Three Monads for Continuations

Richard B. Kieburtz, Borislav Agapiev, James Hook
Pacific Software Research Center

Oregon Graduate Institute
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 92-018

November 1992

Three Monads for Continuations *†

Richard B. Kieburtz Borislav Agapiev James Hook
Oregon Graduate Institute of Science & Technology
19600 N.W. von Neumann Dr.
Beaverton, OR 97006

e-mail: *lastname*@cse.ogi.edu

Technical Reprot CS/E 92-018

November 6, 1992

Abstract

We propose three monads that express the structure of different modes of continuation semantics. The first is the familiar CPS semantics, the second is a semantics for languages with first-class continuations, and in the third we have "composable contexts" that are useful to express the semantics of backtracking such as occurs in the computations of logic programs. The third structure is not actually a monad, as the left identity law fails for reasons that we discuss.

Associated with each monad are certain morphisms that yield values from computations, or latent values. These morphisms are respectively, **eval**, the evaluator of applicative expressions, **call/cc**, a meta-language analog of the `call/cc` control primitive of Scheme. The monads, enriched with these morphisms allow the expression of semantics of languages with explicit control operators, semantics not expressible without the enrichments. In the pre-monad, the semantics of such languages can be expressed without added enrichments.

The pre-monad supports the expression of context-dependent semantics for program structures that use control constructs such as backtracking, context reentry, or Horn-clause resolution. The paper illustrates its use by giving a semantics for a language that combines functional and logic programming styles in an interesting way.

1 Introduction

One of the nice properties of a continuation semantics for a conventional programming language is that the order of evaluation in the object language can be described without imposing an order of evaluation on the meta-language. Thus the semantics of the object language can reflect its operational aspects although the semantics of the meta-language is purely denotational.

*The research reported here was supported in part by the National Science Foundation under grant No. CCR-9101721.

†Submitted to *Lisp and Symbolic Computation* special issue on continuations.

Unfortunately, this property is lost as soon as first-class control constructs appear in the object language. At that point, a continuation can no longer be considered to be an abstract concept expressed only in the denotation of the language, but becomes the same concept expressed both in the object language or the meta-language. We have wondered whether there is a useful, higher-order concept in which to express a continuation-style semantics of languages with first-class control that would enable the semantics to recover its purely denotational flavor. This paper reports the results of our exploration. The higher-order concept is that of a *context*, a concept that has been used in the literature for many years but which has not previously been formalized in the way given here.

It would be very awkward to express the semantics of a programming language if the functions that denote syntactic expressions did not compose uniformly. The motivation for seeking a monadic framework for semantics is that the monad laws guarantee that computations that express latent values can be composed by the so-called Kleisli composition in the monad [10]. Moggi's thesis that "monads are everywhere" is that all kinds of compositional properties found in semantic frameworks can be characterized by the appropriate monads, if one will only look for them.

It is well known that a continuation semantics for an applicative language can be expressed by functions in the monad of the CPS transformation. We shall take that monad as a starting point and consider two additional structures, one a monad and the other almost a monad, for reasons that will be discussed. The formal development of these structures is the topic of Section 2.

The use of the (pre-)monad of composable contexts is illustrated by developing the semantics for a language in which functional and logic programming styles are tightly connected and which uses complex backtracking control for its evaluation. The paper concludes with a brief discussion of some issues in the semantics of languages with explicit control.

2 Monads capture semantic structure

Here we shall review the monad of the CPS-transformation, familiar from the work of Moggi and others, then introduce two structures that have not been previously studied. One of these is a monad of control alternatives, suggested by the call-with-current-continuation primitive of the Scheme language. The monad of control alternatives captures the intuition that a latent computation may either produce a value in its immediate context, or control may escape to heaven-knows-where. The `call/cc` primitive affords a means of specifying the context to which a control escape must return.

More generally, the context of an evaluation might be bound dynamically or might itself be the result of a calculation. The third structure considered is a pre-monad of composable contexts. This pre-monad provides continuations for control-alternative computations. There are compelling reasons to conjecture that it provides the structure needed to formulate semantics for the most general problems of programming with explicit control.

The object map of each monad, if expressed as a formula of propositional logic, forms the hypothesis of an implication from which one can derive full classical propositional logic. The

Curry-Howard analogy between intuitionistic propositions and types of the lambda calculus is well-known. Less well-known is the analogy pointed out by Griffin [6] between classical propositions and types of a computational calculus with first-class continuations, such as the CBV calculus of Filinski [5]. The monads studied here offer three different bases for such a calculus, analogously as the corresponding logic formulas offer three different ways to complete axiom schemes for classical propositional logic.

2.1 Monads of a cartesian-closed category

As a model for programming language semantics, we assume an underlying cartesian-closed category. The intended interpretation is that objects of the category correspond to types and morphisms to functions. State is easily accommodated in such a model [11, 10]. We shall use the following characterization of a monad [8]

Definition 1: A **Kleisli triple** (T, η, $(\_)^*$) in a category $\mathcal{D}$ consists of

- an object mapping function $\mathrm{T} : Obj(\mathcal{D}) \rightarrow Obj(\mathcal{D})$,
- a natural transformation called the **unit**, $\eta_X : X \rightarrow TX$,
- a natural extension operation that takes each morphism $f : X \rightarrow TY$ to a morphism $f^* : TX \rightarrow TY$ in $\mathcal{D}$.

