func [ x, y : val Integer ]
  { let
      z == 3; a == Integer$New[
    in
      a := 5; x * x + 2 * y - ValueOf[a]
    ni }

func [ A : var IntArray; i, j : val Integer ] var Integer { ( A[i] )+j }

  { func[ x : val T ] { (f[Y[f]]) [x] } }
```

The final example of a function declaration shows the flexibility of functions in Russell; the function Y declared above is the "least fixed point" operator taking T functionals as arguments and producing T functions as results.

### 3.8.2. Type Constructions

TypeConstruction ::= Enumeration | Record | Union | Extension | Image

A type is a set of operation values. Thus, a type construction can be viewed as a syntactic mechanism for associating other operation denotations. The meaning of the various type constructions is given below. The meaning of a data type is the meanings of the operations it provides. Throughout this report, we will present the meanings of the operations of a data type in one of two ways:

1. Where the operations have a straightforward interpretation, we will simply say in words what their results are.

2. Where the operations are reasonably complex or unfamiliar, we will present their meaning as a set of (conditional) axiomatic equations, which will define "equivalent program texts". An axiom of the form

   \[ \text{bexp} \Rightarrow \text{exp}_1 \equiv \text{exp}_2 \]

   is to be interpreted as "\text{exp}_1 \text{ and } \text{exp}_2 \text{ may be substituted for one another in a legal Russell program in any context in which evaluation of the} \]
(side-effect free) Boolean expression bexp yields Boolean$\top$.$^
$ If two expressions can always be substituted for one another, we will omit the qualifying "Boolean$\top$". Additionally, we will adopt the following naming convention in the presentation of the axioms: the free variable "exp" in an axiom stands for any Russell denotation having the appropriate signature; otherwise free variables in the axioms stand for any Russell identifier having the correct signature. (The evaluation of an arbitrary denotation in Russell can have side-effects; the evaluation of an identifier cannot.)

3.8.2.1. **Enumerations**

A type construction of the form

\[
\text{Enumeration ::= enum } \{ \text{Id}^+ \}
\]

produces an enumeration type with a set of distinct constants and an ordering among them.

**Signature:**

The signature of an enumeration type construction

\[
\text{enum } \{ a_0, \ldots, a_k \}
\]

is

\[
\text{type } T \{ \\
\quad \text{New : = ValueOf ; alias } < ; > ; = ; \\
\quad \text{First, Last : } \text{func[ ] val T ;} \\
\quad \text{Pred, Succ : } \text{func[ val T ] val T ;} \\
\quad \text{Ord : } \text{func[ val T ] val Integer ;} \\
\quad \text{OrdInv : } \text{func[ val Integer ] val T ;} \\
\quad \text{Card : } \text{func[ ] val Integer ;} \\
\quad a_0, \ldots, a_k : \text{func[ ] val T } \}
\]

**Meaning:**

The operations of enumerations obey the following laws. Pred, Succ, Ord, OrdInv, Card and the relational operators of the type are related according to:
OrdInv[ Ord[x] ] ≡ x
0 ≤ Ord[x] < Card[]
Card[] ≡ k + 1
x < y ≡ Ord[x] Integer$< Ord[y]
x = y ≡ Ord[x] Integer$= Ord[y]
x > y ≡ Ord[x] Integer$> Ord[y]

Thus, Ord and OrdInv can be viewed as defining a canonical representation for the enumeration type. Card gives the number of distinct constants of the type.

The next set of axioms define the meanings of the constants introduced by an enumeration type:

First[] ≡ a_0[
Last[] ≡ a_k[
Succ[ a_i[] ] ≡ a_{i+1}[] for 0≤i<k
Pred[ a_i[] ] ≡ a_{i-1}[] for 0<i≤k

The remaining operations of an enumeration type define the interpretation of variables. In giving the axioms for these operations for enumerations and the other data types in Russell, we will use a "hidden" operation alias with signature

func[ x, y : var T ] val Boolean

where T is the type being defined. (Throughout the following, any hidden operation used in defining a type will be given a boldface name to distinguish it from the other operations of the type.) The reason for making alias hidden is to guarantee that it is not redefined by a type modification. Thus, we can safely assume that it has reasonable properties in cases where we must define alias of composite types in terms of alias of component types. However, we can easily relate the hidden version alias to the alias operation provided by most data types through the trivial axiom:

