The Russell Semantics:

An Exercise in Abstract Data Types

A. Demers

J. Donahue

TR 80-431

Department of Computer Science
Cornell University
Ithaca, New York 14853

September 1980

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

1. Introduction

Russell is a new programming language designed by the authors. The language is described informally in [Demers80c]; the ideas leading to its development are discussed in [Donahue79, Demers80a, 80b]. This paper presents a complete formal semantics for Russell in terms of an interpreter written as a Russell program; thus, some familiarity with [Demers80c] is assumed.

The most obvious feature of the Russell semantics given below is its "metacircularity;" the meaning of Russell programs is given by an interpreting program written in Russell. This Russell interpreter can also be read as a "denotational description" of the language in the style of [Stoy77, Gordon79, Tennent76]. In the presentation below, we will discuss how to give the semantics a denotational reading.

If we intended to give a denotational semantics for Russell (which was our original purpose), why did we adopt the idiosyncratic syntax of Russell instead of the (idiosyncratic) syntax of the "Oxford school?" The obvious answer is that we have grown comfortable with Russell and find it a natural tool for expressing programs. But a far more important reason is that Russell allows us to present a very "abstract" semantics.

One of the most serious problems in presenting the semantics of any "real" programming language is the considerable amount of detail that must be specified; this means that the definition must be carefully structured if it is to be read by any but the most diligent. The literature on programming methodology suggests that one way to organize large complicated programs is to use abstract data types to hide details of representation and implementation from those parts of the program in which they are not relevant. As Mosses notes [77], most existing denotational definitions fail to use this basic idea of program design, even though it seems immediately applicable to the problems of presenting semantics. It has seemed to us that one of the reasons for this is that no language has been available that has both a mechanism to allow information-hiding and a type structure that can express the value spaces commonly used in denotational semantics (for example, denotational semantics frequently use recursively defined function spaces). Russell has both of these necessary properties, so we decided to exploit this fact in defining its semantics. As we describe below, the type structure of the language and the polymorphic features of Russell allow us to partition the semantics into two separately understandable pieces -- a set of function definitions that depend solely on certain abstract properties of the underlying domains (or types) and a set of data type definitions that provide the meaning of the underlying domains.

The paper is organized as follows. In the next section, we present some notational preliminaries used throughout the paper (in particular, we specify the syntax of the defined and the defining languages). We then present the abstract syntax of (defined) Russell and discuss the translation of Russell programs from concrete to abstract form. We then present a polymorphic interpretation function that gives the meaning of Russell programs. We then give definitions of data types that can be used as arguments for the interpretation function. We close with some comments about further research suggested by this exercise.

2. Notational Preliminaries

2.1. Defined and Defining Languages

The language to be defined in this report is described informally in [Demers80c]. The *defining* language is the applicative subset of the defined language. This applicative subset is simply Russell without *vars* -- no use will be made of any identifier with *var* signature (and thus of any operation having a *var* parameter or yielding a *var* result).

Additionally, two builtin type-producing functions will be used in the defining language. The first of these is the List builtin function with signature

```
func[ T : type{} ]
  type L { Nil     : func[] val L;
           MkL     : func[ val T ] val L;
           ||      : func[ val T; val L ] val L;
           empty?  : func[ val L ] val Boolean;
           first   : func[ val L ] val T;
           rest    : func[ val L ] val L  }
```

where the operations satisfy the following set of axioms:

```
MkL[x] = x || Nil[]
empty?[ Nil[] ] = Boolean$True[]
empty?[ x || list ] = Boolean$False[]
first[ x || list ] = x
rest[ x || list ] = list
empty?[ list ] = False ==> (first[list] || rest[list]) = list
```

(The use of the "=" symbol here can be read as "behaves the same as". Thus, the denotation "x||Nil[]" can be substituted for the denotation "MkL[x]" in any denotation. Care must be taken here because "=" itself is often introduced as part of the meaning of a data type. In the following, we will use

"T$=" in axioms if we are defining the meaning of the equality operator defined by a particular type and just "=" to mean "behaves the same as".)

The List function is defined by

```
func[ T : type() ]
 { let
     ConsNode ==
       record{ hd : T; tl : union{ ConsNode, Void } }
         with CN{
           Mk == func[x: val T ; y: val ConsNode]{ Mk[ x, FromConsNode[y] ]};

           Mk == func[x: val T ; y: val Void]{ Mk[ x, FromVoid[y] ] }   }
   in
     union{ ConsNode, Void }
       with List{
         Nil == func[] { List$FromVoid[ Void$Null[] ] };

         MkL == func[ x : val T ] { x || List$Nil[] } ;

         ||  == func[ x : val T; y : val List ]
                 { if
                     IsConsNode?[y] ==> FromConsNode[ Mk[ x, ToConsNode[y] ] ]
                     [] else ==> FromConsNode[ Mk[ x, ToVoid[ y ] ]
                     fi } ;

       empty? == func[ x : val List ] { IsVoid?[x] };

       first == func[ x : val List ] { hd[ ToConsNode[ x ] ] } ;

       rest == func[ x : val List ] { tl[ ToConsNode[ x ] ] }     }
   ni }
```

Note that in the definition of ConsNode, we have "overloaded" the identifier Mk to refer to three functions that each produce new ConsNode values, but take different second arguments. Each of the overloaded operations is defined in terms of the Mk that is part of the meaning of record types.

The second builtin type-producing function is ×, which is simply used to allow some shorthand in record definitions. The function × is defined by

$$\times == \underline{func}[\ T_1,\ T_2 : \underline{type}()\ ]\ \{\ \underline{record}\ Pair\{\ first : T_1;\ scnd : T_2\ \}\ \}$$

In essence, we are simply giving canonical names to the selectors of a record type.

One final point should be noted before we press on to define the semantics of Russell. Both the defining and the defined language in this paper use call-by-value order of application. But since free identifiers appearing in constructions are not evaluated (constructions are not applications), recursive unions and records are provided, both in the defined and in the defining language.

3. Abstract Syntax

The concrete syntax of the Russell language is given in an appendix. The abstract syntax of Russell allows us to remove from the semantic description many purely syntactic matters. These include:

1. signature-matching (the semantics assumes that the signature-matching constraints of [Demers80c] have been satisfied),

2. operator position (we do not care whether a particular operator appears before or between the operands, only that an operator has a particular set of operands),

3. the formation of identifiers from characters,

4. operator overloading (we will assume that all operators appearing in an abstract program have unique names, unlike the concrete syntax where names can be qualified with a signature), and

5. the import rule and the constraints of syntactic validity of a set of declarations (again we will assume that all concrete programs satisfy these constraints).

The abstract syntax of Russell denotations in the defined language is given as follows. Denotations in Russell are either identifiers, combinations, constructions or type modifications. Combinations in Russell include applications, type selections, conditionals, repetitions and blocks.

```
DEN     == union{ ID, COMBIN, CONSTR, MODIFY }
COMBIN  == union{ APPL, SELECT, COND, REPEAT, SEQ, BLOCK }
APPL    == record{ opr : DEN; opnd : List[DEN] }
SELECT  == record{ type : DEN; component : ID }
COND    == record{ arms : List[GDEN] }
GDEN    == record{ guard : DEN; rhs : SEQ }
REPEAT  == record{ arms : List[GDEN] }
SEQ     == record{ dlist : List[DEN] }
BLOCK   == record{ ldecl : DECL; body : SEQ }
DECL    == record{ lhs : List[ID]; rhs : List[DEN] }
```

Constructions in Russell are the basic ways of producing function and type values. Type constructions include enumerations, records, extensions, images and unions.

```
CONSTR  == union{ FUNC, TCONSTR }
FUNC    == record{ parms : List[ID]; body : DEN }
TCONSTR == union{ ENUM, RECORD, EXTEND, IMAGE, UNION }
ENUM    == record{ consts : List[ID] }
RECORD  == record{ lname : ID; fields : List[FIELD] }
FIELD   == record{ fname : ID; ftype : DEN }
EXTEND  == record{ type : DEN }
IMAGE   == record{}
UNION   == record{ tlist : List[ID] }
```

Type modifications allow the set of operations that a data type provides to be altered:

```
MODIFY == record{ type : DEN; modifier : TMOD }
TMOD   == record{ lname : ID; newops : DECL }
```

(We do not include signature modification in this abstract syntax. Signature modifications do not change the meanings of the operations of a type; they only make certain programs syntactically illegal. Thus, we regard signature modifications as purely syntactic matters.)

