A Proof-Theoretic Assessment of Runtime Type Errors

AT&T Bell Laboratories Technical Memo 11261-921230-43TM

November 1993

Carl A. Gunter
AT&T Bell Laboratories and
the University of Pennsylvania

Didier Rémy
INRIA, Rocquencourt

Abstract

We analyze the way in which a result concerning the absence of runtime type errors can be expressed when the semantics of a language is described using proof rules in what is sometimes called a *natural* semantics. We argue that the usual way of expressing such results has conceptual short-comings when compared with similar results for other methods of describing semantics. These short-comings are addressed through a form of operational semantics based on proof rules in what we call a *partial proof* semantics. A partial proof semantics represents steps of evaluation using proofs with logic variables and subgoals. Such a semantics allows a proof-theoretic expression of the absence of runtime type errors that addresses the problems with such results for natural semantics. We demonstrate that there is a close correspondence between partial proof semantics and a form of structural operational semantics that uses a grammar to describe evaluation contexts and rules for the evaluation of redexes that may appear in such contexts. Indeed, partial proof semantics can be seen as an intermediary between such a description and one using natural semantics. Our study is based on a case treatment for a language called *RAVL* for *R*ecords *A*nd *V*ariants *L*anguage, which has a polymorphic type system that supports flexible programming with records and variants.

1 Runtime Type Errors in Natural Semantics

One of the primary purposes for imposing a type discipline on a programming language is to ensure the absence of certain forms of data incompatibilities that might occur at runtime in the evaluation of an ill-typed program. A characterisitic instance of such a problem occurs when a procedure call is made on an actual parameter that fails to have the proper form demanded of the formal parameter of the procedure. Such a guarantee of runtime safety can often be expressed precisely and proved rigorously for a clean language design. One of the earliest examples of such a treatment appeared in Milner's paper [Mil78] introducing the type system that forms the basis for the Standard Meta-Language (SML) [MTH90, MT91]. Similar results have been stated and proved for many subsequent language designs involving types and semantic specifications. The goal of this paper is to study some of the relationships between various approaches to such results and use this perspective as a basis for describing a new form of operational specification for which it is possible to provide a proof-theoretic expression of the guarantees obtained from the type correctness of a program.

A paper by Wright and Felleisen [WF91], which surveys some of the literature on results about runtime type errors, can be taken as a background for our discussion in this paper, which focuses on the form of such results for a semantics like the one given for the SML in [MTH90]. The term *natural* semantics is sometimes used for this form of specification, which is based on rules that look like those of natural deduction (see the survey in [Gun91] for further discussion). For example, the following natural semantics rules describe call-by-value evaluation and the evaluation of local definitions ('let' bindings):

$$\frac{}{\textbf{fun}\ x\ M \downarrow \textbf{fun}\ x\ M} \qquad (\textsc{Arrow})$$

$$\frac{M \downarrow \textbf{fun}\ x\ L \qquad N \downarrow V \qquad [V/x]L \downarrow U}{M\ N \downarrow U} \qquad (\textsc{App})$$

$$\frac{M \downarrow V \qquad [V/x]N \downarrow U}{\textbf{let}\ x = M\ \textbf{in}\ N \downarrow U} \qquad (\textsc{Let})$$

Basic judgements have the form $M \downarrow U$ where M is a *term* (program) and U is a value (typically defined as a term in a normal form) where U is to be viewed as the result of evaluating M. This binary relation is defined to be the least relation satisfying a collection of rules such as those above. For example, the App rule can be viewed as asserting that the result of applying M to N is U provided three other judgments hold: M evaluates to a value $\textbf{fun}\ x\ L$ and N to a value V such that the result $[V/x]L$ of substituting V for x in L evaluates to U. For sequential languages specified in this way it is possible to show that if $M \downarrow U$ and $M \downarrow V$, then U and V are the same.

Given a semantics expressed in this form, a theorem about runtime type errors can be expressed by providing rules illustrating when such errors arise and showing that type-correct programs do not raise type errors (this approach is used in [ACPP91] for example). Such rules have two forms that might be termed error *origination* and *propogation* rules. Here are typical examples:

$$\frac{M \downarrow U \qquad U \text{ is not a procedure}}{M\ N \downarrow \text{error}} \qquad (\textsc{ErrOrig})$$

$$\frac{M \downarrow \textbf{fun}\ x\ L \qquad N \downarrow \text{error}}{M\ N \downarrow \text{error}} \qquad (\textsc{ErrProp})$$

where error is a distinguished value which does not have a type. The rule ErrOrig indicates that an error is raised (originated) if the operator M in an application $M\ N$ is evaluated to a term that fails to have the form $M \downarrow \textbf{fun}\ x\ L$ for some L. The rule ErrProp indicates that if the evaluation of an argument of a procedure raises an error, then the application also raises (propogates) an error. One proves that a type-correct program does not raise a runtime error by showing that subject reduction holds: that is, if $M \downarrow U$ and M has type σ, then U also has type σ. From this it follows that U is not error because error cannot have type σ.

Although this approach does directly express the errors that are being avoided due to the type correctness of a program, it does so in an *ad hoc* manner. To see what we mean by this, note that the particular choice of error rules determines exactly what the theorem indicating the absence of runtime errors actually means. For example, the rule

$$\frac{N \downarrow \text{error}}{M\ N \downarrow \text{error}} \qquad (\textsc{ErrProp-1})$$

is not a consequence of ErrProp and adding it would therefore result in a stronger theorem asserting that a type-correct program does not evaluate to error.[1] In formulating an assertion about what runtime errors are avoided as a result of type correctness, attention must therefore be paid to whether important origination or propogation rules may have been omitted; it is unclear how such rules could be chosen mechanically. Similar decisions must also be made in the kind of semantic formulation used by Milner [Mil78] and others.

