Extensible Denotational Language Specifications

Robert Cartwright, Matthias Felleisen
Department of Computer Science
Rice University
Houston, TX 77251-1892

To appear in: Symposium on *Theoretical Aspects of Computer Software*
Sendai, April 18–21, 1994

Extensible Denotational Language Specifications*

Robert Cartwright[1] and Matthias Felleisen[1,2**]

[1] Department of Computer Science
Rice University
Houston, TX 77251-1892

[2] Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract. Traditional denotational semantics assigns radically different meanings to one and the same phrase depending on the rest of the programming language. If the language is purely functional, the denotation of a numeral is a function from environments to integers. But, in a functional language with imperative control operators, a numeral denotes a function from environments and continuations to integers. This paper introduces a new format for denotational language specifications, *extended direct semantics*, that accommodates orthogonal extensions of a language without changing the denotations of existing phrases. An extended direct semantics always maps a numeral to the same denotation: the injection of the corresponding number into the domain of values. In general, the denotation of a phrase in a functional language is always a projection of the denotation of the same phrase in the semantics of an extended language—no matter what the extension is. Based on extended direct semantics, it is also possible to construct interpreters for complete languages by composing interpreters for language fragments.

1 The Denotational Specifications of Complex Languages

A programming language like Scheme [25], Common LISP [32], or ML [19] consists of a rich functional core, augmented by destructive operations on data objects, control constructs, and possibly other imperative operators. Traditional denotational language specifications [1, 15, 27, 35] cope with these constructs by interpreting program phrases as functions that map *environments* $\times$ *stores* $\times$ *continuations* to *values* $\times$ *stores*. Programmers, however, rely on simpler semantic descriptions when they reason about program phrases. Most program phrases do not exploit the full generality of the language, permitting their semantics to

* The paper is an extended and revised version of Rice Technical Report 90-105, "Extended Direct Semantics", January 1990.

** The authors are partially supported by NSF grant CCR 91-22518. The second author is also partially supported by Arpa grant 8313, issued by ESD/AVS under Contract No. F196228-91-C-0168 under the direction of Robert Harper and Peter Lee.

be analyzed using a simpler semantic model. For example, if a program phrase is purely functional and its free variables are always bound to effect-free procedures, it can be interpreted as a function mapping *environments* to *values*. Similarly, if an imperative program phrase does not use general control operators like `callcc`, `goto`, or `catch` and its free variables are always bound to values and procedures conforming to the same constraint, it can be interpreted as a function mapping *environments* $\times$ *stores* into *values* $\times$ *stores*. Unfortunately, denotational definitions of practical programming languages are written in a form that makes it difficult to extract a simplified definition for a disciplined subset—much less prove that the definitions are equivalent over the restricted language.

Another way to describe the same problem is to observe what happens to the meanings of simple program phrases like the numeral `5` when a language is extended. In a functional language, the numeral `5` denotes what a programmer expects: a function from environments to the integer **5**. But if we add reference cells to the language, `5` denotes a constant function from *environments* $\times$ *stores* to *values* $\times$ *stores*. Moreover, if we add control operators (such as `goto` or `callcc`), the meaning of `5` becomes a constant function from *environments* $\times$ *stores* $\times$ *continuations* to *values* $\times$ *stores*. This annoying property of denotational semantics is well-known among language researchers. Indeed, in a recent survey paper, Peter Mosses [23] has argued that this phenomenon has been a major impediment to the acceptance of denotational semantics as a practical vehicle for defining programming languages.

In this paper, we show how to cast denotational definitions in a form that preserves the simple semantics of important language subsets such as the functional core. The simplified definitions for designated subsets are projections of the general definition. The new form for denotational definitions consists of a trivial *base* definition and a sequence of *extensions*. The base definition describes a trivial language with exactly two programs: $\bot$, which always diverges, and `err`, which always aborts and reports an error. The extensions all fit within the same schema.

The schema critically relies on the distinction between a complete program and a nested program phrase. A complete program is thought of as an agent that interacts with the outside world, *e.g.*, a file system, and that affects global resources, *e.g.*, the store. A central authority administers these resources. The meaning of a program phrase is a *computation*, which may be a *value* or an *effect*. If the meaning of a program phrase is an *effect*, it is propagated to the central authority. The propagation process adds a function to the effect package such that the central authority can resume the suspended calculation. We refer to this component of an effect package as the *handle* since it provides access to the place of origin for the package.

The central authority is implemented via the function `admin`. It performs the *actions* specified by effects. Actions can modify resources, can examine them without changing them, or may simply abort the program execution. Once the action is performed, the administrator extracts the *handle* portion of the *effect* and invokes it, if necessary. The *handle* then performs whatever computation remains following the *action* of the *effect*. Thus, at top-level a handle is roughly a conventional continuation.

Casting a language extension into our framework requires the specification of four components:

- a declaration of the new syntactic constructors for the extension;
- the definition of a domain of new *values*, *resources*, and *actions* produced by the extension;
- the definition of new clauses in the meaning function $\mathcal{M}$ for each new syntactic constructor;
- the definition of new clauses in the function `admin` for the new actions.

In essence, language extensions are determined by adding clauses to the meaning function $\mathcal{M}$ that describe what *value* or *effect* an instance of a new construct denotes. New clauses in the function `admin` determine how *actions* are interpreted as transformations on *resources*.

In the second section we motivate our denotational framework based on our prior work on operational semantics. We use the third section to illustrate our new approach to structuring denotational definitions by presenting an idealized dialect of **Scheme** (**Core ML**) as the sum of a few extensions to a trivial base language. Next we show how to exploit this new technique to write modular interpreters in the fourth section. In the fifth section we address two lines of related work. Section A of the Appendix defines the notation for domain equations; Section B contains the pieces of the implementation that are not explained in Section 4.

2 Extensible Operational Semantics

The intuition underlying our framework of extensible semantic specifications is derived from a generalization of the operational semantics of the λ-calculus to full-fledged Scheme-style programming languages with exceptions, first-class continuations, and assignments [9, 10, 11, 12]. The extensions of these semantics are conservative in the sense that the reduction relation for the extended language is a superset of the original reduction relation. More precisely, the reduction relation for the core language is simply interpreted over the larger syntax but is not changed otherwise. The new linguistic facilities are interpreted by the addition of new reductions.

We illustrate our framework for extensible operational semantics by examining a simple functional programming language, Pure Scheme. Fig. 1 contains the complete specification of Pure Scheme's syntax and reduction semantics. Pure Scheme's term language contains numeric, boolean, and functional constants, variables, λ-abstractions (user-defined procedures), **if**-expressions, and applications. The expression $(\lambda\, x.e)$ *binds* x in e. An occurrence of a variable x that is not bound by a surrounding $\lambda\, x.$ is *free*. An expression is *closed* if it does not contain free variables. If e and e' are expressions, with x possibly free in e,

$e[x/e']$ is the result of substituting all free occurrences of x in e by e' without capturing free variables in the latter.

Syntax

$$
\begin{array}{rcll}
c & ::= & 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots & (\textit{numerals}) \\
 & & \mid \mathsf{true} \mid \mathsf{false} & (\textit{booleans}) \\
f & ::= & \mathsf{zero?} \mid \mathsf{add1} \mid \mathsf{sub1} & (\textit{numeric functions}) \\
v & ::= & c \mid f & (\textit{constants}) \\
 & & \mid x & (\textit{variables}) \\
 & & \mid (\lambda\, x.\ e) & (\textit{abstractions}) \\
e & ::= & v & (\textit{values}) \\
 & & \mid (\mathsf{if}\ e\ e\ e) & (\textit{branching}) \\
 & & \mid (e\ e) & (\textit{applications})
\end{array}
$$

Auxiliary Syntax (Evaluation Contexts)

$$
\begin{array}{rcll}
E & ::= & [\ ] & (\textit{hole}) \\
 & & \mid (E\ e) & (\textit{evaluate function position}) \\
 & & \mid (v\ E) & (\textit{evaluate argument position}) \\
 & & \mid (\mathsf{if}\ E\ e\ e) & (\textit{evaluate test position})
\end{array}
$$

Standard Reductions

$$
\begin{array}{rcll}
E[(f v)] & \longrightarrow & E[\delta(f,v)] \quad \text{if } \delta(f,v) \text{ is defined} & (\delta) \\
E[((\lambda\, x.\ e)\ v)] & \longrightarrow & E[e[x/v]] & (\beta_v) \\
E[(\mathsf{if}\ \mathsf{true}\ e_1\ e_2)] & \longrightarrow & E[e_1] & (\mathbf{iftrue}) \\
E[(\mathsf{if}\ \mathsf{false}\ e_1\ e_2)] & \longrightarrow & E[e_2] & (\mathbf{iffalse})
\end{array}
$$

Constant Interpretation

$$
\begin{aligned}
\delta(\mathsf{zero?}, 0) &= \mathsf{true} \\
\delta(\mathsf{zero?}, n+1) &= \mathsf{false} \\
\delta(\mathsf{add1}, n) &= n+1 \\
\delta(\mathsf{sub1}, n) &= n-1
\end{aligned}
$$

