Relating Models of Backtracking

Mitchell Wand
College of Computer and Information Science
Northeastern University
360 Huntington Ave, Room 202 WVH
Boston, MA 02115
wand@ccs.neu.edu

Dale Vaillancourt
College of Computer and Information Science
Northeastern University
360 Huntington Ave, Room 202 WVH
Boston, MA 02115
dalev@ccs.neu.edu

ABSTRACT

Past attempts to relate two well-known models of backtracking computation have met with only limited success. We relate these two models using logical relations. We accommodate higher-order values and infinite computations. We also provide an operational semantics, and we prove it adequate for both models.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Denotational semantics, Operational semantics*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.6 [**Programming Techniques**]: Logic Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms

Languages, Theory

Keywords

two-continuation semantics, streams, adequacy, logical relations, monads

1. INTRODUCTION

There are two well-known models of backtracking computation: the stream model and the two-continuation model. The stream model represents backtracking computations by a stream of answers, and the two-continuation model uses a success continuation and a failure continuation. Hughes [8] defines a *backtracking monad* to be a monad with additional operations *disj* and *fail* satisfying five additional axioms. Both the stream model and the two-continuation model are backtracking monads, but this fact does not give us any deeper relation between the two.

Past attempts to relate these models have met with limited success: either the types do not work out, or the relation works in one direction but not the other, or the relation does not work for higher-order values.

We show how to relate the monads in a simple way. We relate streams of scalars using a representation inspired by Danvy et al. [4]. We then extend this to higher-order values using logical relations. At observable types this yields the identity relation, and so we get a denotational equivalence between the values in each model. We then provide an operational semantics and prove it adequate for both models.

2. BACKTRACKING COMPUTATION

We are interested in modeling a simply-typed call-by-value PCF that permits backtracking computation, such as a higher-order variant of Icon or Snobol. Such a language has types

$$\gamma ::= nat \;\mid\; \gamma_1 \to \gamma_2$$

and terms

$$\begin{array}{rl} t ::= & x \mid \lambda x.t \mid t_1\, t_2 \mid \mathit{succ} \mid c_n \\ \mid & \mathit{rec}\, f.(\lambda x.t) \mid t_1 \vee t_2 \mid \mathit{fail} \end{array}$$

where $c_0, c_1, c_2, \cdots$ are constants of type *nat* and *succ* is a constant of type $nat \to nat$. We allow recursive functions with $\mathit{rec}\ f.(\lambda x.t)$, and *fail* represents a computation which fails to produce a value. The computation $t_1 \vee t_2$ evaluates to t_1, and, if computation backtracks, t_1 is discarded in favor of t_2. It is a simple exercise to extend our results by considering an object-language with pairs, booleans, conditionals, and other features. We omit them to save space.

The canonical toy example in our calculus is *nats*, a program that evaluates to some natural number. The value of *nats* starts at 0, and each time computation backtracks to *nats* its value increases by 1.

$$\begin{array}{rcl} \mathit{nats\text{-}from} & = & \mathit{rec}\, f.(\lambda n.n \vee f(\mathit{succ}\ n)) \\ \mathit{nats} & = & \mathit{nats\text{-}from}(c_0) \end{array}$$

Note that this language is considerably more general than Prolog. We consider this relationship in section 7.

3. MONADIC METALANGUAGES

Following the lead of Moggi, we give a semantics to an object-language with computational effects by first translating it into a monadic metalanguage (for convenience, we abbreviate this to MML). An MML is just a typed λ-calculus that is equipped with the constants *unit* and *bind*. These two constants are computation constructors: they permit us to treat effectful computations as values of some abstract datatype. Intuitively, *unit* constructs a computation that immediately returns a value, and *bind* sequences two computations (where the second computation is parameterized on the results and effects of the first). We enrich MML with natural numbers, a successor function, and a family of fixed-point operators in order to explore the meaning of infinite computations such as *nats*. We then augment MML with two more computation constructors to handle backtracking.

3.1 Core MML

The core MML has types

$$\alpha, \beta ::= \sigma \;\mid\; \alpha \to \beta \;\mid\; T\alpha$$

where σ ranges over some scalar or base types (including *nat* and $\mathbb{1}$, the type whose only value is $()$). $\alpha \to \beta$ is a function type, and $T\alpha$ is an abstract type of "computations that produce values of type α." The terms are given by:

$$\begin{aligned} e \;::=\; & x \;\mid\; \lambda x.e \;\mid\; e_1\ e_2 \;\mid\; \mathit{succ} \;\mid\; c_n \\ \mid\; & \mathit{unit}_\alpha \;\mid\; \mathit{bind}_{\beta\alpha} \end{aligned}$$

This language is minimal, but it is a straightforward task to extend it with conditionals, pairs, booleans, a predecessor function, etc. The terms have types

$$\begin{array}{ll} \mathit{succ} & : \mathit{nat} \to \mathit{nat} \\ c_n & : \mathit{nat} \quad (\forall n \in \mathbb{N}) \\ \mathit{unit}_\alpha & : \alpha \to T\alpha \\ \mathit{bind}_{\beta\alpha} & : T\beta \to (\beta \to T\alpha) \to T\alpha \end{array}$$

for each pair of types α and β. unit_α takes a value v of type α to a computation which simply produces v when run. $\mathit{bind}_{\beta\alpha}\ c\ f$ denotes a computation that runs c and then runs the computation obtained by feeding c's result to f.

