A Programmer's Introduction to RUSSELL

H. Boehm
A. Demers
J. Donahue

[0]This is a slightly edited and LaTeX-ed version of the original RUSSELL language documentation —B Bantchev, 1996, 2007

Contents

Introduction

Our intent is to provide a highly informal, but hopefully comprehensible, description of the language RUSSELL. No attempt is made to provide a complete definition of the language; it is assumed that the on-line `rhelp` facility [Boe85] is available for this purpose. Formal descriptions of (slightly obsolete versions of) the language semantics can be found in [Dem80c] (operational/denotational style), [Dem83] (equational description), and [Boe84] (axiomatic style). [Hoo84] provides a more mathematical interpretation of the type structure of the language. An overview of the type structure can be found in [Don85].

The description here should be adequate for someone to start using the language. We will describe the implemented version of the language, rather than that of [Boe80].

RUSSELL is a very compact expression language which is nevertheless extremely general and uniform in its treatment of functions and, more importantly, datatypes.

By the term 'expression language' we mean that there is no distinction between expressions and statements. Any executable construct in the language returns a value. In particular, assignment 'statements' and conditionals return values, which may or may not be used. By the same token, most constructs in the language can also potentially modify the value of some variable, and thus play the role of statements.

The following are some of the other important characteristics of the language. The concepts involved are discussed more fully in [Dem79], [Dem80a] and [Dem80b].

- The philosophy underlying RUSSELL datatypes is that *all objects manipulated by a* RUSSELL *program are in fact elements of one universal domain or set.* We may, if we wish, think of this set as being the set of all bit sequences. Thus we can identify an integer with its binary representation, a character string with a sequence of bits representing the ASCII codes for the characters, and a function with a sequence of bits representing the code to be executed to evaluate the function, possibly followed by another sequence of bits describing the environment in which the code is to be executed. Thus the object itself gives us no information about how to interpret it. The same bit string could represent either a character string or a function mapping integers to integers. (For a much more abstract view of such a domain see, for example, [Sto77].)

  What distinguishes one datatype from another then is not the set of its elements, but rather the set of operations (or functions) that may be applied to interpret elements of the universal set. In fact, the RUSSELL view is that *a datatype is just that collection of operations.* Thus, the `Boolean` type is just the collection of functions `and`, `or`, `not`, etc. What makes a value an integer is not any characteristic of the value itself, but the fact that it is intended to be interpreted by exactly these functions.

- The language design permits *static (i.e. compile time) signature checking* ('*type checking*' in PASCAL terminology). Each expression in the language has a *signature*, or *syntactic type* associated with it. Such a signature specifies in which contexts an expression may appear. Signature checking insures that an expression does not appear in a context in which it is not meaningful. Thus, the value produced by an expression cannot be misinterpreted.

  For example, assume we are representing integers using their binary representation. Furthermore, we represent the `Boolean` constants `True` and `False` as the bit strings

`1` and `0` respectively. Signature checking in RUSSELL will prevent us from writing an expression like

```
3 * False
```

in spite of the fact that it produces a well-defined answer. Clearly the `0` value produced by the expression `False` is intended to be used in a `Boolean` context, and is not intended to be treated as an integer. This is reflected by assigning the expression `False` the signature `val Boolean` (`Boolean` value). The signature of `*` is

```
func [val Short; val Short] val Short
```
[1]

This indicates, among other things, that it is a function whose arguments must have signatures `val Short`, and thus must be interpretable as short (i.e. 16 bit) integers. Since the parameter signature for `*` (`val Short`) does not match the signature of the second argument expression (`val Boolean`), the above expression will be rejected by the compiler.

It is important to distinguish signatures, which are associated with expressions, and are purely syntactic objects, from types, which are collections of functions. `Short` is a built-in type, which contains operations, such as `*`. It is convenient to use the same identifier (or expression, in general) in signatures, to indicate appropriate uses of an expression.

- Many programming languages allow users to treat *functions as data objects* in at least some contexts, e.g. as parameters. Since RUSSELL *datatypes* are just collections of functions, they *can similarly be treated as data objects.* Furthermore, since both functions and datatypes are again part of the same universal data domain as other objects, this can be done *uniformly in all contexts.*

- *Variables are data objects* which differ from others primarily in that they identify locations where other values can be stored. The `ValueOf` and `:=` operations included in most built-in types can be applied to them to obtain and change the value stored in the location. This will be discussed further later.

- RUSSELL has a *completely uniform declaration mechanism.* A declaration simply binds an identifier to the value of an arbitrary expression. New variables are obtained by binding an identifier to the result obtained by calling an allocation function.

- Since RUSSELL is an expression language, it has no notion of a procedure, distinct from that of a function. Functions may both examine and change the values of variable parameters, but may not depend on, or alter the values of other variables. In fact, *no expression with function or type signature may mention a variable not local to it.* As a consequence of this, two syntactically identical type expressions always denote the same type. This is fundamental to signature checking in RUSSELL.

- *Any legal expression can be the body of a function*, and any identifier that appears inside it may be made a parameter of the function. Thus *there are no constraints imposed on the argument and result signatures for functions.*

[1]Actually there are several instances of `*` with different signatures. Each of the types `Short`, `Long`, and `Float` has a separate `*` operation. They provide 16-bit integer, unlimited integer, and floating point multiplication, respectively.

- An attempt was made to keep the design as conceptually clean as possible, sometimes at the expense of syntactic convenience. Thus, full RUSSELL syntax can be rather verbose. On the other hand, the compiler understands enough *different kinds of abbreviations*, that real RUSSELL programs do not look too unusual. We will proceed to use abbreviated syntax, where it does not detract from explanations. Nonetheless, some of the syntax given below is unnecessarily verbose. (For a list of what the compiler can infer automatically, and under what conditions, try `rhelp inference`.)

A Simple Example

We'll start with a very simple example which will serve to illustrate some of the points which follow.

In keeping with tradition, we will use a declaration of the factorial function:

```
fact ==                                          1
        func [n: val Short] val Short            2
          {                                      3
           if                                    4
              n > 0  ==>  n*fact[n-1]            5
           #  n = 0  ==>  1                      6
           fi                                    7
          }                                      8
```

Declarations in RUSSELL always have the form `a == b`. This has the effect of binding the identifier `a` to the value produced by the expression `b`. We could use `b == 0` to declare `b` to be equivalent to the integer constant `0`. (This is completely distinct from declaring an integer variable. The identifier `b` is purely a value. It does not make sense to assign to it.)

