MAKING VARIABLES ABSTRACT: AN EQUATIONAL THEORY FOR RUSSELL

Alan Demers
James Donahue†

January 1983

TR 82-534

Department of Computer Science
Cornell University
Ithaca, NY 14853

†Xerox Palo Alto Center
3333 Coyote Hill Road
Palo Alto, CA 94304

This research was supported in part by the National Science Foundation under Grant MCS79-10148.

Making Variables Abstract: An Equational Theory for Russell

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

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

"Once a person has understood the way in which variables are used in programming he has understood the quintessence of programming." - Edsger Dijkstra [72, p.11]

Introduction

One of the fundamental notions of programming, and thus of programming languages, is the variable. Recently, the variable has come under attack. The proponents of "functional programming" have argued that variables are at the root of all our problems in programming. They claim that we must rid our languages of all manifestations of the "vonNeumann bottleneck" and learn to live in the changeless world of functional combinators.

While we may not believe all the claims of the functional programming advocates, there is evidence that the treatment of variables in most programming languages leaves much to be desired. In this paper we discuss how to make variables "abstract", i.e., how to introduce the notion of variable into a language so that variables have reasonable mathematical properties. This paper includes:

- a discussion of language design principles that allow a more mathematical treatment of variables in programming languages, and
- a description of an equational logic for the programming language Russell [Boehm80]. Although this logic follows much of the development of abstract data types (cf. [Guttag78]), it is novel in its treatment of a language in which expressions not only produce values but also can have effects. We discuss the overall structure of the logic and present in detail the rules particularly relevant to the semantics of variables. A complete presentation of the logic appears in [Demers82], and the rule numbers in this paper agree with the numbers used there.

This research was supported in part by the National Science Foundation under Grant MCS79-10148.

In the next section, we discuss the underlying language principles needed to make an equational specification of variables possible. In the sections that follow, we present a logic that does so for Russell.

Language Principles

In designing Russell, we gave considerable thought to Backus's criticisms of "vonNeumann" languages [Backus78]. We were (and remain) convinced that any general-purpose programming language must incorporate some notion of variable and state. However, we were concerned that

- We ought to be able to give proof rules for all of Russell. This concern has led us in two directions. Hans Boehm [Boehm82] has developed a Hoare-style proof system for the language. In this paper we take a different approach to specifying the language - an equational or algebraic theory.
- The proof system should be small and yet treat the entire language. We did not want to restrict ourselves to "small, weak subsets of full vonNeumann languages that have states that are vastly simpler than real ones." [Backus78, p.618]

We feel that we have achieved both of these goals. Below we present the equational theory of Russell. It is small enough to fit on a few pages of text, yet we feel it provides the essential insights a programmer needs.

The development of an equational logic for Russell was significantly aided by two important design decisions. The first of these was to adopt the principle of *type-completeness* [Demers80]. Basically, type-completeness means

(1) Every Russell expression has a type;

To appear in the Tenth Annual ACM Symposium on the Principles of Programming Languages (Austin, Texas, January 24-26, 1983)

(2) For any type in Russell there is an expression of that type; and

(3) Any Russell expression can be parameterized with respect to any free identifier occurring in it to yield a function of a more complex type.

Because Russell is type-complete, it has a simple, clearly identifiable language core of combining forms and then a larger set of interchangeable parts. These interchangeable parts include the builtin types and, most importantly for our purposes, the operations on variables. In most programming languages variables can be manipulated by programs (they can be passed as arguments to procedures, for example) but the operations that manipulate them are not all part of the surface syntax of the language. Assignment usually is a named primitive operation, but other operations like variable allocation usually are not, making it impossible to define the semantics of variables using the same logic that defines the non-variable component of the language. There are compelling reasons for this - few languages have type structures rich enough to describe such operations as variable allocation. Type-completeness makes it possible for all the operations that manipulate variables in Russell to be named functions in the programming language. Making Russell type-complete allowed us to bring the operations on variables up to the surface, so that the same equational theory that defines the language core can also define the parts of the language that manipulate variables.

The other important design decision was to begin developing an equational proof system for Russell before the language design was frozen. While developing the equational system, we found places where critical properties of the operations on variables could not be expressed as equations among expressions of the language. Our decision to develop an equational specification had an interesting consequence - instead of enriching an *assertion language* for describing Russell, we were forced to enrich the *programming language* itself. It seems to us that this is an important point. A good deal has been written about the need to develop a language and its proof rules simultaneously; the hope is that the language will be molded by the desire to keep its proof rules simple. The use of an equational system puts an interesting twist on this idea: the requirements of the proof rules may point out areas where the programming language should be made more expressive.

In the following section we describe the Russell language subset used in this paper. We then present the overall structure of our equational proof system, and discuss the most important inference rules. The complete set of rules is given in [Demers82]. Next we discuss proof rules for operations on simple variables. We give two sample proofs to convey the flavor of reasoning in the system. Finally, we give rules for composite variables (records) which show how their semantics are straightforward compositions of the semantics of their components.

The Programming Language

The language used below is a minor modification of Russell [Boehm80]. As we have already remarked, Russell is a type-complete language. It includes function- and type-valued expressions and treats them on an equal footing with more conventional expressions.

Russell is statically typed. As described below, the Russell logic includes rules by which we associate with each expression e a syntactic type specifier or *signature* S. The signature of e tells whether e denotes a value or a variable, and gives an expression for the type of e. Thus, a signature has the form

var exp or **val** exp

where, informally, "exp" must be a type-valued Russell expression. Formally, S is a legal signature iff the formula

legal S

is provable in the logic.

Expressions have a fairly conventional syntax. In the following description, variables beginning with p,q,...,x,y,z represent identifiers; variables beginning with d,...,h or T represent expressions; and those beginning with S represent signatures. A Russell expression may be any of:

- An *identifier* or *constant*. Special identifiers such as **int**, **bool** and **type**, which are builtin types, and a number of builtin functions to be introduced below, are treated as constants that may not be redeclared.
- A *sequential composition*, of the form

  $(e_1; ...; e_n)$
- A **let**-*block*, of the form

  let ...; x_i = d_i; ... **in** e **ni**
- A *conditional*, of the form

  if e_1 **then** e_2 **else** e_3 **fi**
- A *loop*, of the form

  while e_1 **do** e_2 **od**
- An *application*

  c_0 [e_1, ..., e_n]

  of function e_0 to arguments e_1 through e_n. When no confusion should arise we shall frequently write applications in infix rather than prefix notation.
- A *function type* expression, of the form

  func[...; x_i:S_i; ...] S

  Intuitively, such an expression denotes the type of n-parameter functions, where x_1 through x_n are parameter names, S_1 through S_n are the corresponding parameter signatures, and S is the result signature. For example,

  func[x:**val int**; y:**val int**] **val int**

  is an expression for the type of the integer addition operation.
- A *function construction*, of the form

  func t[...; x_i:S_i; ...] { e }

  Such an expression denotes a function, where x_1 through x_n are parameter names and S_1 through S_n are the corresponding parameter signatures as above; identifier t is a local name for the function, which

allows it to call itself recursively; and expression e is the function body. For example,

```
func square[x:val int] {
    if x<0 then square[-x]
        else if x>0 then square[x-1]+ x+ x-1
            else 0 fi fi }
```

is an expression for an integer squaring function.

The Structure of the Logic

The first task in specifying the semantics of variables is to find a set of operations suitable for characterizing their behavior. What sort of behavior must be captured?

- That one can *assign* to a variable and *extract the value* possessed by a variable.
- That variables can *alias* – two syntactically distinct variable-producing expressions in a program can produce semantically indistinguishable results.
- That new variables can be *allocated*. This allocation does not change the value of any existing variable, nor does the new variable alias with any existing variable.

In Russell the operations of assignment and value extraction, the predicates that test for aliasing and the operation of allocating a new variable are all just particular builtin functions. Thus it seemed natural to apply the ideas of abstract data type specification, where semantics is given by a set of equations relating the results of function applications, to defining Russell. In doing so we encountered two problems not common to the usual equational specifications:

(1) Russell expressions can have effects, as well as produce values – that is the whole point of having variables in the language. Thus, the equations of our system must relate not only the values of expressions but also the effects of evaluating them. In a sense, one of the major problems in the development of an equational theory for Russell is the semantics of the ";" operator.