The equations that must hold in this metalanguage are the usual β and η laws of typed λ-calculus, the standard monad laws, and the successor law:

$$\begin{array}{l} \mathit{bind}\ (\mathit{unit}\ e)\ f = (f\ e) \\ \mathit{bind}\ e\ \mathit{unit} = e \\ \mathit{bind}\ (\mathit{bind}\ c\ f)\ g = \mathit{bind}\ c\ (\lambda v.\, \mathit{bind}\ (f\ v)\ g) \\ \mathit{succ}\ c_n = c_{n+1} \end{array}$$

The operations given so far capture finite sequences of computation, but in order to capture backtracking we must be able to construct computations that represent more than a single answer. We turn to Hughes [8] for two such constructors, and we add a family of fixed-point operators to express infinite computations.

3.2 Backtracking MML

Hughes defines a *backtracking monad* to be a monad with two additional constants *disj* and *fail* subject to additional axioms. The operation *fail* represents a computation that cannot produce further answers, and *disj* constructs a computation that produces answers generated by either of its constituent computations. The constants' types and axioms are:

$$\begin{array}{l} \mathit{fail}_\alpha \ : T\alpha \\ \mathit{disj}_\alpha : T\alpha \to T\alpha \to T\alpha \end{array}$$

$$\begin{array}{l} \mathit{disj}\ \ \mathit{fail}\ c = c \\ \mathit{disj}\ c\ \ \mathit{fail} = c \\ \mathit{disj}\ (\mathit{disj}\ c_1\ c_2)\ c_3 = \mathit{disj}\ c_1\ (\mathit{disj}\ c_2\ c_3) \\ \mathit{bind}\ (\mathit{disj}\ c_1\ c_2)\ f = \mathit{disj}\ (\mathit{bind}\ c_1\ f)\ (\mathit{bind}\ c_2\ f) \end{array}$$

The laws state that *fail* is a left and right unit of *disj*, *disj* is associative, and *bind* distributes through *disj*.

We also want to explore infinite computations such as *nats*, and therefore we add a family of fixed-point operators subject to the following type and axiom:

$$\begin{array}{l} \mathit{fix}_{\alpha\beta} : ((\alpha \to T\beta) \to (\alpha \to T\beta)) \to (\alpha \to T\beta) \\ \mathit{fix}\, e \ = e(\mathit{fix}\, e) \end{array}$$

We will consider a MML with these additional constants and axioms.

The interpretations of scalar types, function types, variables, abstractions, application, numbers, successor, and fixed-points of MML are fixed, but the interpretations of the computation types and monad operations depend on the representation of computations. We give the fixed interpretations here and address the representation of computations in the following section.

We interpret MML types as complete partial orders (not necessarily with bottom), and we write α^M for the interpretation of type α in a monad M. Scalar types are interpreted as discrete orders (i.e., sets whose order is given by an equivalence relation). For example, $\mathit{nat}^M = \mathbb{N}$, and the ordering is the identity relation. Function types are interpreted as cpo's of continuous functions:

$$\begin{aligned} \sigma^M &= D^\sigma \\ (\alpha \to \beta)^M &= [\alpha^M \to \beta^M] \end{aligned}$$

Computation types $T\alpha$ will always be interpreted as cpos with bottom. Our cpos need not have a bottom element, but the function space $[\alpha^M \to \beta^M]$ has a bottom element whenever β^M has one. We write $\sqsubseteq_{\alpha^M}$ for the ordering relation on the domain α^M.

If $\Gamma : \mathit{Var} \to$ Types, then Env_Γ, the set of Γ-consistent environments, is the set of all $\rho \in [\mathit{Var} \to \bigcup_\alpha \alpha^M]$ such that forall $x \in \mathit{dom}(\rho)$, $(\rho(x) \in \Gamma(x))$.

We write e^M for the interpretation of an expression e in the monad M. If $\Gamma \vdash e : \alpha$, then $e^M \in [\mathit{Env}_\Gamma \to \alpha^M]$. When e is closed, we take the liberty of writing e^M in place of $e^M\rho$ (where $\rho \in \mathit{Env}_\emptyset$). e^M is defined as follows:

$$\begin{aligned} x^M\ \rho &= \rho(x) \\ \lambda x.e^M\ \rho &= \lambda v.(e^M \rho[v/x]) \\ (e_0\ e_1)^M\ \rho &= (e_0^M \rho)\ (e_1^M \rho) \\ \mathit{succ}^M &= \lambda n.(n+1) \\ c_n^M &= n \\ \mathit{fix}^M &= \lambda f. \textstyle\bigsqcup_{n \in \omega} f^{(n)}(\bot) \end{aligned}$$

As usual, *fix* is only defined on functions whose domain has a bottom element, and our interpretations for computation types will always have bottom. This is why we assign the type given above for *fix*.