Fig. 1. Pure Scheme and Its Reduction Semantics

The reduction semantics of Pure Scheme determines a partial function from programs to values where a program is a closed expression. There are two sets of *values*: procedures and constants. Following the λ-calculus tradition, the specification of the *eval*-function is based on the transitive closure of the standard reduction function. The latter is a reduction that partitions any non-value into an *evaluation context* and a redex, reduces the redex, and fills the hole of the evaluation context with the contractum. An evaluation context is a special context. The location of the hole in an evaluation context determines which sub-term must be evaluated next. Thus, the definition of the set of evaluation contexts for

Additional Syntax

$$\begin{array}{rll} e ::= & (\mathsf{ref}\ e) & (\textit{reference allocation}) \\ & \mid (!\ \ e) & (\textit{reference dereferencing}) \\ & \mid (e\ \ :=\ \ e) & (\textit{reference update (infix)}) \end{array}$$

$$\begin{array}{rll} s ::= & \{(x, v)\}^* & (\textit{store (identifier-value bindings)}) \end{array}$$

Additional Auxiliary Syntax

$$\begin{array}{rll} E ::= & (\mathsf{ref}\ E) & (\textit{evaluate sub-expression}) \\ & \mid (!\ \ E) & (\textit{evaluate sub-expression}) \\ & \mid (E\ \ :=\ \ e) & (\textit{evaluate location position}) \\ & \mid (v\ \ :=\ \ E) & (\textit{evaluate value position}) \end{array}$$

Standard Reductions

$$\begin{array}{rcll} s\ E[(\mathsf{ref}\ v)] & \longrightarrow & s\,(l, v)\ E[l] & (\mathsf{alloc}) \\ & & \text{if } l \text{ is not used in } s \text{ or } E[v] & \\ s\,(l, v)\,s'\ E[(!\ \ l)] & \longrightarrow & s\,(l, v)\,s'\ E[v] & (\mathsf{lookup}) \\ s\,(l, v)\,s'\ E[(l\ \ :=\ \ v')] & \longrightarrow & s\,(l, v')\,s'\ E[l] & (\mathsf{update}) \end{array}$$

Fig. 2. State Scheme and Its Reduction Semantics

a language defines the order of evaluation.

In a functional language of Pure Scheme, the evaluation context of a redex plays no role until the redex is reduced to a value. For example, the expression $E[((\lambda\ x.(\mathsf{add1}\ (\mathsf{if}\ x\ 0\ 1)))\ \mathsf{true})]$ reduces to $E[1]$, no matter what E is. More generally, all (closed) expressions in a mathematical language (with recursion) either produce a value, signal an error, or diverge. Consequently, a denotational semantics can map an expression in such a language to numbers, boolean values, functions, errors, or bottom.

Once Pure Scheme is extended to a language with ML-style reference cells (see Fig. 2)[3] the reduction of a closed expression may result in a "stuck expression" whose future behavior depends on and affects other portions of the program. Consider the following expression:

$$(\mathsf{ref}\ (\lambda\ x.x)).$$

It is a closed expression that does not diverge, is not a value, and cannot be reduced to a value without affecting the context. We refer to such results as *effects*.

[3] The decision to return the location of an update message to the program as the result value of the action is arbitrary. ML returns a neutral value (`unit`), Scheme's return value is undetermined though many implementations return the assigned value.

An effect is most easily understood as an interaction between a sub-expression and a central authority that administers the global resources of a program.[4] Examples of such resources are stores, heaps, file systems, the place for the final result of a program, and other input/output channels. Given an administrator, an effect can be viewed as a message to the central authority plus enough information to resume the suspended calculation. In the operational semantics, the "stuck expression", (ref $(\lambda\ x.x)$) in the running example, can play the role of the first half of the message; the evaluation context functions as the second part of the message. After extending the store, looking up a value, or modifying the contents of a location, the calculation process can continue. Technically, the operational semantics accomplishes this by filling the evaluation context with some datum, which also communicates information from the administrator to the suspended expression. By translating this operational semantics into a mathematical framework, we can design an extensible denotational semantics for a programming language.

3 Extensible Denotational Specifications

The denotational specification of a programming language consists of two parts. The first part defines the syntactic and semantic domains of the language. The domains are sub-domains of some universal domain, *e.g.*, $\mathcal{P}\omega$ [29], $\mathbf{T}^\omega$ [24], U [28], or $\mathbb{T}$ [17]. The syntactic domains specify the phrases of the language; the semantic domains contain denotations for the various kinds of syntactic phrases. The second part is a functional interpreter that maps elements of the syntactic domains to elements in semantic domains. It is defined in a universal[5] programming language like LAMBDA [29] or KL [17] and satisfies the law of compositionality, *i.e.*, the interpretation of a phrase is a function of the interpretations of its sub-phrases. To keep track of the denotations of free variables, interpreters are also parameterized over an environment argument. Algebraically speaking, the interpreter is (roughly) a homomorphism from syntax to semantics.

We illustrate the methodology of extensible denotational specifications by defining a semantics for Pure Scheme. We first extend this semantics with destructive reference cells and then with first-class continuation objects. Although Pure Scheme is a simple syntactic language with only one kind of phrase type, it is semantically rich and its extensions pose most of the problems that complicate traditional denotational language specifications. The first subsection presents the general schema of an extensible denotational specification for Pure Scheme and its extensions. The second, third, and fourth subsections contain the specifications for Pure Scheme, State Scheme, and Control Scheme, respectively. The latter two are literally enlargements of the semantics of Pure Scheme. The last subsection shows that the denotation of a phrase is stable with respect to all possible extensions.

[4] It is also possible to design an operational semantics that deals with such resources on a more local level [5, 11, 40].

[5] The language is universal relative to the chosen universal domain.

Semantic Domains

$$
\begin{array}{lll}
(\mathit{Values}) & \mathbb{V} = \Sigma_1(\mathbb{V},\mathbb{C}) & \text{(depends on language)} \\
(\mathit{Computations}) & \mathbb{C} = \mathtt{inV}(\mathbb{V}) \oplus \mathtt{inFX}((\mathbb{V} \longrightarrow \mathbb{C})_\bot \otimes \mathbb{A}) & \\
(\mathit{Actions}) & \mathbb{A} = \mathtt{inE}(\mathbb{E}) \oplus \Sigma_2(\mathbb{V},\mathbb{C}) & \text{(depends on language)} \\
(\mathit{Errors}) & \mathbb{E} = \{\bot, \mathtt{err}\} & \\
(\mathit{Resources}) & \mathbb{R} = \Pi(\mathbb{V},\mathbb{C},\ldots) & \text{(depends on language)}
\end{array}
$$

$$(\mathit{Environments}) \quad \mathbb{E}nv = \mathit{Var} \multimap \mathbb{V}$$

Semantic Functions

$$
\begin{array}{l}
\mathcal{P} : \ \mathit{Prog} \longrightarrow ((\mathbb{V} \oplus \mathbb{E}) \times \mathbb{R}) \\
\mathcal{M} : \mathit{Expr} \longrightarrow \mathbb{E}nv \longrightarrow \mathbb{C}
\end{array}
$$

$$\mathcal{P}[\![P]\!] = \mathtt{admin}\langle \mathcal{M}[\![P]\!]\bot, r_0\rangle$$

$$
\begin{aligned}
\mathcal{M}[\![\Omega]\!]\rho &= \mathtt{inV}(\bot) = \bot \\
\mathcal{M}[\![\mathsf{err}]\!]\rho &= \mathtt{inAC}(\mathtt{inE}(\mathtt{err})) \\
\mathcal{M}[\![e]\!]\rho &= \ldots \text{ depends on language } \ldots
\end{aligned}
$$

Auxiliary Semantic Functions

$$
\begin{array}{ll}
\mathtt{admin} : & \mathbb{C} \times \mathbb{R} \longrightarrow ((\mathbb{V} \oplus \mathbb{E}) \times \mathbb{R}) \\
\mathtt{handler} : & \mathbb{C} \longrightarrow (\mathbb{V} \longrightarrow \mathbb{C}) \longrightarrow \mathbb{C} \\
\mathtt{inAC} : & \mathbb{A} \longrightarrow \mathbb{C}
\end{array}
$$

$$
\begin{aligned}
\mathtt{admin}\langle \bot, r\rangle &= \bot \\
\mathtt{admin}\langle \mathtt{inV}(v), r\rangle &= \langle v, r\rangle \\
\mathtt{admin}\langle \mathtt{inFX}(k, \mathtt{inE}(\mathtt{err})), r\rangle &= \langle \mathtt{err}, r\rangle \\
\mathtt{admin}\langle \mathtt{inFX}(k, p), r\rangle &= \ldots \text{ depends on language } \ldots
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{handler}(\bot)f &= \bot \\
\mathtt{handler}(\mathtt{inV}(v))f &= f(v) \\
\mathtt{handler}(\mathtt{inFX}(k,p))f &= \mathtt{inFX}([\underline{\lambda}\, v : \mathbb{V}.\mathtt{handler}(k(v))f], p)
\end{aligned}
$$

$$\mathtt{inAC}(e) = \mathtt{inFX}(\mathtt{inV}, e)$$

Fig. 3. The Semantic Framework

3.1 The Semantic Framework: Extended Direct Semantics

Following the ideas of the previous section, the domain of denotations for phrases $\mathbb{C}$ consists of two disjoint pieces: the sub-domain of value denotations $\mathbb{V}$ and the sub-domain of effect messages (effects). The first part of Fig. 3 displays the general schema for the domain equations. In general, the domain of values is a sum of domains constructed from $\mathbb{V}$ and $\mathbb{C}$; we indicate this with the notation $\Sigma_1(\mathbb{V}, \mathbb{C})$ in Fig. 3. For a language like Pure Scheme, the domain of values contains numbers, boolean values, functions from values to computations, and bottom (the denotation of diverging expressions). For State Scheme, the domain of values also contains a domain of locations, which interpret reference cells.