(2) Equivalences between expressions may hold only in certain contexts. For example, the following equivalence is obviously valid:

(let x = 3 **in** y := 4; x **ni)** ≡ (y := 4; 3)

Here we have substituted the right-hand side of the declaration ("4") for the declared identifier ("y") within block body. On the other hand, consider

(let x = ValOf(y) **in** y := 4; x **ni)**
≡ (y := 4; ValueOf(y) **ni)**

Again we have just substituted the right-hand side of a declaration for the left-hand identifier. However, this equivalence is clearly invalid. The equations of our system must include premises that allow the first substitution but disallow the second.

These constraints led us to a Gentzen-style deductive system having much in common with those of [Bates82] and [Martin-Lof79]. In the remainder of this section we present the basic structure of the system. In later sections we give the rules of inference that define our Russell subset.

Formulas

Our logic is basically an equational one – formulas can have the form $e_1 \equiv e_2$, where e_1 and e_2 are expressions of the programming language. However, to deal with the problems discussed above we need several other kinds of formulas. The formulas of our system include:

- *Typings.* These are formulas of the form $e \sim S$, which assert that e is a legal Russell expression whose signature is S.
- *Legality Assertions.* These are formulas of the form **legal** S, which assert that S is a legal signature – S has the form **var** e or **val** e where e is a type-valued expression whose value is independent of the store.
- *Purity Assertions.* These are formulas of the form **pure** e, which assert that expression e is independent of the values of any variables – e neither has an observable effect nor produces a value that depends on the current state. An expression e is said to be *pure* if **pure** e holds. The notion of purity is an important one and is discussed at some length below.
- *Equivalences.* These are formulas of the form $e_1 \equiv e_2$, which assert that expressions e_1 and e_2 are completely equivalent – that they have the same value and produce the same observable effects when evaluated. If exp is a Boolean expression we will sometimes abbreviate "exp ≡ True" by "exp". Note that True is a pure, total expression – its evaluation always terminates without error or observable effect – so the equivalent expression exp must likewise be pure and total.

It is worth noting that the truth or falsity of any formula is independent of the store. For example, if $e_1 \equiv e_2$ is true the expressions e_1 and e_2 may reference program variables but the two expressions must be equivalent independent of what values are possessed by those variables.

Environments

An *environment* Γ is a set of formulas

$$\Gamma = \Theta \cup \Phi$$

where Θ is a set of typings in which only simple identifiers (not arbitrary expressions) may appear on the left hand sides, and Φ is set of equivalence formulas. When Θ contains a typing $(x \sim S)$ we say that Θ (and hence Γ) *defines* x. Γ is *closed* if Θ defines every identifier that occurs free in a formula of Θ or Φ. All environments arising in correct proofs are closed.

Given Γ and a set of program identifiers x_1 through x_n, we may construct

$$\Gamma/x_1,\ldots,x_n =_{def} \bigcup\{\, \Gamma' \mid (\Gamma' \subseteq \Gamma) \wedge (\Gamma' \text{ is closed}) \wedge (\Gamma' \text{ defines none of } x_1,\ldots,x_n) \,\}.$$

i.e., the maximal closed subset of Γ not defining any of x_1 through x_n. Less formally, this construction simply deletes from Γ definitions of x_1 through x_n and any

formulas that depend on them. By construction $\Gamma/x_1,...,x_n$ is closed. since an arbitrary union of closed environments is easily seen to be closed. Using the above construction, we define

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

which gives the effect of declaring program identifiers on the environment – first all references to previous instances of the identifiers are deleted from the environment, and then new typings are added. Note that $(\Gamma,...,x_i\sim S_i,...)$ is closed if all free identifiers of S_1 through S_n are among x_1 through x_n or are defined in $\Gamma/x_1,...,x_n$.

Similarly, we define $\Gamma\,|\,x_1,...,x_n$, the **var** *restriction* of Γ to x_1 through x_n, by

$$\Gamma\,|\,x_1,...,x_n =_{def} \bigcup\{\,\Gamma' \mid (\Gamma'\subseteq\Gamma) \wedge (\Gamma' \text{ is closed}) \wedge ((y\sim \textbf{var } T)\in\Gamma' \Rightarrow \exists i\; y = x_i)\,\}$$

This restricts Γ so that any free identifiers with **var** signature must be among x_1 through x_n.

Finally, we define

$$\Gamma_{val} =_{def} \Gamma\,|$$

i.e., the restriction of Γ to no **var** identifiers (leaving only identifiers with **val** typings). This operation is useful because expressions that can be typed using only the information in Γ_{val} are guaranteed to be pure.

Goals

A goal G has the form

$$\Gamma \vdash F$$

with the meaning that the formula F is a consequence of the typings and equivalences in Γ.

Inference Rules

An inference rule has the form

$$\frac{G_1 \quad ... \quad G_n}{G}$$

with the meaning that from G_1 through G_n we may conclude G. We call G the *conclusion* of the rule, and G_1 through G_n its *hypotheses* or *subgoals*. We shall frequently abbreviate by allowing lists of formulas to appear in hypotheses and conclusions. We also follow the convention that the variables t,u,...,z represent new identifiers, and that an expression of the form

$$d[...e_i/x_i...]$$

which represents the result of simultaneously substituting e_1 through e_n for x_1 through x_n, is legal only if no capture occurs.

Theorems

The set of *theorems* is the smallest set of goals closed under deduction using the inference rules. This set is nonempty, since there are inference rules with no hypotheses.

Basic Rules

The system includes some logical rules stating obvious basic facts about equivalences and typings. Among them are *assumption* and *modus ponens* rules, rules stating *reflexivity, symmetry* and *transitivity* of $\equiv$, and *substitution* rules that allow provably equivalent expression to be substituted for one another in equivalence formulas and typings. These rules are straightforward, and can be found in [Demers82].

The Russell subset we are using includes builtin types **int** and **bool** with the Boolean, arithmetic and comparison operators +, -, *, /, $\leq$, $\wedge$, $\vee$, etc. These operations treat values rather than variables, and in a Hoare-style system there are equivalent, identically-named functions in the assertion language. To enable us to reason about terms that use these operators, we include all rules of the form

L10 (equivalent terms)

$$\frac{\Theta \vdash (e_1\sim T,\ e_2\sim T)}{\Gamma \vdash e_1 \equiv e_2}$$

where

(1) Θ consists of the typings contained in Γ.

(2) T is either **int** or **bool**,

(3) Expressions e_1 and e_2 are equivalent terms using only the builtin **int** and **bool** operators mentioned above. Free identifiers are considered universally quantified.

These rules correspond roughly to the assumption "all true formulas in the assertion language are axioms" frequently made when discussing relative completeness of systems of proof rules – cf. [deBakker80].

Using the complete set of rules from [Demers82], we can show that if e_1 and e_2 use only the **bool** operators listed above, and if

$$\Gamma \vdash e_1 \equiv e_2$$

is provable, then

$$\Gamma \vdash (e_1 \Leftrightarrow e_2) \equiv \text{True}$$

is provable, and so is

$$\Gamma \vdash e_3 \equiv \text{True}$$

whenever e_3 is a **bool** term such that $e_1 \Leftrightarrow e_2$ tautologically implies e_3.

Purity, Null Effect and Legality

In giving the semantics of a language in which expressions may have effects, it is necessary at times to limit the use of inference rules to only those subexpressions that are provably independent of state. For exam-

ple, an equivalence such as

$$f[e_1,e_1.e_2] \equiv g[e_2,e_1]$$

must be provable only if the meanings of e_1 and e_2 are independent of the number of times they are evaluated and the order in which they are evaluated. We can guarantee this by requiring that e_1 and e_2 be *pure* – intuitively, that evaluation of e_1 or e_2 neither depend on nor affect the value possessed by any variable. Thus, the logic includes purity assertions and a number of rules that allow us to deduce that expressions are pure. Purity assertions are perhaps the most novel aspect of this work, and they appear as hypotheses in many of the rules below.

Purity is not the same as freedom from side-effects. Consider the expression

ValOf[x]

which extracts the value possessed by variable x. This expression has no effect, but its value manifestly depends on the state; thus it is not pure.

Conversely, evaluation of a pure expression can have an effect, since it does not necessarily terminate without error. Thus the plausible *null effect* rule

(pure null effect (UNSOUND))

$$\frac{\Gamma \vdash (e_1; e_2) \sim S \qquad \Gamma \vdash \textbf{pure}\ e_1}{\Gamma \vdash (e_1; e_2) \equiv e_2}$$

is unsound because it fails to account for possible abortion or nontermination in e_1. For example, it could be used to show

$$(1/0;\ 17) \equiv 17$$

which is clearly untrue because the erroneous division by zero occurs in the left side of the equation but not the right.