3.3 From the Object Language to the MML

Equipped with a metalanguage for backtracking, we now turn to the translation from the object language. Since our object-language is call-by-value, we use the standard call-by-value translation [12]. This translation maps object-language functions to metalanguage functions that consume values and produce computations. Dually, the translation of an application intuitively reads: "evaluate t_1, bind the value to f, evaluate t_2, bind the value to x, and finally evaluate fx". The translation on types is:

$$\begin{aligned}\ulcorner\gamma\urcorner &= T\gamma^* \\ \sigma^* &= \sigma \\ (\gamma_1 \to \gamma_2)^* &= \gamma_1^* \to T\gamma_2^*\end{aligned}$$

and the translation on terms is as follows:

$$\begin{aligned}\ulcorner x\urcorner &= \mathit{unit}\ x \\ \ulcorner \lambda x.t\urcorner &= \mathit{unit}\ (\lambda x.\ulcorner t\urcorner) \\ \ulcorner t_1\ t_2\urcorner &= \mathit{bind}\ \ulcorner t_1\urcorner\ (\lambda f.\,\mathit{bind}\ \ulcorner t_2\urcorner\ (\lambda x.f\ x)) \\ \ulcorner \mathit{succ}\urcorner &= \mathit{unit}\ \ \mathit{succ} \\ \ulcorner c_n\urcorner &= \mathit{unit}\ c_n \\ \ulcorner t_1 \vee t_2\urcorner &= \mathit{disj}\ \ulcorner t_1\urcorner\ulcorner t_2\urcorner \\ \ulcorner \mathit{fail}\urcorner &= \mathit{fail} \\ \ulcorner \mathit{rec}\ f.(\lambda x.t)\urcorner &= \mathit{unit}\ (\mathit{fix}\,\lambda f.\lambda x.\ulcorner t\urcorner)\end{aligned}$$

We demonstrate this translation on *nats* and simplify the result with β, η, the monad laws, and the fixed portion of MML's interpretation:

$$\begin{aligned}\ulcorner \mathit{nats}\urcorner &= \ulcorner \mathit{nats\text{-}from}(c_0)\urcorner \\ &= \ulcorner(\mathit{rec}\ f.(\lambda n.n \vee f(\mathit{succ}\ n)))\ c_0\urcorner \\ &= \mathit{bind}\ \ulcorner \mathit{rec}\ f.(\lambda n.n \vee f(\mathit{succ}\ n))\urcorner \\ &\qquad \lambda f.\,\mathit{bind}\ \ulcorner c_0\urcorner\ (\lambda x.f\ x) \\ &= \mathit{bind}\ (\mathit{unit}\ (\mathit{fix}\,\lambda f.\lambda n.\ulcorner n \vee f(\mathit{succ}\ n)\urcorner)) \\ &\qquad \lambda f.\,\mathit{bind}\ (\mathit{unit}\ c_0)\ (\lambda x.f\ x) \\ &= \mathit{bind}\ (\mathit{unit}\ c_0) \\ &\qquad (\mathit{fix}\,\lambda f.\lambda n.\ulcorner n \vee f(\mathit{succ}\ n)\urcorner) \\ &= (\mathit{fix}\,\lambda f.\lambda n.\ulcorner n \vee f(\mathit{succ}\ n)\urcorner)\ c_0 \\ &= (\mathit{fix}\,\lambda f.\lambda n.\ \mathit{disj}\ (\mathit{unit}\ n)\ \ulcorner f(\mathit{succ}\ n)\urcorner)\ c_0 \\ &= \mathit{disj}\ (\mathit{unit}\ c_0) \\ &\qquad \mathit{bind}\ (\mathit{unit}\ (\mathit{fix}\,\lambda f.\lambda n.\ulcorner n \vee f(\mathit{succ}\ n)\urcorner)) \\ &\qquad\quad \lambda g.\,\mathit{bind}\ \ulcorner \mathit{succ}\ c_0\urcorner\ (\lambda x.gx) \\ &= \mathit{disj}\ (\mathit{unit}\ c_0) \\ &\qquad \mathit{bind}\ \ulcorner \mathit{rec}\ f.(\lambda n.n \vee f(\mathit{succ}\ n))\urcorner \\ &\qquad\quad \lambda g.\,\mathit{bind}\ \ulcorner c_1\urcorner\ (\lambda x.gx) \\ &= \mathit{disj}\ (\mathit{unit}\ c_0)\ \ulcorner \mathit{nats\text{-}from}(c_1)\urcorner\end{aligned}$$

We see that the result is a computation that produces either c_0 or the result of evaluating $\ulcorner \mathit{nats\text{-}from}(c_1)\urcorner$, as expected.

4. TWO BACKTRACKING MONADS

There are two well-known models of backtracking computation, and they are, in fact, both backtracking monads. The first one models backtracking computations as a stream of answers, and the monad operations correspond to standard operations on streams. The second model represents backtracking computations as procedures that consume two continuations. The first continuation is a success continuation; it consumes an answer and a second continuation that can be invoked to get another answer. A computation will invoke the second continuation when it has no further answers of its own to produce.