The domain of effect messages is always the product of two domains. The first component of this product represents the evaluation context, the other specifies the action that the administrator has to perform. A straightforward representation of an evaluation context is a function from values to computations, which directly corresponds to its operational usage as a function from syntactic values to expressions. The action component $\mathbb{A}$ of the effect domain is a sum of domains built from $\mathbb{V}$ and $\mathbb{C}$; in Fig. 3 we use the expression $\Sigma_2(\mathbb{V}, \mathbb{C})$ for this purpose. It contains one summand for each kind of action that the administrator has to perform. For the trivial base language, the basic domain of actions contains only an error action (since we believe that signaling an error is the most basic effect). For Pure Scheme, no additional action is required. For State Scheme, there are three additional actions: the allocation of a reference cell filled with some value, the dereferencing of a cell, and the modification of a reference cell.

In general, an interpreter maps phrases $\times$ environments to computations. Environments are needed to interpret the meaning of bound variables. Given the schematic domain definitions, an interpreter can deal with two simple programming constructs: Ω, which represents divergence, and **err**, which signals an error. Other kinds of phrases require an extension of the interpreter.

Sub-phrases of complex phrases are evaluated via recursive calls to the interpreter. Since the result of such a recursive call is a computation, it is necessary to inspect the tag of the result. If it is a plain value, the value component can be consumed locally. If it is an effect, however, it must be propagated to the central administrator, which will interpret its meaning. To deal with this situation uniformly, we introduce a function `handler` that maps a computation and the consumer of its eventual value to computations. The consumer is a function from values to a computation.[6]

The function `handler` performs a simple tag check: if the first argument is in the value sub-domain, it applies its second argument to the value. Otherwise, it creates a new effect message that accounts for the additional consumer with a modified handle. Thus an effect message is propagated from `handler` to `handler` until it eventually reaches the central administrator. The administrator, `admin`, is a function of two arguments: a computation and an (open-ended) list of resources

[6] The function `handler` is roughly a composition combinator for functions from values to computations. Let $f \in \mathbb{V} \longrightarrow \mathbb{V}$, $g \in \mathbb{V} \longrightarrow \mathbb{C}$, and $x \in \mathbb{V}$. Then,

$$\mathtt{handler}(\mathtt{inV}(f(x)))(g) = g(f(x)).$$

On a superficial level `handler` is related to the composition transformation of a monad (cmp. Section 5), but it only partially satisfies the monad laws [E. Moggi; personal communication, August 1989].

Semantic Domains

$$\mathbb{V} = \texttt{inN}(\mathbb{N}_\perp) \oplus \texttt{inB}(\mathbb{T}_\perp) \oplus \texttt{inP}((\mathbb{V} \longrightarrow_s \mathbb{C})_\perp)$$
$$\mathbb{A} = \texttt{inE}(\mathbb{E})$$

Semantic Functions

$$
\begin{aligned}
\mathcal{M}[\![\ulcorner n \urcorner]\!] &= \texttt{inV}(\texttt{inN}(n)) \\
\mathcal{M}[\![\mathsf{true}]\!] &= \texttt{inV}(\texttt{inB}(\texttt{true})) \\
\mathcal{M}[\![\mathsf{false}]\!] &= \texttt{inV}(\texttt{inB}(\texttt{false})) \\
\mathcal{M}[\![\mathsf{add1}]\!]\rho &= \texttt{inV}(\texttt{inP}(\underline{\lambda}\, m : \mathbb{V}. \\
&\qquad\quad \texttt{case}\ m\ \texttt{of} \\
&\qquad\qquad [\texttt{inN}(n) \Rightarrow \texttt{inV}(\texttt{inN}(n+1))] \\
&\qquad\qquad [m \qquad \Rightarrow \texttt{inAC}(\mathrm{err})])) \\
\ldots \quad & \ldots \\
\mathcal{M}[\![x]\!]\rho &= \texttt{inV}(\rho(x)) \\
\mathcal{M}[\![(\lambda\ x.\ e)]\!]\rho &= \texttt{inV}(\texttt{inP}(\underline{\lambda}\, d : \mathbb{V}.\mathcal{M}[\![e]\!]\rho[x/d])) \\
\mathcal{M}[\![(e_1\ e_2)]\!]\rho &= \texttt{handler}(\mathcal{M}[\![e_1]\!]\rho) \\
&\qquad (\underline{\lambda}\, f : \mathbb{V}.\texttt{handler}(\mathcal{M}[\![e_2]\!]\rho) \\
&\qquad\quad (\underline{\lambda}\, a : \mathbb{V}.\ \texttt{case}\ f\ \texttt{of} \\
&\qquad\qquad\qquad [\texttt{inP}(g) \Rightarrow g(a)] \\
&\qquad\qquad\qquad [g \qquad \Rightarrow \texttt{inAC}(\mathrm{err})])) \\
\mathcal{M}[\![(\mathsf{if}\ e_1\ e_2\ e_3)]\!]\rho &= \texttt{handler}(\mathcal{M}[\![e_1]\!]\rho) \\
&\qquad (\underline{\lambda}\, t : \mathbb{V}. \\
&\qquad\quad \texttt{case}\ t\ \texttt{of} \\
&\qquad\qquad [\texttt{inB}(\texttt{true}) \ \ \Rightarrow \mathcal{M}[\![e_2]\!]\rho] \\
&\qquad\qquad [\texttt{inB}(\texttt{false}) \Rightarrow \mathcal{M}[\![e_3]\!]\rho] \\
&\qquad\qquad [f \qquad\qquad \Rightarrow \texttt{inAC}(\mathrm{err})])
\end{aligned}
$$

Fig. 4. An Extended Direct Semantics for Pure Scheme

$(\Pi(\mathbb{V}, \mathbb{C}, \ldots))$. If the computation is a value, the value and the resources are returned as the result of the program. A computation that requires an "error" action yields the error paired with the current resources. The meaning of a program is defined as the composition of the interpreter and the administrator.

Fig. 3 presents the schema of a general language definition. All of the following definitions fit into this schema. Each language fragment demands modifications of this outline in five places: the sum of value sub-domains (Σ_1), the sum of actions (Σ_2), the list of resources, the clauses of the meaning function $\mathcal{M}$, and the clauses of the administrator function `admin`. Everything else remains the same, *e.g.*, the type of the interpreter, the handler function, the top-level meaning function, *etc.* Since the interpreter maps phrases and environments to their meanings, we refer to this style of semantics as *extended direct semantics*.

In the absence of computational effects and errors, the schema actually reduces to a schema for direct semantics.

3.2 Pure Scheme

We first illustrate the abstract ideas of the first subsection with a simple language of arithmetic:

$$M ::= \ulcorner n \urcorner \mid (\texttt{add1}\ M) \mid (\texttt{sub1}\ M) \mid \ldots, \quad n \in \mathbb{N}.$$

This language's value domain only contains the integers and bottom. The effect messages only exist to propagate errors. Thus, the domain equations are:

$$\begin{aligned} \mathbb{V} &= \texttt{inN}(\mathbb{N}_\perp) \\ \mathbb{A} &= \texttt{inE}(\mathbb{E}) \end{aligned}$$

and the meaning function maps numerals to integers:

$$\mathcal{M}[\![\ulcorner n \urcorner]\!]\rho = \texttt{inV}(\texttt{inN}(n)).$$

The interpretation of an `add1`-expression is the result of interpreting its subexpression and handling the result with a function that outputs the successor for all numeric inputs:

$$\begin{aligned} \mathcal{M}[\![(\texttt{add1}\ e)]\!]\rho = \texttt{handler}&(\mathcal{M}[\![e]\!]\rho) \\ &(\underline{\lambda}\, m : \mathbb{V}.\, \texttt{case}\ m\ \texttt{of} \\ &\qquad [\texttt{inN}(n) \Rightarrow \texttt{inV}(\texttt{inN}(n+1))] \\ &\qquad [m \qquad \Rightarrow \texttt{inAC}(\texttt{inE}(\texttt{err}))]) \end{aligned}$$

For potential non-numerical inputs the consumer function outputs error. Other complex expressions are interpreted in the same fashion.

To extend the language of arithmetic to full Pure Scheme, we add the syntax of Fig. 1 and another summand to the domain of values: the domain of procedure denotations. Pure Scheme procedures map non-bottom values to computations. Thus, the new domain of values is the domain of strict functions from values to computations:

$$\mathbb{V} = \texttt{inN}(\mathbb{N}_\perp) \oplus \texttt{inP}((\mathbb{V} \longrightarrow_s \mathbb{C})_\perp).$$

No new effects are required for full Pure Scheme.

The semantic function (see Fig. 4) maps variables to values and procedures to functions in the usual way. For an application $(M\ N)$ the interpreter first recursively determines the meaning of M and coerces it to a value f using `handler`. The corresponding consumer then determines the value a of N in a similar manner. The second consumer function applies f to a if f is a function and raises an error signal otherwise.