Since a pure expression is independent of the state, its effect must be the same – abortion, nontermination, or no effect – every time it is evaluated. Similarly, the value of a pure expression, if it has a value, must be the same every time it is evaluated. In short, a pure expression may be evaluated any number of times, *provided it is evaluated at least once*; The two rules

P1 (pure idempotence)

$$\frac{\Gamma \vdash e \sim S \qquad \Gamma \vdash \textbf{pure}\ e}{\Gamma \vdash e \equiv (e; e)}$$

and

P2 (pure reordering)

$$\frac{\Gamma \vdash (e_1 \sim S_1; e_2 \sim S_2; e_3 \sim S_3) \qquad \Gamma \vdash \textbf{pure}\ e_1}{\Gamma \vdash \textbf{let}\ x_1 = e_1; x_2 = e_2\ \textbf{in}\ e_3\ \textbf{ni} \equiv \textbf{let}\ x_2 = e_2; x_1 = e_1\ \textbf{in}\ e_3\ \textbf{ni}}$$

capture this behavior. Note that only one of two expressions being reordered needs to be pure. In the rules below, a hypothesis of the form

$$\Gamma \vdash \textbf{pure}\ e$$

appears whenever the rule's conclusion involves moving or duplicating (but not entirely eliminating) subexpression e.

Purity Rules

Below are the rules by which one deduces purity of expressions. One of the attractive points of Russell is that there is a large class of syntactically identifiable pure expressions. The following two rules illustrate this. Intuitively, they state that any identifier is pure and that any type-correct Russell expression that imports no **var** identifiers and does not produce a **var** result is pure.

P3 (identifier purity)

$$\frac{\Gamma \vdash x \sim S}{\Gamma \vdash \textbf{pure}\ x}$$

P4 (value purity)

$$\frac{\Gamma_{val} \vdash e \sim \textbf{val}\ T}{\Gamma \vdash \textbf{pure}\ e}$$

Rule P4 is surprisingly powerful. Although it requires a proof of **pure** e to begin in an environment without **var** typings, it does not prohibit introduction of **var** typings in proofs of the subgoals. Thus, the rule can be used to prove that an expression is pure even if the expression allocates variables, provided it does not refer to variables from the surrounding context.

P5 (substitution purity)

$$\frac{\Gamma \vdash e_1 \equiv e_2 \sim S \qquad \Gamma \vdash \textbf{pure}\ e_1}{\Gamma \vdash \textbf{pure}\ e_2}$$

This rule allows us to prove an expression is pure even if it violates the simple syntactic characterization of purity given in the previous two rules, by proving it equivalent to another expression known to be pure.

In addition to the above rules, there are rules that allow us to deduce purity of composite expressions. These rules can be found in [Demers82]. For the most part they are straightforward – they merely state that composite expressions are pure if their components are, and that function type expressions and function constructions are pure unconditionally. For applications the purity rule is more subtle, however.

P12 (application purity)

$$\frac{\Gamma \vdash (..., e_i \sim \textbf{val}\ T_i, ...) \qquad \Gamma \vdash f(..., e_i, ...) \sim \textbf{val}\ T \qquad \Gamma \vdash (\textbf{pure}\ f, ..., \textbf{pure}\ e_i, ...)}{\Gamma \vdash \textbf{pure}\ f(..., e_i, ...)}$$

This rule states that an application is pure if the function and argument expressions are pure and *neither the arguments nor the result are variables.* Although this rule may seem weak at first glance, it is the strongest rule one could hope for in general. Applications that take **var** arguments or produce **var** results may be impure even though all their subexpressions are pure – value extraction and variable allocation, both of which are obviously impure, are simple examples of this.

Applications of certain builtin functions may be pure even though they violate the rather stringent hypotheses of rule P12 above. For these functions there are special application purity rules. Some of these rules appear below; the remaining ones may be found in [Demers82].

Null Effect Rules

We have argued that our rules *cannot* allow a pure expression to be discarded even if its value is never used. However, there are a number of Russell constructs whose evaluation is guaranteed to terminate without error or observable effect, and such expressions *can* safely be discarded when their values are unused. For each such expression e there is a special *null effect* rule of the form

(null effect rule scheme)

$$\frac{\Gamma \vdash (e_1; e_2) \sim S}{\Gamma \vdash (e_1; e_2) \equiv e_2}$$

The most general of these rules is

P13 (identifier null effect)

$$\frac{\Gamma \vdash (x; e) \sim S}{\Gamma \vdash (x; e) \equiv e}$$

which allows us to discard evaluation of an identifier if the value is not subsequently used. There are also null effect rules for function types and function constructions:

P14 (**func** type null effect)

$$\frac{\Gamma \vdash (\textbf{func}[...; x_i{:}S_i; ...]\ S; e') \sim S'}{\Gamma \vdash (\textbf{func}[...; x_i{:}S_i; ...]\ S; e') \equiv e'}$$

P15 (**func** construction null effect)

$$\frac{\Gamma \vdash (\textbf{func}\ f[...; x_i{:}S_i; ...]\{e\}; e') \sim S'}{\Gamma \vdash (\textbf{func}\ f[...; x_i{:}S_i; ...]\{e\}; e') \equiv e'}$$

These rules state that building a function type or a function value in Russell has no effect. Of course, they say nothing about the effect of subsequently applying the function – there can be no general null effect rule for applications, since this depends on the details of the function body. A number of the builtin functions are guaranteed to terminate without error or effect if their arguments do. There is a separate null effect rule for each such function.

There are no specific rules for proving that a composite expression has no effect if its components have none; however, such facts are provable in the system.

Signature Legality Rules

Earlier we remarked informally that a signature **var** e or **val** e is legal only if e is a type expression whose value is independent of the store. We now have the mechanism to make this remark formal:

S1 (signature legality)

$$\frac{\Gamma \vdash T \sim \textbf{val type} \qquad \Gamma \vdash \textbf{pure}\ T}{\Gamma \vdash (\textbf{legal var}\ T, \textbf{legal val}\ T)}$$

Recall that evaluation of a pure expression does not necessarily terminate without error. Thus, for example, the above rule allows a legal signature to contain a nonterminating type expression. This is generally acceptable. We could easily eliminate it by adding to the rule the additional hypothesis

$$\Gamma \vdash (T; \text{True}) \equiv \text{True}$$

which precludes nontermination or abortion of T. This approach is taken in the discussion of records below, as it simplifies the presentation of the rules.

Composite Expressions

Earlier we described the rules by which we may deduce that a composite expression is pure. We shall also require *signature rules* for composite expressions, which allow us to deduce that expression e is type-correct by proving a formula of the form $e \sim S$. Finally, we shall require rules allowing us to deduce equivalences between composite expressions. The most important of the composite expression rules are discussed below; the remaining ones can be found in [Demers82].

The simplest signature rule is the one for sequential composition:

S2 (; signature)

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

Basically, this rule states that a composite expression is type-correct if all its subexpressions are. Most of the other signature rules are similar, and may be found in [Demers82]. The signature rules for applications and **let**-blocks are more interesting, however, as they embody the "signature calculus" rules of [Boehm80].

The signature rule for function applications is as follows:

S3 (application signature)

$$\frac{\Gamma \vdash f \sim \textbf{val func}[...; x_i{:}S_i; ...]\ S \qquad \Gamma \vdash (..., e_i \sim S_i[...e_j/x_j...], ...) \qquad \Gamma \vdash \textbf{legal}\ S[...e_j/x_j...]}{\Gamma \vdash f[..., e_i, ...] \sim S[...e_j/x_j...]}$$

The subgoals and conclusion of this rule involve simultaneous substitution of the argument expressions e_i for the

corresponding parameter names x_i. These substitutions enable the rule to deal with type-valued parameters. For example. let p be the polymorphic identity function

func p[t:**val type**; x:**val** t] {x}

so that

p ~ **val func**[t:**val type**; x:**val** t] **val** t

For the (type-correct) application

p[**int**, 17]

The second and third subgoals of the function signature rule are instantiated as

$\Gamma \vdash$ **int** ~ **val type**[**int**,17/t,x]
$\Gamma \vdash$ 17 ~ **val** t[**int**,17/t,x]
$\Gamma \vdash$ **legal val** t[**int**,17/t,x]

or, equivalently,

$\Gamma \vdash$ **int** ~ **val type**
$\Gamma \vdash$ 17 ~ **val int**
$\Gamma \vdash$ **legal val int**

These allow the conclusion

$\Gamma \vdash$ p[**int**,17] ~ **val int**

as desired.