T$alias ≡ T$alias

The additional axioms satisfied by the operations on variables of enumeration type include the following (where x and y are any identifiers having signature var T and exp is any denotation having signature val T where T is the local name used in the signature of the type):
New[] alias x ≡ x alias New[] ≡ False[]

x alias x ≡ True[]

x alias y => x ≡ y

ValueOf[ ( New[]; x ) ] ≡ ( New[]; ValueOf[ x ] )

x := exp ≡ ( x := exp; ValueOf[x] ) ≡ let z := exp in x := z; z ni

¬(x alias y) =>
  ( x := exp; ValueOf[y] ) ≡ let v := ValueOf[y] in x := exp; v ni

That is, no variable ever aliases with New, every variable aliases with itself and assignment and ValueOf work as expected. The assignment operator produces the value of its second argument as its result.

These axioms are, in fact, generally true for this set of operations provided by all of the type constructions and builtin types of Russell. Thus, when presenting the remainder of the data types of Russell, we will omit the axioms for these operations when they satisfy the same properties.

3.8.2.2. Records

The record construction produces a new type with operations to select components (or fields) from variables and values of the type.

Record ::= record Id# | { Id* : TypeDenotation }* |

The optional identifier in a record construction is the local name of the type and may be used in the TypeDenotations appearing in the construction. It also is used as the local name in the signature of the type (if not provided, a new name is used).

Signature:

The signature of a record construction of the form

record t { F₁ : TypeDen₁; ...; Fₙ : TypeDenₙ }

is

type t { New; :=; ValueOf; alias; = ;

F₁ : field TypeDen₁ ; ...; Fₙ : field TypeDenₙ ;

Mk : func[ val TypeDen₁; ...; val TypeDenₙ ] val t }

(Remember that field is a shorthand signature specification for a pair of functions.) Each type denotation appearing in a record construction must have a signature that matches

type { New; :=; =; alias; ValueOf }
Meaning:

A variable or value of a record type is treated as a Cartesian product of its fields (Mk is the operation to create such a value). Thus, the meanings of the operations New, :=, ValueOf, and = are defined in terms of compositions of those operations for each of the fields of the record. We have the axioms:

\[ F_i[\text{Mk}[v_1, \ldots, v_n]] \equiv v_i \text{ for } 1 \leq i \leq n \]
\[ F_i[\text{New[]}] \equiv \text{TypeDen}_i \text{New[]} \text{ for } 1 \leq i \leq n \]
\[ \text{ValueOf}[x] \equiv \text{Mk}[\text{ValueOf}[F_1[x]], \ldots, \text{ValueOf}[F_n[x]]] \]
\[ (x := y) \equiv \text{Mk}[F_1[x] := F_1[y], \ldots, F_n[x] := F_n[y]] \]
\[ (x = y) \equiv (F_1[x] = F_1[y]) \& \ldots \& (F_n[x] = F_n[y]) \]

In each case, the operator is defined for the record type only if the identically-named operator with the appropriate signature appears in the signature of each field type.

The aliasing relations among the fields of a record variable are defined by:

\[ x \text{ alias } y \equiv (F_1[x] \text{ alias } F_2[y]) \& \ldots \& (F_n[x] \text{ alias } F_n[y]) \]

The alias operation for the record type itself also obeys the usual axioms:

\[ x \text{ alias } \text{New[]} \equiv \text{New[]} \text{ alias } x \equiv \text{False[]} \]
\[ x \text{ alias } x \equiv \text{True[]} \]
\[ \text{alias} \equiv \text{alias} \]

It is important to note here that in producing the value of a record construction, none of the free identifiers appearing in the construction are evaluated. Thus, a record construction may produce a value even though all of the operations of the resulting type will diverge when applied (for example,

\[ \text{record } T\{x, y : T\} \]

This allows straightforward construction of recursively defined record types.

Examples:

\[ \text{record } \{\text{re, im : Real}\} \]
\[ \text{record } T\{\text{Val : Integer; Next : Heap$Ref[T]}\} \]
3.8.2.3. Unions