These components of a monad must satisfy three laws:

$$\eta_X^* = id_{TX} \quad \text{(K1)}$$

$$f^* \circ \eta = f \quad \text{(K2)}$$

$$(g^* \circ f)^* = g^* \circ f^* \quad \text{(K3)}$$

Some authors write $g \odot f$ to express $g^* \circ f$, calling this the *Kleisli composition* of g with f. Laws (K1) and (K2) express that the unit is respectively, a left and a right identity with respect to Kleisli composition. Law (K3) expresses that the natural extension is associative with respect to morphism composition, i.e. that the Kleisli composition is associative. We shall call a monad-like structure a **pre-monad** if it satisfies (K3) and (K2) but not (K1).

A morphism $k : X \rightarrow Y$ of the underlying category can be 'lifted' to a T-monadic morphism by composition on the left with the unit of the monad, $\eta_Y \circ k : X \rightarrow TY$. Such morphisms are called the **proper**, (or existing) morphisms of T. The natural extension of proper morphisms provides a mapping of morphisms $X \rightarrow Y$ to $TX \rightarrow TY$ which together with the object mapping function constitutes a functor $T : \mathcal{D} \rightarrow \mathcal{D}$. The more interesting morphisms of a monad are those of types $X \rightarrow TY$ that are non-proper. For each of the monads we consider, we shall be interested in the interpretation given to its non-proper morphisms.

2.2 What is a monad for continuations?

Each of the monadic structures studied here can be used to transform a direct semantics for the λ-calculus into a call-by-value semantics that uses continuations explicitly. As the language is

extended, we do not expect that every function will denote a proper morphism in the category. Hence, functions given the type $X \to Y$ in the language will correspond to morphisms from X to TY in the category, i.e. they will map values to computations. Furthermore, an expression representing a function value acquires a type $T(X \to TY)$.

When Moggi introduced monads as a tool to structure semantics, he used a notation based on **let** to capture the distinction between computations and values. In Moggi's metalanguage the call-by-value application rule is written:

$$\begin{array}{lll} \mathbf{let}_T & f = [\![M]\!] & \\ \mathbf{in} & \mathbf{let}_T & x = [\![N]\!] \\ & \mathbf{in} & fx \end{array}$$

Here $[\![M]\!]$ and $[\![N]\!]$ are computations of types $T(X \to TY)$ and TX, f and x are bound to the values of type $X \to TY$ and X extracted from them and fx is the computation of type TY representing the result. This metalanguage is translated into the monadic framework above by mapping $\mathbf{let}_T\ v = C\ \mathbf{in}\ B$ to $(\lambda v.B)^*C$.

The interpreter below, called the Kleisli interpreter[3], is obtained from a standard direct semantics. This semantics will be instantiated for each of the three monads discussed in the paper. The symbol π represents a primitive function of type $X \to Y$.

$$\begin{array}{rcl} [\![x]\!]\rho & = & \eta(\rho\, x) \\ [\![\pi]\!]\rho & = & \eta(\eta \circ \pi) \\ [\![\mathbf{if}(M, N_0, N_1)]\!]\rho & = & (\lambda b.\mathbf{if}(b, [\![N_0]\!]\rho, [\![N_1]\!]\rho))^*([\![M]\!]\rho) \\ [\![\lambda x.M]\!]\rho & = & \eta(\lambda v.[\![M]\!]\rho[x \mapsto v]) \\ [\![M\, N]\!]\rho & = & (\lambda f.f^*([\![N]\!]\rho))^*([\![M]\!]\rho) \end{array}$$

2.3 The "answers" object

A perplexing question has been how to characterize the object that is the codomain of all continuations—the "answers" object. Some guidance is provided by intuition:

- This object can be interpreted as a type that includes all observable data elements. Elements of datatypes can be returned as ultimate answers at the top level of a program.
- Ultimate answers cannot be discriminated within a program, but only in the environment in which a program is executed. Thus the "answers" object cannot be analyzed in the object language.
- The class of continuations is rich enough to discriminate all observables. There is at least one continuation for each datatype that is capable of discriminating its elements.

We assume an underlying categorical structure in which to model computation by abstract machines. Let $\mathcal{C}$ be a closed symmetric monoidal category with generator I and tensor $\otimes$. An exponent Y^X in $\mathcal{C}$ is designated by the infix notation $X \multimap Y$. Suppose that $\mathcal{C}$ contains a

full subcategory $\mathcal{D}$ that is bicartesian, generated by $(1, \times, +)$. That is, the tensor product in $\mathcal{D}$ becomes the cartesian product, $\mathcal{D}$ has a terminal object and it also has finite coproducts and morphisms for distribution, $d_{A,B,C} : A \times (B + C) \to (A \times B) + (A \times C)$, natural in A, B and C, with the expected coherence conditions. This provides enough structure in $\mathcal{D}$ to model datatypes, but we shall need more. $\mathcal{D}$ must have exponential objects that can be interpreted as the function spaces whose elements are the denotations of programs.

In this framework, the "answers" object, according to our intuition, must satisfy the axioms:

$$\forall X \in Obj(\mathcal{D}) \text{ there is a unique arrow } \delta_X : X \otimes Ans \to Ans \tag{I1}$$

$$\forall X \in Obj(\mathcal{D}) \text{ there is a monic arrow } \kappa_X : X \to Ans \tag{I2}$$

Axiom (I1) expresses that Ans is really a type of *ultimate* answers that cannot be further analyzed; axiom (I2) expresses that there is a continuation capable of fully discriminating the elements of X. From (I1) by exponentiation we obtain for each $X \in Obj(\mathcal{D})$ the unique arrow $\delta^\star : X \to Ans \multimap Ans$ which tells us that $Ans \multimap Ans$ is a terminal object of $\mathcal{D}$. This observation leads to the following

Proposition: If $Ans \in Obj(\mathcal{D})$ then $\mathcal{D}$ is a preorder category.