The signature rule for **let**-blocks can be developed from the application signature rule. Following Landin's "principle of correspondence" [Landin66], we expect the expression

let ...; x_i:S_i = e_i; ... **in** d **ni**

to be type-correct iff the application

f [..., e_i, ...]

is type-correct, where f has signature

func [...; x_i:S_i; ...] S

and S is the signature of the block body d. The rule

S4 (**let** signature)

(Γ,...x_i~S_i...) $\vdash$ d ~ S
$\Gamma \vdash$ (..., e_i ~ S_i[...e_j/x_j...], ...)
$\Gamma \vdash$ **legal** S[...e_j/x_j...]

$\Gamma \vdash$ **let** ...; x_i:S_i = e_i; ... **in** d **ni**
~ S[...e_j/x_j...]

follows quite directly.

The remaining composite expression rules are those that allow us to deduce equivalences. A few of these rules are described below; a complete list appears in the [Demers82]. The most interesting and, for our purposes, the most important of the rules are the ones dealing with **let**-blocks, function applications and conditionals. Most of the **let** rules are used to rename bound identifiers, to rearrange the components of a **let**-block, or to combine a block with an adjacent expression. For example, the rule

C6 (**let** absorption (right))

$\Gamma \vdash$ **let** ...; x_i = d_i; ... **in** e_1 **ni** ~ S_1
(Γ/...x_i...) $\vdash$ e_2 ~ S_2

$\Gamma,\Phi \vdash$ **let** ...; x_i = d_i; ... **in** e_1 **ni**; e_2
≡ **let** ...; x_i = d_i; ... **in** e_1; e_2 **ni**

allows an expression e_2 to be "absorbed" into a **let**-block from the right. The requirement that e_2 can be typed with an environment from which the new identifiers x_i have been eliminated ensures that capture cannot occur.

The following two rules allow **let**-blocks to be introduced and eliminated.

C8 (**let** elimination (a))

$\Gamma \vdash$ **let** x = d **in** e **ni** ~ S
Γ/x $\vdash$ e ~ S

$\Gamma \vdash$ **let** x = d **in** e **ni** ≡ (d; e)

This rule states that a binding that is never referred to may be eliminated.

C9 (**let** elimination (b))

$\Gamma \vdash$ e ~ S

$\Gamma \vdash$ **let** x = e **in** x **ni** ≡ e

In neither of these rules is there any requirement that the subexpressions be pure.

The most complex of the **let** rules is the following *substitution* rule:

C10 (**let** substitution)

$\Gamma \vdash$ **let** ...; x_i:S_i = d_i; ...**in** e **ni** ~ S
...
(Γ,...,x_i~S_i...) $\vdash$
let ...; x_i:S_i = d_i; ...**in** g_j **ni**
≡ **let** ...; x_i:S_i = d_i; ...**in** h_j **ni**
(Γ,...,x_i~S_i...) $\vdash$ **pure** g_j, **pure** h_j
...
(Γ,...,x_i~S_i...)$\cup${...,g_j≡h_j,...} $\vdash$ e ≡ e'

$\Gamma \vdash$ **let** ...; x_i:S_i = d_i; ...**in** e **ni**
≡ **let** ...; x_i:S_i = d_i; ...**in** e' **ni**

When the number of g_i-h_i pairs is zero, this rule specializes to the obviously sound rule:

C10.1 (**let** equivalence)

$\Gamma \vdash$ **let** ...; x_i:S_i = d_i; ...**in** e **ni** ~ S
(Γ,...,x_i~S_i...) $\vdash$ e ≡ e'

$\Gamma \vdash$ **let** ...; x_i:S_i = d_i; ...**in** e **ni**
≡ **let** ...; x_i:S_i = d_i; ...**in** e' **ni**

Informally, $g_i \equiv h_i$ can be used as a predicate describing the bound values; so the general form of the rule allows us to use knowledge of the bound values in proving the equivalence of e and e'. For example, letting g_1 be x_1 and h_1 be d_1, we can derive the following extremely useful rule:

C10.2 (pure β-conversion)

$\Gamma \vdash$ **let** $x_1{:}S_1 = d_1$; ...**in** e **ni** $\sim$ S
$\Gamma \vdash$ **pure** d_1

$\Gamma \vdash$ **let** $x_1{:}S_1 = d_1$; ...**in** e **ni**
$\equiv$ **let** $x_1{:}S_1 = d_1$; ...**in** $e[d_1/x_1]$ **ni**

provided no capture occurs. This derived rule is related to the β-conversion rule of the lambda calculus.

Two important rules give the meaning of function application:

C18 (application argument evalutation)

$\Gamma \vdash$ $d[..., e_i, ...] \sim$ S

$\Gamma \vdash$ $d[..., e_i, ...]$
$\equiv$ **let** $x = d$; ...; $y_i = e_i$; ... **in** $x[..., y_i, ...]$ **ni**

This rule states that argument evaluation may be done by **let**-binding; thus, the rule implies that parameters are passed by value, and that argument evaluation proceeds from left to right.

C19 (application meaning)

$\Gamma \vdash$ (**func** $p[...; x_i{:}S_i; ...]\{e\}$)$[..., e_i, ...] \sim$ S

$\Gamma \vdash$ (**func** $p[...; x_i{:}S_i; ...]\{e\}$)$[..., e_i, ...]$
$\equiv$ **let** $p =$ (**func** $p[...; x_i{:}S_i; ...]\{e\}$);
...; $x_i = e_i$; ... **in** e **ni ni**

The binding for p in the right hand side of the conclusion is necessary in case e calls p recursively. If there are no recursive calls, this binding can be eliminated using the **let**-elimination rule (C8) and the **func** construction null effect rule (P15).

Another important class of rules define conditional expressions. These include a signature rule, absorption rules analogous to the **let** absorption rules above, and a condition evaluation rule similar to application argument evalutation. There are also three **if** elimination rules:

C13 (**if** elimination (true))

$\Gamma \vdash$ **if** True **then** e_2 **else** e_3 **fi** $\sim$ S

$\Gamma \vdash$ **if** True **then** e_2 **else** e_3 **fi** $\equiv e_2$

C14 (**if** elimination (false))

$\Gamma \vdash$ **if** False **then** e_2 **else** e_3 **fi** $\sim$ S

$\Gamma \vdash$ **if** False **then** e_2 **else** e_3 **fi** $\equiv e_3$

C15 (**if** elimination (parallel))

$\Gamma \vdash$ **if** e_1 **then** e_2 **else** e_2 **fi** $\sim$ S

$\Gamma \vdash$ **if** e_1 **then** e_2 **else** e_2 **fi** $\equiv (e_1; e_2)$

The most complex of the **if** rules is:

C16 (**if** substitution)

$\Gamma \vdash$ **if** e_1 **then** e_2 **else** e_3 **fi** $\sim$ S
$\Gamma \vdash$ **pure** e_1
$\Gamma \cup \{e_1 \equiv \text{True}\} \vdash e_2 \equiv d_2$
$\Gamma \cup \{e_1 \equiv \text{False}\} \vdash e_3 \equiv d_3$

$\Gamma \vdash$ **if** e_1 **then** e_2 **else** e_3 **fi**
$\equiv$ **if** e_1 **then** d_2 **else** d_3 **fi**

Like **let** substitution (C10), this rule introduces new equivalences into the environments of its subgoals, making it possible to use knowledge of the truth or falsity of the condition in proving properties of subexpressions e_1 and e_2. As we illustrate below, the **if** substitution and parallel **if** elimination rules can be used together to perform case analysis in the system.

The Semantics of Variables

In Russell, the operations that manipulate variables are generally components of data types. Each builtin type T includes the components:

:= : **func**[**var** T; **val** T] **val** T
ValOf : **func**[**var** T] **val** T
New : **func**[ ] **var** T

These operations perform assignment, value extraction and allocation for variables of type T. The rules below specify precisely how they behave. We give the rules for the builtin type **int**, but essentially the same rules hold for all builtin types, including composite types such as arrays and records. Similar rules are expected to hold for user-defined types as well. However, one must be somewhat careful here. The fact that a user-defined type happens to have a New operation with the right signature does not mean that it will necessarily obey these rules – there is nothing in the name of an operation that prescribes its semantics.