The union type construction produces a type with projection, injection and inspection operations between the type produced and the type identifiers specified in the construction.

\[ \text{Union} ::= \text{union} \{ \text{Id}^+ \} \]

The identifiers appearing in the construction must be distinct and each identifier must have a signature matching

\[ \text{type} \{ = \} \]

**Signature:**

The signature of a union construction of the form

\[ \text{union} \{ \text{Id}_1, \ldots, \text{Id}_n \} \]

is

\[ \text{type} \ t \{ \]
\[ \text{New} \ ; \ \text{ValueOf} \ ; := \ ; \ \text{alias} \ ; = \ ; \]
\[ \text{ToId}_1 \ : \ \text{func}[ \text{val} \ t ] \ \text{val} \ \text{Id}_1 ; \]
\[ \ldots \]
\[ \text{ToId}_n \ : \ \text{func}[ \text{val} \ t ] \ \text{val} \ \text{Id}_n ; \]
\[ \text{FromId}_1 \ : \ \text{func}[ \text{val} \ \text{Id}_1 ] \ \text{val} \ t ; \]
\[ \ldots \]
\[ \text{FromId}_n \ : \ \text{func}[ \text{val} \ \text{Id}_n ] \ \text{val} \ t ; \]
\[ \text{IsId}_1? \ : \ \text{func}[ \text{val} \ t ] \ \text{val} \ \text{Boolean} ; \]
\[ \ldots \]
\[ \text{IsId}_n? \ : \ \text{func}[ \text{val} \ t ] \ \text{val} \ \text{Boolean} \} \]

**Meaning:**

The axioms constraining these operations are fairly obvious:
ToId₁[ FromId₁[ x ] ] ≡ x for 1 ≤ i ≤ n
IsId₁[ FromId₁[ x ] ] ≡ True[] for 1 ≤ i ≤ n
IsId₁[ FromId₁[ x ] ] ≡ False[] for i ≠ j
IsId₁[ x ] ==> FromId₁[ ToId₁[ x ] ] ≡ x for 1 ≤ i ≤ n

The meaning of equality for union types is given in terms of the meanings of equality for the types of the union:

( x = y ) ≡
   if
     IsId₁[ x ] & IsId₁[ y ] ==> ToId₁[ x ] = ToId₁[ y ]
   .
   .
   .
   IsIdₙ[ x ] & IsIdₙ[ y ] ==> ToIdₙ[ x ] = ToIdₙ[ y ]
   else ==> False[]
   fi

Again, none of the free identifiers appearing in a union construction are evaluated when the construction is evaluated, but only when the equality operation of the construction is applied (the meaning of none of the remaining operations depend on the meaning of the component types). Thus, recursively defined unions like

T == union { Integer, T }

are perfectly legal and well-behaved.

3.8.2.4. Extensions

Type extension creates a new type with operations that interpret values and variables of the type of which it is an extension.

Extension ::= extend { TypeDenotation }

Signature:

The signature of a type extension is the signature of the specified type denotation with the following additional operations:

In : func[ var TypeDenotation ] var T ;
In : func[ val TypeDenotation ] val T ;
Out : func[ var T ] var TypeDenotation ;
Out : func[ val T ] val TypeDenotation
where T is the local name used in the type signature.

Meaning:

The meanings of the operations other than In and Out are those of the type denotation. Again, the type denotation is not evaluated until an operation from the type is applied. The In and Out functions that take var arguments do not produce "new" variables as result, but behave like "field selectors":

\[
x \text{ alias } y \equiv \text{In}[x] \text{ alias } \text{In}[y] \text{ and } x \text{ alias } y \equiv \text{Out}[x] \text{ alias } \text{Out}[y].
\]

Thus, the resulting type behaves like the original type with the addition of a set of conversion operations that allow free interchangeability between variables and values of the original and the new types.

3.8.2.5. Images

The final form of type construction is image.

Image ::= image \{ [Operation]Signature \}

Images construct types with values that are images of [Operation] values, i.e., with transfer functions between values of the type and values with the specified [Operation]Signature and vice-versa.

Signature:

The signature of an image of the form

\[
\text{image} \{ \text{OpSig} \}
\]

is

\[
\text{type Image} \{
\text{New} ; := ; \text{ValueOf} ; \text{alias} ;
\text{In} ; \text{func[ OpSig ] val Image} ;
\text{Out} ; \text{func[ val Image ] OpSig} \}
\]

Note that equality is not provided for image types.

Meaning:

The meanings of In and Out satisfy the following:

\[
\text{let } F \equiv \text{expl in exp2 ni} \equiv \\
\text{let } I \equiv \text{image\{ (*Signature of expl*) \};} \\
F \equiv I\text{Out[ I\text{In[ expl ]]}} \\
in \text{exp2 ni}
\]

Thus, In and Out preserve the identity of the operations they are applied to.
Examples:

```
image { func[ val Integer ] val Integer }
image { type {} }
```

The first example produces a type with conversion functions from Integer functions to elements of the type, i.e., it is a "functional" type. The second type is the "paradoxical" type of Russell; it is the type of all types (since any type can be coerced by "forgetting" to match the signature `type{}`).

3.2. Type Modification

TypeModification ::= OperationModification | SignatureModification

Type modification produces a new type by modification of the operations or the signature of an existing type. This modification may involve

1. adding new operations to a type,
2. replacing operations of a type, or
3. changing the signature of a type to delete operations from it ("hiding" the operations). The effect of deleting an operation by type modification is exactly the same as the "forgetting" matching rule given above.

The two forms of type modification allowed in Russell are described below.

3.2.1. Operation Modifications

An operation modification allows new components to be added to a type or the meanings of existing components to be changed.

```
OperationModification ::= TypePrimary WithList

WithList ::= with Id# | [Operation]Declaration;
```

Signature:

The signature of the result of an operation modification is produced by:

1. substituting in all of the signatures of the WithList components the local name used in the signature of the TypePrimary for the local name used in the WithList (in essence, reconciling the use of possibly different local names for the same type) and then
2. adding the names and modified signatures of the WithList components to the signature of the type being modified. Duplicates (which will arise
in replacing the meaning of a previously defined operation) are ignored.

**Meaning:**

The declarations contained in the WithList are evaluated and a new type is created in which the values of the new components are added to the meaning of the type being modified. Throughout the modifier the local name that prefixes the modifier is used to refer to the modified type; thus, both the operations of the type being modified and the new operations being added may be referenced by selection on the local name. For example, in the type denotation

\[ T' \text{ with } t \{ \ldots t$F \ldots \} \]

the meaning of \( t$F \) is either the \( F \) component of the WithList if one is provided or the operation denoted by \( T'$ \) if \( F \) is not declared in the WithList. In this way, the new operations provided in the WithList can be defined both in terms of the type being modified and in terms of each other.

**Examples:**

```plaintext
extend \{ Boolean \}
with \$\{ 
    Nand == func[ x, y : val B ] \{ B$\neg( x B$& y ) \} ;
    Nor == func[ x, y : val B ] \{ B$\neg( x B$| y ) \} ;
    \sim == func[ x : val B ]
    \{ if B$Out[x] ==> B$In[False[]] \} else ==> B$Out[True[]] fi \}

Integer
with \{ NewNew == func[] \{ let x == Int$New[] in x := Int$0; x \ni \} \}
with \{ New == Int$NewNew \}
```

Note that in the first example, the use of extension produces a type having conversion operators to allow Boolean values to be applied to operations of the resulting type. These \( \text{In} \) and \( \text{Out} \) operations are needed in the body of \( \sim \) to define negation in terms of a conditional. The \( \sim \) operation used inside the body of the modifier is the one that is part of the modified type, i.e., the one defined inside the modifier (this \( \sim \) has the same meaning as the one provided by Boolean). The second example shows how modifiers can be used sequentially to define an operation in terms of its previous value. In this case, we first define \( \text{NewNew} \) in terms of the value of the previous \( \text{New} \) operation and then redefine \( \text{New} \) as the \( \text{NewNew} \) operation that was just constructed.

**Syntactic Constraints:**

Again, the declarations appearing in the WithList must be syntactically valid. The same rules used in deciding the validity of the declarations prefixing a block apply, except that the substitutions performed in signature expansion use the "component name" of the identifiers introduced (e.g., "T$F") where \( T \) is the local name used in the WithList and \( F \) is an identifier declared in the
list). Thus, the following sort of "renaming" can be done:

```haskell
with T { NewType == Integer;
        F : func[ val T$NewType ] val T$NewType == Integer$- }
```

In determining the matching requirements, "Integer" is substituted for "T$NewType" in the signature of F, which then matches the signature of "Integer$-". Doing this renaming allows F to take arguments that have signature `val T$NewType`.

Also, an operation may be redefined in a WithList only if it does not appear free in the signature of any component of the type being modified.

2.2.2. **Signature Modifications**

A signature modification allows operations to be deleted from the signature of a type, thus "hiding them" from use.

```
SignatureModification ::= TypePrimary ExportList
ExportList ::= export Id# { ExportElement+ } |
               hide Id# { ExportElement+ } |
ExportElement ::= Id { << Signature >> }# ExportList#
```

**Signature:**

The signature of a type modified by an ExportList will either contain only those components listed in the ExportList or only those not listed in the ExportList if the list is prefixed with `hide`. If one of the identifiers in the list refers to a type, it may be restricted by specifying an ExportList for it; thus the need for the recursive definition of ExportLists. The optional signature specification is used to specify which of a set of overloaded component names are to be exported. Otherwise, if an overloaded component name appears in the ExportList, all of its meanings are added to or deleted from the resulting type. The optional identifier prefixing an export list is used as a local name for the type being modified in the specification of signatures inside the ExportList. The signature of the resulting type is taken from the signatures of the exported components.

**Meaning:**

The meanings of the operations of the type are unaffected.
Examples:

record{ Val : T; Count : Integer }
    with R{
    Alloc   == func[] { let x == R$New[] in Count[x] := 0; x ni };
    Assign == func[ x : var R; y : val R ]
                      { Val[x] := Val[y]; Count[x] := Count[x] + 1 } ;
    Equal  == func[ x, y : val R ] { Val[x] T$= Val[y] } ;
    ACount == func[ x : var R ] val Integer { ValueOf[ Count[x] ] } } } 
    with R{ New == R$Alloc; := == R$Assign ; = == R$Equal ; Count == ACount }
    hide R{ Alloc, Assign, Equal, 'Val, 
        Count << func[ var R ] var Integer >>,
        Count << func[ val R ] val Integer >> }

In this example, we have constructed a type in which variables of the type keep a count of how many times they have been assigned to. This example uses the name Count several times just to show how function names can be overloaded and selectively hidden. Because the Count and Val field selectors that are provided by the record type are hidden in the hide modification, the user of this type "sees" a type that has no components, i.e., variables and values of the type do not appear as if they were created by a record construction. But, the operation Count that is provided will produce the number of times the := operation was applied to a variable of the type. Note that like any other operation a type provides, equality and assignment can be changed as desired.

Syntactic Constraints:

Signature modifications must obey the export rule described in Section 3.6; thus, all component names appearing free in the signature of a type must be exported, i.e., must be components of the type's signature.
4. Predefined Identifiers

There are eight predefined identifiers in Russell. Three of these, Void, Boolean and Integer, are "primitive" types in the language; they are used to define the meaning of other constructions in Russell. The remaining data types (Character, String, Heap and FileSystem, and the builtin function Array) are not primitive in this sense; there is nothing in Russell that depends on their signature or meaning. In deciding the meanings of these non-primitive identifiers, we have chosen to define them in the simplest manner we could think of. However, any implementation of Russell is free to change the signature or behavior of these non-primitive builtin identifiers.

4.1. Void

The simplest builtin data type is Void.

Signature:

\[
\text{type V \{ Null : func[] val V \}}
\]

Meaning:

Void has only a single constant Null and no other operations.

4.2. Boolean

The meanings of conditionals and repetitions depend on the Boolean data type.

Signature:

The signature of Boolean is:

\[
\text{type Bool \{ := ; ValueOf ; alias ; True, False : func[] val Bool ; ~ : func[ val Bool ] val Bool ; \&, |, = : func[ x, y val Bool ] val Bool \}}
\]

Meaning:

The operators of this type behave as one would expect Booleans to behave -- True and False produce distinct values and &, |, ~ are the usual Boolean connectives.

4.3. Integer

The final "primitive" data type of Russell is Integer, which is used to
give the meaning of the enumeration type construction.

**Signature:**

The signature of the Integer data type is