Proof: By *(I2)* there is a monic arrow $\kappa_X : X \to Ans$. Suppose $x_1, x_2 : 1 \to X$ are two points of X. Then x_1, x_2 are distinct if and only if $\kappa_X \circ x_1$, $\kappa_X \circ x_2 : 1 \to Ans$ are distinct points of Ans. However, if $Ans \in Obj(\mathcal{D})$ then it is easily shown that Ans has only a single point. For, suppose $h, h' : 1 \to Ans$. Then $h \circ \pi_1 = \delta_1 = h' \circ \pi_1 : 1 \times Ans \to Ans$. Since 1 is terminal, $\pi_1 : 1 \times Ans \to 1$ is epic and we have $h = h'$ and hence, $x_1 = x_2$.
□

If the category $\mathcal{D}$ is not to collapse to a preorder, then apparently Ans lies outside of $\mathcal{D}$. We offer the following construction for Ans, again motivated by the intuition that the "answers" object is a sum of all datatypes. Let Ans be defined as a limit of all finite coproducts,

$$Ans = \sum_{X \in Obj(\mathcal{D})} X$$

Notice that $\mathcal{D}$ is a small category, finitely generated. However, Ans is not itself a small object; it is the sum of all small sums. Thus there is no reason to expect this limit to exist in $\mathcal{D}$. It lives in the host category, $\mathcal{C}$. There are no arrows from this large object to the small objects of $\mathcal{D}$, hence no compositions of continuation arrows.

In the following sections, we shall use the symbol "$\multimap$" to designate exponentials in $\mathcal{C}$ and "$\to$" to designate exponentials in $\mathcal{D}$, which is assumed to be cartesian-closed.

2.4 The continuation-passing monad

The functions of CPS semantics are captured in the monad whose object function, unit and natural extension operation are:

$$TX \quad = \quad (X \multimap Ans) \multimap Ans$$

$$\begin{aligned} \eta_X &= \lambda x \lambda c.cx \\ f^* &= \lambda t.\lambda c.t(\lambda x.fxc) \\ &\quad \textbf{where } f : X \to TY \end{aligned}$$

We call this the CPS monad. It has previously been called the monad of continuations [10] but, as we shall see, it is not the only interesting monadic structure that captures computation with continuations. It is, however, the only one of the three structures studied here that results in a "tail recursive" semantics.

In the propositions-as-types analogy between intuitionistic propositional logic and the simply typed λ-calculus, λ-terms of type t correspond to proofs of the formula corresponding to t in the logic. Closed lambda terms correspond to proofs of tautologies. Griffin [6] observed that the analogy extends to one relating classical logic to a λ-calculus extended with typed continuations, and used the analogy to suggest types for control operators.

An "answers" object is analogous to the absurdity proposition of an intuitionistic logic [6]. An object TX is analogous to a double-negation proposition, $\neg\neg X$, in intuitionistic logic. The formula $\neg\neg X \Rightarrow X$, when added as an axiom scheme, yields classical logic. Analogous to this formula would be morphism $\mathbf{eval}_X : TX \to X$ in the category $\mathcal{D}$. For proper computations of T it satisfies:

$$\mathbf{eval}(\lambda c.cx) = x$$

Such a morphism cannot be defined as a closed λ-expression, i.e. it does not necessarily exist as a consequence of the cartesian-closed property of $\mathcal{D}$. As a semantic framework, T + **eval** begs the termination problem. Properties of programs inferred from a T + **eval** semantics are so-called "partial correctness" properties, i.e. properties that hold of terminating computations, but without guarantee of termination.

Intuitively, **eval** installs a computation in an abstract machine, provides it with an initial continutation, runs it, and extracts the answers. An **eval** specific to the SECD machine, for example, would install the program as the control string along with an empty stack and dump, run the machine until the control and dump were empty, then return the value at the top of the stack. If the untyped λ-calculus is used as the abstract machine, **eval** may be realized as the function that provides the identity function $(\lambda x.x)$ as an initial continuation and returns the resulting lambda term. When comparing this semantics to other λ-calculus based treatments of control operators this specialized view of **eval** is illuminating. In particular, the notion of "composable continuations" expressed in [2] is based upon an implicit assumption that **eval** exists as a morphism.

The CPS monad internalizes as objects TX the morphisms that map X-accepting continuations to final results. Such objects are sets of 'latent computations' that provide semantics for applicative expressions. If $f' : X \to Y$ is a morphism of $\mathcal{D}$, then the proper morphism $f = \eta_Y \circ f' : X \to TY$ satisfies the equation

$$f\,x\,c = c(f'\,x)$$

Non-proper morphisms of this monad are those whose codomain element may represent a computation that discards the nominal result continuation and instead uses a different continuation to effect a tail-call or to raise an exception, or which diverges.

2.5 The monad of control alternatives

The second monad we consider is motivated by the desire to provide semantics to expressions abstracted on a continuation variable. The constituents of the monad are:

$$\begin{aligned} SX &= (X \multimap Ans) \to X \\ \eta_X &= \lambda x.\lambda c.x \\ f^* &= \lambda s.\lambda c.f(s(\lambda x.c(fxc)))\,c \\ &\quad \textbf{where } f : X \to SY \end{aligned}$$

As before, Ans is required to be an "answers" object, and an object $(X \multimap Ans)$ is interpreted as a type of X-accepting continuations.