This can be contrasted with the kind of result one may assert when a structural operational semantics (SOS) is used to describe the operational semantics of programs. Under such a semantics, there is a sequence of steps of evaluation (transitions)

$$M \to M_1 \to \cdots \to M_n$$

and M is said to evaluate to M_n just in case M_n is a value (one also shows that if U is a value, then there is no N such that $U \to N$). In such a formulation, a theorem asserting that type correct programs do not have runtime type errors says that if a sequence of relations such as those above holds and M is type correct, then either $M_n \to M_{n+1}$ for some term M_{n+1}, or M_n is a value. This means that *either* the evaluation sequence can be extended, *or* it cannot be extended and the last term in the sequence is a normal form. In this way one rules out, for example, the possibility that M_n has the form $V\ L$ where V is a value that is not a procedure because M_n would then be a non-value to which no rule applies.

Intuitively, type correctness of a term under a natural semantics assures that there is *some* rule that can be applied to the term in searching for the value to which it evaluates. More generally, if we are given a term M and a search is made for a proof of a proposition $M \downarrow U$ for some value U, then an analogy with SOS suggests that the type correctness of M implies that the proof search is never 'blocked' on any of its subgoals. In the remaining sections of this paper we describe how this can be viewed as the proper analogy to the case for a transition system by formalizing the sense in which the search for a proof in natural semantics is like a transition semantics on 'partially completed' proofs, in a system we call a *partial proof* semantics. The absence of runtime type errors in this setting is then described proof-theoretically by saying that a (legitimately formed) partial proof of the evaluation of a well-typed program M is either total (that is, a proof that M has a value U) or it can be developed further by applying additional proof steps to its subgoals. We will illustrate how partial proof semantics can be viewed as an intermediary between natural semantics—with which it shares the characteristic that computation is viewed as the search for a proof—and a form of SOS that is defined by combining rules for the reduction of redexes with a grammar for evaluation contexts.

Our treatment of partial proof semantics is done through a case illustration based on a language called *RAVL,* which has a polymorphic type system that supports flexible programming with records and variants. This language is typical of new language proposals for which the absence of runtime type errors is a key design issue. In RAVL, as in most similar languages for record calculi, the type system is intended to ensure that a type-correct field selection from a record does not raise a runtime error due to the absence of the field and ensure that a case analysis of a variant value covers all of its possible cases.

[1] This rule also has the operational significance that an error is reported even if the evaluation of the operator M diverges. If a similar propogation rule for the operator M is given, then the operational semantics for programs would appear to be non-sequential.

2 RAVL: A Language for Programming with Records and Variants

There has been considerable interest in developing programming languages with strong static type-checking that provide a better treatment of programming with records than languages developed in the 1970's. Seminal research of John Reynolds [Rey80] and Luca Cardelli [Car84, Car88] showed a way to achieve this to a substantial degree using a *subtyping relation* on types. A more recent paper of Mitchell Wand [Wan87] proposed another pleasing calculus for the manipulation of records. His system used parametric polymorphism and the concept of a *row variable* to provide the desired flexibility and also offered the prospect of *type inference.* Subsequently there have been several proposals for similar systems including the Machiavelli language of Atsushi Ohori [OBBT89], the system of Didier Rémy [Rém89] and the ML^{++} language of Jategaonkar and Mitchell [JM88]. RAVL, the language introduced in this paper, is similar to languages studied in [Rém89, Rém90, Rém93].

The new language is the **R**ecords **A**nd **V**ariants **L**anguage—RAVL. Some of the basic design goals of RAVL can be summarized as follows:

1. RAVL is 'upwards compatible' with the functional core fragment of ML,
2. RAVL has a statically inferred polymorphic typing system,
3. type safety properties of RAVL are similar to those that hold for ML,
4. RAVL provides intuitive and flexible constructs for the manipulation of records and variants.

As presented in this paper, RAVL is a theoretical language in the spirit of languages such as the those studied by Cardelli and Milner. Nevertheless, RAVL can be implemented efficiently and we believe that an acceptable concrete syntax for it is also possible.

Syntax of types and terms.

We are given a collection of symbols $\mathcal{L}$ called *labels* which will be used as tags (in variants) and field names (in records). We use the symbols a and b for labels and L for finite sets of labels. The typing system of RAVL is composed of four syntax classes: types, rows, attendances, and schemes. The symbols used for the syntax classes can be summarized as follows:

- σ and τ range over types and α, β, γ over type variables,
- ρ^L ranges over rows and π^L over row variables,
- φ and ψ range over attendances and χ over attendance variables.

Types are defined inductively, based on a given collection $F_1, ..., F_n$ of operators. These operators have a signature indicating the number of arguments of each operator and the sorts of the arguments, where sorts are of types, rows and attendances.

- a type variable α is a type expression,
- $\sigma \to \tau$ is a type expression if σ and τ are type expressions,
- $\Pi\rho^{\emptyset}$ is a type expression,

- $\Sigma\rho^{\emptyset}$ is a type expression,
- $F_i(\bar{\sigma}_i, \bar{\rho}_i, \bar{\varphi}_i)$ is a type expression where F_i is a type operator taking type arguments $\bar{\sigma}$, row arguments $\bar{\rho}$, and attendance arguments $\bar{\varphi}$ where

$$\bar{\sigma}_i \equiv \sigma_1, \ldots, \sigma_{l_i} \quad \text{and} \quad \bar{\rho}_i \equiv \rho_1, \ldots, \rho_{m_i} \quad \text{and} \quad \bar{\varphi}_i \equiv \varphi_1, \ldots, \varphi_{n_i}$$

Rows have the following definition:

- a row variable π^L is a row,
- abs^L is a row,
- $a : \varphi\,;\ \rho^{L\cup\{a\}}$ is a row if $a \notin L$, φ is an attendance and π^L is a row.