Assignment and Value Extraction

The first step in specifying the semantics of variables is to give rules relating the effects and values produced by assignment and value extraction. These include

V1 (:= value (left))

$\Gamma \vdash$ $(e_1 := e_2) \sim$ **val int**
$\Gamma \vdash$ **pure** e_1

$\Gamma \vdash$ $(e_1 := e_2) \equiv$
$(e_1 := e_2;$ ValOf$[e_1])$

V2 (:= value (right))

$\Gamma \vdash$ $(e_1 := e_2) \sim$ **val int**
$\Gamma \vdash$ (**pure** e_1, **pure** e_2)

$\Gamma \vdash$ $(e_1 := e_2) \equiv (e_1 := e_2; e_2)$

These two rules give the value of assignment expressions: the value produced is the value of the right hand side and the value of the right hand side becomes the value produced by ValOf after the assignment.

P16 (ValOf null effect)

$\Gamma \vdash x \sim \textbf{var int}$
$\Gamma \vdash e \sim S$

$\Gamma \vdash (\text{ValOf}[x]; e) \equiv e$

This rule gives the effect of ValOf – it has none.

V3 (:= annihilation)

$\Gamma \vdash (e{:=}e_1) \sim \textbf{val int}$
$\Gamma \vdash (e{:=}e_2) \sim \textbf{val int}$
$\Gamma \vdash (\textbf{pure } e, \textbf{pure } e_2)$

$\Gamma \vdash (e{:=}e_1; e{:=}e_2) \equiv (e_1; e{:=}e_2)$

V4 (self :=)

$\Gamma \vdash (e{:=}\text{ValOf}[e]) \sim \textbf{val int}$
$\Gamma \vdash \textbf{pure } e$

$\Gamma \vdash e{:=}\text{ValOf}[e] \equiv \text{ValOf}[e]$

These two rules give the effect of assignment. Rule V3 states that if two successive assignments are made to the same variable, only the second one can be noticed. The "**pure** e_2" premise to the rule guarantees that the value of e_2 does not depend on the prior value of the variable. Rule V4 states that assigning the value of a variable to itself has no effect.

Aliasing

The next step in developing a specification of variables is to give rules that describe when variables share storage. To do so in an equational system we must add several new builtin functions to the programming language:

alias, contains, disjoint:
val func[T_1,T_2:**val type**]
val func[x:**var** T_1, y:**val** T_2] **val bool**

Informally, alias[T_1,T_2][e_1,e_2] is true if the expressions e_1 and e_2 evaluate to the same variable; contains[T_1,T_2][e_1,e_2] is true if the expression e_1 evaluates to a variable that contains e_2 as a component, so that changing the contents of variable e_1 will change the contents of e_2, while changing the contents of e_2 will change a component of the contents of e_1. Finally, disjoint[T_1,T_2][e_1,e_2] is true if no changes to the contents of e_1 will affect the contents of e_2 and *vice versa*. Formally, these functions are related by inference rules given below.

As mentioned earlier, we feel it is important that our attempt to develop an equational specification of Russell pointed out areas in which the language needed to be made more expressive. The alias, contains and disjoint predicates were introduced to make an equational description of the language possible; but after some consideration they began to seem natural and potentially quite useful in ordinary programming. For example, a program that manipulates large arrays can use them to test whether an efficient in-place algorithm can be used or copying is required.

The following inference rules define alias, contains and disjoint:

V5 (alias definition)

$\Gamma \vdash (e_1 \sim \textbf{var } T_1, e_2 \sim \textbf{var } T_2)$
$\Gamma \vdash (\textbf{pure } e_1, \textbf{pure } e_2)$

$\Gamma \vdash \text{alias}[T_1,T_2][e_1,e_2] \equiv (\text{contains}[T_1,T_2][e_1,e_2] \;\&\; \text{contains}[T_2,T_1][e_2,e_1])$

V6 (disjoint definition)

$\Gamma \vdash (e_1 \sim \textbf{var } T_1, e_2 \sim \textbf{var } T_2)$
$\Gamma \vdash (\textbf{pure } e_1, \textbf{pure } e_2)$

$\Gamma \vdash \text{disjoint}[T_1,T_2][e_1,e_2] \equiv (\neg\text{contains}[T_1,T_2][e_1,e_2] \;\&\; \neg\text{contains}[T_2,T_1][e_2,e_1])$

These two rules define alias and disjoint in terms of contains: two variables alias if each contains the other, and two variables are disjoint if neither contains the other.

P17 (contains purity)

$\Gamma \vdash \text{contains}[T_1,T_2][e_1,e_2] \sim \textbf{val bool}$
$\Gamma \vdash (\textbf{pure } e_1, \textbf{pure } e_2)$

$\Gamma \vdash \textbf{pure } \text{contains}[T_1,T_2][e_1,e_2]$

P18 (contains null effect)

$\Gamma \vdash (\text{contains}[T_1,T_2][x_1,x_2]; e) \sim S$

$\Gamma \vdash (\text{contains}[T_1,T_2][x_1,x_2]; e) \equiv (T_1; T_2; e)$

V7 (contains transitivity)

$\Gamma \vdash (x_1 \sim \textbf{var } T_1, x_2 \sim \textbf{var } T_2, x_3 \sim \textbf{var } T_3)$

$\Gamma \vdash (\,(\,\text{contains}[T_1,T_2][x_1,x_2] \wedge \text{contains}[T_2,T_3][x_2,x_3]\,) \Rightarrow \text{contains}[T_1,T_3][x_1,x_3]\,) \equiv (T_1; T_2; T_3; \text{True})$

V8 (contains reflexivity)

$\Gamma \vdash x \sim \textbf{var } T$

$\Gamma,\Phi \vdash \text{contains}[T,T][x,x] \equiv (T;\text{True})$

V9 (nonoverlap)

$\Gamma \vdash (\ldots, x_i \sim \textbf{var } T_i, \ldots)$

$\Gamma \vdash ((\text{contains}[T_1,T_0][x_1,x_0] \wedge \text{contains}[T_2,T_0][x_2,x_0]) \Rightarrow (\text{contains}[T_1,T_2][x_1,x_2] \vee \text{contains}[T_2,T_1][x_2,x_1])) \equiv (T_0; T_1; T_2; \text{True})$

V10 (acyclic)

$\Gamma \vdash (x_1 \sim \textbf{var } T, x_2 \sim \textbf{var } T)$

$\Gamma \vdash \text{contains}[T_1,T_2][x_1,x_2] \equiv \text{contains}[T_2,T_1][x_2,x_1]$

These rules give the essential properties of contains. Rule P17 states that an application of contains is independent of the store if its arguments are; similarly, by rule P18 an

application of contains has no observable effect if the argument evaluations have none. Rules V7 and V8 state the obvious facts that containment is transitive and reflexive. Rule V9 states that variables do not overlap – if two variables contain anything in common, then one of the two variables must contain the other. Finally, rule V10 is a well-foundedness requirement, stating that a variable cannot be properly contained in another variable of the same type.

The following three rules give the semantic import of the alias, contains and disjoint tests in the programming language.

V11 (alias equivalence)

$$\frac{\Gamma \vdash (e_1 \sim \textbf{var}\ T_1,\ e_2 \sim \textbf{var}\ T_2) \quad \Gamma \vdash (\textbf{pure}\ e_1,\ \textbf{pure}\ e_2) \quad \Gamma \vdash \text{alias}[T_1,T_2][e_1,e_2] \equiv \text{True}}{\Gamma \vdash e_1 \equiv e_2}$$

This rule states that two variables that alias are completely indistinguishable. Note the hypotheses "**pure** e_1" and "**pure** e_2" are essential and are *not* implied by purity of "alias[T_1,T_2][e_1,e_2]" – e_1 and e_2 might be impure but have effects that exactly cancel one another. For example, one can prove that

alias[**int**,**int**]

[(x:=ValOf[x]+ 1; y),(x:=ValOf[x]–1; y)]

is pure and equivalent to True even though the subexpressions

$$x := \text{ValOf}[x] \pm 1$$

are obviously impure.

V3a (:= annihilation (generalized))

$$\frac{\Gamma \vdash (x{:=}e_1) \sim \textbf{val}\ T_1 \quad \Gamma \vdash (e{:=}e_2) \sim \textbf{val}\ T_2 \quad \Gamma \vdash \text{contains}[T_2,T_1][e,x] \equiv \text{True} \quad \Gamma \vdash (\textbf{pure}\ e,\ \textbf{pure}\ e_2)}{\Gamma \vdash (x{:=}e_1;\ e{:=}e_2) \equiv (e_1;\ e{:=}e_2)}$$

This rule states that an assignment to variable e hides the effect of a previous assignment to x if e contains x. As in the annihilation rule given earlier, the purity hypotheses are necessary to ensure that neither e nor e_2 depends on the previous value of x.