Booleans and if-expressions require extensions of the arithmetic language that are analogous to those described above. Fig. 4 contains the full specification of Pure Scheme.

Semantic Domains

$$\begin{array}{lrl}
 & \mathbb{V} &= \mathtt{inN}(\mathbb{N}_\perp) \oplus \mathtt{inB}(\mathbb{T}_\perp) \oplus \mathtt{inP}((\mathbb{V} \longrightarrow \mathbb{C})_\perp) \oplus \mathtt{inL}(\mathbb{L}_\perp) \\
 & \mathbb{A} &= \mathtt{inE}(\mathbb{E}) \oplus \mathtt{inRef}(\mathbb{V}) \oplus \mathtt{inDer}(\mathbb{L}_\perp) \oplus \mathtt{inSet}(\mathbb{L}_\perp \otimes \mathbb{V}) \\
 & \mathbb{R} &= \mathbb{S}to \\
(\mathit{Stores}) & \mathbb{S}to &= \mathbb{L}_\perp \multimap \mathbb{V} \\
(\mathit{Locations}) & \mathbb{L}_\perp &: \text{a flat domain, usually isomorphic to } \mathbb{N}_\perp
\end{array}$$

Semantic Function

$$\begin{aligned}
\mathcal{M}[\![(\mathsf{ref}\ e)]\!]\rho &= \mathtt{handler}(\mathcal{M}[\![e]\!]\rho)(\underline{\lambda}\, v : \mathbb{V}.\mathtt{inAC}(\mathtt{inRef}(v))) \\
\mathcal{M}[\![(e_1 \ := \ e_2)]\!]\rho &= \mathtt{handler}(\mathcal{M}[\![e_1]\!]\rho) \\
&\qquad (\underline{\lambda}\, l : \mathbb{V}.\mathtt{handler}(\mathcal{M}[\![e_2]\!]\rho) \\
&\qquad\quad (\underline{\lambda}\, a : \mathbb{V}.\ \mathtt{case}\ l\ \mathtt{of} \\
&\qquad\qquad\qquad [\mathtt{inL}(l') \Rightarrow \mathtt{inAC}(\mathtt{inSet}(l', a))] \\
&\qquad\qquad\qquad [g \qquad\ \Rightarrow \mathtt{inAC}(\mathtt{inE}(\mathtt{err}))])) \\
\mathcal{M}[\![(!\ \ e)]\!]\rho &= \mathtt{handler}(\mathcal{M}[\![e]\!]\rho) \\
&\qquad (\underline{\lambda}\, l : \mathbb{V}.\ \mathtt{case}\ l\ \mathtt{of} \\
&\qquad\qquad [\mathtt{inL}(l') \Rightarrow \mathtt{inAC}(\mathtt{inDer}(l'))] \\
&\qquad\qquad [g \qquad\ \Rightarrow \mathtt{inAC}(\mathtt{inE}(\mathtt{err}))]))
\end{aligned}$$

Administrator Function

$$\begin{aligned}
\mathtt{admin}\langle \mathtt{inFX}(k, \mathtt{inRef}(v)), r\rangle &= \mathtt{admin}\langle k(\mathtt{inL}(\mathtt{new}(r))), \mathtt{extend}(r, \mathtt{new}(r), v)\rangle \\
\mathtt{admin}\langle \mathtt{inFX}(k, \mathtt{inSet}(l, v)), r\rangle &= \mathtt{admin}\langle k(l), \mathtt{extend}(r, l, v)\rangle \\
\mathtt{admin}\langle \mathtt{inFX}(k, \mathtt{inDer}(l)), r\rangle &= \mathtt{admin}\langle k(\mathtt{lookup}(r, l))), r\rangle
\end{aligned}$$

Fig. 5. An Extended Direct Semantics for State Scheme

3.3 State Scheme

The addition of reference cells to Pure Scheme requires the syntax of Fig.2 and a new class of values for the interpretation of reference cells. Following tradition, we call these values *locations*. Locations are bound in the *store* ($\mathbb{S}to$), which is a part of the global resource pool. Like all other resources, the store is managed by the administrator. Thus, the creation of a cell requires a message to the central administrator. It allocates a new location and associates it with the value of the allocation message. The result of the allocation is the new location l, that is, the administrator uses the handle to return l to the program. Similarly, the acts of dereferencing and modifying the contents of a cell also require an exchange of messages between the program phrase and the administrator.

Translated into the language of our denotational framework, the definition of State Scheme requires a new summand for the domain of values, the domain of locations, and three new summands for the domain of actions, the second component of effect messages:

1. $\texttt{inRef}(\mathbb{V})$, which represents an allocation message;
2. $\texttt{inSet}(\mathbb{L}_\perp, \mathbb{V})$, which represents an update message; and
3. $\texttt{inDer}(\mathbb{L}_\perp)$, which represents a dereferencing message.

Moreover, the list of resources (thus far empty) needs to be extended with a store. See the first part of Fig. 5 for the detailed specification of the extended domains.

The interpretation of the three new syntactic forms is straightforward. For each form, the interpreter recursively determines the computation of the subexpressions with a handler that eventually issues an effect message concerning the allocation, modification, or dereferencing of a location. The extended administrator accepts precisely these three new effect messages. If the effect message is an allocation message, it proceeds as described at the outset of this section. The other two effect messages are treated in a similar vein. The second part of Fig. 5 presents the new clauses for the interpreter and administrator functions; the clauses in $\mathcal{M}$ and `admin` for Pure Scheme syntax and effect messages remain the same!

Syntax

$$e ::= (\mathsf{catch}\ x\ e) \mid (\mathsf{throw}\ x\ e)$$

Semantic Domains

$$\begin{aligned}\mathbb{V} &= \texttt{inN}(\mathbb{N}_\perp) \oplus \texttt{inB}(\mathbb{T}_\perp) \oplus \texttt{inP}((\mathbb{V} \longrightarrow \mathbb{C})_\perp) \oplus \texttt{inK}((\mathbb{V} \longrightarrow \mathbb{C})_\perp)\\ \mathbb{A} &= \texttt{inE}(\mathbb{E}) \oplus \texttt{inCon}(((\mathbb{V} \longrightarrow \mathbb{C}) \longrightarrow \mathbb{C})_\perp)\end{aligned}$$

Semantic Function

$$\begin{aligned}\mathcal{M}[\![(\mathsf{catch}\ x\ e)]\!]\rho &= \texttt{inAC}(\texttt{inCon}(\underline{\lambda}\, k : \mathbb{V} \longrightarrow \mathbb{C}.\texttt{handler}(\mathcal{M}[\![e]\!]\rho[x/\texttt{inK}(k)])\ k))\\ \mathcal{M}[\![(\mathsf{throw}\ x\ e)]\!]\rho &= \texttt{handler}(\mathcal{M}[\![e]\!]\rho)\\ &\qquad (\underline{\lambda}\, v : \mathbb{V}.\,\texttt{case}\ \rho(x)\ \texttt{of}\\ &\qquad\qquad [\texttt{inK}(k) \texttt{ => } \texttt{inAC}(\texttt{inCon}(\underline{\lambda}\, k' : \mathbb{V} \longrightarrow \mathbb{C}.k(v)))]\\ &\qquad\qquad [k \qquad \texttt{ => } \texttt{inAC}(\texttt{inE}(\mathsf{err}))])\end{aligned}$$

Administrator Function

$$\texttt{admin}\langle \texttt{inFX}(k, \texttt{inCon}(f)), r\rangle = \texttt{admin}\langle f(k), r\rangle$$

Fig. 6. An Extended Direct Semantics for Control Scheme

3.4 Control Scheme

Standard Scheme and SML of New Jersey provide constructs for programming with first-class continuation objects. A continuation object is an abstraction of the control state of a program. Gaining access to this control abstraction is often called *catching* a continuation. Using this abstraction, or *throwing* to a continuation in the jargon of Scheme and SML, a program can discard its current continuation and transmit a value to a previously caught continuation object. Control Scheme employs the syntax (catch x e) for catching the continuation and binding it to x and the syntax (throw x e) for throwing the value of e to the continuation bound to x [37].[7] Modeling first-class continuation objects according to the traditional method requires the use of a continuation-semantics [36, 1, 15, 27, 35]. A continuation semantics parameterizes the denotation of *every* phrase type over a *continuation*, which is a function that maps intermediate results to final answers.

In the framework of extended direct semantics the output of the final answer is under the control of the central administrator. Sending an effect message from any place in the program to the administrator by definition constructs a handle, which is a functional abstraction of the rest of the computation. Hence continuation objects are easily accommodated in the framework. A catch-expression denotes an effect message that contains the recipient of the continuation, which is roughly the body of the catch-expression (e) parameterized over the name for the continuation object (x). The administrator applies the recipient to the handle relinquishing control over the rest of the computation. A throw-expression denotes a similar package but the continuation recipient of the package *ignores* the continuation it receives from the administrator and uses the object bound to the throw variable instead.

With regards to domains, the extension of Pure Scheme to Control Scheme requires the addition of one value summand, the domain of continuations, and one effect, a control message. The latter contains the recipient of the continuation, which is a function from handles to computations. The extension of the meaning function $\mathcal{M}$ and the administrator **admin** implement the above schema. The details are given in Fig. 6.