Attendances have the following definition:

- an attendance variable χ is an attendance,
- abs is an attendance,
- $\text{pre}(\sigma)$ is an attendance if σ is a type.

The definition of RAVL is relative to a given collection of recursive type equations:

$$\begin{aligned} F_1(\bar{\alpha}_1, \bar{\pi}_1, \bar{\chi}_1) &= \mathcal{F}_1(\bar{\alpha}_1, \bar{\pi}_1, \bar{\chi}_1) \\ &\vdots \\ F_m(\bar{\alpha}_m, \bar{\pi}_m, \bar{\chi}_m) &= \mathcal{F}_m(\bar{\alpha}_m, \bar{\pi}_m, \bar{\chi}_m) \end{aligned}$$

where $\mathcal{F}_i(\bar{\alpha}_i, \bar{\pi}_i, \bar{\chi}_i)$ is a type whose free variables lie among $\bar{\alpha}_i, \bar{\pi}_i, \bar{\chi}_i$. We write $\mathcal{F}_i(\bar{\sigma}_i, \bar{\rho}_i, \bar{\varphi}_i)$ for the type obtained by simultaneously substituting types $\bar{\sigma}_i, \bar{\rho}_i, \bar{\varphi}_i$ for variables $\bar{\alpha}_i, \bar{\pi}_i, \bar{\chi}_i$ respectively in this expression. As usual, a type scheme is either a type or a quantified variable followed by a type scheme. The syntax of types is summarized in a pseudo-BNF grammar in Table 1.

Table 1: Syntax of RAVL types

$$\begin{aligned} \sigma &::= \alpha \mid \sigma \to \sigma \mid \Pi\rho^{\emptyset} \mid \Sigma\rho^{\emptyset} \mid F_i(\sigma_i, r0_i, f0_i) \\ \rho^L &::= \pi^L \mid \text{abs}^L \mid a : \varphi\,;\ \rho^{L\cup\{a\}} \qquad a \notin L \\ \varphi &::= \chi \mid \text{abs} \mid \text{pre}(\sigma) \\ s &::= \sigma \mid \forall\,\alpha.\,s \mid \forall\,\chi.\,s \mid \forall\,\pi.\,s \end{aligned}$$

The rows are defined by a rule scheme in which L can be instantiated by any finite set of labels. Because of the labels, the grammar for the language is context sensitive. The superscripts in row expressions are intended to exclude expressions such as $\Pi\,(a : \chi\,;\ a : \chi'\,;\ \pi^L)$ in which a field is defined twice or $\Pi\,(a : \chi\,;\ \pi^L) \to \Pi\,(\pi^L)$ in which L must be both $\emptyset$ and $\{a\}$. All occurrences of a given row variable should be preceded by the same set of labels (possibly in various different orders). The labels on the inside of a row can be omitted when the outermost superscript is given.

In particular, this is the case for any row expression embedded in a type expression. Hence it will be convenient to write $\Pi\,(a:\chi\,;\ \pi)$, for example, rather than writing $\Pi\,(a:\chi\,;\ \pi^{\{a\}})$. Another way of expressing the rules concerning the formation of such expressions is to treat the types as a sorted algebra as in [Rém93]. Expressions for rows are taken modulo left commutativity,

$$\left(a:\varphi\,;\ b:\psi\,;\ \rho^{(\{a,b\}\cup L)}\right) = \left(b:\psi\,;\ a:\varphi\,;\ \rho^{(\{a,b\}\cup L)}\right)$$

and distributivity,

$$\mathrm{abs}^L = \left(a:\mathrm{abs}\,;\ \mathrm{abs}^{L\cup\{a\}}\right)$$

Ordinarily we omit the labels on such expressions and write,

$$(a:\varphi\,;\ b:\psi\,;\ \rho) = (b:\psi\,;\ a:\varphi\,;\ \rho)$$

and

$$\mathrm{abs} = a:\mathrm{abs}\,;\ \mathrm{abs}$$

The language of expressions is defined by the BNF grammar in Table 2. The typing judgments

Table 2: Syntax of RAVL terms.

$$\begin{aligned} M ::= {} & x \mid \mathbf{fun}\ x\ M \mid M\ M \mid \mathbf{let}\ x = M\ \mathbf{in}\ M \mid \mathbf{intro}\,[F_i]\ M \mid \mathbf{elim}\,[F_i]\ M \\ & \mid \{\} \mid \{M\ \mathbf{with}\ a = M\} \mid M \setminus a \mid M.a \\ & \mid a\ \mathbf{tag}\ M \mid \mathbf{lastcase}\ M\ \mathbf{of}\ a\ \mathbf{tag}\ x \Rightarrow M \mid \mathbf{case}\ M\ \mathbf{of}\ a\ \mathbf{tag}\ x \Rightarrow M\ \mathbf{else}\ x \Rightarrow M \end{aligned}$$

have the form $H \vdash M : s$ where H is a list of assertions $x : s$. The full list of rules is given in Appendix A. The rules listed there fall into three groups. First of all, there are the rules for the Damas-Milner type inference of ML. Second there are rules for the MacQueen-Sethi treatment of recursive types using introduction and elimination constructors. And third there are rules for records and variants that are special to RAVL.

Although RAVL is more expressive than ML, it is still possible to perform ML-style type inference on terms. We omit details here; the result can be obtained from arguments in [Rém93] and [Rém90].

Natural semantics.

A *natural semantics* presents the evaluation of a language through a set of rules for inferring a binary relation between programs and values. This is often an intuitive and flexible way to describe an abstract evaluator for a language; in particular, such rules were used to provide the formal definition of the ML Standard [MTH90]. We now sketch how one might describe a natural semantics for RAVL. The goal is to define a binary relation $M \downarrow U$ between terms M and a special class of terms U called *values*. Values are defined by the grammar in Table 3. We indicated in the right column the name of the set of values of the corresponding shape. Letters U, V and W range over values.