V12 (reordering)

$$\frac{\Gamma \vdash (..., x_i \sim \textbf{var}\ T_i, ...) \quad \Gamma \vdash (..., y_j \sim \textbf{var}\ U_j, ...) \quad \Gamma \vdash (..., \text{disjoint}[T_i,U_j][x_i,y_j], ...) \quad (\Gamma \mid ...x_i...) \vdash e_1 \sim S_1 \quad (\Gamma \mid ...y_j...) \vdash e_2 \sim S_2 \quad \Gamma \vdash e_3 \sim S_3}{\Gamma \vdash \textbf{let}\ t_1 = e_1;\ t_2 = e_2;\ \textbf{in}\ e_3\ \textbf{ni} \equiv \textbf{let}\ t_2 = e_2;\ t_1 = e_1;\ \textbf{in}\ e_3\ \textbf{ni}}$$

This rule allows us to conclude that the effects of two expressions do not interfere with one another, so that the expressions may be evaluated in either order. The hypotheses require that the free **var** identifiers of e_1 are among x_1 through x_n, the free **var** identifiers of e_2 are among y_1 through y_n, and all x's are disjoint from all y's. There is no requirement that the x's or y's themselves be pairwise disjoint, as this is not necessary to show that e_1 and e_2 do not interfere.

Variable Allocation

We can now give the rules that define the allocation function for variables of type **int**:

V13 (New disjoint)

$$\frac{\Gamma \vdash e \sim \textbf{var}\ T}{\Gamma \vdash \text{disjoint}[\textbf{int},T][\text{New}[\,],e] \equiv (T;\ e;\ \text{True})}$$

P17 (New null effect)

$$\frac{\Gamma \vdash e \sim S}{\Gamma \vdash (\text{New}[\,];\ e) \equiv e}$$

V14 (New inaccessible)

$$\frac{\Gamma \vdash e \sim S}{\Gamma \vdash \text{New}[\,]{:=}e \equiv e}$$

These three rules give the important properties of the value and effect of New – a new variable is disjoint from any other variable, and allocating or even assigning to a new variable has no observable effect. There is no rule stating that New is pure – it is *not* pure, since it produces a new variable each time it is invoked. However, as noted above, it may be possible in the system to prove that an expression is pure even if it creates new variables, provided it does not use any variables from the surrounding context. For example, one can use the value purity rule (P4) to show that the expression

let x: **var int** := New[ ] **in** x:=17 **ni**

is pure; and, with some manipulation, one can use the New inaccessible rule (V14) to show that it is equivalent to the constant 17.

Note that the above rules leave many things unsaid about variables in Russell. They do not specify an implementation, but only the observable characteristics of variables in programs. Variable allocation may be done using a stack or a heap; the definition of New is insensitive to such implementation decisions. The operations of assignment or value extraction could have "benevolent side-effects" [Hoare72] like garbage collection; the only obligation is that after an assignment the only variable to yield a new value is the one to which the assignment was made.

Two Examples

To demonstrate the use of our equational system, we give two substantial examples.

A Derived Rule

Our first example illustrates how case analysis can be performed within the system. We shall prove the following derived inference rule:

$\Gamma \vdash (x_1{:=}e_1 \sim \textbf{val}\ T,\ x_2{:=}e_2 \sim \textbf{val}\ T)$
$\Gamma \vdash (\textbf{pure}\ e_1,\ \textbf{pure}\ e_2)$
$\Gamma \vdash (T;\ True) \equiv True$

$\Gamma \vdash (x_1{:=}e_1;\ x_2{:=}e_2;\ ValOf[x_1]$
$\quad \equiv (x_1{:=}e_1;\ x_2{:=}e_2;$
$\quad\quad \textbf{if}\ disjoint[T,T][x_1,x_2]\ \textbf{then}\ e_1\ \textbf{else}\ e_2\ \textbf{fi}$

Intuitively this rule is true because two variables of the same type are either disjoint or alias - they cannot overlap, nor can one of them properly contain the other.

The proof of this derived rule proceeds as follows. Using disjoint definition (V6), contains null effect (P18), and **if** elimination (C15), we show

$x_1{:=}e_1;\ x_2{:=}e_2;\ ValOf[x_1]$
$\quad \equiv disjoint[T,T][x_1;\ x_2];\ x_1{:=}e_1;\ x_2{:=}e_2;\ ValOf[x_1]$
$\quad\quad \equiv \textbf{if}\ disjoint[T,T][x_1;\ x_2]$
$\quad\quad\quad \textbf{then}\ x_1{:=}e_1;\ x_2{:=}e_2;\ ValOf[x_1]$
$\quad\quad\quad \textbf{else}\ x_1{:=}e_1;\ x_2{:=}e_2;\ ValOf[x_1]\ \textbf{fi}$

Next, using **if** substitution (C16) with subgoals

$\Gamma \cup \{disjoint[T,T][x_1,x_2] \equiv True\} \vdash$
$\quad (x_1{:=}e_1;\ x_2{:=}e_2;\ ValOf[x_1])$
$\quad\quad \equiv (x_1{:=}e_1;\ x_2{:=}e_2;\ e_1)$
$\Gamma \cup \{disjoint[T,T][x_1,x_2] \equiv False\} \vdash$
$\quad (x_1{:=}e_1;\ x_2{:=}e_2;\ ValOf[x_1])$
$\quad\quad \equiv (x_1{:=}e_1;\ x_2{:=}e_2;\ e_2)$

(which are discussed below) we obtain

$\quad \equiv \textbf{if}\ disjoint[T,T][x_1;\ x_2]$
$\quad\quad \textbf{then}\ x_1{:=}e_1;\ x_2{:=}e_2;\ e_1$
$\quad\quad\quad \textbf{else}\ x_1{:=}e_1;\ x_2{:=}e_2;\ e_2\ \textbf{fi}$

from which we can obtain the desired result using **if** absorption rules.

The first of the two subgoals is straightforward, since the assumption that x_1 and x_2 are disjoint allows the assignment expressions to be reordered. The second subgoal is a bit more complicated. Using disjoint definition (V6) and the hypothesis that

$$disjoint[T,T][x_1,x_2] \equiv False$$

we obtain, after some manipulation,

$$contains[T,T][x_1,x_2] \vee contains[T,T][x_2,x_1] \equiv True$$

Using contains acyclic (V10) we can show the two disjuncts on the left hand side are equivalent. After more manipulation this leads to

$$(contains[T,T][x_1,x_2] \wedge contains[T,T][x_2,x_1])$$
$$\equiv alias[T,T][x_1,x_2] \equiv True$$

Now that x_1 and x_2 have been shown to alias, the desired result follows directly.

A Simple Procedure

As a second example of the use of our system, we consider a theorem presented in [Cartwright78]. Let SWAP represent the Russell expression

func swap[y_1,y_2: **var int**] **val int**
{ **let** t := New[ ] **in** t := ValOf[y_2];
y_2 := ValOf[y_1]; y_1 := ValOf[t] **ni** }

which is a standard function for swapping the values of two variables. We would like to show that swap works correctly whether or not x_1 and x_2 are disjoint - roughly speaking, we would like to prove something analogous to the Hoare-style correctness formula

$$\{x_1{=}a_1 \wedge x_2{=}a_2\}\ swap[x_1,x_2]\ \{x_2{=}a_1 \wedge x_1{=}a_2\}$$

Since the formulas of our system are equivalences between programming language expressions, we cannot even state precisely this formula. What we *can* show is that for any Γ from which there are deductions

$\Gamma \vdash (x_1 \sim \textbf{var int},\ x_2 \sim \textbf{var int})$
$\Gamma \vdash (e_1 \sim \textbf{val int},\ e_2 \sim \textbf{val int})$
$\Gamma \vdash (\textbf{pure}\ e_1,\ \textbf{pure}\ e_2)$
$\Gamma \vdash swap \equiv SWAP$

there is also a deduction

$\Gamma \vdash (x_1{:=}e_1;\ x_2{:=}e_2;\ swap[x_1,\ x_2])$
$\quad\quad \equiv (x_2{:=}e_1;\ x_1{:=}e_2)$

This equivalence formula appears similar to the Hoare formula, and a proof of it should convince the reader that calling swap actually does exchange the values of the arguments. However, the two formulas really are technically quite different. For example, if x_1 and x_2 are aliased, the Hoare formula

$$\{x_1{=}11 \wedge x_2{=}17\}\ swap[x_1,x_2]\ \{x_2{=}11 \wedge x_1{=}17\}$$

holds because the precondition cannot possibly be satisfied; the equivalence

$$(x_1{:=}11;\ x_2{:=}17;\ swap[x_1,\ x_2]) \equiv (x_2{:=}11;\ x_1{:=}17)$$

holds because 17 is stored over 11 on both sides. The similar equivalence

$$(x_1{:=}11;\ x_2{:=}17;\ swap[x_1,\ x_2]) \equiv (x_1{:=}17;\ x_2{:=}11)$$

does *not* hold when x_1 and x_2 are aliased.