Extending State Scheme to Core Scheme, a Scheme-like language that contains reference cells as well as catch and throw, is similarly easy. It is necessary to extend the domains of State Scheme as follows:

$$\begin{aligned}
\mathbb{V} &= \mathtt{inN}(\mathbb{N}_\perp) \oplus \mathtt{inB}(\mathbb{T}_\perp) \oplus \mathtt{inP}((\mathbb{V} \longrightarrow \mathbb{C})_\perp) \oplus \mathtt{inL}(\mathbb{L}_\perp) \oplus \mathtt{inK}((\mathbb{V} \longrightarrow \mathbb{C})_\perp) \\
\mathbb{A} &= \mathtt{inE}(\mathbb{E}) \\
&\quad \oplus \mathtt{inRef}(\mathbb{V}) \oplus \mathtt{inDer}(\mathbb{L}_\perp) \oplus \mathtt{inSet}(\mathbb{L}_\perp \otimes \mathbb{V}) \\
&\quad \oplus \mathtt{inCon}(((\mathbb{V} \longrightarrow \mathbb{C}) \longrightarrow \mathbb{C})_\perp)
\end{aligned}$$

The new clauses for the meaning function $\mathcal{M}$ and the administrator **admin** remain the same.

The merger of State Scheme and Control Scheme is symmetric. Merging the reference domains into the domains of Control Scheme yields domains that are isomorphic to those of Core Scheme. The resulting meaning function and the

[7] With first-class continuations objects it is easy to simulate many other control constructs and patterns, *e.g.*, loop exits, blind and non-blind backtracking [14], coroutines [16], light-weight threads [39], and time-preempted computations [8].

administrator would be identical to the ones we just described. In general, the addition of new orthogonal linguistic constructs is as straightforward as the extension of Pure Scheme to Control Scheme or of State Scheme to Core Scheme. It requires the addition of summands in the domain of values, the domain of effects, the interpreter and the administrator function. No modification of the previous specifications is needed. We therefore believe that the attribute "extensible" is fully justified for our new style of denotational specifications. The following subsection provides the basis for a more formal justification of this attribute.

3.5 Stable Denotations

Recall that the semantic domains for an extended direct definition have the form

$$\begin{array}{ll} (\mathit{Values}) & \mathbb{V} = \Sigma_1(\mathbb{V}, \mathbb{C}) \\ (\mathit{Actions}) & \mathbb{C} = \mathtt{inE}(\mathbb{E}) \oplus \Sigma_2(\mathbb{V}, \mathbb{C}) \\ (\mathit{Resources}) & \mathbb{R} = \Pi(\mathbb{V}, \mathbb{C}, \ldots) \end{array}$$

where $\Sigma_1(\mathbb{V}, \mathbb{C})$ and $\Sigma_2(\mathbb{V}, \mathbb{C})$ are disjoint sums of terms constructed using domain operations and subsets of $\mathbb{V}$ and $\mathbb{C}$. An orthogonal *extension* of a semantic definition written in this framework consists of six parts:

- a set of syntactic constructors defining the syntax of the extension;
- a collection of new values designated by new terms in the sum $\Sigma_1(\mathbb{V}, \mathbb{C})$;
- a collection of new actions designated by new terms in the sum $\Sigma_2(\mathbb{V}, \mathbb{C})$;
- a new clause in the meaning function $\mathcal{M}$ for each new syntactic constructor;
- a new component of the list of resources;
- a new clause in the *admin* function for each new kind of action.

Any of these components except the first can be omitted.

By the form of the domain specifications, the semantic domain for a language extension is a superset of the semantic domain for the original language. The extension process adds new elements to the semantic domain. More precisely, there are two pairs of functions that relate the original domains $\mathbb{V}$, $\mathbb{C}$ to corresponding enlarged domains $\mathbb{V}'$, $\mathbb{C}'$. First, there is a pair of functions $\Phi_{\mathbb{V} \longrightarrow \mathbb{V}'} : \mathbb{V} \longrightarrow \mathbb{V}'$ and $\Phi_{\mathbb{C} \longrightarrow \mathbb{C}'} : \mathbb{C} \longrightarrow \mathbb{C}'$, that inject values and computations from the smaller domains to the values and computations in the larger domains. These injections are actually identities (interpreted on the universal domain). Second, there is a pair of functions $\Psi_{\mathbb{V}' \longrightarrow \mathbb{V}} : \mathbb{V}' \longrightarrow \mathbb{V}$ and $\Psi_{\mathbb{C}' \longrightarrow \mathbb{C}} : \mathbb{C}' \longrightarrow \mathbb{C}$, that project the values and computations from the larger domains onto values and computations of the smaller domains. These projections map any components of values and computations with new tags to $\perp$ and leave all other components unchanged. In summary,

$$\Phi_{\mathbb{V} \longrightarrow \mathbb{V}'} \circ \Psi_{\mathbb{V}' \longrightarrow \mathbb{V}} = \Psi_{\mathbb{V}' \longrightarrow \mathbb{V}} \sqsubseteq I_{\mathbb{V}'} \text{ and } \Phi_{\mathbb{V} \longrightarrow \mathbb{V}'} = I_V$$

as well as

$$\Phi_{\mathbb{C} \longrightarrow \mathbb{C}'} \circ \Psi_{\mathbb{C}' \longrightarrow \mathbb{C}} = \Psi_{\mathbb{C}' \longrightarrow \mathbb{C}} \sqsubseteq I_{C'} \text{ and } \Phi_{\mathbb{C} \longrightarrow \mathbb{C}'} = I_C.$$

```
(define-const-structure (Loop))
(define-const-structure (Err))

(define baseM
  (module (Interpreter Admin Resources)

    (define Interpreter
      (lambda (exp env)
        (match exp
          [($ Loop) (printf "looping~n") (Interpreter exp env)]
          [($ Err) (inExc 'error)]
          [exp (error 'Interpreter "can't interpret ~s" exp)])))

    (define Admin
      (lambda (comp resources)
        (match comp
          [($ Val v) (cons v resources)]
          [($ FX continuation ($ Exc n))
           (error 'Admin "exception raise:~~s~n" n)]
          [else (error 'Admin "bad effect message: ~s" comp)])))

    (define Resources '()) ))
```

Fig. 7. Module for Base Language

where I_X is the identity function on the domain X.

Informally speaking, this property asserts that in the framework of extensible semantics, the addition of a programming construct corresponds to the addition of a new "dimension" to the space of meanings. This addition satisfies the following property. Let $\mathcal{M}$ be the meaning function for the core language, and let $\mathcal{M}'$ be the meaning function of an *arbitrary* extension. If e is an expression in the core language, ρ an environment in $Var \multimap \mathbb{V}$, and ρ' in $Var \multimap \mathbb{V}'$, then

$$\mathcal{M}[\![e]\!]\rho = \Psi_{\mathbb{C}' \longrightarrow \mathbb{C}}(\mathcal{M}'[\![e]\!](\Phi_{\mathbb{V} \longrightarrow \mathbb{V}'} \circ \rho))$$

and

$$\Phi_{\mathbb{C} \longrightarrow \mathbb{C}'}(\mathcal{M}[\![e]\!](\Psi_{\mathbb{V}' \longrightarrow \mathbb{V}} \circ \rho')) \sqsubseteq \mathcal{M}'[\![e]\!]\rho',$$

The reader may want to contrast this general statement with the numerous papers on the relationship between direct and continuation semantics [13, 18, 26, 30, 34].

4 Composing Interpreters

Given an extended direct semantics of Core Scheme, we should be able to implement an interpreter for Core Scheme in a modular fashion. One obvious possibility is to concatenate the domain summands and the clauses of the functions $\mathcal{M}$

```
;; Syntax: e ::= (var x) | (lam x e) | (app e e)

(define-const-structure (var x))
(define-const-structure (lam x M))
(define-const-structure (app M N))

(define CBVM
  (lambda (language)
    (import ([language (InterpPrev Interpreter) Admin Resources])
      (module (Interpreter Admin Resources)

        ;; Semantics: V = Proc(V → C)

        (define-const-structure (Proc closure))
        (define inProc make-Proc)

        (define Interpreter
          (lambda (exp env)
            (match exp
              [($ var x)
               (inVal (env x))]
              [($ lam x exp)
               (inVal (inProc (lambda (v) (InterpTop exp (Extend env x v)))))]
              [($ app exp1 exp2)
               (Handler (InterpTop exp1 env)
                 (lambda (f)
                   (Handler (InterpTop exp2 env)
                     (lambda (a)
                       (match f
                         [($ Proc g) (g a)]
                         [_ (inExc 'error)])))))]
              [exp (InterpPrev exp env)])))))))
```

Fig. 8. Module for Call-by-Value Lambda Notation

and `admin`. This method yields monolithic interpreters defined from independent pieces, but the method for gluing them together is based on program *text* and is thus beyond the scope of most programming languages.

One alternative is to use an object-oriented programming language in the spirit of CLOS [6]. Each language fragment would extend the list of resources, and the interpreter and administrator methods of an existing language. In addition a language fragment would also introduce new local data constructors.

Here we present another alternative based on the module system of Rice

Scheme.[8] It is inspired by Steele's upcoming paper on composing monads [33].[9] Fig. 7 through Fig. 14 in this section and Section B contain the complete code.

A language is implemented as a module that exports three bindings:

1. an interpreter function;
2. an administrator function; and
3. a list of resources.