Table 3: Syntax of values

$U ::= A \mid I \mid R \mid B$	normal values	$\mathcal{U}$
$A ::= \mathbf{fun}\ x\ M$	function values	$\mathcal{A}$
$I ::= \mathbf{intro}\,[F]\ U$	intro values	$\mathcal{I}$
$R ::= \{\} \mid \{R\ \mathbf{with}\ a = U\}$	record values	$\mathcal{R}$
$B ::= a\ \mathbf{tag}\ U$	variant values	$\mathcal{B}$

The (non-error) rules given in the first section can be taken as rules for the semantics of RAVL. Some further examples for the record part of the language would look as follows:

$$\frac{M \downarrow \{R\ \mathbf{with}\ a = U\}}{M.a \downarrow U} \qquad \text{(Project-1)}$$

$$\frac{M \downarrow \{R\ \mathbf{with}\ b = V\} \qquad R.a \downarrow U}{M.a \downarrow U} \qquad \text{(Project-2)}$$

$$\frac{M \downarrow R \qquad N \downarrow U}{\{M\ \mathbf{with}\ a = N\} \downarrow \{R\ \mathbf{with}\ a = U\}} \qquad \text{(Extend)}$$

3 Type Errors and SOS

As mentioned earlier, it is possible to formulate a good theorem concerning the absence of runtime errors for a transition semantics of the right kind. Let us now look at an SOS for RAVL in the style urged by Wright and Felleisen [WF91]. It can be seen as an instance of term rewriting in which a grammar of evaluation contexts is used to control the order in which redexes in a term are to be reduced. For RAVL, the evaluation contexts are listed in Table 4. (Our evaluation contexts are slightly more general than left-to-right or right-to-left evaluation of application, but they do describe call-by-value evaluation.) The SOS of RAVL is obtained by connecting these evaluation contexts with reduction rules for redexes as given in Table 5 via the following rule:

$$\frac{M \longrightarrow N}{E[M] \longrightarrow E[N]}$$

which asserts that a term M' evaluates to N' (in one step) if there is an evaluation context E such that M', N' have the forms $E[M], E[N]$ respectively for terms M and N such that $M \longrightarrow N$.

It is possible to show that the resulting reduction system is deterministic in the following sense. Suppose $\stackrel{*}{\longrightarrow}$ is the transitive closure of $\longrightarrow$.

Theorem 1 *If $M \stackrel{*}{\longrightarrow} U$ and $M \stackrel{*}{\longrightarrow} V$ for values U and V, then U and V are the same (up to renaming of bound variables).*

This result is proved by demonstrating local confluence. (The result is not difficult because evaluation contexts cannot be 'embedded' within one another.) The following theorem states precisely the desired property.

Theorem 2 *If $\vdash M : \sigma$ and $M \stackrel{*}{\longrightarrow} N$ and there is no term N' such that $N \longrightarrow N'$, then N is a value.*

Table 4: Evaluation Contexts

$$
\begin{array}{l}
E ::= [\,] \\
\quad \mid E\ M \mid M\ E \mid \textbf{let}\ x = E\ \textbf{in}\ M \mid \textbf{intro}\,[F_i]\ E \mid \textbf{elim}\,[F_i]\ E \\
\quad \mid \{E\ \textbf{with}\ a = M\} \mid \{M\ \textbf{with}\ a = E\} \mid E \setminus a \mid E.a \\
\quad \mid a\ \textbf{tag}\ E \mid \textbf{case}\ E\ \textbf{of}\ a\ \textbf{tag}\ x \Rightarrow M\ \textbf{else}\ x \Rightarrow M \mid \textbf{lastcase}\ E\ \textbf{of}\ a\ \textbf{tag}\ x \Rightarrow M
\end{array}
$$

Table 5: Redex Rules

$$
\begin{array}{rl}
(\textbf{fun}\ x\ M)\ V & \longrightarrow [V/x]M \\
\textbf{let}\ x = V\ \textbf{in}\ M & \longrightarrow [V/x]M \\
\textbf{elim}\,[F_i]\ \textbf{intro}\,[F_i]\ V & \longrightarrow V \\
\{V\ \textbf{with}\ a = U\}.a & \longrightarrow U \\
\{V\ \textbf{with}\ b = U\}.a & \longrightarrow V.a \\
\{V\ \textbf{with}\ a = U\} \setminus a & \longrightarrow V \\
\{V\ \textbf{with}\ b = U\} \setminus a & \longrightarrow \{V \setminus a\ \textbf{with}\ b = U\} \\
\textbf{case}\ a\ \textbf{tag}\ V\ \textbf{of}\ a\ \textbf{tag}\ x \Rightarrow M\ \textbf{else}\ y \Rightarrow N & \longrightarrow [V/x]M \\
\textbf{case}\ b\ \textbf{tag}\ V\ \textbf{of}\ a\ \textbf{tag}\ x \Rightarrow M\ \textbf{else}\ y \Rightarrow N & \longrightarrow [V/y]N \\
\textbf{lastcase}\ a\ \textbf{tag}\ V\ \textbf{of}\ a\ \textbf{tag}\ x \Rightarrow M & \longrightarrow [V/x]M
\end{array}
$$

It can be viewed as a guarantee of safety from runtime errors of type correct programs and proved by demonstrating that subject reduction holds.

The more reduction contexts are permitted the stronger the Subject Reduction Lemma is, since safe evaluation is an easy consequence of the reduction lemma. Thus it is interesting to prove the subject reduction for a general semantics, even a non-deterministic one: the result for any deterministic semantics is then obtained by restricting the evaluation contexts will be a corollary, and the safe evaluation theorem will follow easily. For RAVL, we could have proved the subject reduction lemma for the full calculus where all contexts are admissible.