```haskell
type Int { New ; := ; ValueOf ; alias ; < ; > ; = ;
    + , - , * , / , mod , ^ : func[ x , y : val Int ] val Int ;
    '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' , '8' , '9' : func[] val Int }
```

**Meaning:**

These are the usual integer operations (note that "-" is overloaded to be both the negation and subtraction operators). Although only ten Integer constants are specified, larger numbers can be composed through use of value selections, as described in Section 3.4.2. By the rules described there, the denotation

```
Integer$51
```

is interpreted as

```
(Integer$'5'[[]) Integer$^ (Integer$'1'[[]])
```

The meaning of Integer$^ is to produce as a result 10 times its left argument plus its right argument.

**4.4. Character**

The Character data type of Russell is predefined, but implementation-dependent. For any implementation, Character must provide at least all of the constants of the Russell character set described in Section 1. The type is simply an enumeration of these constants in some implementation-defined order, i.e., Character is a type defined by

```
enum{ .... , 'a' , 'b' , .... , 'z' , 'A' , .... , 'Z' , '0' , .... , '9' , ... }
```

for some choice of constants and ordering. The signature and meaning of this type is that of the enumeration, as defined in 3.8.2.1.

**4.5. String**

Strings in the Russell language are sequences of Characters.

**Signature:**

String has signature
type String {
    New ; := ; ValueOf ; < ; = ; > ; alias ;
    ^ : func[ x,y : val String ] val String ;
    Substr : func[ val String; i,j : val Integer ] val String ;
    Length : func[ val String ] val Integer ;
    Char : func[ val String ] val Character ;
    MkStr : func[ val Character ] val String ;
    "", 'A', 'B', ..., 'a', ..., 'z', ... : func[ ] val String }

Meaning:

The constants of the type have the same names as the Character constants, with
the obvious axioms:

    MkStr[ 'A' ] ≡ 'A'       Char[ 'A' ] ≡ 'A'
    MkStr[ 'B' ] ≡ 'B'       Char[ 'B' ] ≡ 'B'

and so forth for each String constant other than '}'. The meanings of the
other operations provided by String are obvious: ^ is the catenation operator,
Length produces the number of characters in a string and Substr produces the
substring of its first argument beginning at the position specified by the
second argument for a length specified by the third argument. (The indexing
is 0-based, so:

    Substr[ s, 0, Length[s] ] ≡ s .)

The ordering on strings is lexicographic:

    Length[x] = 1 & Length[y] = 1 ==> x < y ≡ Char[x] Character$< Char[y]
    ( x ^ y ) < ( x ^ z ) ≡ y < z
    Length[y] > 0 ==> x < ( x ^ y ) .

4.6. Heap

The Heap builtin data type of Russell allows the dynamic creation of
variables. A heap can be viewed as a named store, in which one can produce
new variables. Heap provides a type-producing function Ref which yields a
type that has operations to create new variables in a a Heap (the Create func-
tion) and to reference these new variables (the + function).

Signature:

The signature of Heap is
type H {
    New : alias;
    Ref : func[ T : type{ New } ]
}

type R
{ New ; := ; ValueOf ; alias ;
  Nil : func[ ] val R ;
  Create : func[ var H ] val R ;
  InHeap? : func[ var H ; val R ] val Boolean ;
  SameRef : func[ var H ; x, y : val R ] val Boolean ;
  * : func[ var H ; val R ] var T }

Meaning:

Heaps only have New and alias as primitive operations, so that little can be
done with heap variables (as one would expect for local stores). In fact, one
cannot create heap values, as ValueOf does not exist. However, the Ref type
component of Heap allows the definition of reference values that can then be
used to refer to variables inside a heap. Note that heaps are untyped (many
types of variables may be created in a single heap), while references to vari-
ables in a heap are typed (i.e., a reference produces a variable of a particu-
lar type). The value of htp is defined only if p was produced by an evalu-
ation of Create[h], i.e., references are bound to particular heaps. We have
the additional axioms:

InHeap?[ h, Create[h] ] ≡ True[]

InHeap?[ h, p ] => SameRef[ h, p, p ] ≡ True[]

SameRef[ h, Create[h], Create[h] ] ≡ False[

SameRef[ h, p1, p2 ] ≡ htp1 alias htp2

The final axiom says that equal references alias; note that here the "hidden"
alias operation is used instead of any operation with the same name that may
be included in the signature of the type of the variable produced. (Indeed,
that type may not even have an alias operation.) The axioms

New[] alias x ≡ x alias New[] ≡ False[

x alias x ≡ True[]

given above for alias are assumed to hold for the type of the variable pro-
duced by htp1, while no such assumptions can be made about a programmer-
defined alias operation with the same signature. Additionally, we have the
obvious

( Heap$Ref[T] )$alias ≡ alias

4.7. File Systems

The final builtin data type of Russell is FileSystem, which provides a
means of external communication for Russell programs. The file system
signature given below is meant to provide a minimal facility for communication -- other sorts of files can be added in particular Russell implementations.

Signature:

The signature of FileSystem is:

```plaintext
type FS{
  RandomFile : func[ T : type{} ]
    type F{
      alias : func[ f1, f2 : var F ] val Boolean ;
      Open : func[ var FS; val FileName ] var F ;
      Read : func[ var F; val Integer ] val T ;
      Write : func[ var F; val Integer; val T ] val T ;
    }
  InputFile : func[ T : type{} ]
    type F{
      alias : func[ f1, f2 : var F ] val Boolean ;
      Open : func[ var FS; val FileName ] var F ;
      Get : func[ var F ] val T ;
      Empty?: func[ var F ] val Boolean ;
    }
  OutputFile : func[ T : type{} ]
    type F{
      alias : func[ f1, f2 : var F ] val Boolean ;
      Open : func[ var FS; val FileName ] var F ;
      Put : func[ var F; val T ] val T ;
    }
}
```

The data type FileName, used to identify a particular file, is left unspecified.

Meaning:

The only visible operations provided by FileSystem are functions that yield file types when applied to a type argument. How FileSystem variables are created is not specified. The Open operation of any of the file types creates a file variable that can be used by the other operations of that file type; Open can be regarded as the file equivalent of the New operation. The FileName provided is used to associate the file opened with some component of the file system. For each of the file types provided, we have the axioms:

```plaintext
Open[ fs, fname ] alias Open[ fs, fname ] ≡ f alias f ≡ True[

f1 alias f2 ==> f1 ≡ f2
```

There are three types of files provided. The first is RandomFiles, in which records are referenced by Integer indices. The axioms for the operations of this type are:

```plaintext
( Write[ f, i, v ]; Read[ f, i ] ) ≡ ( Write[ f, i, v ]; v )

~(i Integer$= j) ==> (Write[f, i, v]; Read[f, j]) ≡ let v := Read[f, j] in Write[f, i, v]; v fi
```
The remaining file types, InputFile and OutputFile, provide the normal operations on sequential files. To axiomatize these operations, we first need to introduce a list data type (which will be used to represent sequential file values) and a hidden \texttt{ValueOf} operation that produces the value associated with a file variable. The List type-producing function we will use is essentially that of [Demers80c], having signature

\begin{verbatim}
func[ T : type{} ]
type L {
    Nil    : func[] val L ;
    empty? : func[ val L ] val Boolean ;
    first  : func[ val L ] val T ;
    rest   : func[ val L ] val L ;
    append : func[ val L; val T ] val L }
\end{verbatim}

with the usual axioms relating these operations. The hidden \texttt{ValueOf} operation will have signature:

\begin{verbatim}
func[ var F ] val List[T]
\end{verbatim}

(where \( F \) is the local name of the file type and \( T \) is the type of elements of the file).

For InputFiles, we have the following axioms:

\begin{verbatim}
Empty?[f] \equiv empty?[ ValueOf[f] ]
\end{verbatim}

\begin{verbatim}
\neg \text{Empty?[f]} \implies \text{Get[f] \equiv let fv == ValueOf[f] in Get[f]; first[ fv ] ni}
\end{verbatim}

\begin{verbatim}
\neg \text{Empty?[f]} \implies ( \text{Get[f]; ValueOf[f] } \equiv \text{let fv == ValueOf[f] in Get[f]; rest[ fv ] ni}
\end{verbatim}

And for OutputFiles, we have:

\begin{verbatim}
Put[ f, x ] \equiv ( Put[ f, x ]; x )
\end{verbatim}

\begin{verbatim}
(\text{Put[f, x]; ValueOf[f]} \equiv \text{let fv == ValueOf[f] in Put[f, x]; fv append x ni}
\end{verbatim}

4.8. Arrays

A builtin function Array allows the construction of single-dimension arrays with integer subscripts Multi-dimension arrays and arrays with other index types are easily constructed from compositions of single-dimension arrays by the use of type modification.

\textbf{Signature:}

The signature of Array is
```plaintext
func[ Size : val Integer ; ComponentType : type { New; := ; = ; ValueOf } ]
  type A
  { New ; := ; = ; ValueOf ; alias ;
    ↑ : func[ val A ; val IndexType ] val ComponentType ;
    ↑ : func[ var A ; val IndexType ] var ComponentType }```