4.1 Stream Monad

The stream monad S is given by the following domain and interpretations for types and the monad operations. The domain $S(A)$ of streams over elements of A is defined as follows:

$$\begin{aligned}S(A) = \bot &\cup \{\langle a_1,\ldots,a_n\rangle \ \mid\ n \geq 0, a_i \in A\} \\ &\cup \{\langle a_1,\ldots,a_n\rangle \,\hat{}\, \bot \ \mid\ n \geq 1, a_i \in A\} \\ &\cup \{\langle a_1,\ldots\ \rangle \ \mid\ a_i \in A\}\end{aligned}$$

where $\hat{}$ is stream append (with $w_1 \,\hat{}\, w_2 = w_1$ when w_1 is infinite or $\bot$). The element $\bot$ represents a computation that runs infinitely without deciding if there are any answers. The element $\langle a_1,\ldots,a_n\rangle$ represents a finite stream with exactly n elements. The element $\langle a_1,\ldots,a_n\rangle \,\hat{}\, \bot$ represents a computation that has determined the first n elements in the stream but runs infinitely when trying to decide if there are any more answers. Last, $\langle a_1,\ldots\ \rangle$ represents an infinite stream.

The ordering on streams is a prefix ordering, $\sqsubseteq_{S(A)}$, defined as follows.
For all $s \in S(A)$:

$$\bot \sqsubseteq_{S(A)} s$$

$$\begin{array}{c}\langle a_1,a_2,\ldots,a_n\rangle \sqsubseteq_{S(A)} \langle b_1,b_2,\ldots,b_n\rangle \\ \Longleftrightarrow \\ a_i \ \sqsubseteq_A\ b_i \qquad (1 \leq i \leq n)\end{array}$$

$$\begin{array}{c}\langle a_1,a_2,\ldots,a_n\rangle \,\hat{}\, \bot \sqsubseteq_{S(A)} \langle b_1,b_2,\ldots,b_n\rangle \,\hat{}\, s \\ \Longleftrightarrow \\ a_i \ \sqsubseteq_A\ b_i \qquad (1 \leq i \leq n)\end{array}$$

$$\begin{array}{c}\langle a_1,a_2,\ldots\rangle \sqsubseteq_{S(A)} \langle b_1,b_2,\ldots\rangle \\ \Longleftrightarrow \\ a_i \ \sqsubseteq_A\ b_i \qquad (\forall i \in \omega)\end{array}$$

The first clause says that $\bot$ is related to all streams. The second clause says that two finite streams are related if and only if they are the same length, and the elements are pointwise related. The third clause relates streams whose tail is undefined to streams which are "at least as defined," pointwise. Infinite streams are related if and only if their elements are pointwise related.[1]

We write α^S or e^S for the interpretation of type α or metalanguage term e in the monad S. Scalar and function types are interpreted as stated in section 3.2, and computation types are interpreted as stream domains:

$$(T\alpha)^S = S(\alpha^S)$$

The constant *unit* constructs a singleton stream, *bind* is map-append, *fail* is the empty stream, and *disj* is stream-append:

$$\begin{aligned}\mathit{unit}^S &= \lambda a.\langle a\rangle \\ \mathit{bind}^S\ \bot\ f &= \bot \\ \mathit{bind}^S\ \langle a_1,\ldots,a_n\rangle\ f &= f(a_1)\,\hat{}\ \ \ldots\ \ \hat{}\, f(a_n) \\ \mathit{bind}^S\ \langle a_1,\ldots,a_n\rangle \,\hat{}\, \bot\ f &= f(a_1)\,\hat{}\ \ \ldots\ \ \hat{}\, f(a_n)\,\hat{}\,\bot \\ \mathit{bind}^S\ \langle a_1,\ldots\ \rangle\ f &= f(a_1)\,\hat{}\ \ \ldots \\ \mathit{fail}^S &= \langle\rangle \\ \mathit{disj}^S &= \lambda c_1 c_2.(c_1 \,\hat{}\, c_2)\end{aligned}$$

The stream monad yields an interpretation of *nats* that is equivalent to the infinite stream of natural numbers in

[1]Note that $\langle 1,2\rangle \sqsubseteq_{S(\mathbb{N})} \langle 1,2,3\rangle$ does not hold under this ordering. The following does hold: $\langle 1,2\rangle \,\hat{}\, \bot \sqsubseteq_{S(\mathbb{N})} \langle 1,2,3\rangle$.

ascending order:

$$\begin{aligned}\ulcorner nats \urcorner^S &= \ulcorner nats\text{-}from(c_0)\urcorner^S \\ &= (disj\ (unit\ c_0)\ \ulcorner nats\text{-}from(c_1)\urcorner)^S \\ &= disj^S\ (unit\ c_0)^S\ \ulcorner nats\text{-}from(c_1)\urcorner^S \\ &= \langle c_0\rangle\ \hat{}\ \ulcorner nats\text{-}from(c_1)\urcorner^S\end{aligned}$$