4 Partial Proof Semantics

Let us turn now to the question of how a result such as 2 can be expressed for a system based on proof rules (as opposed to transitions). To get the intuition for what we will capture rigorously, consider how one might calculate the result of evaluating a program in a language described using natural semantics. If, for example, an application $M\ N$ is to be evaluated, then the last step of the evaluation may come from an application of the rule App. This means that it is necessary to calculate the values of M and N, then combine these into a term obtained from a substitution involving these values, and then evaluate this term. The calculation therefore involves a *subgoal* $M \downarrow X$ where X is a variable whose value must be found, and a subgoal $N \downarrow Y$. This essentially

corresponds to the clause in the grammar for evaluation contexts (given in Table 4) of the form $E ::= \cdots \mid E\ M \mid M\ E \mid \cdots$ Once values have been found for X and Y, we may view the situation as that of a redex analogous to the β-reduction rule at the beginning of Table 5. In the semantics we are about to describe, there are two relations written $M \downarrow u$ and $M \Downarrow u$. The first of these is analogous to the clauses of the grammar for evaluation in SOS, and we call the rules that apply to it *search* rules. The second is analogous to the rules for redex reduction in SOS and we call these the *redex* rules. There are some rules for how the search and redex rules are combined. We call this style of semantic description a *Partial Proof Semantics (PPS)*.

As an example, let us consider the rules associated with application in a partial proof semantics for RAVL. Here are the search and redex rules respectively:

$$\frac{M \downarrow v \qquad N \downarrow w \qquad v\ w \Downarrow u}{M\ N \downarrow u} \tag{App-S}$$

$$\frac{[V/x]M \downarrow u}{(\mathbf{fun}\ x\ M)\ V \Downarrow u} \tag{App-R}$$

The variables u, v, w range over a syntax class of *partial values* that consists either of values U (as given by the grammar in Table 3) or *logic variables* X. The idea being that a logic variable represents an unknown value which will be determined through the resolution of subgoals of the proof. A rule of the form

$$\frac{}{V \Downarrow V} \tag{Value}$$

indicates that a value is related to itself. Two additional rules apply to the formation of subgoals, which are leaves of a partial evaluation tree in which the result of evaluation is a variable:

$$\frac{}{M \downarrow X} \qquad\qquad \frac{}{M \Downarrow X} \tag{Subgoal}$$

The full set of PPS rules for RAVL is given in the appendix. Some definitions are needed to make the rules clear. The grammar of terms for RAVL is expanded to include a new syntax class of *logic variables* which are denoted by X, Y, Z. Here is an example of a program in the expanded grammar

$$\{X\ \mathbf{with}\ a = Y\}$$

A term with no logic variables is said to be a *total program;* these are the ones whose semantics particularly interests us. Similarly, let us refer to values as defined in Table 3 as *total values*. Recall that letters U, V and W range over total values. As in the table, letters such as A, I, R, B range over specific shapes of values. It can be shown that types and values match with one another: values of function types are function values, values of recursive types are intro values, and so on. *Logic variables* are drawn from a new syntax class; letters X, Y and Z range over logic variables. A *Partial value* is a total value or logic variable:

$$u ::= X \mid U$$

Total values are partial values that do not contain any logic variables.

A partial proof is viewed as a way of representing intermediate steps in the search for a proof $\bigtriangledown$ of a relation of the form $M \Downarrow V$ where $\bigtriangledown$ has no logic variables. A proof is said to be *total* if it has no logic variables; otherwise it is said to be *partial*. It can be shown that the conclusion of

a proof $\bigtriangledown$ has the form $M \downarrow V$ or $M \Downarrow U$ where U is a total value if, and only if, $\bigtriangledown$ has no logic variables. A *subgoal* is a leaf of the form $M \downarrow X$ or $M \Downarrow X$ where M is total; it can also be shown that a proof has subgoals if, and only if, it has logic variables. In PPS, the search for a proof can be viewed as a sequence of proofs:

$$\bigtriangledown_1 \longrightarrow \bigtriangledown_2 \longrightarrow \cdots \longrightarrow \bigtriangledown_n$$

where $\bigtriangledown_1$ is a partial proof of the form

$$\frac{}{M \Downarrow X}$$

and each $\bigtriangledown_{i+1}$ can be obtained by resolving or extending subgoals in $\bigtriangledown_i$. The search is to be viewed as complete (terminated) if the proof $\bigtriangledown_n$ is total; its conclusion will have the form $M \Downarrow U$ where U is a (total) value.

Some examples may help to illustrate the idea. The following is a partial proof with two logic variables:

$$\frac{\{\{\}\ \textbf{with}\ a = \textbf{fun}\ x\ x\} \downarrow X \qquad X.a \Downarrow Y}{\{\{\}\ \textbf{with}\ a = \textbf{fun}\ x\ x\}.a \downarrow Y}$$

A refinement of this partial proof can be obtained by carrying out further steps on the left subgoal:

$$\frac{\bigtriangledown \qquad \{\{\}\ \textbf{with}\ a = \textbf{fun}\ x\ x\}.a \Downarrow Y}{\{\{\}\ \textbf{with}\ a = \textbf{fun}\ x\ x\}.a \downarrow Y}$$

where $\bigtriangledown$ is

$$\frac{\begin{array}{c}\{\} \downarrow \{\} \qquad \textbf{fun}\ x\ x \downarrow \textbf{fun}\ x\ x \\ \{\{\}\ \textbf{with}\ a = \textbf{fun}\ x\ x\} \Downarrow \{\{\}\ \textbf{with}\ a = \textbf{fun}\ x\ x\}\end{array}}{\{\{\}\ \textbf{with}\ a = \textbf{fun}\ x\ x\} \downarrow \{\{\}\ \textbf{with}\ a = \textbf{fun}\ x\ x\}}$$

When the logic variable Y is replaced by the total value $\textbf{fun}\ x\ x$ in accordance with redex rule PROJECT-R-1, then the proof becomes total.