Thus, arrays may be built from any component type that allows assignment and equality.

**Meaning:**

The signature of the resulting array type includes the usual basic operations and ↑ functions to select components of array variables or values. The basic operations for the array type are defined in terms of the ComponentType operations. To give the axioms that these operations must satisfy, we will use the following "hidden" operations:

- **Update** : `func[ val A; val Integer; val ComponentType ] val A ;`
- **InitA** : `func[] val A`

`Update` changes the value of a component of an array value; `InitA` is an unspecified array constant, which will be used in building up the results of array operations.

The axioms satisfied by these operations are:
\textbf{Update}[a, i, v] + j \equiv \textit{if} i = j \implies v \land \neg (i = j) \implies a + j \textit{fi}

0 \leq i < \text{Size} \implies \text{New}[] + i \equiv \text{ComponentType}$\text{New}[]$

\text{ValueOf}[a] \equiv
\begin{align*}
\text{let } f &= \text{func}[x : \text{var} A; n : \text{val} \text{ Integer}; \text{result : val A}] \\
&\quad \text{if} \\
&\quad \mathrm{n} \geq \text{Size} \implies \text{result} \\
&\quad \neg \mathrm{n} < \text{Size} \implies f[a, n+1, \textbf{Update}[\text{result}, n, \text{ValueOf}[a+n]]] \\
&\quad \text{fi}\}
\end{align*}
\begin{align*}
in &\quad f[a, \text{Integer}$$\text{S}0[]], \text{InitA}] ni
\end{align*}

a := b \equiv
\begin{align*}
\text{let } f &= \text{func}[x : \text{var} A; n : \text{val} \text{ Integer}; \text{result : val A}] \\
&\quad \text{if} \\
&\quad \mathrm{n} \geq \text{Size} \implies \text{result} \\
&\quad \neg \mathrm{n} < \text{Size} \implies f[a, n+1, \textbf{Update}[\text{result}, n, a + n := b + n]]] \\
&\quad \text{fi}\}
\end{align*}
\begin{align*}
in &\quad f[a, \text{Integer}$$\text{S}0[]], \text{InitA}] ni
\end{align*}

0 \leq i < \text{Size} \quad \text{cand} (a + i = b + i) \implies (a = b) \equiv \text{False}[]

a = b \implies (i < 0 \mid i \geq \text{Size}) \quad \text{cor} (a + i = b + i) \equiv \text{True}[]

a \textbf{alias} b \implies (i < 0 \mid i \geq \text{Size}) \quad \text{cor} (a + i) \textbf{alias} (b + i) \equiv \text{True}[

\neg (a \textbf{alias} b) \& (0 \leq i, j < \text{Size}) \implies a + i \textbf{alias} b + j \equiv \text{False}[]

(0 \leq i, j < \text{Size}) \implies (a + i) \textbf{alias} (a + j) \equiv (i \text{ Integer}$$\text{S} = j)

The axiom for \textbf{Update} is obvious. The axioms about assignment and \text{ValueOf} state that the result of these operations is produced by applying the ComponentType operation to every component of the array and producing the array result that is the component-by-component result of these operations. Equality and aliasing are also decided component-wise (if two arrays do not alias, they do not share any components). The final axiom specifies that two array elements alias iff they have equal indices.
5. Programs

A Russell program is a function declaration to be evaluated and applied.

Program ::= Id == FuncConstruction

The details of how arguments are constructed to be passed to the function and of how the result of the function is communicated are implementation-defined.
References

[Demers 78]

[Demers 79]

[Demers 80a]

[Demers 80b]

[Demers 80c]

[Donahue 79]

[Jensen 75]

[Scott 76]
Scott, Dana. Data Types as Lattices. SIAM Journal on Computing 5:3 (September 1976).