4.2 Two-Continuation Monad

The two-continuation monad K is slightly more involved than the stream monad; it is constructed as follows. Let O be a domain of observations; that is, O will contain the values that we can observe when a computation terminates. A computation must terminate in order to observe a result. We therefore assume that O does not contain a bottom element signifying non-termination. Sensible choices for O include any domain of scalars and the domain of streams over scalars (consider the stream of answers printed in the Prolog REPL). For the latter, we set $O = S(nat^S) - \{\bot\}$. We write $O_\bot$ to denote O with a bottom element (re-)adjoined. The process of choosing an O is coupled with the process of choosing initial continuations in which to run computations. We explore this dimension towards the end of section 6. Now we define

$$K(A) = [[A \to [\mathbb{1} \to O_\bot] \to O_\bot] \to [\mathbb{1} \to O_\bot] \to O_\bot]$$

The domain $[\mathbb{1} \to O_\bot]$ is the domain of failure continuations, and $[A \to [\mathbb{1} \to O_\bot] \to O_\bot]$ is the domain of success continuations. A success continuation takes an answer in A and a failure continuation. If the computation has an answer to produce, it will send it to the success continuation. It will also pass along a failure continuation that, if invoked, will continue computation with the next answer. If a computation cannot produce any more answers, it will invoke its failure continuation to produce an answer.

We interpret scalar and function types as in the stream monad, but the computation types are interpreted using K instead of S:

$$\begin{aligned}\sigma^K &= D^\sigma \\ (\alpha \to \beta)^K &= [\alpha^K \to \beta^K] \\ (T\alpha)^K &= K(\alpha^K)\end{aligned}$$

The monad operations are interpreted by:

$$\begin{aligned}unit^K &= \lambda a.(\lambda\kappa\phi.\kappa a\phi) \\ bind^K &= \lambda cf.(\lambda\kappa\phi.c\ (\lambda b\phi.fb\kappa\phi)\ \phi) \\ fail^K &= \lambda\kappa\phi.\phi() \\ disj^K &= \lambda c_1c_2.(\lambda\kappa\phi.c_1\kappa(\lambda().c_2\kappa\phi))\end{aligned}$$

The constant $unit^K$ constructs a computation that succeeds just once with the given value a, and $bind^K$ extends the success continuation of c by applying f to c's result in the original continuations. Note that when c fails, the entire $bind^K$ expression fails; this is useful for modeling conjunction. $fail^K$ is a computation which simply invokes its failure continuation, and $disj^K$ evaluates to its first computation c_1 in an extended failure continuation. Should c_1 fail, c_2's answers will be sent to the success continuation. The argument to the failure continuation in the last case must be $()$, the single element of type $\mathbb{1}$, hence we write $(\lambda().\ldots)$ in the definition of $disj^K$.

The two-continuation monad yields an interpretation of *nats* that, when handed a success and a failure continuation, will pass c_0 and a new failure continuation to the success continuation. If invoked, the new failure continuation will produce the rest of the answers.

$$\begin{aligned}\ulcorner nats\urcorner^K\ \kappa\ \phi &= \ulcorner nats\text{-}from(c_0)\urcorner^K\ \kappa\ \phi \\ &= (disj\ (unit\ c_0)\ \ulcorner nats\text{-}from(c_1)\urcorner)^K\ \kappa\ \phi \\ &= disj^K\ (unit\ c_0)^K\ \ulcorner nats\text{-}from(c_1)\urcorner^K\ \kappa\ \phi \\ &= (\lambda\kappa\phi.\kappa\ c_0\ \phi)\ \kappa\ (\lambda().\ulcorner nats\text{-}from(c_1)\urcorner^K\ \kappa\ \phi) \\ &= \kappa\ c_0\ (\lambda().\ulcorner nats\text{-}from(c_1)\urcorner^K\ \kappa\ \phi)\end{aligned}$$

5. RELATING THE TWO MODELS

We relate the two semantics with a logical relation. A logical relation R consists of a type-indexed family of relations R_α for each type α in the language. At scalar type we choose R_σ to be the identity relation on D^σ because the models share a representation of scalars. We then define the relation at higher-order types inductively. The first well-known use of this technique is in Plotkin [13], though he credits Troelstra [18]. Mitchell [11] provides a thorough account of logical relations.

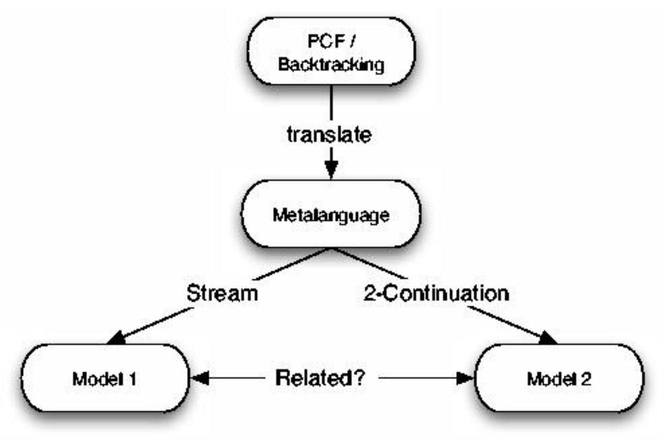

Figure 1: A logical relation relates the models of two semantics.