The intuitionistic formula analogous to an object SX is $\neg X \Rightarrow X$, which in classical logic is abbreviated as $X \vee X$. A morphism $SX \to X$ can be interpreted as evaluating a computation that might produce a value of type X in two different ways, either by a direct evaluation, ignoring the continuation argument, or by invoking the argument continuation. The analogy with a disjunctive formula of logic hints that SX may be related to a disjoint sum, $X + X$. This is indeed the case, provided there is added to the set of monad morphisms a constructor $\mathcal{A}_X : Ans \to X$, called 'abort' [4]. Then we can define

$$\begin{aligned} inl &= \eta = \lambda x.\lambda c.x \\ inr &= \lambda x.\lambda c.\mathcal{A}(cx) \end{aligned}$$

The discriminator is

$$\textbf{case}(s, f, g) = \lambda c.f\,(s(\lambda x.c(g\,x\,c)))\,c$$

in which $s : SX$, $f : X \to Y$ and $g : X \to Y$. Notice the similarity in form between the discriminator and the natural extension of a function in the monad S,

$$f^* = \lambda s.\textbf{case}(s, f, f).$$

It is informative to compare this formulation with Griffin's construction of disjunctive types [6] in the CPS monad. That construction requires the explicit addition of both the operator $\mathcal{A}$ and of Felleisen's control operator [4], $\mathcal{C}$, while in the monad S we need add only $\mathcal{A}$ as an explicit operator. However, since $\mathcal{A}$ is conventionally defined in terms of $\mathcal{C}$, independent axioms are needed for $\mathcal{A}$ if it is to be defined without $\mathcal{C}$. We propose the following axioms:

$$c'(\mathcal{A}(cx)) = cx \quad \text{provided } x : X,\ c, c' : X \multimap Ans. \tag{A1}$$

$$\lambda c.f(\mathcal{A}(cx))c =_{SX} \lambda c.x \quad \text{provided } x : X,\ f : X \to SX. \tag{A2}$$

Axiom (A1) says that continuations are strict and (A2) says that functions are strict in the meta-language. Care must be taken to ensure that the axioms do not entail collapse of the category. Even a slight generalization of (A2) causes collapse. We have not proved soundness for (A1) and (A2) as given above, but neither have we been able to show that they cause collapse in a cartesian-closed category.

Abbreviating $\lambda s.\mathbf{case}(s, f, g)$ as $[f, g]$, it is now easy to check that the following properties of sums hold:

$$[f,g] \circ inl \;=\; f \qquad [f,g] \circ inr \;=\; g \qquad [u^* \circ inl, u^* \circ inr] \;=\; u^*$$

Note that a general categorical coproduct would require the stronger property $[u \circ inl, u \circ inr] = u$ in which the codomain of u is arbitrary. This sum is a true coproduct if and only if S is a full functor from the category $\mathcal{D}$ to itself.

Additionally, one can simply postulate a constructor that injects expressions of type SX into a λ-calculus. The introduction rule is

$$\frac{\Gamma \vdash s : SX}{\Gamma \vdash \mathbf{call/cc}_X\, s : X}$$

Although **call/cc** cannot be expressed as a closed formula in the lambda-calculus, it is closely related to morphisms of the monad S. It is a structure function for initial S-algebras, and satisfies the equation:

$$\eta^S_X \circ \mathbf{call/cc}_X = id_{SX}$$

The operational explanation of **call/cc** is that when applied to an abstraction, $\lambda c.e$, it binds the abstraction variable, c, to the current continuation. Any subexpression of the form $c\,e'$ is interpreted as a 'throw' of the value of expression e' to the bound continuation.

But what if the value of e' is itself constructed with **call/cc**? The semantics of composite expressions in this monad are explained by the Kleisli composition. This is composition of the natural extension in the monad S of morphisms that may produce either normal values (proper morphisms) or may abort with an alternative continuation.

In Scheme, **call/cc** has been lifted from its status as a semantic operator of the meta-language to become a syntactic operator of the programming language. The Kleisli interpreter for the monad S can be extended to account for this language construct:

$$[\![\texttt{call/cc}\,\lambda x.M]\!]\,\rho \;=\; \lambda c.([\![M]\!]\,\rho[x \mapsto c])\,c$$

To complete the analogy with formulae of logic, note that the logical formula $(\neg X \Rightarrow X) \Rightarrow X$ is Peirce's law, also sufficient to yield full classical logic when added to intuitionistic logic as an axiom scheme. This formula corresponds to the type of $\mathbf{call/cc}_X \;:\; SX \to X$.

2.6 The pre-monad of composable contexts

The third structure is intended to provide a complete foundation for a semantics of logic programs, or of a language with first-class control primitives. This structure is a composite of the two previous ones, with constituents:

$$\begin{aligned} RX &= T(SX) = (((X \multimap Ans) \to X) \multimap Ans) \multimap Ans \\ \eta_X &= \eta^T_{SX} \circ \eta^S_X = \lambda x.\lambda h.h(\lambda c.x) \\ f^* &= \lambda r.\lambda h.r(\lambda s.f(s(\lambda x.fxh))h) \\ &\quad \mathbf{where}\ f : X \to RY \end{aligned}$$

This structure is not a monad, as the left identity law (K1) fails, but it is a pre-monad. The left identity law would be provable if elements of type SX were restricted to those constructed by application of η^S_X, but then the monad R would be isomorphic to the CPS monad. We conjecture that the left identity law may also be provable in a category without fixpoints, which would imply that it is connected with the uniform termination problem for R-computations.

An object RX is a space of computations that take SX-expecting continuations to final results. We call an SX-accepting continuation an X-expecting *context*. A context supplies its SX-typed argument with both an X-expecting continuation for a result produced by normal evaluation and a second continuation of the same type for use if the evaluation aborts. Thus an aborted computation need not escape to the 'top level', but may backtrack. Aborting a computation with an alternate continuation is equivalent to continuing the computation in another context. This intuition is summarized in the CPS transformation of SX-typed expressions:

$$\begin{aligned} [\![inl\, x]\!]_T\, h &= h(inl\, x) \\ [\![inr\, x]\!]_T\, h &= h_0(inl\, x) \end{aligned}$$

where $[\![\_]\!]_T$ denotes the Kleisli interpreter instantiated on the monad T and h_0 is a context constant, or initial context. (There is no closed λ-term of type $SX \multimap Ans$.)