Finally, a Russell program is simply a function to be evaluated:

```
PROG == record{ name : ID; func : FUNC }
```

Although the domain ID of identifiers will be left unspecified, we will assume the existence of two functions

```
ToID : func[ val String ] val ID
```

and

ToString : func[val ID] val String

that convert in the obvious fashion between Strings (as defined in [Demers80c]) and identifiers.

4. The Meaning of Russell Programs

4.1. Basic Domains

The semantics of Russell is given as a set of functions that interpret denotations in the abstract syntax as elements of a set of "abstract value spaces" or "domains." Before giving the details of the interpretation function for programs, we discuss the signatures and semantic properties of the type parameters of the semantics.

4.1.1. Values

One of the basic premisses of the design of Russell was that the value space that was manipulated by Russell programs is typeless, e.g., one cannot ask whether a value is an integer or a function. Moreover, data types in Russell are themselves values in this amorphous value space.

Because values can be used in many different ways and the results of all denotations in the language are values, the "value space" data type VAL must have projection and injection functions from itself to functions, data types, etc. We therefore will require VAL to have signature

```
type V{ ToBoolean   : func[ val V ] val Boolean;
        FromBoolean : func[ val Boolean ] val V;
        ToInteger   : func[ val V ] val Integer;
        FromInteger : func[ val Integer ] val V;
        ToVoid      : func[ val V ] val Void;
        FromVoid    : func[ val Void ] val V;
        ToFVAL      : func[ val V ] val FVAL;
        FromFVAL    : func[ val FVAL ] val V;
        ToTVAL      : func[ val V ] val TVAL;
        FromTVAL    : func[ val TVAL ] val V  }
```

where

1. Boolean, Integer and Void are primitive domains (they are left unspecified),

2. FVAL and TVAL are the domains of function values and type values, respectively. The specifications of each of these types is given below.

   The only constraints on the operations of VAL is that

   ToBoolean[FromBoolean[x]] = x

   ToInteger[FromInteger[x]] = x

and so forth for each pair of projections and injections. It is important to notice that while VAL looks like it could be formed from a disjoint union, this is not necessary, because there is no operation available to _inspect_ a value, asking about its type.

4.1.2. _Environments and Stores_

Given the value space VAL, we can define the domains of environments and stores. Environments are used to provide the values of free identifiers appearing in denotations, stores to give the current values of program variables. A state is simply a pair consisting of an environment and a store; the state provides all of the values necessary to evaluate a denotation.

4.1.2.1. Environments

The data type ENV provides the following set of operations to manipulate the association of identifiers and values.

```
type Env{
    Extend   : func[ val Env; val List[ID]; val List[VAL] ] val Env;

    RecExtend: func[ EvDen : func[ val DEN; val Env×STORE ] val VAL×STORE ]
               func[ val Env; val List[ID]; val List[DEN] ] val Env;

    Apply    : func[ Env : val Env; Name : val ID ] val VAL  }
```

The axioms that these operations must obey are:

```
Apply[ Extend[ e, MkL[id], MkL[v] ], id ] = v

id ≠ id' ==> Apply[ Extend[ e, MkL[id'], MkL[v] ], id ] = Apply[ e, id ]

let
  RecEnv == (RecExtend[ EvDen ])[ e, MkL[id], MkL[den] ]
in Apply[ RecEnv, id ] = ValOf[ EvDen[ den, RecEnv, s ] ] ni

id ≠ id' ==>
  Apply[ (RecExtend[EvDen])[ e, MkL[id'], MkL[den] ], id ] = Apply[ e, id ]
```

Extend and Apply work together as one would expect. The most difficult of the operations to understand is clearly RecExtend, which builds an environment in which each new identifier is bound to the value of its corresponding denotation in the *extended* environment. The definition of RecExtend is complicated somewhat because evaluation of a denotation in Russell produces both a value and a new store. So we need a store in which to evaluate denotations being bound by RecExtend and we need to produce only the value produced when taking the value of the identifier. The import rule of Russell, which prohibits variables in operation denotations, and the fact that only operations may be recursively declared guarantees that

1. the store used to evaluate recursively defined denotations is irrelevant and that

2. these evaluations may have no visible effect on the store. Thus, in the axiom for RecExtend, the store used to evaluate a denotation to be bound is unspecified.

4.1.2.2. Stores

The other major component of the state is the store, which maps variables into their current values. In Russell, the semantics of the combining forms (which is what we are to present below) do not depend at all on the details of the store; it is only the meaning of the data type constructions of the language that specify the "interpretation" of variables. Thus, in the meaning function for programs the data type STORE can have the simplest of signatures:

type { }

4.1.3. Function and Type Values

For each of the data types FVAL and TVAL the operations provided give the meaning of application and of the evaluation of the Russell constructions of the type.

The signature of FVAL, the space of function meanings, is

```
type fv{
    EvalFunc: func[ EvDen : func[val DEN; val ENV; val STORE] val VAL×STORE ]
                func[ Func : val FUNC; Env : val ENV ] val fv ;

    Apply   : func[ Fval : val fv;
                    Args : val List[VAL]; Store : val STORE ] val VAL×STORE ]
```

A Russell function takes a list of values (the arguments) and a store and produces both a value and a store. Functions in Russell can have effects, but only on variables passed as arguments. The only axiom for these operations is the obvious

```
Apply[ (EvalFunc[f])[ StrToFunc[ "func[ id ] { den }" ], e ], v, s ] =
    f[ den, Mk[ Extend[ e, id, v ], s ]
```

(where StrToFunc is a function that translates a string to its corresponding function denotation in the obvious manner). That is, the application of a function produces the result of evaluating the function body in the environment of the function value extended by binding the argument values to the parameter identifiers.

The data type TVAL provides operations much like an environment, except that it also provides operations that evaluate the various type constructions of the defined language. It has signature

```
type tv{
    Extend
      : func[ val tv; val List[ID]; val List[VAL] ] val tv;

    Apply
      : func[ Tval : val tv; Name : val ID ] val VAL ;

    EvalImage
      : func[] val tv ;

    EvalEnum
      : func[ Enum : val ENUM ] val tv;

    EvalUnion
      : func[ Union : val UNION; Env : val ENV ] val tv ;

    EvalRecord
      : func[ EvDen : func[val DEN; val ENV; val STORE] val VAL×STORE ]
          func[ Record : val RECORD; Env : val ENV ] val tv ;

    EvalExtension
      : func[ EvDen : func[val DEN; val ENV; val STORE] val VAL×STORE ]
          func[ Exten : val EXTEND; Env : val ENV ] val tv ;

    EvalModifier
      : func[ EvDen : func[val DEN; val ENV; val STORE] val VAL×STORE ]
          func[ Type : val tv; Mod : val TMOD; Env : val ENV ] val tv    }
```

The axioms for these operations are extremely cumbersome to present (except for the obvious axioms for Extend and Apply, which behave like Extend and Apply for environments). This is because much of the meaning of a Russell program is tied up in the sets of operations that the type constructions provide; thus, the specifications of the results of EvalRecord, EvalUnion, etc. include the behavior of many operations. To give an example of the sort of specifications one would have to write to axiomatize the operations of TVAL completely, we present a simple example from the meaning of _images_.

The In and Out functions provided by an image data type must satisfy the axiom

Out[In[x]] = x

(taking the operation value of the image of an operation yields the same operation). To say this using the operations provided by FVAL and TVAL, we need

```
let
  In == ToFVAL[ Apply[ EvalImage[], MkID["In"] ] ] ;
  Out == ToFVAL[ Apply[ EvalImage[], MkID["Out"] ] ]
in
  let
    InResult == Apply[ In, x, s ]
  in
    Apply[ Out, ValOf[InResult], StoreOf[InResult] ] = MkResult[ x, s ]
  ni
ni
```

where MkResult simply forms a value, store pair and ValOf and StoreOf produce the Val and the Store components respectively of such a pair. (In other words, applying the composition of In and Out yields the same value and leaves the store unchanged.) The constraints on all of the other operations provided by the other constructions in Russell can be similarly expressed, but only with a substantial amount of notation. Thus, we leave the behavior of the meanings of the type constructions unspecified.