Our logical relation R is defined as follows. For each type α in the metalanguage, we define a relation $R_\alpha \subseteq (\alpha^S \times \alpha^K)$ by induction on α, as follows:

1. R_σ is the identity relation on the shared interpretation of scalar types σ.
2. $R_{\alpha\to\beta} = \{(f,g)\ \mid\ \forall(a,a') \in R_\alpha.(fa, ga') \in R_\beta\}$
3. $R_{T\alpha}$ is the smallest relation on $S(\alpha^S) \times K(\alpha^K)$ such that:
   - $(\bot,\bot) \in R_{T\alpha}$.
   - $(\langle\rangle, \lambda\kappa\phi.\phi()) \in R_{T\alpha}$.
   - if $(a_1,a_1'),\ldots,(a_n,a_n') \in R_\alpha$, then
     - $(\langle a_1,\ldots,a_n\rangle,$ $(\lambda k.(k\ a_1') \star (k\ a_2') \star \cdots \star (k\ a_n'))) \in R_{T\alpha}$ $(n \geq 1)$
     - $(\langle a_1,\ldots,a_n\rangle\ \hat{}\ \bot,$ $(\lambda k.(k\ a_1') \star (k\ a_2') \star \cdots \star (k\ a_n') \star \bot)) \in R_{T\alpha}$
   - $R_{T\alpha}$ is closed under limits of ω-chains in $(T\alpha)^S \times (T\alpha)^K$.

$\star$ is a right-associative composition-like operator; it is defined by

$$f \star g \equiv \lambda x.f(\lambda().gx)$$

We use $\star$ to "compose" two functions that each consume a failure continuation:

$$\lambda\kappa.(\kappa\ a) \star (\kappa\ b) = \lambda\kappa\phi.\kappa a(\lambda().\kappa b\phi)$$

Lemma 1 states that $\star$ amounts to a representation of stream-append.

LEMMA 1. *If* $(c_1, c_1'), (c_2, c_2') \in R_{T\alpha}$, *then* $(c_1\ \hat{}\ c_2, (\lambda k.(c_1'\ k) \star (c_2'\ k))) \in R_{T\alpha}$

Proof: By induction on the structure of c_1 and closure of $R_{T\alpha}$ under limits of ω-chains. We illustrate the case where c_1 is infinite. c_1 and c_1' are both limits of ω-chains:

$$\begin{aligned} c_1 &= \bigsqcup_{n\in\omega} s_n, \text{where } s_n \sqsubseteq s_{n+1} \\ c_1' &= \bigsqcup_{n\in\omega} s_n', \text{where } s_n' \sqsubseteq s_{n+1}' \end{aligned}$$

where

$$\begin{aligned} s_n &= \langle a_1, \ldots, a_n\rangle\ \hat{}\ \bot \\ s_n' &= \lambda k.(k\ a_1') \star \cdots \star (k\ a_n') \star \bot \end{aligned}$$

Since $(c_1, c_1') \in R_{T\alpha}$ and $R_{T\alpha}$ is closed under limits of ω-chains of related values, we know

$$\forall n \in \omega.(s_n, s_n') \in R_{T\alpha}$$

It is easy to show, by induction that for all n,

$$(s_n\ \hat{}\ c_2, \lambda\kappa.(s_n'\ \kappa) \star c_2') \in R_{T\alpha}$$

It is easy to verify that $s_n\ \hat{}\ c_2$ and $\lambda\kappa.(s_n'\kappa) \star c_2'$ are both ω-chains. We note that $\star$ is continuous since it is definable using just abstraction and application. Since $R_{T\alpha}$ is closed under limits of ω-chains of related values, we deduce

$$\begin{aligned} &(c_1\ \hat{}\ c_2, \lambda k.(c_1'\ \kappa) \star (c_2'\ \kappa)) \\ &= \textstyle\bigsqcup_{n\in\omega}(s_n\ \hat{}\ c_2, \lambda\kappa.(s_n'\ \kappa) \star (c_2'\ \kappa)) \in R_{T\alpha} \end{aligned}$$

■

LEMMA 2. *R is a logical relation.*

Proof: We must show that the denotations of each constant are related. We illustrate the case for *disj*.

We must show $(disj^S, disj^K) \in R_{T\alpha\to T\alpha\to T\alpha}$. Assume (s, s'), $(t, t') \in R_{T\alpha}$. Then

$$\begin{aligned} disj^S\ s\ t &= s\ \hat{}\ t \\ disj^K\ s'\ t' &= \lambda\kappa.(s'\kappa) \star (t'\kappa) \end{aligned}$$

and so, by Lemma 1, $(disj^S\ s\ t, disj^K\ s'\ t') \in R_{T\alpha}$. But since we chose (s, s') and (t, t') arbitrarily, we use the definition of logical relation at function type twice to conclude that $(disj^S, disj^K) \in R_{T\alpha\to T\alpha\to T\alpha}$. ■

Two environments ρ and ρ' are logically related with respect to some type environment Γ when their domains agree and they map variables to related values:

$$\begin{aligned} (\rho, \rho') \in R_\Gamma \iff & dom(\rho) = dom(\rho') = dom(\Gamma) \\ & \wedge \forall x \in dom(\Gamma).(\rho(x), \rho'(x)) \in R_{\Gamma(x)} \end{aligned}$$

THEOREM 1. *If* $\Gamma \vdash e : \alpha$ *and* $(\rho, \rho') \in R_\Gamma$, *then* $(e^S\rho, e^K\rho') \in R_\alpha$.