In this case, the expression appearing in the declaration is an explicit function construction. Its value is the factorial function whose body consists of lines 4-7. The heading of the function construction (line 2) gives its signature, which indicates it is 'a function mapping a single short integer value `n` to a short integer value'.

The body of the `fact` function is an expression whose value will be returned as the value of a function call. As expected, it consists of a conditional. RUSSELL uses a slightly extended version of Dijkstra's guarded command notation [Dij76]: the keyword `else` is allowed as an abbreviation for the negation of all the other guards. Since RUSSELL is an expression language, conditionals return a value. In this latter respect RUSSELL conditionals resemble ALGOL conditional expressions.

The conditional here should be interpreted as follows: if the first guard is true, i.e. `n` is greater than `0`, then the value returned is the value of the expression `n*fact[n-1]`. Note that arguments to function applications (calls) are enclosed in square brackets rather than parentheses.[2]

If the second guard is true, then the integer `1` is returned. If neither guard is true, a run-time error occurs. If two or more guards of some conditional are all true, then any one of the corresponding expressions may be chosen to be evaluated.

Assuming $\texttt{n} \geq \texttt{0}$, the conditional could also have been written as:

[2]Function application syntax is actually quite flexible. Brackets may frequently be omitted. This is discussed below.

```
if  n > 0  ==>  n*fact[n-1]
#   else   ==>  1
fi
```

An Introduction to Imperative Russell

It is much easier to write applicative (variable-free) programs in Russell than in most other languages. Our experience indicates that small Russell programs tend to naturally be applicative. Nonetheless, it is easy to introduce variables if one prefers. We illustrate this by rewriting the `fact` function so that it uses a loop rather than recursion. (Clearly loops are not useful without variables.)

A (short) integer variable can be declared by means of a declaration, such as

```
N == Short$New[ ]
```

The type `Short` is, like all Russell types, nothing but a collection of functions. One of these functions is `New`. It allocates a new integer variable and returns it. Thus, its signature is

```
func [ ] var Short
```

The `$` symbol is used to indicate a selection from a type. `Short$New` explicitly specifies the `New` function from the type `Short`, and `Short$New[ ]` is an expression which returns a new integer variable. `N` is then simply bound to the result of this expression.

The loop construct has the same syntax as the conditional, except that `if` and `fi` are replaced by `do` and `od`, respectively (see [Dij76]). It indicates that while one of the guards is true, the corresponding expression should be executed. It always returns the special value `Null`.

An imperative version of the factorial function can be written as:

```
fact == func [n: val Short] val Short  {
  let
    N == Short$New[ ];
    F == Short$New[ ]
  in
    N := 2;  F := 1;
    do  N <= n  ==>  F := F*N;  N := N+1  od;
    V[F]
  ni
}
```

Some other language characteristics are illustrated by this example. A block can be used anywhere an expression is legal. Its general syntax is

`let` *declarations* `in` *expressions* `ni`

where both declarations and expressions are separated by semicolons. The value of the block is the value of the last expression. Thus, the value returned by the block in the example is the final value of the variable `F`. As before this becomes the value returned by the function.

We have mentioned the `V` function explicitly at the end of the block. It takes a variable as its argument and returns the value of the variable. Such functions exist for most built-in types, including `Short`. It does not usually need to be mentioned explicitly. In fact, the 'statement'

```
F := F*N
```

should technically have been written as

```
F := V[F]*V[N]
```

Assignment 'statements' have the value of the right hand side as their value. (In the example neither of these matters since they are both discarded.)

All functions referenced by the program (e.g. `+`, `<=`, `:=`, `V`) are actually components of the type `Short`, like the `New` function. Thus, technically, we should have written `Short$+` instead of `+`, `Short$V` instead of `V`, etc. In their case, unlike in the case of `New`, it is possible for a compiler to deduce that a selection from `Short` was actually meant, and we therefore omitted that information. (Constants like `1` also fit into this framework and do not have to be treated as special cases. This however is a little more complicated, and won't be discussed until later.)

Russell Lexical Structure

Now that we have hopefully conveyed some rough feeling for the language, we finally start over at the beginning.

A Russell program consists of a sequence of identifiers, keywords, strings, comments, and punctuation characters. Upper versus lower case distinctions are significant in all contexts. Identifiers may have one of three forms:

- Any Algol- or Fortran-style identifier is also a Russell identifier. More precisely, an identifier can be any letter (upper and lower case are distinct), followed by some number of letters and digits. `_` is considered to be a letter.
- Any sequence of characters enclosed in quotes (`'`) constitute an identifier. These identifiers are most commonly used for constants, as will be described in the next section. A quote character can be included in such an identifier by doubling it, or preceding it by a backslash (\) character.
- Any sequence of *operator characters.* These are:

  ```
  ! % & * + - . / : < = > ? @ \ ^ ‘ | ~
  ```

  By convention these are used as function names.

An identifier may be used in one and the same type to denote two or more of its functions, provided that each two of these functions differ in their signatures. Such an identifier is said to be *overloaded within the type.* According with this, any identifier occurring as an expression may optionally be followed by

`<<` *signature* `>>`

to explicitly indicate which, of a number of overloaded instances of the identifier within a type, was intended.[3]

Note that Russell allows for two kinds of identifier overloading. On the one hand, an identifier may denote functions in different types. On the other hand, it may have different instances within the same type.[4]

[3]The signature itself may not mention an overloaded type component, since, for any identifier inside a signature, it must be apparent which of its instances is meant, but the compiler does not consider signature information in making such a determination.

[4]The `New` identifier exemplifies both kinds of overloading. Within almost each built-in type – `ChStr`, `Short`, `Boolean`, etc. – there are two functions named `New`: besides the one that we already used, another function *allocates a variable and initializes it* to a proper value. The signature of the second function, e.g.

In most cases, none of the two kinds of overloading requires explicit use of `$` (selection) or `<< ... >>` (resolution), respectively. Wherever possible, the compiler chooses the proper function automatically.