4.2. The Meaning of Russell

The meaning of Russell programs is given below as a polymorphic Russell function.

```
EvalProg ==

func[ VAL   : type V{ (* signature given above *) } ;

      ENV   : type E{ (* signature given above *) } ;

      STORE : type { } ;

      FVAL  : type fv{ (* signature given above *) } ;

      TVAL  : type tv{ (* signature given above *) }   ]

   (* the semantics takes the types VAL, ENV, STORE, FVAL and TVAL as
      arguments and produces a function that gives the meaning of the
      evaluation of a Russell program  *)

{ let
    (* some commonly used functions to compose VAL, STORE pairs include *)

    MkResult == (VAL × STORE)$Mk;

    ValOf == (VAL × STORE)$first;

    StoreOf == (VAL × STORE)$scnd;

    MkResult == (List[VAL] × STORE)$Mk;

    ValListOf == (List[VAL] × STORE)$first;

    StoreOf == (List[VAL] × STORE)$scnd;

   (* We first give the meaning of each of the primitive combining forms in
      Russell, e.g, application, conditional, selection.
      In each case, the meaning of the combining form is to yield a value
      and a new store based on the current state *)
```

```
EvalAppl ==
  func[ Appl : val APPL; Env : val ENV; Store : val STORE ]
    { let
        OprResult == EvalDen[ opr[Appl], Env, Store ];
      in
        let
          OpndResult == EvalDenList[opnd[Appl], Env, StoreOf[OprResult]];
        in
          let
            Args  == ValListOf[ OpndResult ];
            Store == StoreOf[ OpndResult ]
          in
            FVAL$Apply[ ToFVAL[ ValOf[ OperResult ] ], Args, Store ]
          ni
        ni
      ni };
(* The value of the operator is treated as a function and applied to the
   values of the operands and the store produced by evaluating both
   operator and operands  *)

EvalCond ==
  func[ Cond : val COND; Env : val ENV; Store : val STORE ]
    { let
        Glist == COND$arms[ Cond ]
      in
        if
           ~ empty?[Glist] ==>
            let
              TestResult == EvalDen[ guard[ first[Glist] ], Env. Store ]
            in
              if
                VAL$ToBoolean[ ValOf[TestResult] ] ==>
                  EvalSeq[ rhs[ first[Glist] ], Env, StoreOf[TestResult] ]

              [] else ==>
                  EvalCond[COND$Mk[rest[Glist]], Env. StoreOf[TestResult]]
              fi
            ni
        fi
      ni } ;
(* Note: the particular order of evaluation of the guards used above is
   arbitrary. If all of the guards are false, the result is unspecified.*)
```

```
EvalBlock ==
  func[ Block : val BLOCK; Env : val ENV; Store : val STORE ]
    { let
        Decls == ldecl[ Block ];
        Body == body[ Block ]
      in
        let
          RecEnv == (RecExtend[EvalDen])[ Env, lhs[Decls], rhs[Decls] ]
        in
          let
            ResultList == EvalDenList[ rhs[Decls], RecEnv, Store ]
          in
            EvalSeq[ Body,
                     Extend[ Env, lhs[Decls], ValListOf[ResultList] ],
                     StoreOf[ ResultList ] ]
          ni
        ni
      ni } ;
(* Blocks produce the result of the body in the environment extended by
   binding the rhs value of each declaration to the corresponding
   lhs identifier.  The use of RecExtend provides an environment in
   which recursively defined operation can be evaluated. *)

EvalRepeat ==
  func[ Repeat : val REPEAT; Env : val ENV; Store : val STORE ]
    { let
        Iterate == func[ S : val STORE ] { EvalRepeat[ Repeat, Env, S ] };
        EvalBody ==
          func[ Glist : val List[GDEN]; S : val STORE ]
            { if
                empty?[Glist] ==> MkResult[ FromVoid[ Void$Null[] ], S ]
              [] else ==>
                let
                  TestResult == EvalDen[ guard[ first[Glist] ], Env, S ];
                  Gden == rhs[ first[ Glist ] ]
                in
                  if
                    ToBoolean[ ValOf[TestResult] ] ==>
                      Iterate
                       [ StoreOf
                          [ EvalSeq[ Gden, Env, StoreOf[TestResult] ] ] ]
                  [] else ==>
                      EvalBody[ rest[Glist], StoreOf[TestResult] ]
                  fi
                ni
              fi }
      in
        EvalBody[ REPEAT$arms[ Repeat ], Store ]
      ni };
(* Repetitions always produce Void$Null as result (they have signature
   val Void) and are executed only for their effect on the store.
   Again, the order of evaluation of the guards is arbitrary.  *)
```

```
EvalSeq ==
  func[ Seq : val SEQUENCE ; Env : val ENV; Store : val STORE ]
    { let
        Result == EvalDen[ first[ Seq ], Env, Store ]
      in
        if
          empty?[ rest[Seq] ] ==> Result
        [] else ==>
           EvalSeq[ SEQUENCE$Mk[ rest[Seq] ], Env, StoreOf[Result] ]
        fi
      ni } ;
(*A sequence produces the value of its last denotation as result.*)

EvalSelect ==
  func[ Sel : val SELECT; Env : val ENV; Store : val STORE ]
    { let
        TypeResult == EvalDen[ type[ Sel ], Env, Store ]
      in
        let
          Val == TVAL$Apply[ ToTVAL[ ValOf[TypeResult] ], component[Sel] ]
        in
          MkResult[ Val, StoreOf[ TypeResult ] ]
        ni
      ni } ;
(* A selection applies an identifier to a type value. *)
```

```
EvalDen ==
  func[ Den : val DEN; Env : val ENV; Store : val STORE ]
    { if
        IsID?[Den] ==> MkResult[ Apply[ Env, ToID[Den] ], Store ]

      [] IsCOMBIN[Den] ==>
          let
            Comb == ToCOMBIN[ Den ]
          in
            if
              IsAPPL?[Comb] ==> EvalAppl[ ToAPPL[Comb], Env, Store ]

            [] IsSELECT?[Comb] ==> EvalSelect[ ToSELECT[Comb], Env, Store ]

            [] IsCOND?[Comb] ==> EvalCond[ ToCOND[Comb], Env, Store ]

            [] IsSEQUENCE?[Comb] ==> EvalSeq[ ToSEQUENCE[Comb], Env, Store ]

            [] IsREPEAT?[Comb] ==> EvalRepeat[ ToREPEAT[Comb], Env, Store ]

            [] IsBLOCK?[Comb] ==> EvalBlock[ ToBLOCK[Comb], Env, Store ]
            fi
          ni

      [] IsMODIFY[Den] ==> EvalModify[ ToMODIFY[Den], Env, Store ],

      else ==> MkResult[ EvalConstr[ ToCONSTR[Den], Env ], Store ]

      fi } ;
(* The meaning of a denotation is simply the meaning of the appropriate
   combining form, type modification or construction.
   Note that the meanings of constructions (given below) does not depend
   on the store. *)
```

```
EvalModify ==
  func[ Mod : val MODIFY; Env : val ENV; Store : val STORE ]
    { let
        TResult == EvalDen[ type[Mod], Env, Store ];
        NewOps == newops[ modifier[ Mod ] ];
        Lname  == lname[ modifier[ Mod ] ]
      in
        let
          RecType == (EvalModifier[EvalDen])
                       [ ToTVAL[ ValOf[TResult] ], modifier[Mod], Env ]
        in
          let
            OpList == EvalDenList
                       [ rhs[NewOps],
                         Extend[ Env, MkL[Lname], MkL[FromTVAL[RecType]],
                         StoreOf[ TResult ] ]
          in
            MkResult
              [ FromTVAL[ Extend[ ToTVAL[ ValOf[ TResult ] ],
                                  lhs[NewOps], ValListOf[OpResults] ] ],
                StoreOf[ OpResults ] ]
          ni
        ni
      ni } ;
(* Note that in evaluating the delarations in the modifier of a type
   modification, the local name of the modifier is bound to the
   modified type. *)
```

```
EvalConstr ==
  func[ Cons : val CONSTR; Env : val ENV ]
    { if
        IsFUNC?[Cons] ==>
          FromFVAL[ (EvalFunc[EvalDen]) [ ToFUNC[Cons], Env ] ]

      [] else ==>
          let
            Type == ToTCONSTR[ Cons ]
          in
            if
              IsRECORD?[Type] ==>
                FromTVAL[ (EvalRecord[EvalDen])[ ToRECORD[Type], Env ] ]

            [] IsIMAGE?[Type] ==> FromTVAL[ EvalImage[] ]

            [] IsEXTEND?[Type] ==>
                FromTVAL[ (EvalExtension[EvalDen])[ToEXTEND[Type], Env] ]

            [] IsUNION?[Type] ==>
                FromTVAL[ (EvalUnion[ EvalDen ] )[ ToUNION[Type], Env ] ]

            [] IsENUM?[Type] ==> FromTVAL[ EvalEnum[ ToENUM[Type] ] ]
            fi
          ni
      fi } ;
(* The meanings of constructions are part of the data types FVAL and TVAL,
   so to evaluate a construction we simply apply the appropriate operation
   from these data types. *)


EvalDenList ==
  func[ DenList : val List[DEN]; Env : val ENV; Store : val STORE ]
    { if
        empty?[DenList] ==> MkResult[ List[VAL]$Nil[], Store ]

      [] else ==>
          let
            First == EvalDen[ first[ DenList ], Env, Store ]
          in
            let
              Rest == EvalDenList[ rest[DenList], Env, StoreOf[First] ]
            in
              MkResult[ ValOf[First] || ValListOf[Rest], StoreOf[ Rest ] ]
            ni
          ni
      fi };
(* Produce a list of values and a new store. *)
```

```
(* given the previous definitions, we can now give the meaning of Russell
   programs *)
in
  func[ Prog : val PROG; Env : val Env ]
    { let
        InitEnv == (RecExtend[ EvalDen ])[ name[Prog], func[Prog], Env ]
      in
        let
          Fval == (EvalFunc[ EvalDen ]) [ func[Prog], InitEnv ]
        in
          func[ Args : val List[VAL]; Store : val STORE ]
            { FVAL$Apply[ Fval, Args, Store ] }
        ni
      ni }
ni }  (* end of the Russell semantics *)
```