The basic language module in Fig. 7 interprets two constants, *Err* and *Loop*, with obvious meanings, implements the administrator of Fig. 3, and exports an empty list of resources.

Each *language fragment* consists of a file with two parts. The first part is a set of (abstract) syntax constructors. The syntax constructors are global for ease of access for the interactive user of the language. The second part is a *language transformer*, a function that maps a language implementation to a richer language implementation. The interpreter functions in language transformers include a clause for each new syntactic facility that the language fragment introduces. When the local interpreter function interprets a sub-expression, it must call the interpreter for the *complete* language since sub-expressions may contain facilities from a language fragment that is yet to be added. Similarly, the local administrator of a language transformer must deal with all new resources and effects that the transformer introduces. But again, it must call the administrator for the *complete* language after the action is performed such that all possible computations can be administered.

For a typical language fragment, consider Fig. 8, which defines the interpretation of the call-by-value λ-notation of Pure Scheme. The signature of the language transformer is

$$
\begin{array}{l}
(\mathit{CBVM} : ((\textbf{module} \\
\qquad (\mathit{Resources}\ \mathbb{R}) \\
\qquad (\mathit{Admin}\ (\mathbb{C}_x \times\ \mathbb{R} \longrightarrow \mathbb{V} \times \mathbb{R})) \\
\qquad (\mathit{Interpreter}\ (\mathit{exps}\ \times \mathbb{E}\mathit{nv} \longrightarrow \mathbb{C}))) \\
\quad \longrightarrow \\
\quad (\textbf{module} \\
\qquad (\mathit{Resources}\ \mathbb{R}) \\
\qquad (\mathit{Admin}\ (\mathbb{C}_x \times\ \mathbb{R} \longrightarrow \mathbb{V} \times \mathbb{R})) \\
\qquad (\mathit{Interpreter}\ (\mathit{exps}\ \times \mathbb{E}\mathit{nv} \longrightarrow \mathbb{C})))))
\end{array}
$$

where $\mathbb{C}_x = \mathbb{V} \oplus ((\mathbb{V} \longrightarrow \mathbb{C}) \otimes \mathbb{E})$. That is, the administrator only handles the basic computations, returning a value and an error. Once the resulting language module is fed into the language transformer for the control language, this will

[8] Rice Scheme is an extension of Chez Scheme [7] that includes modules and data constructor definitions for immutable structures. It also comes with a soft type system [41] that infers types and eliminates type checks where possible. Our implementation of Core Scheme type-checks statically and is thus independent of a run-time tag checking.

[9] See the next section for a brief comparison with his technique.

```
;; Syntax: e ::= (catch x e) | (throw x e)

(define-const-structure (catch x M))
(define-const-structure (throw x N))

(define controlM
  (lambda (language)
    (import ([language (InterpPrev Interpreter) (AdminPrev Admin) Resources])
      (module (Interpreter Admin Resources)

        ;; Semantics: V = K(V ⟶ C); C = FX(V ⟶ C, Con(V ⟶ C))

        (define-const-structure (K v->c))
        (define inK make-K)

        (define-const-structure (Con v->c))
        (define inCon (lambda (x) (inCC (make-Con x))))

        (define Interpreter
          (lambda (exp env)
            (match exp
              [($ catch x exp)
               (inCon
                 (lambda (k)
                   (Handler (InterpTop exp (Extend env x (inK k))) k)))]
              [($ throw x exp)
               (Handler (InterpTop exp env)
                 (lambda (v)
                   (match (env x)
                     [($ K k) (inCon (lambda (kp) (k v)))]
                     [_ (inExc 'error)])))]
              [exp (InterpPrev exp env)])))

        (define Admin
          (lambda (comp resources)
            (match comp
              [($ FX continuation ($ Con r))
               (AdminTop (r continuation) resources)]
              [comp (AdminPrev comp resources)])))))))
```

Fig. 9. Module for Catch and Throw

change. The signature for the latter transformer clearly shows this transformation:

```
(controlM : ((module
                (Resources ℝ)
                (Admin (ℂx × ℝ ⟶ 𝕍 × ℝ))
                (Interpreter (exps × 𝔼nv ⟶ ℂ)))
             ⟶
             (module
                (Resources ℝ)
                (Admin (ℂc × ℝ ⟶ 𝕍 × ℝ))
                (Interpreter (exps × 𝔼nv ⟶ ℂ)))))
```

(where $\mathbb{C}_c = \mathbb{V} \oplus ((\mathbb{V} \longrightarrow \mathbb{C}) \otimes (\mathbb{E} \oplus ((\mathbb{V} \longrightarrow \mathbb{C}) \longrightarrow \mathbb{C})))$. Now the administrator also handles control messages, according to the specification in Fig. 6. For the rest of the language implementation, including the definition of the value constructor, of *Handler* and of environments, we refer to Section B in the appendix.

Finally, the implementation of the complete language consists of four definitions:

1. the module for the complete language, defined by composing language fragments functionally and applying them to the base module;
2. a global definition for the interpreter of the complete language;
3. a global definition for the administrator of the complete set of resources; and
4. a meaning function for programs.

Here is the definition for Core Scheme:

```
(define CoreScheme (storeM (controlM (CBVM (arithmM baseM)))))

;; The global interpreter:
(define InterpTop (import ([CoreScheme Interpreter]) Interpreter))

;; The complete administrator
(define AdminTop (import ([CoreScheme Admin]) Admin))

;; The Program Meaning Function:
(define Program
  (import ([CoreScheme Resources])
    (lambda (P)
      (AdminTop (InterpTop P Empty) Resources))))
```

The order of function composition in the definition of *CoreScheme* is irrelevant; omitting a language fragment in the chain defines a sublanguage of Core Scheme without the respective constructs.

5 Related Work

Monads. Moggi's [21, 22, 20] recent work on formulating notions of computations as monads, popularized by Wadler [38], is partly motivated by the lack of modularity in denotational specifications and a resulting lack of understanding of the logical relationships between computations. Roughly speaking, a monad corresponds to a triple consisting of a domain constructor T that maps a domain of values to a domain of computations, an injection η from values to computations, and a combination transformation μ for computations. Different monads express different styles of denotational specifications and thus computations. Some typical monads are the store-passing monad, the continuation-passing monad, and the monad for non-deterministic calculations.

Paremeterizing denotational direct models over monads leads to more flexible language specifications. The parameterized domain equation of a "monadic semantics" is

$$\mathbb{V} = \mathbb{N}_\perp \oplus (\mathbb{V} \longrightarrow_s T(\mathbb{V}));$$

the meaning assignment for numerals maps a numeral to its injection into the computation domain:

$$\mathcal{M}[\![\ulcorner n \urcorner]\!]\rho = \eta(n).$$

Now, if $T(\mathbb{V}) = (\mathbb{V} \longrightarrow \mathbb{A}) \longrightarrow \mathbb{A}$ for some fixed domain $\mathbb{A}$ and

$$\eta(n) = \underline{\lambda}\, k : \mathbb{V} \longrightarrow \mathbb{A}.k(n),$$

then the semantics is a continuation semantics. If $T(\mathbb{V}) = \mathbb{S}to \longrightarrow (\mathbb{V} \otimes \mathbb{S}to)$ and

$$\eta(n) = \underline{\lambda}\, \sigma : \mathbb{S}to.\langle n, \sigma \rangle,$$

then instantiating the semantics yields a store-passing semantics. In general, given a direct semantics parameterized over a monad, it is easily possible to move to a semantics based on a different style, but as a result, a numeral is assigned different domain elements—depending on the chosen monad.

Unfortunately, the combination of monads poses severe difficulties. Since monads cannot be composed directly, Moggi [20:ch.4] introduced the notion of a monad constructor, which are roughly monad transformers and can be combined. Combining the same two monad constructors in a different order, however, often yields different results. A typical example is the combination of an exception monad with a store monad. One combination undoes side-effects in the process of raising an exception, the other one does not.

To overcome the problems associated with combining monads, Steele [33] introduces the idea of a *pseudo-monad*, a relaxation of Moggi's notion. Based on the functional composition of pseudo-monads, Steele shows how to build interpreters for languages from interpreters for language fragments. Section 4 was inspired by his approach. But pseudo-monads have the same problems as monads: denotations are still not stable and combinations are not symmetric. At this point it is not clear what the trade-offs between the monad approach and our approach are.

Vienna School of Denotational Semantics. The extended direct semantics for Control Scheme is related to the approach of the Vienna School of Denotational Semantics [3, 4, 15:52–55] to modeling jumps and labels, a method that was partially rediscovered by Allison [2]. Indeed, the treatment of labels and gotos in block-structured languages according to this school of thought can be seen as an instance of our technique. In their framework, a statement denotes a state transformer that either returns a new state or a syntactic label (combined with a state). Returning a state corresponds to returning a value; returning a label is an effect message. The responsibility of propagating effects rests with the composition function for statements and other "glue" functions. When a label reaches the top-level, an exception handler that corresponds to our administrator effects the appropriate jump. Blikle and Tarleki [4] showed that by using the VDM technique, denotational specifications can use sets instead of domains and partial functions, instead of continuous ones. The Vienna School has not applied this technique to the specification of other imperative language constructs [D. Bjørner; letter, June 5, 1990].