Some of the sequences formed using the first and the third forms above are actually reserved for use as keywords and may not be used for other purposes. These are:

Keywords	*Purpose*
`cand cor`	conditional and, or
`if fi do od else ==>`	guarded commands
`then elsif`	conventional conditionals
`enum record prod union extend`	building types
`export hide with constants`	modifying types
`let use in ni ==`	declarations
`val var func type field readonly characters`	signatures
`<< >>`	explicit overload resolution
`:`	parameters, declarations

The syntax of strings is complementary to that of identifiers. The following are legal strings:
— any sequence of characters enclosed in double quotes (");
— any sequence of letters and digits starting with a digit.
The following are examples of strings:

```
123      "123"      0A1FB      "A1FB"      "Hello"      "'"
""""     ("" represents a single " inside the string)
"\""     (same as """")
```

The sequences `\n`, `\r`, and `\t` may be used to denote linefeed, carriage return, and tab characters inside strings and quoted identifiers.

The next section discusses the meaning of strings in the language. Usually they are interpreted as in other programming languages. In particular, `123` will usually represent the integer and `"Hello"` – the corresponding character string. The formal definitions differ however, both to allow treatment of infinite sets of constants within the Russell framework, and to allow the user the same mechanism in constructing his own types.

Russell comments are delimited by `(*` and `*)`. Unlike comments in most other languages, it is possible to nest comments. Thus, the following is legal:

```
(* This is a comment (* This is a nested comment *) *)
```

but not this:

```
(* (* This is an improperly constructed comment *)
```

for the `Short` type, is thus `func [val Short] var Short` and we could have written the imperative version of the factorial function as:

```
fact == func [n: val Short] val Short  {
  let
    N == Short$New[2];  F == Short$New[1]
  in
    do  N <= n  ==>  F := F*N;  N := N+1  od;
    V[F]
  ni
}
```

Constants and Strings

As we pointed out before, types in RUSSELL are just collections of functions. Thus, the only way one can talk about particular values is by having functions in that type that produce them.

Consider the built-in type `Boolean`. It contains two functions, named `True` and `False`, with signatures (roughly speaking)

```
func [ ] val Boolean
```

I.e., these are functions with no arguments which produce a `Boolean` result. Thus, we can always get the `Boolean` value `False` by writing

`Boolean$False[ ]` [5]

This allows us to deal with finite sets of constants for a given type. Since we always want to consider types as finite sets of operations, we need to extend this idea to handle infinite sets of constants (such as those in the built-in `Short` type). This is done by treating strings, in the sense described above, as abbreviations.

The only `Short` constants provided explicitly (i.e. in the same way as `True` and `False` for the `Boolean` type) are `'0'` through `'9'`. (The quotes are part of the identifiers.) Also provided is a concatenation operation `^+` which, in this case, gives the value of the integer obtained by writing an integer next to another digit. Roughly what happens then is that the expression `123` is treated as an abbreviation for:

```
('1'[ ] ^+ '2'[ ]) ^+ '3'[ ]
```

If the concatenation operator were not already built-in, it could be declared as:

```
^+ == func[x,y: val Short]  {
  let
    '10' == 5+5
  in
    '10'*x+y
  ni
}
```

Note that using `10` instead of `'10'` would result in infinite recursion.

In the above example we have again omitted the selections of the constants `'1'`, `'2'`, and `'3'`, as well as the `^+` operator, from the `Short` type. Formally strings are always selected from a type. Thus, we should have written `Short$123`. The compiler will normally infer this selection, so that, in practice, writing `123` is sufficient.

All this allows us to define the meaning of (unquoted) strings a little more precisely. In general,

`T$`$a_1 a_2 \ldots a_n$

is expanded to

`(...(T$'`a_1`'[ ] T$^+ T$'`a_2`'[ ])...) T$^+ T$'`a_n`'[ ]`

Quoted strings are treated only slightly differently. Since we want `""` to be legal,

[5]The compiler allows this to be shortened to just `False` in virtually all contexts. The declaration `a == False` would result in `a` having signature `func [ ] val Boolean`. But even such a ‘mistake’ would be unlikely to effect the correctness or efficiency of a program. The compiler would simply convert occurrences of `a` to `a[ ]`. Try `rhelp inference` for more details.

we have to generalize the treatment to handle this in a reasonable way. We do this by agreeing that `''` (two single quotes) will implicitly be concatenated onto the left of any such string. Furthermore, to distinguish such strings more explicitly from unquoted ones, `^*` will be used as the concatenation operator. Thus, `T$"ab"` is expanded to

```
(T$''[ ] T$^* T$'a'[ ]) T$^* T$'b'[ ]
```

In accordance with this scheme, the built-in type `ChStr` (character strings) has constants, i.e. nil-ary functions, with names `''` and `'c'`, for all characters `c` in the character set. It also includes a `^*` concatenation function, which, unlike the one for integers, really does character string concatenation.

We have gained something besides uniformity in this approach. It is possible to make use of the string mechanism for user-defined types. For example, we can define an 'octal integer' type by simply modifying the built-in type `Short` in the following two ways: first, the constants `'8'` and `'9'` have to be deleted. Secondly, `^+` now has to multiply by 8 rather than 10. As will be described later, RUSSELL allows the user to easily construct such new types.

Some Notes on Expression Syntax

RUSSELL provides the following primitives for building expressions:

• Selection from a type. A component function `f` of a type `T` may be selected by writing

```
T$f
```

• Function construction (lambda abstraction in lambda calculus terminology). Any expression `E` can be turned into a function by writing

function_signature[6] `{E}`

Any identifiers appearing in `E` can be treated as parameters by including them as such in the signature. The above declarations of factorial functions are simple examples of this.

• Function application. A function `f` may be applied to arguments $a_1, \ldots, a_n$ by writing

`[`$a_1, \ldots, a_i$`] f [`$a_{i+1}, \ldots, a_n$`]`

Which arguments go before the function and which go after is a choice that can be made arbitrarily (though hopefully consistently) by the programmer. If one of the argument lists is empty, it can always be omitted. (As mentioned above, if both are empty, they can usually, but not always, both be omitted.) Thus, any function can be treated as an infix, prefix, or postfix function. (We could introduce more reasonable syntax for the factorial function by declaring it as `! == func` ...)

• Blocks. The use of blocks to introduce declarations was illustrated above. RUSSELL allows two other kinds of blocks. The block

`let in` *expressions* `ni`

can be abbreviated as

`(`*expressions*`)`

[6]The result signature may be omitted if it can be inferred by the compiler.