By constrast, let us consider the search for a proof of a program that is not type correct. The first step in the partial evaluation of $\{\}\ (\textbf{fun}\ x\ x)$ is the following partial proof:

$$\frac{\{\} \downarrow X \qquad \textbf{fun}\ x\ x \downarrow Y \qquad X\ Y \Downarrow Z}{M\ N \downarrow Z}$$

This can be extended to:

$$\frac{\{\} \downarrow \{\} \qquad \textbf{fun}\ x\ x \downarrow \textbf{fun}\ x\ x \qquad \{\}\ (\textbf{fun}\ x\ x) \Downarrow Z}{M\ N \downarrow Z}$$

Although this proof is not total, its remaining subgoal does not match any of the rules; there is no way of refining further to a total proof or even continuing the effort to resolve the subgoal. The theorem asserting absence of runtime errors for PPS says that this situation does not occur for type-correct programs.

Some care must be taken about the names of logic variables in subgoals when stating the desired result about the resolution of subgoals in a partial proof involving a type-correct program. There

is a problem that can occur if the same variable is used in two different subgoals. For example, a partial proof that has the form

$$\frac{M \downarrow X \qquad N \downarrow X \qquad X\ X \Downarrow Z}{M\ N \downarrow Z}$$

is an instance of APP-S, but an unwanted connection is made between the results of evaluating M and N. A simple technical restriction resolves this problem. A partial proof is said to be *linear* if the logic variables appearing in its subgoals are all distinct. Putting this another way, linear proofs are those in which each logic variable appears in exactly one subgoal. The following lemmas and theorem now express the desired conclusion in a manner similar to that used for SOS.

Lemma 1 (Confluence) *Let M be a RAVL term and suppose tere are proofs*

$$\frac{\bigtriangledown}{M \downarrow U} \qquad \frac{\bigtriangledown'}{M \downarrow U'},$$

Then $U = U'$ and $\bigtriangledown = \bigtriangledown'$.

This is stronger in some sense that the equivalent lemma for SOS approach, since the proofs are also identified. Different choices of reduction order are expressed as different ways in which parts of the proof are completed. Local confluence could be proved for the full calculus, but the proof would be more difficult than that for SOS since redexes can be embbeded. Local confluence and determinism are not required for type safety, but they are useful properties of the semantics. The operational semantics of RAVL does not specifiy the order of evaluation between the argument and the function, but the Confluence Lemma says it is still deterministic. For languages with states or concurrency, leaving the order of evaluation unspecified would make the semantics non-deterministic: a program in such a language could return several answers.

The two central results are the following:

Lemma 2 (Subject reduction) *If $M : \sigma$ and $\dfrac{\bigtriangledown}{M \downarrow u}$, then all of the total programs in $\bigtriangledown$ are well-typed. Also, if u is total, then $u : \sigma$.*

Theorem 3 (Safe evaluation) *Suppose $\vdash M : \sigma$ and $\dfrac{\bigtriangledown}{M \downarrow u}$ is linear. If there is no $\bigtriangledown'$ that extends $\bigtriangledown$ such that $\dfrac{\bigtriangledown'}{M : u'}$, then $\bigtriangledown$ is total (which implies that u is also total).*

5 Discussion and Acknowledgements

Of course, it can be shown that the SOS and PPS specifications of RAVL are equivalent. The primary difference between them is that PPS rules derive an evaluation relation while SOS describes the evaluation relation as the transitive closure of the 'redex reduction' relation. It is possible to modify the SOS description so that rules are used to serve the role of evaluation contexts. For example, the following pair of rules would suffice in place of the clause $E ::= \cdots \mid E\ M \mid M\ E \mid \cdots$ from Table 4:

$$\frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N} \qquad \frac{N \longrightarrow N'}{M\ N \longrightarrow M\ N'}$$

The resulting formulation would then be more like the ones in [Plo81]; the connection between SOS rules and those of PPS would be less clear, however.

The primary significance of PPS comes from the ability to express intermediate stages in the search for a proof of an evaluation relation $M \Downarrow U$. PPS proofs are similar in spirit to the *goal stack* structures in goal-directed theorem proving; they differ in allowing only a very restricted form of proposition and in the fact that the theorem to be proved is not known before the proof is complete (because the result of evaluation is constructed along with the proof). Ideally we would have prefered to retain the natural semantics rules and simply introduce logic variables and subgoals for them, but technical difficulties force the distinction between search and redex rules. To see where the problem arises, suppose we had expressed the App-S as follows:

$$\frac{M \downarrow v \qquad N \downarrow w \qquad v\ w \downarrow u}{M\ N \downarrow u} \qquad \text{(App-S')}$$

Then the same rule would again be applicable to its right-most hypothesis $v\ w \downarrow u$ allowing an infinite regress of partial resolutions of the application into subgoals. This would permit an infinite proof search that does not arise from a divergent evaluation and invalidate the property that a total proofs are uniquely determined by their conclusions.

Some of the basic ideas in PPS have appeared in other work on programming languages and their semantics. A paper by Howe [How91] studies the meta-theory of natural semantic proofs. His work includes the notion of a logic variables in natural semantics and the linearity restriction. Howe also makes a restriction on proof search that has basically the same effect as our use of two relations $\downarrow$ and $\Downarrow$. A somewhat similar idea is embodied in logic programming constructs such as the *freeze, constrain,* and in *wait* declarations [Nai85]. These constructs restrain the use of a clause until logic variables have been instantiated.

What is the merit of PPS relative to the other approaches to the semantics of programming languages? We have stressed that PPS provides a link between natural semantics and SOS for RAVL, but more work will be necessary to determine its robustness as a specification technique for other kinds of languages. For example, the presentation of SOS using evaluation contexts is convenient for describing the semantics of control features: does PPS share or lack this property? What about the semantics of state or concurrency? These are questions that will occupy our future investigations of PPS as a specification formalism.