The proof (of the *true* equivalence) proceeds as follows. Using substitution we can show the left hand side of the desired result is equivalent to

$$(x_1{:=}e_1;\ x_2{:=}e_2;\ SWAP[x_1,\ x_2])$$

By application meaning (C19) this is equivalent to

$(x_1{:=}e_1;$ **let** $y_1 = x_1;\ y_2 = x_2$
in ... (body of SWAP) ... **ni**

Using the derived β-conversion rule (C10.2), **let** elimination (C8) and identifier null effect (P13) y_1 and y_2 can be eliminated to yield

$x_1{:=}e_1;\ x_2{:=}e_2;$ **let** t = New[ ] **in**
t:=ValOf[x_2]; x_2:=ValOf[x_1];
x_2:=ValOf[t] **ni**

Now **let** motion can be used to convert this to

$$\begin{array}{l}\textbf{let } t = New[\,]\ \textbf{in}\\ \quad x_1{:=}e_1;\ x_2{:=}e_2;\ t{:=}ValOf[x_2];\\ \quad x_2{:=}ValOf[x_1];\ x_1{:=}ValOf[t]\ \textbf{ni}\end{array}$$

(A use of **let** renaming may be required here to avoid capture of free variables of e.) At this point we invoke the **let** substitution rule in its full generality. We first prove the subgoal

$$\textbf{let } t = New[\,]\ \textbf{in}\ disjoint[\textbf{int},\textbf{int}][t,x_i]\ \textbf{ni} \equiv \textbf{let } t = New[\,]\ \textbf{in True ni}$$

which is straightforward; we then prove

$$\begin{array}{l}(\Gamma, t{\sim}\textbf{var int}) \cup \{disjoint[\textbf{int}.\textbf{int}][t.x_i] \equiv True\} \vdash\\ \quad (x_1{:=}e_1;\ x_2{:=}e_2;\ t{:=}ValOf[x_2];\\ \qquad x_2{:=}ValOf[x_1];\ x_1{:=}ValOf[t])\\ \quad \equiv\ (t{:=}e_2;\ x_2{:=}e_1;\ x_1{:=}e_2)\end{array}$$

This subgoal is more substantial than the first. It is fairly straightforward to transform the left side to

$$(t{:=}e_2;\ x_1{:=}e_1;\ x_2{:=}e_2;\ x_2{:=}ValOf[x_1];\ x_1{:=}e_2)$$

At this point we perform a case analysis. Just as in the previous example, we obtain one case in which

$$disjoint[T,T][x_1,x_2] \equiv True$$

and another case in which

$$alias[T,T][x_1,x_2] \equiv True$$

The first case can be dealt with easily by reordering the assignments to x_1 and x_2. The second case can be handled just as easily by using := annihilation (V3) to eliminate the unwanted assignments.

From the two major subgoals above, **let** substitution (C10) allows us to conclude that the original expression is equivalent to

$$\textbf{let } t = New[\,]\ \textbf{in } t{:=}e_2;\ x_2{:=}e_1;\ x_1{:=}e_2\ \textbf{ni}$$

This is easily transformed to

$$\textbf{let } t = New[\,]\ \textbf{in } t{:=}e_2\ \textbf{ni};\ x_2{:=}e_1;\ x_1{:=}e_2$$

from which the **let** block can be eliminated using argument evaluation (C18) and New inaccessible (V14). This gives us the desired equivalence.

Structured Variables

The set of operations used above for simple variables forms the basis of the characterization of composite variables like records or arrays. Below we present the definition of records in Russell; arrays can be treated by a similar approach.

A Russell expression for a record type has the form

$$\textbf{record}\ (\ f_1{:}T_1;\ ...;\ f_n{:}T_n\ )$$

This is a slight simplification of the actual Russell syntax – in particular, recursively defined types are not allowed – but it will suffice for our purposes. Such an expression denotes a record type with fields f_1 through f_n of types T_1 through T_n respectively. We shall use the signature rule

S9 (**record** type signature)

$$\frac{\begin{array}{l}\ldots\\ \Gamma \vdash \textbf{legal val } T_i\\ \Gamma \vdash (T_i;\ True) \equiv True\\ \ldots\end{array}}{\Gamma \vdash \textbf{record } (...;\ f_i{:}T_i;\ ...) \sim \textbf{val type}}$$

The hypotheses of this rule demand that the expressions for the types of the fields (T_1 through T_n) be pure and free of side effects. While not strictly necessary, this restriction considerably simplifies the rules below; without it the conclusions of the rules would have to contain additional occurrences of the T_i with no purpose other than to ensure soundness of the rules in the face of erroneous or nonterminating type expressions.

P18 (**record** type purity)

$$\frac{\Gamma \vdash \textbf{record } (...;\ f_i{:}T_i;\ ...) \sim \textbf{val type}}{\Gamma \vdash \textbf{pure record } (...;\ f_i{:}T_i;\ ...)}$$

P19 (**record** type null effect)

$$\frac{\Gamma \vdash \textbf{record } (...;\ f_i{:}T_i;\ ...) \sim \textbf{val type}}{\Gamma \vdash (\textbf{record } (...;\ f_i{:}T_i;\ ...);\ e) \equiv e}$$

These are standard purity and null effect rules, reflecting the fact that constructing a record type value is independent of the store and terminates without error.

For this section, we let

$$RR =_{def} \textbf{record } (...;\ f_i{:}T_i;\ ...)$$

be a fixed record type expression. Associated with RR are the usual operations of assignment, value extraction and variable allocation:

$$\begin{array}{l}:= : \textbf{func}[\textbf{var } RR;\ \textbf{val } RR]\ \textbf{val } RR\\ ValOf : \textbf{func}[\textbf{var } RR]\ \textbf{val } RR\\ New : \textbf{func}[\,]\ \textbf{var } RR\end{array}$$

In addition, there are constructor and field selector functions:

$$\begin{array}{l}Mk : \textbf{func}[\textbf{val } T_1;\ ...;\ \textbf{val } T_n]\ \textbf{val } RR\\ f_1^{val} : \textbf{func}[\textbf{val } RR]\ \textbf{val } T_1\\ f_1^{var} : \textbf{func}[\textbf{var } RR]\ \textbf{var } T_1\\ \ldots\\ f_n^{val} : \textbf{func}[\textbf{val } RR]\ \textbf{val } T_n\\ f_n^{var} : \textbf{func}[\textbf{var } RR]\ \textbf{var } T_n\end{array}$$

In Russell, function names can be *overloaded* – two or more functions can have the same name – as long as they have different types so they can be disambiguated. Thus, the identifying superscripts "var" and "val" on the field names are technically unnecessary. We retain them below for clarity.

To specify the meaning of the record type constructor, we must give meanings for each of the above operations. These meanings are defined in terms of the operations for the types T_i. In the rules that follow, we use the

notation T_i:=, T_iValOf and T_iNew for the assignment, value extraction and allocation functions for variables of type T_i.

Construction and field selection for record values are described by two simple rules:

V14 (**record val** selection)

$$\frac{\Gamma \vdash (..., x_i \sim \textbf{val}\ T_i, ...)}{\Gamma \vdash f_i^{val}[Mk[.... x_j,...]] \equiv x_i}$$

V15 (**record val** construction)

$$\frac{\Gamma \vdash e \sim \textbf{val}\ RR \qquad \Gamma \vdash \textbf{pure}\ e}{\Gamma \vdash Mk[.... f_i^{val}[e], ...] \equiv e}$$

These are the simple McCarthy rules for the LISP constructors [McCarthy60] restated in Russell terms. The additional rule

V16 (**record var** selection)

$$\frac{\Gamma \vdash e \sim \textbf{var}\ RR \qquad \Gamma \vdash \textbf{pure}\ e}{\Gamma \vdash ValOf[e] \equiv Mk[..., T_i\$ValOf[f_i^{var}[e]], ...]}$$

shows how the **var** and **val** field selection functions relate to one another. Note that the identity

$$f_i^{val}[ValOf[e]] \equiv T_i\$ValOf[f_i^{var}[e]]$$

follows immediately from rules V14 and V16.