The construct

```
use comma − separated_list_of_type_expressions in
    expressions
ni
```

tells the compiler that it should use the types given in inferring omitted selections in the second list of expressions. Otherwise it is equivalent to

```
(expressions)
```

Note that in inferring such selections the compiler will first try to use the types of the arguments and then search surrounding `use` lists inside out. (As it turns out this means that `use` lists are primarily useful for implicit selections of constants, which have no arguments.)

Furthermore, any user program is treated as if it were embedded in the following construct:

```
use  Float    in
  use  ChStr    in
    use  Boolean  in
      use  Short    in
          user_program
ni ni ni ni
```

• Sequence control constructs. Loops and conditionals were described above. Conditionals can frequently be abbreviated by the *conditional and/or* constructs:

```
expression_1 cand expression_2
expression_1 cor expression_2
```

Here all expressions return `Boolean` values. Intuitively, these constructs are similar to the operations `and` and `or` of the built-in type `Boolean`. The difference (hinted at by the absence of `[ ]` around the 'arguments') is that they are not call-by-value operations. The second 'argument' is only evaluated if necessary. Thus, e_1 `cand` e_2 is actually equivalent to

```
if
     e_1  ==>  e_2
#  else  ==>  Boolean$False[ ]
fi
```

`cand` and `cor`, though resembling operations by their syntax, are nevertheless *partially evaluated expressions*, and therefore can be used, as well as `if ... fi` and other constructs, to implement *lazy evaluation.* (Also try `rhelp LList`).

• Type constructions and modifications. These provide ways to build new types out of existing ones. They are discussed below.

A few more remarks on the syntax of applications and selections are in order at this point. First, an expression, such as `[a]b$c` is, in principle, ambiguous. The function `b` could produce a type, and thus the expression could be interpreted as `([a]b)$c`. However, by having selection bind more tightly than application, the correct interpretation is `[a](b$c)`.

The second problem is that the above syntax would require us to write the statement

```
x := x+1
```

as

```
[x] := [[x]+[1]]
```

even if we assume that the constant application and all selections are automatically inferred.[7] This would, at best, be acceptable only to Lisp programmers. Thus, the actual syntax allows dropping the brackets for functions used as either binary infix, or as unary prefix operators. A standard Fortran-style precedence scheme is used to disambiguate the resulting expressions. It is worth noting that this scheme relies purely on the identifiers appearing in the expression, and not at all on any semantic or signature information. This applies to the entire Russell parser.

It is not worth discussing the exact precedences here. In general, standard operator symbols have conventional (not Pascal-like) precedences. It is always safe to bracket expressions containing nonstandard operators. For details, try `rhelp expressions`.

The Anatomy of a Signature

As previously mentioned, each Russell expression has a signature associated with it. This signature describes how the result produced by that expression should, and should not, be interpreted.

Russell expressions always produce objects which are intended to be used as either first-order (i.e. non-operation) values, variables, functions, or types. In order to distinguish between these, there are four different kinds of signatures, corresponding to the above four categories. They are described somewhat informally below.

Value signatures

The general form of such a signature is

```
val T
```

where `T` is an expression denoting a type. Informally, it indicates that the value produced by the corresponding expression should be interpreted as a value of type `T`. More formally, it means only that the result produced by the expression should only be interpreted by (i.e. passed as a parameter to) a function that expects an argument of signature `val T`. Thus, one technically correct way of viewing the whole issue is that the expression `T` is just a tag used to match up functions with proper arguments. The use of a type expression as tag turns out to be particularly convenient.

An alternate, and probably more enlightening view is the following: we want to guarantee that the first-class value in question is passed only to functions that know how to interpret it. An obvious way to do that is to keep track, in its signature, of all the functions that can be applied to it. Since usually all these functions are components of a type, we use the expression representing that type as a shorthand. (In fact, some functions not in `T` may also expect `val T` arguments. These however are usually built out of the 'primitive' functions in `T`.)

As an example consider the following expressions, where it is assumed that all identifiers have the natural meanings:

(a) `1+2`

[7] The full unabbreviated version of the above statement is

```
[x] Short$:= [[Short$V[x]] Short$+ [Short$'1'[ ]]]
```

(b) `BoundedStack[Short, 10]$push[S,3]`
(c) `BoundedStack[Short,5+5]$push[S,3]`

(a) is a simple integer expression. It has signature `val Short`.

In (b) and (c) we assume that `BoundedStack` is a function which takes two arguments. The first is the type of the individual elements to be pushed onto the stack. The second is the maximum size of the stack. Its result is the corresponding type of bounded stacks. We assume that everything works in an applicative fashion, so that the push operation returns a stack value. The signature of (b) then is `val BoundedStack[Short,10]`. That of (c) is `val BoundedStack[Short,5+5]`.

It should be emphasized that signatures are purely syntactic objects, which are determined using some simple rules described below. In particular, nothing in the whole signature mechanism knows anything about the semantics of `+`. Thus, the signatures of (b) and (c) are completely distinct, and the functions in `BoundedStack[Short,5+5]` cannot be used to interpret objects of signature `val BoundedStack[Short,10]`. (This design decision rarely causes any real inconvenience. Furthermore, if we want to do any syntactic 'type checking' at all, it is clearly essential. Semantic equivalence of RUSSELL type expressions is in general undecidable.)

⊔ Variable signatures

The general form of a variable signature is

```
var T
```

where `T` is a type expression, as above. This indicates that the result of the expression is a variable (or location) which can hold an object to be interpreted as by the functions of type `T`. Usually (though not always) the only thing that can be done with an object of signature `var T` is to pass it as an argument to `T$:=` or `T$V`, or to bind an identifier to it.

The most common example of an expression of signature `var T` (other than a simple variable) is

```
T$New[ ]
```

where `New` is the function in `T` that allocates a new variable. Another example is

```
A.1
```

where `A` is a *variable* of an appropriate array type, and `.` is the name of the subscription operator in the (built-in) array type. It produces the *location* of the first element of the array. (Array types usually have two versions of the `.` operation. In addition to the one mentioned here, there would normally be an operation mapping an array *value* and an index into a component *value*. Try `rhelp Array`.)

⊔ Function signatures

In order to insure that functions are passed only appropriate arguments, it is clearly necessary that their signature include the signatures of the arguments, as well as the signature of the result.