[Wulf 78]
Appendix A: Collected Syntax

Basic Vocabulary

Letter ::= a | b | c | ... | z | A | B | ... | Z
Digit ::= 0 | 1 | 2 | ... | 9
PunctuationChar ::= $ | ( | ) | _ | ` | [ | ] | << | >> | . | , | ; | @ | :
OpChar ::= + | - | * | /= | + | < | = | > | : | & | | \ | ~ | .
WhiteSpace ::= <blank>

Alphanumeric ::= Letter | Digit
Character ::= Alphanumeric | PunctuationChar | OpChar | WhiteSpace

Id ::= WordId | OperatorId | QuotedId
WordId ::= Letter Alphanumeric
OperatorId ::= OpChar
QuotedId ::= ' Character '
String ::= Number | QuotedString
Number ::= Digit Alphanumeric
QuotedString ::= " Character "

Signatures

Signature ::= [Any]Signature
VarSignature ::= var TypeDenotation
ValSignature ::= val TypeDenotation
FuncSignature ::= func [ Parameter ] [Any]Signature
TypeSignature ::= Type Id # { TypeSignatureComponent }
TypeSignatureComponent ::= Id : [Operation]Signature
 | Id : field TypeDenotation
 | New ::= ::= < | > | ValueOf | alias
**Parameters**

Parameter ::= [Any]Parameter

[Any]Parameter ::= \{ Id\^+ , \}\# [Any]Signature

**Denotations**

Denotation ::= [Any]Denotation


[Any]Primary ::= [Any]Id | [Any]Conditional

\| [Any]Sequence | [Any]Block | [Any]Selection

ValPrimary ::= Repetition

[Operation]Primary ::= [Operation]Id \{ << [Operation]Signature >> \}\#

\| [Operation]Selection

\| [Operation]Construction

TypePrimary ::= TypeModification

**Applications**

[Any]Application ::= \{ [ Denotation ] \}\# FuncPrimary \{ [ Denotation\^* ] \}\#

**Conditionals**

[Any]Conditional ::= if \{ Guard ==> Denotation\^+ [Any]Denotation \}^+ fi

\| ValDenotation \textit{and} ValDenotation

\| ValDenotation \textit{or} ValDenotation

Guard ::= ValDenotation | \textit{else}

**Repetitions**

Repetition ::= do \{ Guard ==> Denotation\^+ \}^+ od

**Sequences**

[Any]Sequence ::= ( Denotation\^* [Any]Denotation )
Blocks

[Any]Block ::= let Declaration* in Denotation* [Any]Denotation ni
Declaration ::= [Any]Declaration

[Any]Declaration ::= Id* { : Signature }# == [Any]Denotation

Selections

[Operation]Selection ::= {TypePrimary $}# Id {<<[Operation]Signature>>};#

[Any]Selection ::= { TypePrimary $ }# String

Constructions

FuncConstruction ::= func[ Parameter* ] Signature# { [Any]Denotation ]
TypeConstruction ::= Enumeration | Record | Extension | Image | Union
Enumeration ::= enum [ Id* ]
Record ::= record Id* { Id* : TypeDenotation }* ]
Extension ::= extend [ TypeDenotation ]
Image ::= image [ [Operation]Signature ]
Union ::= union [ Id* ]

Type Modifications

TypeModification ::= OperationModification | SignatureModification
OperationModification ::= TypePrimary WithList
WithList ::= with Id* { [Operation]Declaration* ];
SignatureModification ::= TypePrimary ExportList
ExportList ::= export Id* { ExportElement* } |
hide Id* { ExportElement* }
ExportElement ::= Id { << Signature >> }# ExportList#

Programs

Program ::= Id == FuncConstruction
Appendix B: Collected Syntactic Constraints

The syntactic constraints that must be satisfied by legal Russell programs are described briefly below, with references to the section and page numbers of their complete descriptions in this Report.

1. Import rule: (Sec. 3, p. 11) No name having \texttt{var} signature may be imported into an operation denotation.

2. Export rules: (Sec. 3.6, p. 18; Sec. 3.9.1, p. 31) A name \texttt{x} may be exported from a scope only if the signature of \texttt{x} contains no free names that are local to the scope.

3. Overloading rules: (Sec. 3.2, p. 14; Sec. 3.6, p. 19) An identifier may be \texttt{overloaded} (i.e., given several different meanings) provided that all of its meanings can be given distinct denotations by supplying (optional) type selections and signature specifications.

Type checking rules: Type checking in Russell is performed by analyzing the \texttt{signatures} of denotations (Sec. 2, pp. 8-10). Type checking is performed on the arguments and parameters in applications, as well as on the left and right hand sides of declarations. The process proceeds as follows:

(a) A signature is assigned to each denotation. Rules for doing this are given separately for each category of denotation (Sec. 3).

(b) For each application (or declaration), parameter signatures are \texttt{expanded} by substitution of argument denotations for parameter names (Sec. 3.1, p. 13).

(c) Argument and parameter signatures are compared using the three \texttt{signature calculus} rules: \texttt{reordering}, \texttt{renaming} and \texttt{forgetting} (Sec. 3.1, pp. 13-14).