Proof: Immediate from Lemma 2 and the Basic Lemma of Logical Relations [11]. ■

THEOREM 2. *Let e be a closed backtracking MML term of type* $T\sigma$. *Then* $e^S = (e^K\ \ cons\ \ nil)$, *where*

$$\begin{aligned} cons &= \lambda a\phi.\langle a\rangle\ \hat{}\ \phi() \\ nil &= \lambda().\langle\rangle \end{aligned}$$

Proof: Let $O = S(D^\sigma) - \{\bot\}$. If e is a closed backtracking MML term of type $T\sigma$, and $e^S = \langle a_1, \ldots, a_n\rangle$, then $e^K = (\lambda k.(k\ a_1) \star (k\ a_2) \star \cdots \star (k\ a_n))$. So

$$e^K\ \ cons = \lambda s.\ cons\ a_1\ (cons\ a_2\ (cons\ a_3\ \ldots\ (cons\ a_n\ s)\ldots))$$

and therefore

$$\begin{aligned} & e^K\ \ cons\ \ nil \\ = \ & cons\ a_1\ (cons\ a_2\ (cons\ a_3 \ldots (cons\ a_n\ \ nil)\ldots)) \\ = \ & \langle a_1, \ldots, a_n\rangle \\ = \ & e^S \end{aligned}$$

The other cases follow by continuity. ■

Since this is a denotational equivalence, an operational semantics for metalanguage terms that is adequate for one denotational semantics will be adequate for the other. We explore this result in section 6.

From Failure Continuations to Failure Computations. Another model K' can be obtained by replacing the failure continuations $[1 \to O_\bot]$ by failure computations $O_\bot$, so $K'(A) = (A \to O_\bot \to O_\bot) \to O_\bot \to O_\bot$, and

$$\begin{aligned} unit^{K'} &= \lambda a.(\lambda\kappa\phi.\kappa a\phi) \\ bind^{K'} &= \lambda cf.(\lambda\kappa\phi.c\ (\lambda b\phi.fb\kappa\phi)\ \phi) \\ fail^{K'} &= \lambda\kappa\phi.\phi \\ disj^{K'} &= \lambda c_1c_2.(\lambda\kappa\phi.c_1\ \kappa\ (c_2\kappa\phi))) \end{aligned}$$

The logical relation is obtained by replacing the composition-like operator $\star$ with standard functional composition $\circ$ in the definition of $R_{T\alpha}$, to get

$$(\langle a_1, \ldots, a_n\rangle, (\lambda k.(k\ a_1') \circ (k\ a_2') \circ \cdots \circ (k\ a_n'))) \in R_{T\alpha}$$

etc.

This leads to a version of the story in which one passes along expressions to be evaluated in case of failure rather than procedures that must be invoked to retrieve more answers.

6. AN ADEQUATE OPERATIONAL SEMANTICS

We supply an operational semantics for the backtracking MML in three stages. First, we define a monadic metalanguage mPCF that is simpler than the backtracking MML. We then translate the latter into mPCF in such a way that the K semantics is equal to this translation followed by mPCF's semantics, L. Finally, we provide an adequate operational semantics for mPCF. This tactic greatly simplifies both the operational semantics and the proof of adequacy that follows. The adequacy proof demonstrates that the given operational semantics is adequate with respect to the L semantics, and therefore it is adequate for the K semantics. Since the two-continuation semantics and the stream semantics are logically related, the adequacy result extends from K to S.

6.1 mPCF

mPCF is a MML that captures nontermination as a computational effect. We include a type *Snat* for streams of natural numbers and stream constructors *cons* and *nil* with these types[2]:

$$\begin{aligned} nil &: Snat \\ cons &: nat \to L(Snat) \to Snat \end{aligned}$$

The types and terms are generated by the following grammar:

$$\begin{aligned} \tau \; &::= \sigma \mid \tau_1 \to \tau_2 \mid L\tau \mid Snat \\ M \; &::= x \mid \lambda x.M \mid M_1 M_2 \mid succ \mid c_n \\ &\quad \mid unit \mid bind \mid fix \mid cons \mid nil \end{aligned}$$

We have renamed the type constructor T to L to avoid confusion between backtracking MML and mPCF computation types. We write M^τ for the set of closed terms of type τ. We interpret mPCF in the lift monad. The interpretations for scalar and function types are as before, and L interprets computation types as lifted domains. Finally, L interprets *Snat* as streams of natural numbers.

$$\begin{aligned} \sigma^L &= D^\sigma \\ (\tau_1 \to \tau_2)^L &= [\tau_1^L \to \tau_2^L] \\ (L\tau)^L &= (\tau^L)_\perp \\ (Snat)^L &= S(nat^S) - \{\perp\} \end{aligned}$$