Function signatures may also include the names of the formal parameters. Although these are not important in determining the correctness of a particular application, there are nevertheless two reasons for putting them here. First, the syntax of function constructions requires it. (There's nowhere else to put them.) The second, more important one, should become apparent when we state the rule for determining the signature of an application.

The syntax for function signatures is

> `func[`$param_1$`;` ... `;` $param_n$`]` *result_signature*

where each $param_i$ is a list of parameter names with identical signature, and the signature itself:

> id_1 `,` ... `,` id_m`:` *parameter_signature*

If one of the $param_i$ includes only a single parameter name which is not otherwise used, then both it and the `:` can be omitted. A signature, such as

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

is viewed as being identical to

```
func[x: val Short; y: val Short] val Short
```

and therefore to

```
func[val Short; val Short] val Short
```

Function signatures are usually written explicitly only in function headings. Even in this case, the result signature may be omitted whenever it can be easily determined from the body of the function.

Some examples of function signatures were already given above. More interesting examples will be presented below in conjunction with type signatures.

◘ Type signatures

If we want to specify how a type expression can be used, we need two kinds of information. First, we need to know what operations can be selected from it. Second, we need to know how those operations themselves can be used. Thus, a type signature consists of operation names, and of the signatures corresponding to those names. The syntax is:

> `type {`op_1`;` ... `;` op_n`}`

The syntax for the individual op_i is the same as that used for parameters in a function signature (except that no names can be omitted, and all signatures have to be either type or function signatures).

The simplest example is the built-in type `Void`, which has no operations as part of the type. Its signature is therefore:

```
type {}
```

There is a built-in function `Null`, which has signature `func[ ] val Void`. (Recall that a `Void` value is also produced by the `do` ... `od` loop.)

Now consider a type `VOID` which has the function `Null` as its only component.[8] It would have signature

```
type {Null: func [ ] val VOID}
```

Unfortunately, this notion of a type signature won't get us very far.

[8]Such a type could be declared as `VOID == Void with {Null == Null}`. It would make sense to use this as the built-in type. The other alternative was chosen only so that we could write `Null[ ]` instead of `Void$Null[ ]`

It is frequently useful to build a new type which behaves like an existing one, but is nevertheless distinct from it. For example, we may wish to have two types, `Meters` and `Feet` with the same operations of addition, subtraction, etc. By keeping them distinct we can use the signature checking mechanism to guarantee that we don't get the two mixed up.

We should be able to build both of these types by first constructing `Meters`, and then simply using the declaration

```
Feet == Meters
```

to get the type `Feet`. (Note once more that the signature mechanism is purely syntactic. Thus, `val Feet` and `val Meters` are still distinct signatures, so we can achieve the desired protection. In fact, this is probably too much protection. In reality, we would want to introduce some conversion functions at some point. Again the signature checking mechanism can assure that these are used exactly where they are appropriate.)

To illustrate what goes wrong in a simple context, let's try the analogous exercise with the type `VOID`. We use the declaration

```
VOID2 == VOID
```

to get a second type with identical characteristics. Certainly its signature has to be the same as that of `VOID` (see above), but what we wanted was

```
type {Null: func [ ] val VOID2}
```

The problem is that `Null` should return value of whatever type it is a component of, not the specific type `VOID`. Therefore, we need to give this type *a name in the signature*. Such a 'local type name' is written immediately after the `type` keyword in the signature. Thus, a more appropriate signature for `VOID` would be

```
type V {Null: func [ ] val V}
```
[9]

Now the construction of `VOID2` works properly.

We conclude with two more examples. The built-in type `Short` will serve as the first. Its signature is

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

The above makes use of yet another form of abbreviation. Certain operation names tend to occur in many types with the same signature. They therefore have default signatures associated with them which can be omitted from a type signature. They are:

Identifier	*Default signature*
`New`	`func [ ] var T`
`:=`	`func [var T; val T] val T`
`=  <  >  <=  >=  <>`	`func [x,y: val T] val Boolean`

[9]This version of `VOID` is harder to obtain. It might be declared as

```
VOID = extend  {Void}
       with V  {Null == func[ ] {V$In[Null]}}
       export  {Null}
```

These constructs are discussed below.

`^+  ^*`	`func [x,y: val T] val T`
`V`	`func [var T] val T`
any quoted identifier	`func [ ] val T`

In all of the above, `T` represents the local type name. If none is explicitly specified, it can be thought of as being generated by the compiler.

As a final illustration, the following is a possible signature of the `BoundedStack` function used above:

```
func [T: type {New; V; :=}; val Short]
    type S  {push : func [val S; val T] val S;
             top  : func [val S] val T;
             pop  : func [val S] val S;
             empty: func [ ] val S}
```

The signature indicates that the `BoundedStack` function expects an integer and a type argument. The type argument used must include the functions `New`, `V`, and `:=`.

The signature checking rules described below allow the actual type argument to have additional components as well. Thus, the built-in `Short` type would be an acceptable argument. In fact, if we were willing to be slightly clever about the implementation of the `BoundedStack` function, we would not even have to require that the parameter include the three functions specified. The parameter signature would then be given simply as

```
type {}
```

and any type whatsoever could be passed to it. (If the three operations are provided, then an array implementation could be used.)

The Signature Calculus

The RUSSELL type checking system can be described by a pair of rules for each language construct. The first rule gives constraints on the signatures of the subexpressions. These constraints must be satisfied for the expression to be signature correct. The second gives the signature of the construct itself.

Signature constraints usually require that two or more signatures be 'the same'. This means that they should be syntactically identical, with the following exceptions:[10]

- Parameter names and local type identifiers may differ, provided they are uniformly renamed.
- Type signatures may list components in different order. (The components must, however, have the same names.)
- Parameters with identical signatures may, or may not, be grouped together.

As we pointed out earlier, type expressions may not depend on the values of variables. Thus, if two signatures are the same, then all type expressions they contain must actually denote the same types.

As an example of the general approach, consider a function construction, such as

```
func [x: val Short] val Short  {
  if  x%2 = 0  ==>  x/2   #  else  ==>  3*x+1  fi
}
```

[10]The compiler allows some other minor differences, such as reordered guards in conditionals. We list only those that are likely to be of interest.

(`x%2` yields the remainder of dividing `x` by `2`.)