5. Type Arguments for the Russell EvalProg Function

The EvalProg function given above was polymorphic with respect to the basic value spaces of the semantics, so a complete semantics for Russell must define a suitable set of type arguments to which we may apply the meaning function for programs. It is in this part of the semantics that we give the meanings of the various constructions of Russell and also where the store must be specified in more detail -- it is the meanings of the various type constructions in Russell that define the interpretation of the store.

Below, STORE will be a type with the following signature:

```
type S{ InitStore : func[] val S;
        Update    : func[ val S; loc, v : val VAL ] val S;
        Apply     : func[ val S; val VAL ] val VAL;
        Alloc     : func[ val S ] val VAL×S }
```

Additionally, STORE will have an "hidden" function Allocated? with signature

```
func[ val STORE; val val ] val Boolean
```

which will give the "allocation status" of locations in the store; Allocated? will not be part of the signature of the store that is known in the remainder of the semantics, but is used only to give the necessary axioms for the store. These axioms are:

```
Apply[ Update[ s, loc, val ], loc ] = val

loc ≠ loc' ==> Apply[ Update[ s, loc, val ], loc' ] = Apply[ s, loc' ]

Apply[ StoreOf[ Alloc[ s ] ], loc ] = Apply[ s, loc ]

Allocated?[ InitStore[], loc ] = False[]

Allocated?[ StoreOf[ Alloc[s] ], ValOf[ Alloc[s] ] ] = True[]

Allocated?[ s, ValOf[ Alloc[s] ] ] = False[]
```

The choice of the bindings of values to locations in the initial store is arbitrary; the only property that the initial store will have is that all locations in it are "unallocated". The axioms for allocation state that Alloc always returns a previously unallocated location and makes the status of the location it returns "allocated."

One complete semantics for Russell is given by the following Russell denotation:

```
let

  ValOf    == VAL×STORE$first;

  StoreOf == VAL×STORE$scnd;

  MkResult == VAL×STORE$Mk;

 (* We begin the definition of the domains of the semantics with the
    the definition of VAL, the basic domain of values. Note that all of the
    inspection operations are hidden, as is consistent with the basic ideas
    the Russell semantics *)
```

```
VAL ==
  let
    ValList == extend{ List[VAL] } ;

    ValPair == extend{ VAL × VAL } ;

    ValType ==
       union{ Integer, Boolean, Void, LOC, FVAL, TVAL, ValList, ValPair }
         hide{ IsInteger?, IsBoolean?, IsVoid?, IsLOC?, IsFVAL?, IsTVAL?,
               IsValList?, IsValPair? }
  in
    ValType
      with Val{
        ToPAIR == func[ x : val Val ] val Val×Val
                    { ValPair$Out[ Val$ToValPair[x] ] };

        FromPAIR == func[ x : val Val×Val ] val Val
                     { Val$FromValPair[ ValPair$In[x] ] } ;

        ToLIST == func[ x : val Val ] val List[Val]
                    { ValList$Out[ Val$ValToValList[x] ] } ;

        FromLIST == func[ x : val List[Val] ] val Val
                     { Val$FromValList[ ValList$In[x] ] } }

      hide { ToValList, FromValList }
  ni ;
```

(* This definition of VAL produces a type that can be seen as the solution of the type equation
VAL = Integer + Boolean + Void + LOC + FVAL + TVAL + VAL×VAL + List[VAL].
The use of extend in the definition of VAL allows us to break the recursion in the evaluation of the type expression and provides the In and Out operations necessary to hide the fact that the expression uses local definitions of ValList and ValPair -- note that they do not appear in the signature of the result. LOC is the domain of locations, given below .*)