A semantics of either applicative or relational expressions built with this monad allows contexts to be composed incrementally. Incremental composition of continuations was not possible in either of the monads T or S, because continuations do not compose as ordinary functions. It is possible in R, because higher-order continuations are available as contexts. The Kleisli composition in R allows context abstractions to occur as arguments of functions, in effect subsuming higher-order CPS transformations.

3 Semantics of F+L—a functional language with Horn-clause logic

To illustrate the use of the pre-monad R, we shall give a semantics for a language that integrates functional and logic programming styles. In this language, expressions may be qualified by a declarative proposition involving existentially quantified variables. A qualified expression has a value if there is a valuation for the existential variables that satisfies the constraint imposed by the proposition. Since any such valuation is sufficient, expressions are multi-valued. However, valuations may be calculated in a particular order by imposing an order of evaluation on applicative expressions, and a search strategy for satisfaction of logical constraints. F+L uses normal-order evaluation and depth-first search.

What makes this language different from other attempts to combine logic and functional programming is (1) that logical variables are first-class and (2) that arbitrary expressions (of non-functional types) can be used as arguments of a predicate. In the presence of first-class logical variables, the constraints imposed upon variables are not necessarily restricted to the qualification clause in which a variable is introduced. Additional constraints may be imposed by the context into which a variable is passed. If a variable is passed as an argument to a function,

it may also be required to satisfy constraints implicit in the declaration of the function's body. An applicative expression may produce a multi-value as its result, by returning an unconstrained logical variable. This liberal treatment of logical variables has been adopted in F+L so that the language does not sacrifice completeness of its logic fragment, other than by specialization of its search strategy, which can be viewed as an implementation decision.

Continuation semantics in the CPS monad is not a convenient formalism in which to express the meaning of F+L. Although it is possible to express the backtracking control implied by alternate clauses defining a predicate, it is not easy to express nested backtracking control. Simple backtracking requires only an abort primitive together with multiple continuations. The continuation alternatives are tried in sequence until one of them does not abort. However, when constraints can be composed dynamically as well as statically (a consequence of first-class logical variables), it is necessary to express a composition of dynamic contexts that cannot be easily expressed with continuations. Continuations are not composable.

The monad R has the mechanism needed to represent the required composition. We shall give a semantics for the kernel of F+L. The kernel, called Mini-F+L, has as its core a polymorphically typed functional programming language with lazy evaluation rules, pattern-matching syntax and local definitions, i.e. **let** expressions. To this is added predicate definitions in the form of Horn clauses, and qualified expressions, whose form is

$$\textbf{let } P(x_1, \ldots, x_k, e_{k+1}, \ldots, e_n) \textbf{ var } x_1, \ldots, x_n \textbf{ in } e$$

where P is a predicate symbol. The variables declared in a **var** clause are existentially quantified. The existentially quantified variables and the expressions $e_{k+1}, \ldots, e_n$ must each have an equality type.

Informally, the value of a qualified expression is a value of the subject expression together with a satisfying valuation for its existential variables. A satisfying valuation is computed by depth-first search of the Horn clause definitions for a proof of the qualifying proposition. Since existential variables may also have occurrences within the propositional part of a qualified expression nested within the subject expression e, it is not assured that every proof of $P(x_1, \ldots, x_k, e_{k+1}, \ldots, e_n)$ will produce a satisfying valuation. A satisfying valuation must satisfy all propositions in which the existential variables occur. When a proof of $P(x_1, \ldots, x_k, e_{k+1}, \ldots, e_n)$ fails to produce a satisfying valuation, it may be that backtracking to find additional proofs will yield a valuation that is satisfying.

If there were no possibility of multiple occurrences of a variable in the expressions of Mini-F+L, then a call-by-name semantics would suffice. In a call-by-name semantics, the environment of a computation would be a set of equations for the existential variables. Actually this set would be a set of sets of equations, for the use of multiple clauses in the definition of a predicate symbol will give rise to an independent set of equations for each clause. To avoid inconsistency in the valuation of different occurrences of a particular variable, the choices made among multiple clauses in arriving at a value for each instance of a variable would need to be recorded along with its value. This makes a call-by-need strategy more attractive, because the consistency problem can be circumvented while at the same time avoiding possible recalculation of valuation at different occurrences of the same variable.

In a call-by-need semantics, variables are bound to values in a state component, a *store*, which must be updated whenever a value binding is made. One aspect of the call-by-name semantics remains, however. When two previously unbound variables are equated, the "value" binding for each of them should be a thunk which will examine a store argument at the other variable's name to find its binding. To obtain uniformity, it is necessary that all value bindings in the store must become thunks.

Bindings to actual values must be distinguishable in the store from the initial "value" given to an unbound variable. We indicate this distinction by tags **u** (for unbound) and **v** (for value). Thus the bindings that would correspond to equations

$$u = 3 \qquad w = (\mathit{unbound}) \qquad x = y$$

are

$$\{u : \lambda\sigma.(\mathbf{v}, 3), \quad w : \lambda\sigma.(\mathbf{u}, w), \quad x : \lambda\sigma.\sigma\, y\, \sigma, \quad y : \lambda\sigma.\sigma\, x\, \sigma\}$$

where σ : *Store* and

$$\mathit{Store} = \mathit{Identifier} \rightarrow (\mathit{Store} \rightarrow \mathit{Value})$$

Notice that the bindings for the identified variables x and y form a cycle. Thus, recursive use of this store to search for a binding for either of these variables would not terminate.