The next few rules describe the aliasing behavior of the fields of record variables.

V17 (field new)

$$\frac{}{\Gamma \vdash f_i^{var}[New[\,]] \equiv T_i\$New[\,]}$$

This rule states that a field selected from a new record variable is indistinguishable from a new variable of the field type. Thus, by new disjoint (V13), the fields of new records are disjoint from all existing variables.

V18 (field disjoint)

$$\frac{\Gamma \vdash e \sim \textbf{var}RR \qquad \Gamma \vdash \textbf{pure}\ e}{\Gamma \vdash disjoint[T_i,T_j][f_i^{var}[e],f_j^{var}[e]] \equiv True}$$

for all $i \neq j$. This rule states that the fields of a record variable are disjoint from one another. Note this fact is *not* a consequence of the preceding rule.

V19 (field containment (a))

$$\frac{\Gamma \vdash (e_1 \sim \textbf{var}\ U, e_2 \sim \textbf{var}\ RR) \qquad \Gamma \vdash (\textbf{pure}\ e_1, \textbf{pure}\ e_2)}{\Gamma \vdash contains[U,RR][e_1,e_2] \equiv \ ... \wedge contains[U,T_i][e_1, f_i^{var}[e_2]] \wedge ...}$$

This rule states that a record variable contains all of its fields, and that it contains nothing else - any variable that contains all the fields contains the record.

V20 (field containment (b))

$$\frac{\Gamma \vdash (e_1 \sim \textbf{var}\ RR.e_2 \sim \textbf{var}\ U) \qquad \Gamma \vdash (\textbf{pure}\ e_1, \textbf{pure}\ e_2)}{\Gamma \vdash contains[RR,U][e_1,e_2] \equiv alias[RR,U][e_1,e_2] \vee ... \vee contains[T_i,U][f_i^{var}[e_1], e_2] \vee ...}$$

This rule states that there is nothing between a record and its fields - a variable that is contained in a record either aliases with the entire record or is contained in one of the fields.

Finally, there are purity and null effect rules for the **val** and **var** field selector functions. The **var** rules

P22 (field purity (**var**))

$$\frac{\Gamma \vdash f_i^{var}[e] \sim \textbf{var}\ T_i \qquad \Gamma \vdash \textbf{pure}\ e}{\Gamma \vdash \textbf{pure}\ f_i^{var}[e]}$$

P23 (field null effect (**var**))

$$\frac{\Gamma \vdash (f_i^{var}[e]; g) \sim S}{\Gamma \vdash (f_i^{var}[e]; g) \equiv (e; g)}$$

may appear strange at first glance, but they are both sound and reasonable - even though a **var** field selector function is passed a variable as an argument, it neither examines nor modifies the value possessed by that variable.

As an extended example, we illustrate that the obviously sound rule

$$\frac{\Gamma \vdash x{:=}e \sim RR \qquad \Gamma \vdash \textbf{pure}\ e}{\Gamma \vdash x{:=}e \equiv Mk[..., f_i^{var}[x]{:=}f_i^{val}[e], ...]}$$

can be derived from the above rules. Beginning from x := e, we first use field containment (V19) and := annihilation (V3) to show

$$x{:=}e \equiv (...; f_i^{var}[x]{:=}f_i^{val}[e]; ...); x{:=}e$$

we then use **record val** construction (V15) to obtain

$$\equiv (...; f_i^{var}[x]{:=}f_i^{val}[e]; ...); x{:=}Mk[..., f_i^{val}[e], ...]$$

By argument evaluation (C18) this becomes

$$\equiv (...; f_i^{var}[x]{:=}f_i^{val}[e]; ...); \textbf{let}\ ...; t_i = f_i^{val}[e]; ...\ \textbf{in}\ Mk[..., t_i, ...]\ \textbf{ni}$$

Using **let** motion rules - note that e is pure and thus commutes with any expression - we obtain

$$\equiv \textbf{let}\ t_1 = f_1^{val}[e]; ...; t_{n-1} = f_{n-1}^{val}[e]; t_n = (...; f_i^{var}[x]{:=}f_i^{val}[e]; ...; f_n^{val}[e])\ \textbf{in}\ Mk[..., t_i, ...]\ \textbf{ni}$$

It is easily shown that

$$(f_n^{var}[x] := f_n^{val}[e]; f_n^{val}[e]) \equiv (f_n^{var}[x] := f_n^{val}[e])$$

Using this fact and **let** motion n times we can reduce the above expression to

$$\equiv \textbf{let } ...;\ t_i = (f_i^{var}[x] := f_i^{val}[e]);\ ... \textbf{ in } Mk[....\ t_i,\ ...]\ \textbf{ni}$$

which by argument evaluation (C18) finally reduces to

$$Mk[\ ...;\ (f_i^{var}[x] := f_i^{val}[e]);\ ...\ ]$$

as desired.

Conclusions

In this paper we have presented an equational theory for Russell and argued that

- It presents a useful *mathematical* specification of variables in Russell, and
- This sort of equational theory can be construed to cover all of even a very rich language like Russell.

The equational logic presented here is certainly not a complete replacement for a conventional Hoare-style system. Most importantly, there are no induction principles to allow proofs of general properties of program correctness. Theorems in this system can give the final results of program execution for particular inputs; but statements such as " for any integer n, the program computes n!" are outside its scope. On the other hand, the fact that it is *always* possible to deduce the result of a terminating computation in the system argues that it gives a complete specification of variables; and this form of specification is a particularly pithy way to get across important properties of the programming language to potential users. In particular, there are components of the Russell equational theory that present the language better than any of our attempts at informal description. When writing the informal description of Russell and explaining the language to others, we often found that describing New as "allocating a new variable" was inadequate. It is far more edifying to give inference rules such as

$$\frac{\Gamma \vdash e \sim \textbf{var } T}{\Gamma \vdash \text{disjoint}[\textbf{int},T][\textbf{int\$New}[\ ],e] \equiv (T;\ e;\ \text{True})}$$

and

$$\frac{\Gamma \vdash e \sim S}{\Gamma \vdash (\text{New}[\ ];\ e) \equiv e}$$

which say very clearly that New produces a result that is disjoint from any other variable, and that applying New has no other effect.

In addition, as we noted above, the process of producing the equational theory led us to make some important changes in the language. It was only when we began to work on the theory that we found the need to introduce alias, contains and disjoint into the programming language. We now consider these operations to be perfectly natural and potentially quite useful.

References

[Backus78]
Backus, J. Can programming be liberated from the vonNeumann style? A functional style and its algebra of programs. *CACM* 21 (1978), pp. 612-641.

[Bates82]
Bates, J. and R. Constable. The definition of μprl. TR-82-492, Department of Computer Science, Cornell University, 1982.

[Boehm80]
Boehm, H., A. Demers and J. Donahue. An informal description of Russell. TR-80-430, Department of Computer Science, Cornell University, 1980.

[Boehm82]
Boehm, H. A logic for expressions with side-effects. *Proceedings Ninth Symposium on Principles of Programming Languages* (1982), pp. 268-280.

[Cartwright78]
Cartwright, R. and D. Oppen. Unrestricted procedure calls in Hoare's logic. *Proceedings Fifth Symposium on Principles of Programming Languages* (1978), pp. 131-140.

[deBakker80]
J. deBakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[Demers80]
Demers, A. and J. Donahue. Type completeness as a language principle. *Proceedings Seventh Symposium on Principles of Programming Languages* (1980), pp. 234-244.

[Demers82]
Demers, A. and J. Donahue. An equational theory for Russell. TR-82-534, Department of Computer Science, Cornell University, 1982.

[Dijkstra72]
Dijkstra, E. Notes on structured programming. In Dahl, O-J., E.W. Dijkstra and C.A.R. Hoare, *Structured Programming*, Academic Press, 1972.

[Guttag78]
Guttag, J. and J. Horning. The algebraic specification of data types. *Acta Informatica* 10 (1978), pp. 27-52.

[Hoare72]
Hoare, C.A.R. Proofs of correctness of data representations. *Acta Informatica* 1 (1972), pp. 271-281.

[Landin66]
Landin, P.J. The next 700 programming languages. *CACM* 9 (1966).

[Martin-Lof79]
Constructive mathematics and computer programming. *Sixth International Congress for Logic, Methodology and Philosophy of Science*, Hanover, Germany (1979).

[McCarthy60]
McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part I. *CACM* 3 (1960), pp. 184-195.