A large number of constraints needs to be checked to insure that this is signature correct. We will elaborate on a few of them.

The checking rule for the conditional requires that all its arms have the same signature.[11] In our example this is satisfied since `x/2` and `3*x+1` both have signature `val Short` (for reasons discussed below). The checking rule also requires all guards to have signature `val Boolean`. The guard `x%2 = 0` satisfies this constraint. The second guard abbreviates `not x%2 = 0` and thus also has the appropriate signature.

The signature of the conditional itself is that of the arms. In our example, the conditional, and thus the body of the function, has signature `val Short`.

Similar rules exist for function constructions. We require that the specified result signature (if any) match the signature of the body. The signature of the construction is that specified by the heading (with the possible addition of an inferred result signature). Thus, the above function construction has signature

```
        func [x: val Short] val Short
```

which may be abbreviated to

```
        func [val Short] val Short
```

since the parameter name is not significant for signature checking.

Signature calculus rules for the other language constructs are given by the corresponding `rhelp` files. Only two of them are non-obvious, and they are described below. We first examine function applications.

Given our examples so far, an obvious set of rules for function applications would be:

- the parameter signatures in the function signature must be the same as the signatures of the corresponding argument subexpressions;
- the signature of the function application is the result signature of the function.

Unfortunately, this is no longer adequate, once we start taking advantage of the Russell type structure. As a simple example, consider writing an identity function. A first attempt might be:

```
        identity == func [x: val ?] val ?  {x}
```

Clearly, it should be possible to apply this function to values of *any* type.[12] Thus, substituting for `?` a specific type, say, `Short`, is not acceptable.

This situation is dealt with in Russell by passing the type associated with the argument as another argument. Thus, we would rewrite the identity function as

```
        identity == func [x: val T; T: type {}] val T
```

If we wanted to apply the identity function to the (short) integer `17`, we would write

[11]This constraint is not enforced when the value of the conditional is immediately discarded. Specifically, if `x`, `y`, and `z` are `Short` variables, then

```
        if  x > 0  ==>  y := x   #  else  ==>  put["error"]  fi;
        z := y
```

is signature correct, in spite of the fact that the first arm of the conditional has signature `val Short`, and the second has signature `val ChStr`. A similar situation occurs if the conditional is the body of a function with `val Void` signature.

[12]It should actually work for functions, types, and variables as well. Unfortunately, the implemented version of Russell does not deal with the variable case, and it deals with the other two clumsily. An implementation of a later version of the language would resolve this problem.

```
identity[17,Short]
```

which the compiler allows us to abbreviate as

```
identity[17]
```
[13]

This application is not signature correct by the first rule above. First, `17` has signature `val Short`, and not `val T`. Clearly, the `T` in the parameter signature is intended to denote the second parameter, and should not be used literally in the comparison. Thus, the checking rule needs to be adapted to read:

- The signatures of the argument subexpressions must match the parameter signatures *after any parameter names in the parameter signatures have been replaced by the corresponding actual argument.*

Thus, we first replace the `T` in `val T` by `Short`, and then check that the resulting parameter signature matches the signature of `17`.

A second problem occurs when we check that the argument `Short` matches the second parameter signature. The type `Short` contains a large number of operations, but the parameter signature calls for none. As mentioned above, this should not matter; the signature `type{}` was intended to indicate that we did not require there to be any operations in the type `T`. Thus, the word 'match' in our revised rule should be interpreted to mean that the two signatures are either the same, or they are both type signatures, and the argument signature contains a superset of the operations in the parameter signature. Any type signature 'matches' the signature `type {}`.

Finally, the second rule needs to be updated to correspond to the revised checking rule. The application `identity[17,Short]` should have signature `val Short`, rather than `val T`. Thus we write:

- The signature of the function application is the result signature of the function *with parameter names replaced by actual arguments.*

A situation similar to that occurring for applications occurs for selections of an operation from a type. The `+` component of the type `Short` has signature

```
func [val S; val S] val S
```

where `S` is the local type identifier. But the signature of `Short$+` should be

```
func [val Short; val Short] val Short
```

(In particular, we need to insure that `1+2` has signature `val Short` and not `val S`.) Thus, the signature of a selection is the signature of the component, with the local type identifier replaced by the type expression preceding the `$` sign.

More About Types

Much of Russell is devoted to the creation and manipulation of types. This is what gives the language most of its flexibility. Rather than discussing the necessary facilities in detail, we give an overview of what's available, and again refer to the `rhelp` facility to fill in the details.

Types in Russell are obtained:

- as built-in types,
- by applying built-in type-producing functions,
- through the use of type constructors provided by the language, or

[13]Trailing arguments may be omitted if they can be inferred from the explicit ones.

- by modifying some existing type.

⊔ Built-in types

- `Void`

Provides no operations. `Void` values are used where the value of an expression is not of interest. (`Void` variables are abused by the current implementation to mean 'the whole machine state'. See `rhelp Void`.)

- `Boolean`

Provides standard Boolean (logical) operations.

- `ChStr`

Provides character strings. It implements string constants, single character input, string output, concatenation, element selection, and substring operations. Functions to convert between strings of length 1 and the ASCII code of the character they contain are also present.

- `Short`

Provides 16 bit integer operations.

- `Float`

Provides (64 bit) floating point operations.

- `Long`

Provides operations on integers of virtually unlimited size ('bignums' in LISP terminology).

⊔ Built-in type producing functions

- `Array`

Requires a (`Short`) size, and an element type as arguments. The result is a type, containing operations to access arrays of the given size and element type. A two-dimensional array of 100 by 100 floating point numbers might be declared as:

```
matrix == Array[100,Array[100,Float]];
A == matrix$New[ ];
```

The element $\texttt{A}_{ij}$ can then be referenced as `A.i.j`.

- `List`, `LList`

`List` produces a type that provides operations, such as `head`, `tail`, `cons`, `is_nil`, etc. on linear lists, consisting of elements of the specified, as an argument to `List`, type. This function is provided only for convenience and implementation efficiency; it can easily be implemented within RUSSELL.

A polymorphic function, named `^`, is defined (with two instances, each through `List`'s `cons`) in the initial environment. Because of the right associativity of `^`, it is possible to write the list with elements `3`, `5`, and `8` as `3^5^8` without further type specification or other baroque syntax. This allows a convenient syntax for procedures with a variable number of arguments of uniform type.