3.1 Syntax of the expression language

A Mini-F+L program consists of definitions of logic predicates followed by an expression. The syntax of expressions is:

Expr ::= Ident
| Expr Expr
| "λ" Ident . Expr
| "π" Expr
| **let** Goal **var** Ident-list **in** Expr
| **let** Goal **in** Expr

where an anything-list is a nonempty list of anythings. The variables introduced in a **var** clause are implicitly existentially quantified and their scope extends over the preceding Goal and the following Expr. A Goal is a conjunction of one or more propositions, each formed by applying a defined predicate letter to a sequence of expressions. Predicate letters have arities and typings fixed by their definitions. The syntax and semantics of the logic fragment of Mini-F+L will be given later.

3.2 Semantics of Mini-F+L

The semantic functions for Mini-F+L are typed as:

$$\mathcal{E} : \mathit{Expr} \rightarrow \mathit{Store} \rightarrow R(\mathit{Value} \times \mathit{Store})$$

$$\mathcal{L} : \mathit{Prop} \rightarrow \mathit{Store} \rightarrow R(\mathit{Value} \times \mathit{Store})$$

$$\mathcal{V} : \mathit{Ident\text{-}list} \rightarrow \mathit{Store} \rightarrow \mathit{Store}$$

The semantics of the (abbreviated) expression language is given below. Rules (*app*), (*abs*) and (*strict*) are calculated by the Kleisli interpreter.

(*var*) $\mathcal{E}\,[\![x]\!]\,\sigma\,h = h\,(\lambda c.\sigma\,x\,(\sigma[x \mapsto \lambda\sigma'.(\mathbf{u}, x)]),\ \sigma)$

(*app*) $\mathcal{E}\,[\![e_1\,e_2]\!]\,\sigma\,h = \mathcal{E}\,[\![e_1]\!]\,\sigma\,(\lambda s_1.\mathcal{E}\,[\![e_2]\!]\,\sigma\,(\lambda s_2.$
$\quad$ **let** $(f', \sigma') = s_1\,(\lambda(f, \sigma).\mathcal{E}\,[\![e_2]\!]\,\sigma\,(\lambda s.f\,(s\,(\lambda x.f\,x\,h))\,h))$
$\quad$ **in** $f'\,(s_2\,(\lambda x.f'\,x\,h))\,h))$

(*abs*) $\mathcal{E}\,[\![\lambda x.e]\!]\,\sigma\,h = h\,(\lambda c.(\lambda(v, \sigma').\mathcal{E}\,[\![e]\!]\,(\sigma'[x \mapsto \lambda\sigma.(\mathbf{v}, v)])))$

(*strict*) $\mathcal{E}\,[\![\pi]\!]\,\sigma\,h = h\,(\lambda c.\lambda(b, \sigma').\lambda h'.h'\,(\lambda c'.$ **case** b **is**
$\quad$ $(\mathbf{u}, x) \Rightarrow \mathcal{A}(c(\pi, \sigma))$
$\quad$ | $(\mathbf{v}, a) \Rightarrow \pi\,a,\ \sigma'))$

(*qual*) $\mathcal{E}\,[\![$**let** q **in** e **var** $xs$$]\!]\,\sigma\,h = \mathcal{L}\,[\![q]\!]\,(\mathcal{V}[\![xs]\!]\,\sigma)\,(\lambda s.\mathcal{E}\,[\![e]\!]\,(s\,(\lambda\sigma'.\mathcal{E}\,[\![e]\!]\,\sigma'\,h))\,h)$

(*ext*) $\mathcal{V}[\![\ x_1, x_2 \ldots x_n]\!]\,\sigma = \mathcal{V}[\![\ x_2, \ldots x_n]\!]\,(\sigma[x \mapsto \lambda\sigma'.\sigma'\,x_1\,\sigma'])$

(*ext'*) $\mathcal{V}[\![\ ]\!]\,\sigma = \sigma$

The rule for existentially quantified variables, (*ext*), updates a store with a binding that forms an elementary cycle for each new variable. It does this in order that variables with no actual value bindings can easily be bound into cyclic search paths in the store. However, this means that unless other steps were taken, an attempt to query a store for the binding of an as-yet-unbound existential variable would result in an infinite, cyclic search.

The rule (*var*) applies the binding of a variable x in a store to the store updated so that if queried for its binding for x, it will return $(\mathbf{u}, x)$. The tag $\mathbf{u}$ indicates that x has no proper value. In the query $\sigma\,x$, if the original store, σ, contains a proper value binding for x, that value will be returned. However, if σ does not contain a proper binding for x, then it will eventually query the updated store for its binding for x. This avoids cyclic search. The technique is due to Gary Lindstrom [7].

The rule (*strict*) contains a check that the argument of a strict operator has a proper value. If not, the computation invokes the backtrack continuation provided by the context.

In the rule for a qualified expression, (*qual*), evaluation of the proposition q furnishes a satisfying binding in the store variable. The context for q is the expression e, which receives the store it needs for evaluation from q. It provides an explicit backtracking continuation for q, namely the continuation willing to retry the evaluation of e if given a new store.