(* One very useful function to have is the function that diverges when asked to produce a VAL *)

```
Diverge == func[] val VAL { Diverge[] };
```

```
(* Next we define the meaning of environments. *)

 ENV ==
   let
     RecCmpnt == image{ func[ val STORE ] val VAL×STORE }
     EnvResult == union{ VAL, RecCmpnt }
   in
   (* an environment is a map from identifiers to VALs or "recursive
      components." Recursive components are "unevaluated expressions" that
      are evaluated only when necessary (when Apply needs to produce their
      value as a result). These recursive components are added to the
      environment by RecExtend to form an environment in which recursive
      declarations are to be evaluated. *)

     image{ func[ val ID ] val EnvResult }

     with env{
       Update == func[ Env : val env; Id : val ID; Val : val VAL ]
                   { env$In[ func[ x : val ID ]
                               { if x = Id ==> FromVAL[Val]
                                 □ else  ==> (env$Out[Env])[x]
                                 fi } ] } ;
     (* Update only allows VALs to be added to an environment -- the
        existence of RecCmpnt values is hidden outside ENV. *)

       Empty == func[] { env$In[ func[ val ID ] { FromVAL[ Diverge[] ]} ] };

       Extend == func[ Env: val env; Ids: val List[ID]; Vals: val List[VAL] ]
                   { if
                       empty?[Ids] ==> Env
                     □ else ==>
                         let
                           NewEnv == Update[ Env, first[Ids], first[Vals] ]
                         in
                           env$Extend[ NewEnv, rest[Ids], rest[Vals] ]
                         ni
                     fi };

       Apply == func[ Env : val env; Id : val ID ] val VAL
                  { let
                      Result == (env$Out[ Env ])[Id]
                    in
                      if
                        IsVAL?[Result] ==> ToVAL[ Result ]
                      □ else ==>
                          ValOf[ (Out[ ToRecCmpnt[Result] ]) [ InitStore[] ] ]
                      fi
                    ni } ;
     (* Note that Apply hides the existence of RecCmpnts -- it forces their
        evaluation in the initial store. One theorem to prove about Russell
        is that the import rule and the restriction that only operation
        denotations to be recursively defined is sufficient to allow the
        InitStore used here to be arbitrary. *)
```

```
      RecExtend ==
        func[ EvDen: func[val DEN; val env; val STORE] val VAL×STORE]
          { let
              ExtendEnv ==
                func[ Env  : val env; Ids : val List[ID];
                      Dens : val List[DEN]; RecEnv : val env ]
                  { if
                      empty?[Ids] ==> Env
                    □ else ==>
                      let
                        RC == func[ S : val STORE ]
                                { EvDen[ first[Dens], RecEnv, S ] };
                        NewEnv ==
                          env$In
                           [ func[ x : val ID ] val EnvResult
                               { if
                                   x = first[Ids] ==> FromRecCmpnt[In[RC]]
                                 □ else ==> FromVAL[ Apply[ Env, x ] ]
                                 fi }  ]
                      in
                        ExtendEnv[ NewEnv, rest[Ids], rest[Dens], RecEnv ]
                      ni
                    fi }
           in
             func[ Env: val Env; Ids: val List[ID]; Dens: val List[DEN] ]
               { let
                   RecEnv ==
                     func[ x : val ID ] val EnvResult
                       { let
                           RE == (RecExtend[EvDen])[ E, Ids, Dens ]
                         in
                           FromVAL[ Apply[ RE, x ] ]
                         ni }
                 in
                   ExtendEnv[ E, Ids, Dens, env$In[RecEnv] ]
                 ni }
           ni }
    (* RecExtend adds recursive components to the environment; the values
       of the denotations are not evaluated (they may be by Apply later).
       Note that the environment used in evaluating the denotations
       (when they are evaluated) will be the recursive extension of Env,
       so recursive definitions work properly. *)

       }   (* the end of the additions to ENV *)
  ni

  hide{ In, Out } (* No operations exported use RecCmpnt *)

(* the next major domain of the semantics is the store *)

 LOC == Integer;
```

```
STORE ==
  let
    (* a storage map is a function from locations (Integers) to VALs *)
    SMAP == image{ func[ val LOC ] val VAL }
  in

     (*a store is a storage map and the index of the "next free" location*)
    record{ Map : SMAP; Next : LOC }

     (* to which we add all of the necessary operationa *)
    with S{
      InitStore ==
        func[] { let InitMap == func[ val LOC ] { Diverge[] }
                 in Mk[ SMAP$In[ InitMap ], LOC$0 ] ni };

      Apply ==
        func[ Store : val S; Val : val VAL ]
          { let
              CurrMap == SMAP$Out[ Map[ Store ] ]
            in
              CurrMap[ ToLOC[ Val ] ]
            ni } ;

      Update ==
        func[ Store : val S; Loc, Val : val VAL ]
         { let
             LVal == ToLOC[ Loc ];
             CurrMap == SMAP$Out[ Map[ store ] ]
           in
             let
               NewMap == func[ x : val LOC ]
                          { if x = LVal ==> Val [] else ==> CurrMap[ x ] fi }
             in
               Mk[ SMAP$In[ NewMap ], Next[ store ] ]
             ni
           ni } ;

      Alloc ==
        func[ Store : val S ]
          { let
              Loc == Next[ Store ];
              CurrMap == SMAP$Out[ Map[ Store ] ]
            in
              MkResult[ FromInteger[Loc], Mk[ CurrMap, Loc LOC$+ 1 ] ]
            ni }

(* The Allocated? predicate that is needed to specify the axioms for the
   type is easily given.  *)
    Allocated? ==
      func[ Store : val S; loc : val VAL ] { ToLOC[loc] ≤ Next[S] } }
  ni

        (* Now, hide the details of the representation of stores *)
    hide{ Map, Next, Mk, Allocated? }
```

```
(* the space FVAL of function values is defined as the image of functions
   that take an argument list and a store and yield a VAL and a store. *)

 FVAL ==
   image{ func[ val List[VAL]; val STORE ] val VAL×STORE }

   with fv{
     Apply ==
       func[ Fval : val fv; Args : val List[VAL]; Store : val STORE ]
         { ( fv$Out[ Fval ] )[ Args, Store ] } ;

     EvalFunc ==
       func[ EvDen : func[val DEN; val ENV; val STORE] val VAL×STORE ]
         { func[ Func : val FUNC; Env : val ENV ]
            { fv$In
               [ func[ Args : val List[VAL]; Store : val STORE ]
                   { EvDen[ body[Func],
                            Extend[Env, parms[Func], Args], Store ] } ] } };

     PrimFunc ==
       func[ f : func[ val List[VAL] ] val VAL ]
         { fv$In[ func[ Args : val List[VAL]; S : val STORE ]
                   { MkResult[ f[ Args ], S ] } ] } }
   (* PrimFunc takes a List[VAL] → VAL function and turns it into an FVAL
      that does not depend on or affect the store. *)

(* the end of the definition of FVAL. Note that we are not hiding In and Out
   in this type -- this is because the meaning of the primitive data type
   constructions need to build function values as part of their meanings. *)

(* the domain TVAL is the most complicated part of the semantics.
   It includes the meaning of all of the primitive data types and data type
   constructions in the language.  Note that TVAL is the only part of the
   semantics that uses the operations provided by STORE -- thus, it is the
   data types of Russell that provide the interpretation of the store  *)

 TVAL ==
   let
     RecCmpnt == image{ func[ val STORE ] val VAL×STORE } ;
     TResult  == union{ VAL, RecCmpnt }
   in
     image{ func[ val ID ] val TResult }
   (* TVAL is basically the same as ENV.  The differences are the addition of
      operation MkType and a more complicated form of recursive extension. *)

     with tv{
       Update == func[ Type : val tv; Id : val ID; Val : val VAL ]
                   { tv$In[ func[ x : val ID ] val TResult
                              { if
                                   x = Id ==> FromVAL[ Val ]
                                 □ else ==> (tv$Out[Type])[x]
                                 fi } ] } ;

       Empty == func[] { tv$In[ func[ val ID ] { FromVAL[ Diverge[] ] } ] };
```

```
MkType ==
  func[ f : func[ val ID ] val VAL ]
    { tv$In[ func[ x : val ID ] val TResult{ FromVAL[ f[x] ] } ] } ;
(* Note that MkType is a less powerful operation that In; it does not
   allow the construction of TVALs with recursive components --
   these can only be added by RecExtend. *)

Extend == func[ Type: val tv; Ids: val List[ID]; Vals: val List[VAL] ]
            { if
                empty?[Ids] ==> Type
              □ else ==>
                 let
                   Newtype == tv$Update[Type, first[Ids], first[Vals]]
                 in