`LList` differs from `List` in that the resulting type provides a version of `cons` which is *lazy* in its second argument: the tail of a list is not necessarily computed until it is actually needed. This allows the manipulation of *conceptually infinite objects.*

- `Ref`

Provides operations on references (pointers). It takes a type argument and produces a type, containing functions that create pointers and do assignments, comparisons, referencing and dereferencing.

It is frequently useful to build a `create` function that allocates a new object and returns a pointer to it. This can be most efficiently done by defining a macro `create(T)` to be `(Ref[T])$In[T$New[]]`. It is also possible to add a `create` function to the reference type (using a `with` clause).

⊔ Type constructors

Note that the above types and functions are simply predeclared identifiers, and are not otherwise special. Type constructors, on the other hand, are really language primitives, with associated syntax.

- `prod`

The `prod` constructor creates Cartesian product types. Cartesian products are tuples of objects with named components. Each product type provides functions to build a tuple, giving values for all its fields, to allocate (an empty) tuple, to assign a tuple value to a tuple variable, and to extract the value of either the tuple or any of its components. Note, that individual fields may *not* be modified.

A product construction has syntax:

`prod` *local_type_name* `{` *parameter_list* `}`

where *parameter_list* is a semi-colon spearated list, with each element of the form

name`:` *signature*

`var` signatures are *not* allowed.

For example, the resulting type of

```
prod {i: val Short; x: val Float}
```

contains operations on pairs, the first of which represents an integer, and the second – a floating point number. These would include `Mk` to build a pair, `i` to obtain the first component, etc.

Expressions appearing inside component signatures are not evaluated. Thus, recursive product types are acceptable.

The signature of one component may depend on another component. We may build a product, such as:

```
prod {x: val T; T: type { ... }}
```

Values corresponding to such a type are, in a sense, self-describing. They contain both a value (`x`) and information about how to interpret it (`T`).

Because of the possible dependence of one component on another, Russell product types are not Cartesian products in the strict sense.

Since the result of an expression with product type can be assigned to a variable, and a product may have function or type components, product types may also be used simply to convert between functions and assignable objects.

As an example, the function `fact`, defined earlier, cannot be directly assigned to a variable. (All assignment operations provided by the language require the second subexpression to have `val`, and not `func` signature.) On the other hand, if we let `T` be

```
prod {x: func [val Short] val Short}
```

then `T$Mk` is an operation which builds a product, i.e. has signature

```
func [func [val Short] val Short] val T
```

Thus, `T$Mk[fact]` has signature `val T`, and so it can be assigned to a `var T` variable (using the assignment operation provided by the product). The `T$x` operation can be used to convert back.

- `union`

The `union` constructor allows the creation of disjoint union types. Syntactically, a union construction is like a product construction, in which `prod` is replaced by `union`.

An object of a union type can be thought of as having (any) one of the types given in the union construction. In the RUSSELL view, a union type contains operations to convert back and forth between the union type and a number of other types – its disjoint components. Thus, if `T` is defined as

```
T == union {i: val Short; x: val Float}
```

then `T$from_i` will convert from `val Short` to `val T`, `T$to_i` will convert in the other direction, `T$from_x` and `T$to_x` will perform the analogous functions between `val Float` and `val T`. `T` will also contain operations `is_i` and `is_x` to test which of the `to` operations may be applied.

It is allowable to give the union a local name to facilitate recursion. For example

```
union U  {
  end: val Void;
  other: val prod {first: val Short; rest: val U}
}
```

operates on elements which are either `Null` or consist of an integer and another similar element. Thus, such an object is effectively a list of integers.

- `record`

The `record` constructor is similar to `prod` but, unlike a product variable, a record variable can be viewed as a collection of component variables, each of which can be separately updated. It is still possible that an individual field be protected from such an updating.

A `record` construction differs from the `prod` and the `union` constructions in that individual fields are specified by type-producing expressions instead of signatures. It is written as:

`record {`$name_1$`:` $type\_expression_1$`; ...;` $name_n$`:` $type\_expression_n$`}`

Each $type\_expression_i$ must have a signature that includes `New`, `V`, and `:=`, with their standard signatures, so that the corresponding field might be treated ‘as a variable’.

- `enum`

Builds enumeration types, i.e. finite scalar types with named elements. The syntax is

`enum {`$name_1$`, ... ,`$name_n$`}`

The type that a call to `enum` returns provides the functionality of the analogous PASCAL scalar types.

• `extend`

Creates a copy of an exisiting type. The copy possesses two more functions, with respect to the original. They are named `In` and `Out`, and provide conversion from the original type to the new one and vice versa.

The syntax is:

`extend` {*type_expression*}

The `extend` type constructor is frequently used in combination with the `with` and/or `export`/`hide` type modifiers.

Type modifiers

• `with`

Adds operations to, or replaces operations in, (a copy of) an existing type. The syntax is

old − type_epression `with` *local_type_name*
`{` *new_operation_declarations* `}`

The local name is optional, and may be used to refer to newly introduced components (including self-referencing, i.e. recursive function definitions), or old components which have not been replaced. (*All* old components are accessible by selecting from the old-type name.)

• `hide` / `export`

Remove specified/unmentioned operations from (a copy of) an existing type. The syntax for `hide` is:

type_expression `hide` *local_type_name*
{*hide_element*; ... ; *hide_element*}

and similarly for `export`. Try `rhelp exp_hide` for details.

Examples

We conclude with some examples.

• *Example 1*

The following is the factorial function we saw before, modified to compute results of essentially unbounded size, and embedded in enough context to turn it into a complete program:

```
let
  ! == func [n: val Short ]  {
    if
       n > 0  ==>  Long$In[n] * ((n-1)!)
    #  n = 0  ==>  Long$1
    fi
  };
  x == Short$New[ ];
in
  do  (put["Factorial of?"];  x := get[FS]) >= 0
          ==>  put[x!];  put["\n"]
  od
ni
```

The program will calculate factorials until it is given a negative input. Note that the `do` ... `od` construct in RUSSELL can be used to simulate a number of other loop constructs, including a `repeat` ... `until` loop, by moving more of the loop body into the guard.[14]

- *Example 2*

This, and the following example, illustrate the use of functions as objects to be manipulated by the program.

Certain operations are naturally viewed as mapping functions to functions. Many programming languages force us to modify this view, and to recast them in a different framework. RUSSELL allows them to be represented directly.