We would like to acknowledge some remarks of Val Breazu-Tannen that inspired us to think about the problem of expressing freedom from runtime errors in natural semantics style and useful conversations with Amy Felty, Elsa Gunter, Doug Howe, Myra VanInwegen, Xavier Leroy, Luc Maranget, and Fernando Pereira. Gunter's research was partially supported by NSF grant INT-8819598 and by an ONR Young Investigator grant.

References

[ACPP91] M. Abadi, L. Cardelli, B. Pierce, and G. D. Plotkin. Dynamic typing in a statically typed language. *Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991. Preliminary version in Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (Austin, TX), January, 1989.

[Car84] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, pages 51–67. *Lecture Notes in Computer Science vol. 173,* Springer, 1984.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Gun91] C. A. Gunter. Forms of semantic specification. *Bulletin of the European Association for Theoretical Computer Science*, 45:98–113, 1991.

[How91] Douglas J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 162–172. IEEE Computer Society, 1991.

[JM88] L. Jategaonkar and J. C. Mitchell. ML with extended pattern matching and subtypes. In R. Cartwright, editor, *Symposium on LISP and Functional Programming*, pages 198–211. ACM, 1988.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML.* The MIT Press, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[Nai85] L. Naish. *Negation and Control in Prolog*, volume 238 of *Lecture Notes in Computer Science*. Springer, 1985.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Datbase programming in Machiavelli — a polymorphic language with static type inference. In *SIGMOD Conference on the Management of Data*, pages 46–57. ACM, 1989.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University, Ny Munkegade—DK 8000 Aarhus C—Denmark, 1981.

[Rém89] Didier Rémy. Records and variants as a natural extension of ML. In *Sixteenth Annual Symposium on Principles Of Programming Languages*, 1989.

[Rém90] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objects Enregistrements dans les Langages Fonctionnels.* Thèse de doctorat, Université de Paris 7, 1990.

[Rém93] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993.

[Rey80] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 211–258. *Lecture Notes in Computer Science vol. 94,* Springer, 1980.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In D. Gries, editor, *Second Symposium on Logic In Computer Science*, pages 207–276, Ithaca, New York, June 1987. IEEE Computer Society Press.

[WF91] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report COMP TR91-160, Department of Computer, Rice University, 1991.

A Typing Rules for RAVL

$\alpha, \chi, \pi \notin H$ $\quad \dfrac{H \vdash M : s}{H \vdash M : \forall\, \alpha\,.\, s}$ $\quad \dfrac{H \vdash M : s}{H \vdash M : \forall\, \chi\,.\, s}$ $\quad \dfrac{H \vdash M : s}{H \vdash M : \forall\, \pi\,.\, s}$ (All-Intro)

$\dfrac{H \vdash M : \forall\, \alpha\,.\, s}{H \vdash M : [\sigma/\alpha]s}$ $\quad \dfrac{H \vdash M : \forall\, \chi\,.\, s}{H \vdash M : [\varphi/\chi]s}$ $\quad \dfrac{H \vdash M : \forall\, \pi\,.\, s}{H \vdash M : [\rho/\pi]s}$ (All-Elim)

$\dfrac{}{H \vdash x : H(x)}$ (Var) $\quad \dfrac{H[x : \sigma] \vdash M : \tau}{H \vdash \mathbf{fun}\ x\ M : \sigma \to \tau}$ (Fun) $\quad \dfrac{H \vdash M : \sigma \to \tau \qquad H \vdash N : \sigma}{H \vdash M\ N : \tau}$ (App)

$\dfrac{H \vdash M : s \qquad H[x : s] \vdash N : \sigma}{H \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : \sigma}$ (Let) $\quad \dfrac{H \vdash M : \sigma \qquad \sigma = \tau}{H \vdash M : \tau}$ (Equal)

$\dfrac{H \vdash M : \mathcal{F}_i(\bar{\sigma}_i, \bar{\rho}_i, \bar{\varphi}_i)}{H \vdash \mathbf{intro}\,[F_i]\ M : F_i(\bar{\sigma}_i, \bar{\rho}_i, \bar{\varphi}_i)}$ (Rec-I) $\quad \dfrac{H \vdash M : F_i(\bar{\sigma}_i, \bar{\rho}_i, \bar{\varphi}_i)}{H \vdash \mathbf{elim}\,[F_i]\ M : \mathcal{F}_i(\bar{\sigma}_i, \bar{\rho}_i, \bar{\varphi}_i)}$ (Rec-E)

$\dfrac{}{H \vdash \{\} : \Pi\ (\mathrm{abs})}$ (Empty) $\quad \dfrac{H \vdash M : \Pi\ (a : \mathrm{abs}\,;\ \rho) \qquad H \vdash N : \sigma}{H \vdash \{M\ \mathbf{with}\ a = N\} : \Pi\ (a : \mathrm{pre}(\sigma)\,;\ \rho)}$ (Extend)

$\dfrac{H \vdash M : \Pi\ (a : \sigma\,;\ \rho)}{H \vdash M \setminus a : \Pi\ (a : \mathrm{abs}\,;\ \rho)}$ (Restrict) $\quad \dfrac{H \vdash M : \Pi\ (a : \mathrm{pre}(\sigma)\,;\ \rho)}{H \vdash M.a : \sigma}$ (Project)