                   tv$Extend[ Newtype, rest[Ids], rest[Vals] ]
                 ni
              fi };

Apply == func[ Type : val tv; Id : val ID ] val VAL
           { let
               Result == (tv$Out[Type]) [Id]
             in
               if
                 IsVAL?[ Result ] ==> ToVAL[ Result ]
               □ else ==>
                   ValOf[ (Out[ ToRecCmpnt[Result] ])[InitStore[]] ]
               fi
             ni } ;
(* Note that Apply always produces a VAL as a result.  If the identifier
   is bound to a RecCmpnt, it is evaluated to give the result. *)
```

```
RecExtend ==
  func[ EvDen: func[val DEN; val ENV; val STORE] val VAL×STORE ]
    { let
        ExtendType ==
          func[ Type: val tv;
                Ids : val List[ID];
                Dens: val List[DEN]; Env: val ENV ]
            { if
                empty?[Ids] ==> Type
              □ else ==>
                  let
                    RCVal == func[ S : val STORE ]
                               { EvDen[ first[Dens], Env, S ] } ;
                    NewT ==
                      func[ x : val ID ] val TResult
                        { if
                            x = first[Id] ==>
                               FromRecCmpnt[ In[ RCVal ] ]
                          □ else ==>
                               FromVAL[ Apply[ Type, x ] ]
                          fi }
                  in
                    ExtendType[ In[NewT], rest[Ids], rest[Dens], Env ]
                  ni
              fi }
      in
        func[ Type: val tv; Lname: val ID;
              Ids: val List[ID]; Dens: val List[DEN]; Env: val ENV ]
          { let
              RecType  ==
                TVAL$MkType
                  [ func[ id : val ID ]
                      { tv$Apply
                          [ (RecExtend[ EvDen ])
                              [Type, Lname, Ids, Dens, Env], id ] } ]
            in
              ExtendType[ Type, Id, Den,
                          Update[ Env, Lname, FromTVAL[ RecType ] ] ]
            ni }
      ni }  }
 (* Recursive extension of a type results in a new type in which all of
    the denotations to be added are evaluated in an environment in which
    the recursive extension of the type is bound to the local name that
    was supplied. *)

ni (* This is the end of the basic part of TVAL *)
```

```
with tv{
  EvalModifier ==
    func[ EvDen : func[val DEN; val ENV; val STORE] val VAL×STORE ]
      { func[ Type : val tv; TMod : val TMOD; Env : val ENV ]
          { (tv$RecExtend[ EvDen ])
               [ Type, lname[ TMod ], lhs[ newops[ TMod ] ],
                  rhs[ newops[ TMod ] ], Env ] } } ;
(* Evaluation of a type modifier simply produces a recursive extension of
   the type. *)

  EvalExtension ==
    func[ EvDen: func[val DEN; val ENV; val STORE] val VAL×STORE ]
      { let
          Ident == PrimFunc[ func[ x : val List[VAL] ] { first[x] } ]
        in
          func[ Ext : val EXTEND; Env : val ENV ]
            { let
                Etype ==
                  func[ x : val ID ]
                    { let
                        ExtResult == EvDen[ type[Ext], Env, InitStore[] ]
                      in
                        TVAL$Apply[ ToTVAL[ ValOf[ExtResult] ], x ]
                      ni }
              in
                Extend[ tv$MkType[ Etype ],
                        MkID["In"] || MkL[ MkID["Out"] ],
                        FromFVAL[Ident] || MkL[ FromFVAL[Ident] ] ]
        ni } }
(* extend{ Type } produces a new type with the same operations as
   the type supplied plus new In and Out operations to transfer between
   the two types -- In and Out each just produce the value of their
   argument.  Note that extend does not evaluate the type; it is
   evaluated only when applied. *)
```

```
(* To give the meaning of the remaining type contructions, we first define a
   "primitive data type" that provides the meanings of the basic operations
   that interpret the store.  Some of the meanings of the type constructions
   that follow will be extensions of this simple type.  *)
    PrimType ==
      let
        Oplist == MkID["New"] || MkID[":="] ||
                  MkID["ValueOf"] || MkL[ MkID["alias"] ];

        Vlist == FromFVAL[ In[ New ] ] ||  FromFVAL[ In[ Assign ] ]
                 || FromFVAL[ In[ Fetch ] ] || MkL[ FromFVAL[ Equal ] ]

        New == func[ val List[VAL]; Store : val STORE ] { Alloc[ Store ] };

        Assign == func[ Args : val List[VAL]; Store : val STORE ]
                    { let
                        Loc == first[ Args ] ;
                        Val == first[ rest[ Args ] ]
                      in
                        MkResult[ Val, Update[ Store, Loc, Val ] ]
                      ni };

        Fetch == func[ Args : val List[VAL]; Store : val STORE ]
                   { MkResult[ Apply[ Store, first[Args] ], Store ] } ;

        Equal == func[ Args : val List[VAL]; Store : val STORE ]
                   { FromBoolean[ first[Args] Loc$= first[rest[Args]] ] }
      in
        func[] { tv$Extend[ tv$Empty[], Oplist, Vlist ] }
      ni ;

    EvalImage ==
      let
        Ident == PrimFunc[ func[ x : val List[VAL] ] { first[x] } ]
      in
        func[]
          { Extend[ PrimType[],
                    (MkID["In"] || MkL[ MkID["Out"] ]),
                    (FromFVAL[ Ident ] || MkL[ FromFVAL[ Ident ] ]) ] }
      ni ;
   (* Images provide the basic store operations, plus In and Out to go
      from operation values to images and vice-versa.
      As with extend, these operations are simply identity functions.*)

   (* The remainder of the definition of TVAL comprises the definitions of
      EvalUnion, EvalRecord and EvalEnum, which specify the final type
      constructions of Russell.  These are the most intricate pieces of the
      semantics, primarily because each of these type constructions produces
      a type with a fairly large number of operations. *)
```

```
EvalUnion ==
  (* Union values will be represented in this semantics by elements of
     VAL×VAL, where the first component is a "tag" and the second
     component the "value". This representation is implicit, in that it
     is simply uniformly used throughout the operations provided by
     a union type. *)
  let
    To == func[ x : val List[VAL] ] { scnd[ ToPAIR[ first[x] ] ] } ;
  (* To "projects," producing the second component of its argument. *)

    From == func[ Tag : val Integer ]
              { func[ x : val List[VAL] ]
                  { FromPAIR[ Mk[ FromInteger[ Tag ], first[x] ] ] } } ;
  (* From "injects," forming a pair with the given tag. *)

    Is == func[ Tag : val Integer ]
            { func[ x : val List[VAL] ]
                let
                  TagVal == ToInteger[ first[ ToPAIR[ first[x] ] ] ]
                in
                  FromBoolean[ Tag = TagVal ]
                ni } } ;
  (* Is "inspects," testing the first component of its first argument.*)

    Equal ==
      func[ Tag   : val Integer; Env : val ENV;
            TName : val ID; Else : func[ val List[VAL] ] val VAL ]
        { func[ Args : val List[VAL] ]
            { let
                x == first[ Args ];
                y == first[ rest[ Args ] ];
                TEqual == Apply[ ToTVAL[Apply[Env, TName]], MkID["="] ]
              in
                let
                  TagsEqual? == ToBoolean[ ( Is[Tag] )[ MkL[x] ] ] &
                                ToBoolean[ ( Is[Tag] )[ MkL[y] ] ]
                in
                  if
                    TagsEqual? ==>
                      Apply[ ToFVAL[TEqual], x || MkL[y], InitStore[] ]
                  □ else ==> Else[ x || MkL[y] ]
                  fi
                ni
              ni } } ;
  (* Equal produces a function that tests for TName$= if the tags of the
     first and second arguments are right and otherwise calls Else.
     Note that the type TName is not evaluated until the resulting
     function is actually applied; thus, type declarations of the form
             T == union{ Integer, T }
     are perfectly legal in Russell. *)
```

```
      ExtendUnion ==
        func[ UType : val tw; Ids : val List[ID]; Env : val ENV;
              Tag   : val Integer; Eq : func[ val List[VAL] ] val VAL ]
          { if
              empty?[Ids] ==>
                Update[ UType, MkID["="], FromFVAL[ PrimFunc[ Eq ] ] ]
            [] else ==>
                let
                  OpList == FromFVAL[ PrimFunc[ To ] ] ||
                            FromFVAL[ PrimFunc[ From[Tag] ] ] ||
                            MkL[ FromFVAL[ PrimFunc[ Is[ Tag ] ] ] ] ;

                  NameList == MkID[ "To" cat ToStr[first[Ids]] ] ||
                          MkID[ "From" cat ToStr[first[Ids]] ] ||
                          MkL[MkID[ "Is" cat ToStr[first[Ids]] cat "?" ]]
                in
                  ExtendUnion[ Extend[ UType, NameList, OpList ],
                               rest[Ids], Env, Tag + 1,
                               Equal[ Tag, Env, first[Ids], Eq ] ]
                ni
            fi }
      (* Union types are builtup by adding new "To", "From" and "Is"