3.2.1 A value comparison function

A semantic function that compares the values of two expressions given as arguments is fundamental to a computational logic. This function also binds variables to values by updating the store in case one of its arguments is a variable with no proper value.

$$
\begin{array}{l}
Comp(e_1, e_2)\,\sigma\, h \;= \\
\qquad \mathcal{E}\,[\![e_1]\!]\,\sigma\,(\lambda s.\, h(\lambda c.\, \mathbf{let}\ (w_1, \sigma') = s\, c\ \mathbf{in} \\
\qquad \mathbf{case}\ w_1\ \mathbf{is} \\
\qquad\qquad (\mathbf{u}, x) \Rightarrow \mathcal{E}\,[\![e_2]\!]\,\sigma'\,(\lambda s.\, h(\lambda c.\quad \mathbf{let}\ (w_2, \sigma''). = s\, c\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{case}\ w_2\ \mathbf{is} \\
\qquad\qquad\qquad\qquad (\mathbf{u}, y) \Rightarrow (w_2,\, \sigma'[x \mapsto \lambda\sigma.\, \sigma\, y\, \sigma,\ y \mapsto \lambda\sigma.\, \sigma\, x\, \sigma]) \\
\qquad\qquad\qquad \mid\ (\mathbf{v}, a) \Rightarrow (w_2,\, \sigma'[x \mapsto \lambda\sigma. w_2]))) \\
\qquad \mid\ (\mathbf{v}, a) \Rightarrow \mathcal{E}\,[\![e_2]\!]\,\sigma'\,(\lambda s. h(\lambda c.\quad \mathbf{let}\ (w_2, \sigma''). = s\, c\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{case}\ w_2\ \mathbf{is} \\
\qquad\qquad\qquad\qquad (\mathbf{u}, y) \Rightarrow \quad (w_1,\, \sigma'[y \mapsto \lambda\sigma.\, w_1]) \\
\qquad\qquad\qquad \mid\ (\mathbf{v}, a') \Rightarrow \quad \mathbf{if}\ a = a'\ \mathbf{then}\ (w_1, \sigma'') \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathcal{A}(c(sc)))))
\end{array}
$$

In the first case instance, e_1 has evaluated to an unbound logical variable, x. (Note that x may be bound to other unbound variables.) There are two subcases. If the evaluation of e_2 equals $(\mathbf{u}, y)$, then y is also an unbound existential variable. In this case, the bindings of x and y are exchanged in the updated store, creating a single, cyclic lookup path that includes both x and y, along with any other variables to which either has previously been bound. Otherwise, e_2 evaluates to a proper value which is bound to x in the updated store. The updated store is made available for use in evaluating further instances of x.

The second case instance is one in which e_1 has a proper value in the store. Again there are two subcases. The expression e_2, with which the value of e_1 is to be compared, may be an unbound variable or it may have a proper value. If e_2 evaluates to an unbound variable, y, then the result updates the store to include a binding for y. If e_2 has a proper value which is equal to the value of e_1, the comparison succeeds with no change to the store; otherwise the comparison fails by invoking the backtracking continuation supplied by the context, h. This is indicated by an application of the abort primitive, $\mathcal{A}$.

The reader will notice the resemblance of *Comp* to a unification algorithm. The resemblance is, of course, not accidental. Unification has been internalized in the semantics of Mini-F+L, However, the only remnant of symbolic interpretation associated with *Comp* is value-tagging in the store where existential variables are bound, to distinguish actual values from unbound variable names, or locations.

3.2.2 Semantics of the logic fragment

The logic fragment of Mini-F+L consists of predicate declarations whose syntax is

Def	::=	Clause-list
Clause	::=	Head .
	|	Head :- Goal-list .
Head	::=	Ident (Pat-list)
Goal	::=	Ident (Expr-list)
Pat	::=	Expr

A pattern (Pat) has the syntax of an expression restricted to variables and data constructors. The identifiers of heads and atomic goals are predicate symbols. All occurrences of a particular predicate symbol must be consistent with respect to arity.

There is a distinguished predicate symbol, `EQ`, that is given a standard computational interpretation. Its definition is the fact,

$$\texttt{EQ}(X, X).$$

and its interpretation is

$$\mathcal{L}[\![\texttt{EQ}(e_1, e_2)]\!]\,\sigma\,h = Comp(e_1, e_2)\,\sigma\,h$$

For evaluation, each F+L predicate declaration is translated to a logically equivalent form in which the body (sequence of goals) of each clause consists of independent goals, conjoined with dependency constraints expressed explicitly as equality propositions [1]. The translation linearizes each clause. A clause is rewritten to eliminate every repeated occurrence of a variable in its head or within its body. Repeated occurrences are replaced by new existential variables and an `EQ` atom is introduced as a constraint, to require that the same value must be bound to the introduced variable as to the one whose occurrence it replaces. After linearization, the residual clause has the following properties:

- No variable occurs more than once in the head of a clause;
- No variable occurs more than once in the residual body;
- The set of variables occurring in the `EQ` constraints is a subset of those that occur in the residual clause.

The clause as rewritten, with the generated `EQ` constraints conjoined to its body, is logically equivalent to the original clause. The motivation for this normalization of clauses is to simplify constraints so that they can be calculated by pairwise comparisons.