$\dfrac{H \vdash M : \sigma}{H \vdash a\ \mathbf{tag}\ M : \Sigma(a : \mathrm{pre}(\sigma)\,;\ \rho)}$ (Tag) $\quad \dfrac{H \vdash L : \Sigma(a : \mathrm{pre}(\sigma)\,;\ \mathrm{abs}) \qquad H[x : \sigma] \vdash M : \tau}{H \vdash \mathbf{lastcase}\ L\ \mathbf{of}\ a\ \mathbf{tag}\ x \Rightarrow M : \tau}$ (Lastcase)

$$\frac{H \vdash L : \Sigma(a : \mathrm{pre}(\sigma)\,;\ \rho) \qquad H[x : \sigma] \vdash M : \tau \qquad H[y : \Sigma(a : \varphi\,;\ \rho)] \vdash N : \tau}{H \vdash \mathbf{case}\ L\ \mathbf{of}\ a\ \mathbf{tag}\ x \Rightarrow M\ \mathbf{else}\ y \Rightarrow N : \tau}\ (\textsc{Case})$$

B Partial Proof Semantics for RAVL

B.1 Search rules

$\dfrac{\mathbf{fun}\ x\ M \Downarrow w}{\mathbf{fun}\ x\ M \downarrow w}$ (Fun-S) $\quad \dfrac{M \downarrow v \qquad N \downarrow w \qquad v\ w \Downarrow u}{M\ N \downarrow u}$ (App-S) $\quad \dfrac{M \downarrow v \qquad \mathbf{let}\ x = v\ \mathbf{in}\ N \Downarrow u}{\mathbf{let}\ x = M\ \mathbf{in}\ N \downarrow u}$ (Let-S)

$\dfrac{M \downarrow v \qquad \mathbf{intro}\,[F_i]\ v \Downarrow w}{\mathbf{intro}\,[F_i]\ M \downarrow w}$ (Intro-S) $\quad \dfrac{M \downarrow v \qquad \mathbf{elim}\,[F_i]\ v \Downarrow u}{\mathbf{elim}\,[F_i]\ M \downarrow u}$ (Elim-S)

$\dfrac{\{\} \Downarrow w}{\{\} \downarrow w}$ (Empty-S) $\quad \dfrac{M \downarrow u \qquad N \downarrow v \qquad \{u\ \mathbf{with}\ a = v\} \Downarrow w}{\{M\ \mathbf{with}\ a = N\} \downarrow w}$ (Extend-S)

$\dfrac{M \downarrow v \qquad v \setminus a \Downarrow u}{M \setminus a \downarrow u}$ (Restrict-S) $\quad \dfrac{M \downarrow v \qquad v.a \Downarrow u}{M.a \downarrow u}$ (Project-S)

$\dfrac{M \downarrow v \qquad a\ \mathbf{tag}\ v \Downarrow w}{a\ \mathbf{tag}\ M \downarrow w}$ (Tag-S) $\quad \dfrac{L \downarrow v \qquad \mathbf{case}\ v\ \mathbf{of}\ a\ \mathbf{tag}\ x \Rightarrow M\ \mathbf{else}\ y \Rightarrow N \Downarrow u}{\mathbf{case}\ L\ \mathbf{of}\ a\ \mathbf{tag}\ x \Rightarrow M\ \mathbf{else}\ y \Rightarrow N \downarrow u}$ (Case-S)

$$\frac{L \downarrow v \qquad \mathbf{lastcase}\ v\ \mathbf{of}\ a\ \mathbf{tag}\ x \Rightarrow M \Downarrow u}{\mathbf{lastcase}\ L\ \mathbf{of}\ a\ \mathbf{tag}\ x \Rightarrow M \downarrow u}\ (\textsc{Lastcase-S})$$

B.2 Redex rules

$$\frac{[V/x]M \downarrow u}{(\mathbf{fun}\ x\ M)\ V \Downarrow u}\ (\textsc{App-R}) \qquad \frac{[V/x]M \downarrow u}{\mathbf{let}\ x = V\ \mathbf{in}\ M \Downarrow u}\ (\textsc{Let-R})$$

$$\frac{}{\mathbf{elim}\,[F_i]\ (\mathbf{intro}\,[F_i]\ U) \Downarrow U}\ (\textsc{Elim-R}) \qquad \frac{}{\{V\ \mathbf{with}\ a = U\}.a \Downarrow U} \qquad \frac{V.a \Downarrow u}{\{V\ \mathbf{with}\ b = U\}.a \Downarrow u}\ (\textsc{Project-R})$$

$$\frac{}{\{\} \setminus a \Downarrow \{\}} \qquad \frac{}{\{V\ \mathbf{with}\ a = U\} \setminus a \Downarrow V} \qquad \frac{V \setminus a \Downarrow v \qquad \{v\ \mathbf{with}\ b = U\} \Downarrow w}{\{V\ \mathbf{with}\ b = U\} \setminus a \Downarrow w}\ (\textsc{Restrict-R})$$

$$a \neq b \qquad \frac{[V/x]M \downarrow u}{\mathbf{case}\ a\ \mathbf{tag}\ V\ \mathbf{of}\ a\ \mathbf{tag}\ x \Rightarrow M\ \mathbf{else}\ y \Rightarrow N \Downarrow u}\ (\textsc{Case-R})$$

$$\frac{[b\ \mathbf{tag}\ V/y]N \downarrow u}{\mathbf{case}\ b\ \mathbf{tag}\ V\ \mathbf{of}\ a\ \mathbf{tag}\ x \Rightarrow M\ \mathbf{else}\ y \Rightarrow N \Downarrow u} \qquad \frac{[V/x]M \downarrow u}{\mathbf{lastcase}\ a\ \mathbf{tag}\ V\ \mathbf{of}\ a\ \mathbf{tag}\ x \Rightarrow M \Downarrow u}\ (\textsc{Lastcase-R})$$

B.3 Other rules

$$\frac{}{M \downarrow X}\ (\textsc{Subgoal-S}) \qquad \frac{}{M \Downarrow X}\ (\textsc{Subgoal-R}) \qquad \frac{}{V \Downarrow V}\ (\textsc{Value})$$