6 Conclusions

Extended direct semantics is a new method for formulating denotational specifications of programming languages. Unlike the conventional approach, a language specification based on extended direct semantics is easily extensible with orthogonal language constructs. Most importantly, the extensions preserve the denotations of phrases in the core language.

Even though our illustration concentrates on extensions of functional languages, imperative languages can be specified in a similar fashion. A denotational model of an Algol-like language would be easy to construct. Indeed, the model may be closer to an implementation given that the store is a central resource rather than a first-class object that is passed around.

A more comprehensive comparison of the two approaches to denotational specifications will have to address several questions. First, we should investigate whether extended direct models characterize languages as precisely as the alternative continuation and store semantics, that is, we should check whether the new models have different full abstraction properties.[10] Second, we need to understand what implementation advantages each approach offers. Finally, we should study the design of modular logics for languages since the raison d'être for a denotational semantics is its usefulness in reasoning about programs.

Acknowledgements. The authors thank Mitch Wand for a helpful discussion about the stability of denotations in extended direct models at POPL'89. Andrew Wright's Soft Type System was an important tool for the verification of type properties of language transformers. We are also grateful to Dan Friedman, Brian

[10] Since we wrote the original report on extended direct semantics, Dorai Sitaram and the second author have explored this question for one special case (continuations) and have published a report on the results [31].

Milnes, Amr Sabry, and Steve Weeks for comments on an early draft of the paper. The Scheme code was type-set with Dorai Sitaram's SLaTeX.

References

1. ALLISON, L. *A Practical Introduction to Denotational Semantics.* Cambridge Computer Science Texts **23**. Cambridge University Press, Cambridge, England. 1986.
2. ALLISON, L. Direct semantics and exceptions define jumps and coroutines. *Information Processing Letters* **31**, 1989, 327–330.
3. BJØRNER, D. AND C. JONES. *Formal Specification and Software Development.* Prentice Hall International, 1982.
4. BLIKLE, A. AND A. TARLECKI. Naive denotational semantics. In *Proc. IFIP 9th World Computer Congress: Information Processing 83*, North-Holland, Amsterdam, 1983, 345-355.
5. CRANK, E. AND M. FELLEISEN. Parameter-passing and the lambda-calculus. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, 1991, 233–245.
6. DEMICHIEL, L.G. Overview: The Common Lisp Object System. *Lisp and Symbolic Computation* **1**(3/4), 1988, 227–244.
7. DYBVIG, R. K. *The Scheme Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.
8. DYBVIG, R. K. AND R. HIEB. Engines from continuations. *Journal of Computer Languages* (Pergamon Press) **14**(2), 1989, 109–124.
9. FELLEISEN, M. AND D.P. FRIEDMAN. A reduction semantics for imperative higher-order languages. In *Proc. Conference on Parallel Architectures and Languages Europe, Volume II: Parallel Languages.* Lecture Notes in Computer Science 259. Springer-Verlag, Heidelberg, 1987, 206–223.
10. FELLEISEN, M. AND D.P. FRIEDMAN. A syntactic theory of sequential state. *Theor. Comput. Sci.* **69**(3), 1989, 243–287. Preliminary version in: *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987, 314-325.
11. FELLEISEN, M. AND R. HIEB. The revised report on the syntactic theories of sequential control and state. Technical Report 100, Rice University, June 1989. *Theor. Comput. Sci.* **102**, 1992, 235–271.
12. FELLEISEN, M., D.P. FRIEDMAN, E. KOHLBECKER, AND B. DUBA. A syntactic theory of sequential control. *Theor. Comput. Sci.* **52**(3), 1987, 205–237. Preliminary version in: *Proc. Symposium on Logic in Computer Science*, 1986, 131–141.
13. FILINSKI, A. Representing monads. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, 1994, to appear.

14. Friedman, D.P., C.T. Haynes, and E. Kohlbecker. Programming with continuations. In *Program Transformations and Programming Environments*, edited by P. Pepper. Springer-Verlag, Heidelberg, 1985, 263–274.

15. Gordon, M.J. *The Denotational Description of Programming Languages*, Springer-Verlag, New York, 1979.

16. Haynes, C.T., D.P. Friedman, and M. Wand. Obtaining coroutines from continuations. *Journal of Computer Languages* (Pergamon Press) **11**, 1986, 143–153. Preliminary Version in *Lisp and Functional Programming*, 1984, Austin, TX.

17. Kanneganti, R., and R. Cartwright. What is a universal higher-order programming language? In *Proc. International Conference on Automata, Languages, and Programming.* Lecture Notes in Computer Science 700. Springer Verlag, Berlin, 1993, 682–695.

18. Meyer, A.R. and M. Wand. Continuation semantics in typed lambda-calculi. *Proc. Workshop Logics of Programs*, Lecture Notes in Computer Science 193, Springer-Verlag, Heidelberg, 1985, 219–224.

19. Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts and London, England, 1990.

20. Moggi, E. An abstract view of programming languages. LFCS Report ECS-LFCS-90-113, University of Edinburgh, 1990.

21. Moggi, E. Notions of computations and monads. *Inf. and Comp.* **93**, 1991, 55-92.

22. Moggi, E. Computational lambda-calculus and monads. In *Proc. Fourth Symposium on Logic in Computer Science*, 1989, 14–23.

23. Mosses, P. Denotational Semantics. In *Handbook of Theoretical Computer Science*, North-Holland, Amsterdam, 1991, 575–631.

24. Plotkin, G. $\mathbf{T}^\omega$ as a Universal Domain. *J. Comput. Syst. Sci.* **17**, 1978, 209–236.

25. Rees, J. and W. Clinger (Eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices* **21**(12), 1986, 37–79.

26. Reynolds, J.C. On the relation between direct and continuation semantics. In *Proc. International Conference on Automata, Languages and Programming*, 1974, 141–156.

27. Schmidt, D.A. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Newton, Mass., 1986.

28. Scott, D. S. Domains for denotational semantics. In *Proc. International Conference on Automata, Languages, and Programming*, Lecture Notes in Mathematics 140, Springer Verlag, Berlin, 1982, 577–613.

29. Scott, D.S. Data types as lattices. *SIAM J. Comput.* **5**(3), 1976, 522–587.

30. Sethi R. and A. Tang. Constructing call-by-value continuation semantics. *J. ACM* **27**(3), 1980, 580–597.

31. Sitaram, D. and M. Felleisen. Modeling continuations without continuations. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, 1991, 185–196.

32. Steele, G.L., Jr. *Common Lisp—The Language.* Digital Press, 1984.

33. Steele, G.L., Jr. Building interpreters by composing monads. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, 1994, to appear.

34. Stoy, J. The congruence of two programming language definitions. *Theor. Comput. Sci.* **13**, 1981, 151–174.

35. Stoy, J.E. *Denotational Semantics: The Scott-Strachey Approach to Programming Languages.* The MIT Press, Cambridge, Mass. 1981.

36. Strachey, C. and C.P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, 1974.

37. Sussman, G.J. and G.L. Steele Jr. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.

38. Wadler, P. The essence of functional programming. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, 1992, 1–14.

39. Wand, M. Continuation-based multiprocessing. In *Proc. 1980 ACM Conference on Lisp and Functional Programming*, 1980, 19–28.

40. Wright, A. and M. Felleisen. A syntactic approach to type soundness. Technical Report 160. Rice University, 1991. *Information and Computation*, 1993, to appear.

41. Wright, A.K. and R. Cartwright. A practical soft type system for Scheme. Technical Report Rice CS 93-218. Rice University, December 1993.

A Notation for Domain Specifications

The semantic definitions presented in the paper rely on the domain constructors $\otimes$ (smash product), $\oplus$ (coalesced sum), $\cdot_\perp$ (lifting), $\longrightarrow$ (continuous functions), and $\multimap$ (finite continuous functions). In this appendix, we briefly describe each of these constructions and associated notation.

The domain operators $\otimes$, $\oplus$, and $\cdot_\perp$ are defined by the following equations:

$$\begin{aligned}
\mathbb{A} \otimes \mathbb{B} &= \{(a, b) \mid a \in \mathbb{A},\ b \in \mathbb{B}, a \neq \perp, b \neq \perp\} \cup \{\perp\} \\
\mathbb{A} \oplus \mathbb{B} &= \{(\mathsf{true}, a) \mid a \in \mathbb{A} \setminus \{\perp\}\} \cup \{(\mathsf{false}, b) \mid b \in \mathbb{B} \setminus \{\perp\}\} \cup \{\perp\} \\
\mathbb{A}_\perp &= \{\perp\} \cup \{(\mathsf{true}, a) \mid a \in \mathbb{A}\}
\end{aligned}$$

The values true and false are used as "tags" in the construction of composite objects; the objects (true, a) and (false, a) are distinct from $\perp$ regardless of the value of a (including $a = \perp$). The notation

$$\mathtt{inA}(\mathbb{A}) \oplus \mathtt{inB}(\mathbb{B})$$

denotes exactly the same domain as

$$\mathbb{A} \oplus \mathbb{B}\,;$$

the tags inA and inB implicitly define the functions $\mathtt{inA} : \mathbb{A} \longrightarrow \mathbb{A} \oplus \mathbb{B}$ and $\mathtt{inB} : \mathbb{B} \longrightarrow \mathbb{A} \oplus \mathbb{B}$ where

$$\mathtt{inA}(x) = \begin{cases} (\mathsf{true}, x) & \text{if } x \in \mathbb{A} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathtt{inB}(x) = \begin{cases} (\mathsf{false}, x) & \text{if } x \in \mathbb{B} \\ \bot & \text{otherwise} \end{cases}$$

Finally, the notation

$$\mathtt{inA}(\mathbb{A})$$

has exactly the same meaning as

$$\mathtt{inA}(\mathbb{A}) \oplus \mathtt{in}\Phi(\Phi)$$

where Φ is the domain containing only the divergent element $\bot$.