Here we look at (a naive view of) numerical differentiation. We give a function `derivative` which returns an approximation to the derivative of a given function. We illustrate its use by embedding it in a program which uses it to multiply 13 by 2, the hard way:

```
let
  epsilon == 0.0001;
  derivative ==
     func [f: func [val Float] val Float]  {
       func [x: val Float] val Float  {
         (f[x] - f[x-epsilon])/epsilon
       }
     };
  square == func [x: val Float]  {x*x};
  double == derivative[square]
in
  put[double[13.0]];  put["\n"];
ni
```

- *Example 3*

The last illustration is an unusual implementation of binary trees. The following is a function which expects a type, describing values, stored at leaves as an argument, and produces a type of binary trees as its result. The result type contains functions to obtain the left or right subtree of a given tree, to obtain the value stored at a leaf, to build a leaf containing a given value, to combine two subtrees into a new tree, and to inquire whether a tree consists solely of a leaf. (The latter is provided directly by the `union` construction and not explicitly implemented.)

A non-leaf tree could be represented as an explicit `prod` or `record` type. We instead use a function which maps `{lft,rgt}` to the left and right subtrees:

```
func [L: type {}]  {
  let
    lr == enum {lft,rgt};
  in  use  lr  in
    union  B  {leaf: val L; interior: func [val lr] val B}
    with  B  {
      lft_subtr == func [x: val B]  {B$to_interior[x][lft]};
      rgt_subtr == func [x: val B]  {B$to_interior[x][rgt]};
```

[14]This is (intentionally) not true of Dijkstra's original version of the construct.

```
          leaf_value == B$to_leaf;
          make_leaf == B$from_leaf;
          make_tree == func [l,r: val B] val B  {
            B$from_interior [
              func [x: val lr]  {
                if x = lft  ==>  l
                #  x = rgt  ==>  r
                fi
              }
            ]
          }
        }
        export  {New; :=; V; is_leaf; lft_subtr; rgt_subtr;
                 leaf_value; make_leaf; make_tree}
      ni ni
    }
```

Other Facilities

The implementation also provides facilities for limited separate compilation of both RUSSELL and non-RUSSELL program segments (`rhelp extern`). Limited direct access to operating system facilities is also provided (`eof`, `argc`, `argv`).

References

[Boe80] Boehm H., A. Demers, J. Donahue. *An Informal Description of* RUSSELL*.* Tech. Rep. 80-430, Comp. Sci. Dept., Cornell University, 1980.

[Boe84] Boehm H. *A Logic for the* RUSSELL *Programming Language.* Thesis, Cornell University, 1984.

[Boe85] Boehm H. RUSSELL *on-line* `rhelp` *facility.* Distributed with the RUSSELL Compiler.

[Dem79] Demers A., J. Donahue. *Data Types are Values.* Tech. Rep. 79-393, Comp. Sci. Dept., Cornell University, 1979.

[Dem80a] Demers A., J. Donahue. *Data Types, Parameters, and Type-Checking.* Proc. Seventh Ann. Principles of Progr. Lang. Symp., 1980, pp.12-23.

[Dem80b] Demers A., J. Donahue. *Type-Completeness as a Language Principle.* Proc. Seventh Ann. Principles of Progr. Lang. Symp., 1980, pp.234-244.

[Dem80c] Demers A., J. Donahue. *The Semantics of* RUSSELL*: An Exercise in Abstract Data Types.* Tech. Rep. 80-431, Comp. Sci. Dept., Cornell University, 1980.

[Dem83] Demers A., J. Donahue. *Making variables abstract: an equational theory for* RUSSELL. Proc. Tenth Ann. Principles of Progr. Lang. Symp., 1983.

[Don85] Donahue J., A. Demers. *Data Types are Values.* ACM TOPLAS 7, 3 (July 1985), pp.426-445.

[Dij76] Dijkstra E. *A Discipline of Programming.* Prentice-Hall, 1976.

[Hoo84] Hook J. *Understanding* RUSSELL *– A First Attempt.* Semantics of Data Types, Proceedings, Lect. Notes in Comp. Sci. bf 173, 1984, pp. 69-86.

[Sto77] Stoy J. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977. See esp. ch.7.

Appendix: Available On-line Help

The following help files are available through the `rhelp` facility.

`abort`
: Function to ungracefully terminate execution

`alias`
: Function to test whether two variables refer to the same location

`applications`
: Function calls

`arg_c_v`
: Access command line arguments

`Array`
: Function for constructing array types

`Boolean`
: Built-in type

`Callcc`
: Call with current continuation

`cand_cor`
: Partial evaluation of and/or functions

`ChStr`
: Built-in character string type

`compiling`
: How to invoke the compiler

`do`
: The Russell loop construct

`enum`
: Enumeration type constructor

`eof`
: Test for end of file on the standard input

`exp_hide`
: Removing components from types

`Expand_Hp`
: Fuction to expand heap

`expressions`
: Types of expressions and how they're parsed

`extend`
: Building new types with conversion functions

`extern`
: How to access separately compiled Russell and non-Russell programs

`File`
: Built-in datatype to provide o.s. file manipulation

`Float`
: Built-in double precision floating point datatype

`func`
: Function constructions

`general`
: How to get started; also printed if no arguments are given

`identifiers`
: Lexical rules

`if`
: Russell conditionals, both guarded command and conventional

`import_rule`
: Restriction on the use of global variables in functions

`inference`
: Acceptable abbreviations

`initialization`
: Definition of (names in) the global environment

`intro`
: A copy of this introduction

`let`
: Blocks and declarations

`limits`
: Limits imposed by the compiler

`List`, `LList`
: Functions for producing linear list types

`Long`
: Unlimited size integers (bignums)

`problems`
: Common problems

`prod`
: Product type constructor

`record`
: Record type constructor

`Ref`
: Function for producing pointer types

`selections`
: Accessing a component of a type

`Short`
: The built-in 16 bit integer datatype

`Signal`
: Utilization of UNIX signals

`signatures`
: Brief description and syntax

`strings`
: The unorthodox treatment of character strings and numeric constants

`trace`
: Some simple debugging facilities

`union`
: The type constructor

`Void`
: The built-in type and the objects, related to it: `Null`, `FS`, `impure`

`with`
: Adding operations to types