         operations for each type in the list (using different tag values)
         and composing the necessary "=" operation by adding a test for
         each type in the list. *)
    in
      func[ Union : val UNION; Env : val ENV ]
        { let
            NotEqual ==
              func[ val List[VAL] ] val VAL { FromBoolean[ False[] ] }
          in
            ExtendUnion[ PrimType[], tlist[Union], Env, 0, NotEqual ]
          ni }
    ni } ;

  EvalRecord ==
  (* Records are represented in this semantics by lists of values, i.e.,
     by values in List[VAL] (again, this representation is only implicit).
     The meaning of records is complicated by the fact that record
     variables must have components (be lists of locations) so that
     component selectors can also be applied to them.  For this reason,
     the meanings of "New", ":=", "ValueOf" and "alias" must be built up
     by building lists of the results of component type operations.  *)
    let
      Last == func[ x : val List[VAL] ]
                { if
                    empty?[rest[x]] ==> first[x]
                  [] else ==> Last[rest[x]]
                  fi } ;
     (* The meanings of the functions New, := and ValueOf will be
        constructed by adding new results to the end of the list
        representing a record value.  For this, we first define a set of
        functions that work on the end of a list.*)
```

```
ExtendEqual ==
  func[ Type : val DEN; Env : val ENV; Rest : val FVAL ]
    { let
        EqualT ==
          func[ x,y : val VAL; S : val STORE ]
            { let
                TVal == ValOf[ EvDen[Type, Env, InitStore[] ]]
              in
                let
                  FVAL$Apply
                   [ ToFVAL[Apply[ ToTVAL[TVal], MkID["="]]],
                     x || MkL[y], S]
                ni
              ni } } ;

        NewEqual ==
          func[ x, y : val VAL; S : val STORE ]
            { let
                Lx == Last[ ToLIST[x] ] ;
                Ly == Last[ ToLIST[y] ] ;
                Fx == FromLIST[ Front[ ToLIST[x] ] ] ;
                Fy == FromLIST[ Front[ ToLIST[y] ] ]
              in
                let
                  Fval == Apply[Rest, Fx || MkL[Fy], S]
                in
                  let
                    Lval ==
                      EqualT[ Lx || MkL[Ly], StoreOf[Fval] ]
                  in
                    MkResult
                     [ FromBoolean[ValOf[Fval] & ValOf[Lval]],
                       StoreOf[ Lval ] ]
                  ni
                ni
              ni }
      in
        FVAL$In
         [ func[ Args : val List[VAL]; S : val STORE ]
             { NewEqual[first[Args], first[rest[Args]], S] } ]
      ni } ;
```

```
ExtendAlias ==
  func[ Type : val DEN; Env : val ENV; Rest : val FVAL ]
    { let
        AliasT ==
          func[ x,y : val VAL; S : val STORE ]
            { let
                TVal == ValOf[ EvDen[Type, Env, InitStore[] ]]
              in
                let
                  FVAL$Apply
                   [ ToFVAL
                       [Apply[ ToTVAL[TVal], MkID["alias"] ]],
                     x || MkL[y], S ]
                ni
              ni } } ;

        NewAlias ==
          func[ x, y : val VAL; S : val STORE ]
            { let
                Lx == Last[ ToLIST[x] ] ;
                Ly == Last[ ToLIST[y] ] ;
                Fx == FromLIST[ Front[ ToLIST[x] ] ] ;
                Fy == FromLIST[ Front[ ToLIST[y] ] ]
              in
                let
                  Fval == Apply[Rest, Fx || MkL[Fy], S]
                in
                  let
                    Lval ==
                      AliasT[ Lx || MkL[Ly], StoreOf[Fval] ]
                  in
                    MkResult
                     [ FromBoolean[ValOf[Fval] & ValOf[Lval]],
                       StoreOf[ Lval ] ]
                  ni
                ni
              ni }
      in
        FVAL$In
         [ func[ Args : val List[VAL]; S : val STORE ]
             { NewEqual[first[Args], first[rest[Args]], S] } ]
      ni } ;
```

```
ExtendRecord ==
  func[ Type : val tv; Pos : val Integer;
        Flist: val List[FIELD]; Env : val ENV;
        Alloc, Assign, Fetch, Equal, Alias : val FVAL ]
    { if
        empty?[Flist] ==>
          Extend[ Type,
                  MkID["New"] || MkID[":="] || MkID["="]
                   || MkID["ValueOf"] || MkL[MkID["alias"]] ],
                  FromFVAL[Alloc] || FromFVAL[Assign]
                   || FromFVAL[Equal] || FromFVAL[Fetch]
                    || MkL[ FromFVAL[Alias] ] ]
      [] else ==>
          let
            NewAlloc ==
              ExtendAlloc[ ftype[first[Flist]], Env, Alloc ];
            NewAssign ==
              ExtendAssign[ftype[first[Flist]], Env, Assign ];
            NewFetch ==
              ExtendFetch[ ftype[first[Flist]], Env, Fetch ];
            NewEqual ==
              ExtendEqual[ ftype[first[Flist]], Env, Equal ];
            NewAlias ==
              ExtendAlias[ ftype[first[Flist]], Env, Alias ];
            NewType ==
              Update[ Type, fname[ first[Flist] ],
                      FromFVAL[ PrimFunc[ Select[Pos] ] ] ]
          in
            ExtendRecord[ NewType, Pos + 1, rest[Flist], Env,
                          NewAlloc, NewAssign, NewFetch,
                          NewEqual, NewAlias ]
          ni
      fi }
(* To add to a record type, we build a selector for the field
   that is to be added and extend the basic operations for the
   type to include operating on the new field. *)
```

```
        in
          func[ Record : val RECORD; Env : val ENV ]
            { let
                RecType ==
                  func[ x : val ID ]
                    { Apply[ (EvalRecord[EvDen])[ Record, Env ], x ] };

                NewEnv ==
                  Update[ Env, lname[ Record ],
                          FromTVAL[ TVAL$MkType[ RecType ] ] ] ;

                AllocNil, AssignNil, FetchNil ==
                  PrimFunc[ func[ val List[VAL] ] { FromLIST[Nil[]] } ];

                EqualNil, AliasNil ==
                  PrimFunc[ func[ val List[VAL] ]
                                { FromBoolean[ True[] ] } ];

                Make ==
                  PrimFunc[ func[ x : val List[VAL] ] { FromLIST[x] } ]
              in
                ExtendRecord
                  [ Update[ Empty[], MkID["Mk"], FromFVAL[Make] ],
                    0, fields[ Record ], NewEnv,
                    AllocNil, AssignNil, FetchNil, EqualNil, AliasNil ]
              ni }
        ni }
(* Finally, the meaning of records is given by beginning with basic
   operations that do nothing on the empty list and then extending these
   operations for each of the fields in the record.  Note that the
   environment used to evaluate each of the field types has the meaning
   of the record bound to the local name of the record construction. *)
  ni ;

EvalEnum ==
  let
    First ==
      PrimFunc[ func[ val List[VAL] ]{ FromInteger[ 0 ] } ] ;

    Ord, OrdInv ==
      PrimFunc[ func[ Args : val List[VAL] ]{ first[Args] } ] ;

    Pred ==
      PrimFunc[ func[ Args : val List[VAL] ]
                  { FromInteger[ ToInteger[first[Args]] - 1 ] } ];

    Succ ==
      PrimFunc[ func[ Args : val List[VAL] ]
                  { FromInteger[ ToInteger[first[Args]] + 1 ] } ];
```

```
        Equal ==
          PrimFunc[ func[ Args : val List[VAL] ]
                      { let
                          x == ToInteger[ first[Args] ] ;
                          y == ToInteger[ first[rest[Args]] ]
                        in
                          FromBoolean[ x = y ]
                        ni }  ] ;

        ExtendEnum ==
          func[ Type : val tv; Consts : val List[ID]; Count : val Integer ]
            { let
                IntConst == func[ val List[VAL] ] { FromInteger[ Count ] } }
              in
                if
                  empty?[Consts] ==>
                     let
                       Card, Last == FromFVAL[ PrimFunc[ IntConst ] ]
                     in
                       Extend[ Type, MkID["Card"] || MkL[ MkID["Last"] ],
                               Card || MkL[ Last ] ]
                     ni
                [] else ==>
                     let
                       Const == FromFVAL[ PrimFunc[ IntConst ] ]
                     in
                       ExtendEnum[ Update[ Type, first[Consts], Const ],
                                   rest[Consts], Count + 1 ]
                     ni
                fi
              ni }
      in
        func[ Enum : val ENUM ]
          { ExtendEnum
              [ Extend[ PrimType[],
                        MkID["First"] || MkID["Ord"] || MkID["OrdInv"] ||
                          MkID["Pred"] || MkID["Succ"] || MkL[ MkID["="] ],
                        FromFVAL[First] || FromFVAL[Ord] ||
                          FromFVAL[OrdInv] || FromFVAL[Pred] ||
                          FromFVAL[Succ] || MkL[ FromFVAL[Equal] ] ],
                consts[Enum], 0 ] }
      ni }  (* the end of the definition of TVAL *)
in
  EvalProg[ VAL, ENV, STORE, FVAL, TVAL ]
ni
```