The binary domain constructions $\otimes$ and $\oplus$ obviously generalize to any finite arity $n > 2$. In this case, we use n distinct tags such as the numbers $1, 2, \ldots, n-1$. The n-ary product of $\mathbb{A}_1, \ldots, \mathbb{A}_n$ is written

$$\mathbb{A}_1 \otimes \ldots \otimes \mathbb{A}_n\,.$$

Similarly, the n-ary sum of $\mathbb{A}_1, \ldots, \mathbb{A}_n$ with injections $\mathtt{inA}_1, \ldots, \mathtt{inA}_n$ is written

$$\mathtt{inA}_1(\mathbb{A}_1) \oplus \ \ldots \oplus \mathtt{inA}_n(\mathbb{A}_n)\,.$$

Since the choice of tag values in sums is arbitrary, we use *pattern matching notation* to identify components of a sum. It also provides a convenient mechanism for projecting a sum onto a component domain. Given the sum

$$\mathtt{inA}_1(\mathbb{A}_1) \oplus \ \ldots \oplus \mathtt{inA}_n(\mathbb{A}_n)\,,$$

the expression

$$\begin{array}{l} \mathtt{case}\ y\ \mathtt{of} \\ \quad [\mathtt{inA}_1(x_1) \ \Rightarrow e_1] \\ \quad \ldots \qquad\quad \Rightarrow \\ \quad [\mathtt{inA}_n(x_n) \Rightarrow e_n])) \end{array}$$

means:
if y has the form $\mathtt{inA}_1(x_1)$, then e_1
else ...
else if y has the form $\mathtt{inA}_1(x_1)$, then e_n
else $\bot$.

Each subexpression e_i can contain the variable x_i as well as any variables bound in the context enclosing the case expression.

The domain $\mathbb{A} \longrightarrow \mathbb{B}$ is the set of all *continuous* functions $f : \mathbb{A} \longrightarrow \mathbb{B}$ such that $f(\bigsqcup X) = \bigsqcup f(X)$ for all chains X. The domain of continuous functions is ordered by set inclusion (viewing functions as sets of ordered pairs). The notation $\underline{\lambda}\, x : \mathbb{A}.e : \mathbb{B}$ defines the anonymous function $f(x) = e : \mathbb{A} \longrightarrow \mathbb{B}$.

$\mathbb{A} \longrightarrow_s \mathbb{B}$ denotes the domain of strict functions from domain $\mathbb{A}$ to domain $\mathbb{B}$. A function in the domain $\mathbb{A} \longrightarrow \mathbb{B}$ is strict if $f(\bot_{\mathbb{A}}) = \bot_{\mathbb{B}}$.

The finite function domain construction $\mathbb{A} \multimap \mathbb{B}$ is defined only for *flat* domains $\mathbb{A}$:

$$\mathbb{A} \multimap \mathbb{B} = \{S \in \mathbb{A} \times \mathbb{B} \,|\, S \text{ is finite, } S \in \mathbb{A} \longrightarrow_s \mathbb{B}\}$$

Given two finite functions $f, g \in \mathbb{A} \multimap \mathbb{B}$, $f \sqsubseteq g$ if either (i) f is $\bot$ or (ii) f and g have the same domain D and for all $d \in D$, $f(d) \sqsubseteq g(d)$. We frequently interpret elements of $\mathbb{A} \multimap \mathbb{B}$ as functions in $\mathbb{A} \longrightarrow \mathbb{B}$.

B The Store Module and Global Definitions

This section presents the remainder of the modular interpreter for Core Scheme. Figs. 10 and 11 contain the global definitions, *i.e.*, the definition of the handler and of functions dealing with environments. Fig. 12 is the language fragment that deals with arithmetic; a similar fragment for boolean expressions was omitted for space reasons. Finally, Fig. 14 (Parts 1 and 2) presents the reference cell module.

```
;; Semantic Framework:
;; V = ⊥
;; C = Val(V) + Comp(V → C, Exc)

(define-const-structure (Val Num_n))
(define inVal make-Val)

(define-const-structure (FX V->C Receiver))
(define inCC (lambda (x) (make-FX inVal x)))

(define-const-structure (Exc num))
(define inExc (lambda (x) (inCC (make-Exc x))))

(define Handler
  (lambda (computation consumer)
    (match computation
      [($ Val v) (consumer v)]
      [($ FX continuation request)
       (make-FX
         (lambda (x) (Handler (continuation x) consumer)) request)])))
```

Fig. 10. Global Definitions

```
;; Environments:
(define Empty
  (lambda (x)
    (error 'env "Impossible: an open program ~s" x)))

(define Extend
  (lambda (env x v)
    (match x
      [(? symbol?) (match-lambda [(? symbol? y) (if (eq? x y) v (env y))])]
      [else (error 'Extend "not a variable: ~s" x)])))
```

Fig. 11. Environments

```
;; Syntax: e ::= (int n) | (add1 e) | (sub1 e)

(define-const-structure (int n))
(define-const-structure (add1 M))
(define-const-structure (sub1 M))

(define arithmM
  (lambda (language)
    (import ([language (InterpPrev Interpreter) Admin Resources])
      (module (Interpreter Admin Resources)

        ;; Semantics: V = Num(num)

        (define-const-structure (Num num))
        (define inNum make-Num)

        (define Interpreter
          (lambda (exp env)
            (match exp
              [($ int m) (inVal (inNum m))]
              [($ add1 exp) (Handler (InterpTop exp env) (op add1))]
              [($ sub1 exp) (Handler (InterpTop exp env) (op sub1))]
              [exp (InterpPrev exp env)])))

        (define op
          (lambda (f)
              (lambda (n)
                (match n
                  [($ Num m) (inVal (inNum (f m)))]
                  [_ (inExc 'error)]))))))))
```

Fig. 12. Module for Arithmetic

```
;; Syntax e ::= (ref e) | (deref e) | (setref e e)

(define-const-structure (ref x))
(define-const-structure (deref M))
(define-const-structure (setref M N))

(define storeM
  (lambda (language)
    (import ([language
              (InterpPrev Interpreter)
              (AdminPrev Admin)
              (ResPrev Resources)])
      (module (Interpreter Admin Resources)

        ;; Semantics:
        ;; V = Loc(L)
        ;; C = FX(V ⟶ C, Ref(V) + Der(L) + Set(L, V))

        (define-const-structure (Loc num))
        (define inLoc make-Loc)

        (define-const-structure (Ref v))
        (define inRef (lambda (x) (inCC (make-Ref x))))

        (define-const-structure (Der l))
        (define inDer (lambda (l) (inCC (make-Der l))))

        (define-const-structure (Set l v))
        (define inSet (lambda (l v) (inCC (make-Set l v))))

        (define Interpreter
          (lambda (exp env)
            (match exp
              [($ ref exp) (Handler (InterpTop exp env) inRef)]
              [($ deref exp)
               (Handler (InterpTop exp env)
                 (lambda (v)
                   (match v
                     [($ Loc l) (inDer l)]
                     [_ (inExc 'error)])))]
              [($ setref exp1 exp2)
               (Handler (InterpTop exp1 env)
                 (lambda (l)
                   (Handler (InterpTop exp2 env)
                     (lambda (v)
                       (match l
                         [($ Loc l) (inSet l v)]
                         [_ (inExc 'error)])))))]
              [exp (InterpPrev exp env)])))
```

Fig. 13. Module for Reference Cells: Part 1

```
(define Admin
  (lambda (comp resources)
    (let ([store (ExtractStore resources)])
      (match comp
        [($ FX continuation ($ Ref v))
         (AdminTop (continuation (inLoc (new store)))
           (cons (make-StoreRes (cons (cons (new store) v) store))
             resources))]
        [($ FX continuation ($ Der l))
         (AdminTop (continuation (lookup store l)) resources)]
        [($ FX continuation ($ Set l v))
         (AdminTop (continuation (inLoc l))
           (cons (make-StoreRes (update store l v)) resources))]
        [comp (AdminPrev comp resources)]))))

(define-const-structure (StoreRes s))

(define Resources (cons (make-StoreRes '()) ResPrev))

(define ExtractStore
  (lambda (resources)
    (match resources
      [(($ StoreRes s) . res) s]
      [(_ . res) (ExtractStore res)]
      ['() (error 'Extract "impossible")] )))

(define new length)

(define lookup
  (lambda (store l)
    (let L ([store store]);; assq yields less precise types
      (match store
        [(((? number? m) . v) . s) (if (= m l) v (L s))]
        ['() (error 'lookup "impossible")]))))

(define update
  (lambda (store l v)
    (match store
      [(((? number? m) . u) . s)
       (if (= l m)
           (cons (cons l v) s)
           (cons (cons m u) (update s l v)))]
      ['() (error 'update "impossible")])) )))
```

Fig. 14. Module for Reference Cells: Part 2

This article was processed using the LaTeX macro package with LLNCS style