An informal presentation of the translation rules is:

$$P(..X, \ldots, X..) :\!- body. \implies P(..X, \ldots, Y..) :\!- body \wedge \texttt{EQ}(X, Y)\ \textbf{var}\ Y.$$

$$head :\!- \ldots Q(..X, \ldots, X..) \ldots \implies head :\!- \ldots Q(..X, \ldots, Y..) \ldots \wedge \texttt{EQ}(X, Y)\ \textbf{var}\ Y.$$

$$head :\!- \ldots Q(..X..) \wedge \ldots \wedge R(..X..) \ldots \implies head :\!- \ldots Q(..Y..) \wedge \ldots \wedge R(..Y..) \ldots \wedge \texttt{EQ}(X, Y)\ \textbf{var}\ Y.$$

An atomic goal also translates into a sequence of EQ constraints. Given a goal $P(e_1, \ldots, e_n)$, create a fresh instance of the definition of P, that is, a copy of the clauses that constitute the definiton of P, in which the variables occurring in the head are replaced by fresh variables $x_1, \ldots, x_m$. The head is $P(p_1, \ldots, p_n)$. Then the goal is translated to

$$P(p_1, \ldots, p_n) \vee \mathtt{EQ}(e_1, p_1) \vee \ldots \vee \mathtt{EQ}(e_n, p_n)$$

To satisfy the goal by sequential trial of the clauses for P, first replace the head, $P(p_1, \ldots, p_n)$, by the right-hand side of the first clause and seek a satisfying binding for the variables. If that fails, replace the head in the goal by the right-hand side of the second clause and try again, continuing to try each clause in turn until either the goal is satisfied or no more clauses remain to be tried.

Notice that in this description of a solution procedure, there are only two kinds of steps: (1) elaboration of atomic goals by the clauses they represent, and (2) calculation of a satisfying binding. The second step applies the comparison function when an atomic goal is an instance of the EQ predicate. This procedure calculates a satisfying binding incrementally, using pairwise comparisons. Comparisons corresponding to distinct atomic EQ goals may be performed in an arbitrary order chosen to optimize performance, or may be concurrent. The order in which alternate clauses are tried in seeking to satisfy a propositional goal is also arbitrary, but it should be sequential if the economy of depth-first search is desired.

The following semantic rules formalize the preceding description of an operational semantics for calculating a satisfying binding by depth-first construction of a deduction tree.

$$(\textit{conjunct}) \qquad \mathcal{L}[\![g_1, g_2]\!]\,\sigma\,h = \mathcal{L}[\![g_1]\!]\,\sigma\,(\lambda s.h\,(\lambda c.\,\textbf{let}\ (\_, \sigma_1) = s\,c\ \textbf{in}\ \mathcal{L}[\![g_2]\!]\,\sigma_1\,h))$$

$$(\textit{disjunct}) \qquad \mathcal{L}[\![cl_1 \mid cl_2]\!]\,\sigma\,h = \mathcal{L}[\![cl_1]\!]\,\sigma\,(\lambda s.\,h\,(\lambda c.\,s\,(\lambda(\_, \sigma').\mathcal{L}[\![cl_2]\!]\,\sigma\,h)))$$

In the rule (*conjunct*), the first goal is evaluated for its updated store, which is propagated to the second goal. The backtrack continuation, in case either goal fails to be satisfied, is that provided by the context h.

In the rule (*disjunct*), the first clause is tried and if successful, the store it calculates is passed to the context. If unsuccessful, the explicit backtrack continuation provided is that which discards the store on which the calculation failed, and tries the second clause, restoring the original store.

4 Conclusions

By relying on the structure of monads to enforce compositionality, we have been led to propose a new formulation of continuation semantics that appears to solve some problems in the denotation of programming languages with explicit control primitives. The use of *context* semantics for languages that involve backtracking or other means for abstraction of context restores the purely denotational property of the semantics that was mentioned in the introduction to this paper. The reader will have noticed that the higher-order, context semantics are even less

intuitive than are continuation semantics and are not suggested as a formalism for simpler programming languages.

In the pre-monad of composable contexts, the left identity law, (K1), fails, hence R is not a monad and the Kleisli composition cannot be assured. This fact would seem to call into question the whole approach if it had no further explanation. However, we conjecture that the failure of the left identity law is connected with the termination problem. If restricted to provably terminating computations, the left identity law appears to hold. This makes intuitive sense, for the semantics in the monad R are capable of accounting for computation of an expression relative to all contexts. The Kleisli composition must be valid not only locally, but with respect to all contexts in which the composed expressions may be evaluated. Since it is undecidable whether an arbitrary expression may or may not terminate, it is undecidable whether an arbitrary computation will reach its context.

Another approach uses order-enriched categories to provide a framework for the semantics of a rich variety of programming languages [9]. There too, the axioms of adjunction are weakened to those of pre-adjunction because of the termination problem for arbitrary computations. We have not yet explored the relation between that framework and the one presented here.

5 Acknowledgements

We wish to thank the program committee of the workshop for the interest shown in this work by several pages of technical comments. We are particularly grateful to Olivier Danvy, Jürgen Koslowski, Andrzej Filinksi and Robin Cockett for helpful and stimulating discussions. Philip Wadler's critique of an earlier version of our work was also very useful in the preparation of this paper.

References

[1] Borislav Agapiev. *Logic Programming—a Functional Approach.* PhD thesis, Oregon Graduate Institute, April 1992.

[2] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, June 1990.

[3] Olivier Danvy, Jürgen Koslowski, and Karoline Malmkjær. Compiling monads. Technical Report CIS-92-3, Kansas State University, Manhattan, Kansas, December 1991.

[4] Matthias Felleisen, Daniel Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[5] Andrzej Filinski. *Declarative Continuations and Categorical Duality.* M.s. thesis, Computer Science Department, University of Copenhagen, July 1989.

[6] Timothy Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, January 1990.

[7] Gary Lindstrom. Implementing logical variables on a graph reduction architecture. In Joseph Fasel and Robert M. Keller, editors, *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 382–400. Springer-Verlag, September 1986.

[8] Saunders MacLane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[9] C. E. Martin, C. A. R. Hoare, and He Jifeng. Pre-adjunctions in order-enriched categories. *Mathematical Structures in Computer Science*, 1(2):141–158, July 1991.

[10] Eugenio Moggi. Notions of computations and monads. *Information and Computation*, 93(1):55–92, July 1991.

[11] Philip Wadler. Comprehending monads. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78, 1990.