6. Further Research

In this report, we have given a complete formal semantics for the programming language Russell. This semantics can be regarded as the final step in the design of Russell -- in fact, the development of the metacircular semantics has resulted in many changes to the language of [Demers79], which was our starting point. The Russell semantics now suggests to us a number of interesting questions for further research, which we discuss below. These fall into three categories:

1. verifying properties of the Russell design,

2. using Russell as a language description tool and

3. using Russell as a language design tool.

6.1. Verifying Properties of Russell

One of the primary reasons for producing a semantics for Russell is that one of our major goals has been to prove that Russell is "strongly typed" in the sense of [Demers80a]. Obviously, it is necessary to have a formal semantics to be able to construct a rigorous proof of such a property. One of the attractive aspects of the semantics that we have presented above is that its structure makes such a proof feasible (but not trivial).

As discussed in [Demers80a], for Russell to be strongly typed using the signature-matching rules of the language, variable-free expressions (those that do not import identifiers with var signature) in the language must satisfy a substitution property. In essence, we must guarantee that identical variable-free expressions

1. produce semantically identical results when evaluated in the same environment (independent of the store) and

2. produce no observable side-effects when evaluated. Notice that our notion of variable-free expressions does not rule out expressions that perform operations on locally delcared variables. To prove this formally requires careful definitions of "semantically identical" and "observable effects", but we can sketch how such a proof can be done based on the structure of the semantics.

A variable-free denotation in Russell is either a construction or the use of one of the combining forms (e.g., conditional) with a set of variable-free component expressions. By looking at EvalConstr, we can see that the meanings of constructions in Russell are independent of the store. Moreover, by looking at the meanings of the various type constructions (given as part of TVAL), we can easily verify that the only operations provided by these types that inspect or affect the store are those that have _var_ parameters in their signatures. (For example,

_union_{ T1, T2 }$FromT1[x]

has no effect on the store.) Now, we simply need to examine the meanings of each of the combining forms to verify that they preserve the substitution property for variable-free expressions. And this is particularly simple to do because of the loose "coupling" of the store and the meanings of the Russell combining forms.

6.2. Russell as a Language Description Tool

On the basis of one example, Russell looks like an attractive language for writing denotational semantics. It has both the facility for defining rather complex data types (including recursively defined function spaces) and a "hiding" mechanism (type parameterization) that allows many of the details of a data type to be hidden in the remainder of the semantics. We have begun looking at the semantics of other languages (LISP in particular, using the semantics of Gordon[75]) to see whether the approach used in giving the Russell semantics can be applied to other programming languages. At present, we see no reason why not.

Russell as a semantics metalanguage is particularly attractive because it will soon be implemented. Thus, like Mosses's SIS system [79], it will (in principle) be possible for a language designer to "experiment" with a proposed design before embarking on a long implementation project. The benefits of this remain to be proven, but we, the Russell designers, believe such a tool would be quite useful.

6.3. Russell as a Language Design Tool

In writing the Russell semantics, we found it natural to make the meaning function EvalProg polymorphic with respect to the underlying data types of the semantics and to define these data types separately. The interface between the meaning functions and the data type definitions, the specification of the signatures of the data types, turned out to be a useful way of understanding the language -- in a sense, it captures the "tightness" of the coupling between the primitives and combining forms of a language and the choice of the underlying value spaces.

One example from the Russell semantics shows this phenomonon particularly well -- the fact that the signature of the STORE data type in EvalProg is type{}, the simplest type signature. This means that while the meanings of the various Russell combining forms may take a STORE as an argument and yield a STORE as a result, they do not depend on any properties of the STORE. One of the design principles of Russell was that the meaning of data types provided the interpretation of variables, e.g., gave a meaning to assignment and allocation. The fact that we could use the trivial type signature for STORE in EvalProg was conformation of the "correctness" of the Russell design in this regard. Moreover, it indicates that we can form an applicative Russell simply by deleting operations on vars from the signatures of the type constructors -- the meanings of the combining forms do not need to change. (In other words, the STORE could just as well be constant!) We hope to explore how other design decisions affect the definition of the interface between the meaning functions and the domain specifications, something that has not been explored in denotational semantics previously (with the possible exception of [Constable79]).

Appendix A: Collected Syntax

Basic Vocabulary

Letter ::= a | b | c | ... | z | A | B | ... | Z

Digit ::= 0 | 1 | 2 | ... | 9

PunctuationChar ::= $ | (|) | { | } | [|] | << | >> | , | ; | □ | :

OpChar ::= + | - | * | / | ↑ | < | = | > | : | & | ⊥ | ~ | .

WhiteSpace ::= <blank>

Alphameric ::= Letter | Digit

Character ::= Alphameric | PunctuationChar | OpChar | WhiteSpace

Id ::= WordId | OperatorId | NumberId | CharId | StringId

WordId ::= Letter Alphameric*

OperatorId ::= OpChar$^+$

NumberId ::= Digit$^+$

CharId ::= ' Character '

StringId ::= " 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]Denotation ::= [Any]Application | [Any]Primary

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

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 $\}^{+}_{\Box}$ __fi__
| ValDenotation __cand__ ValDenotation
| ValDenotation __cor__ ValDenotation

Guard ::= ValDenotation | __else__

Repetitions

Repetition :: = __do__ $\{$ Guard ==> Denotation$^{+}_{;}$ $\}^{+}_{\Box}$ __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>>}$^{\#}$

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$^{\#}$ | constants

Programs

Program ::= Id == FuncConstruction

1. References

[Demers80a]
Demers, A. and J. Donahue. Data Types, Parameters and Type-Checking. Proceedings Seventh Annual Principles of Programming Languages Symposium, 1980, pp. 12 - 23.

[Demers80b]
Demers, A. and J. Donahue. Type-Completeness as a Language Principle. Proceedings Seventh Annual Principles of Programming Languages Symposium, 1980, pp. 234 - 244.

[Demers80c]
Demers, A., J. Donahue and H. Boehm. An Informal Introduction to Russell. Technical Report TR80-430, Computer Science Department, Cornell University, 1980.

[Donahue79]
Donahue, James. On the Semantics of "Data Type." SIAM J. Computing 8:4 (November 1979), pp. 546 - 560.

[Gordon75]
Gordon, M.J.C. Towards a Semantic Theory of Dynamic Binding. Technical Report CS-75-507, Computer Science Department, Stanford University, 1975.

[Gordon79]
Gordon, M.J.C. The Denotational Description of Programming Languages. Springer-Verlag, 1979.

[Mosses77]
Mosses, P. Making Denotational Semantics Less Concrete. Workshop on Semantics of Programming Languages, Bad Honnef, Germany, 1977.

[Mosses79]
Mosses, P. SIS -- Semantics Implementation System: Reference Manual and User Guide. Technical Report MD-30, Computer Science Department, Aarhus University, 1979.

[Scott76]
Scott, Dana. Data Types as Lattices. SIAM Journal on Computing 5:3 (September 1976).

[Stoy77]
Stoy, J. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, 1977.

[Tennent76]
Tennent, R.D. The Denotational Semantics of Programming Languages. CACM 19:8 (August 1976).