The interpretations for variables, abstraction, application, *succ*, c_n, and *fix* remain as they are in the monads S and K. In L, *unit* takes a value to a computation that terminates with that value, and *bind* encodes strict application:

$$\begin{aligned} unit^L &= \lambda d.\, lift(d) \\ bind^L &= \lambda cf.\, case\, c\, of \perp \to \perp \;\mid\; lift(d) \to f(d) \end{aligned}$$

Observe that $(L(Snat))^L = (S(\mathbb{N}) - \{\perp\})_\perp = S(\mathbb{N})$. Therefore we can define nil^L and $cons^L$ in the usual way.

$$\begin{aligned} nil^L &= \langle\rangle \in (L(Snat))^L \\ cons^L &= \lambda xy.\langle x\rangle \,\hat{}\, y \in (L(Snat))^L \end{aligned}$$

6.2 Translating from MML to mPCF

Our translation from MML to mPCF is straightforward; we simply implement the two-continuation semantics in mPCF. The translation on types is crafted such that $\lceil\alpha\rceil^L = \alpha^K$. Let Ω be any mPCF type such that $\Omega^L = O$ (e.g., $\Omega = Snat$ or nat).

$$\begin{aligned} \lceil\sigma\rceil &= \sigma \\ \lceil\alpha \to \beta\rceil &= \lceil\alpha\rceil \to \lceil\beta\rceil \\ \lceil T\alpha\rceil &= (\lceil\alpha\rceil \to (1 \to L\Omega) \to L\Omega) \to (1 \to L\Omega) \to L\Omega \end{aligned}$$

$$\begin{aligned} \lceil unit\rceil &= \lambda a.\lambda\kappa\phi.\kappa a\phi \\ \lceil bind\rceil &= \lambda cf.\lambda\kappa\phi.c\;(\lambda b\phi.fb\kappa\phi)\;\phi \\ \lceil disj\rceil &= \lambda c_1c_2.\lambda\kappa\phi.c_1\;\kappa\;(\lambda().c_2\;\kappa\;\phi) \\ \lceil fail\rceil &= \lambda\kappa\phi.\phi() \\ \lceil fix\rceil &= fix \\ \lceil succ\rceil &= succ \\ \lceil c_n\rceil &= c_n \\ \lceil x\rceil &= x \\ \lceil\lambda x.e\rceil &= \lambda x.\lceil e\rceil \\ \lceil e_1\; e_2\rceil &= \lceil e_1\rceil\;\lceil e_2\rceil \end{aligned}$$

As in section 4.2, we restrict the argument of failure continuations to be (). Also, note that this translation never produces any *unit* or *bind* terms in mPCF.

The K interpretation of types is preserved by $\lceil\cdot\rceil$:

LEMMA 3. *For any backtracking MML types α, we have:*

$$\alpha^K = \lceil\alpha\rceil^L$$

Proof: By a trivial induction on α. ■

The K interpretation of terms is also preserved by $\lceil\cdot\rceil$:

THEOREM 3. *For MML terms e, if $\Gamma \vdash e : \alpha$ and ρ is Γ-consistent, then $e^K\rho = \lceil e\rceil^L\rho$*

Proof: By induction on the structure of e. No translation on ρ is necessary, due to Lemma 3. ■

In the next section we give an operational semantics for mPCF and prove that it is adequate with respect to L. Theorem 3 is significant in this context because it says that we can evaluate an MML program by compiling it into mPCF and applying mPCF's evaluator.

6.3 Operational Semantics

We provide a deterministic call-by-name operational semantics for mPCF. The grammar for values V and evaluation contexts E is:

$$\begin{aligned} V \; &::= c_n \mid \lambda x.M \mid unit\; M \mid cons\; V\, M \mid nil \\ &\quad \mid cons\; V \mid bind\; V \\ &\quad \mid succ \mid unit \mid bind \mid fix \\ E \; &::= [] \mid EM \mid bind\, E \mid succ\, E \mid cons\, E \end{aligned}$$

We write V^τ to refer to the set of closed values of type τ. Reduction halts if the term reaches any constant, an abstraction, or *unit* of some term.[3] Evaluation is permitted in an empty context, in the operator position of an application, in the first argument of *bind*, and in the operand position of successor. (We permit the latter because the type of *succ* precludes the possibility that its argument diverges.)

The reduction for application is standard β reduction, and we augment this rule with reductions for successor and the standard reduction for a call-by-name fixed-point operator. The reduction and context for *bind* encodes strict application.

$$\begin{aligned} (\lambda x.M_1)\; M_2 \;&\to\; M_1[M_2/x] \\ succ\; c_n \;&\to\; c_{n+1} \\ fix\, M \;&\to\; M(fix\; M) \\ bind\; (unit\; M_1)\; M_2 \;&\to\; M_2\; M_1 \\ bind\; (bind\; M_1\; M_2)\; M_3 \;&\to\; bind\; M_1\; \lambda v.\, bind\; (M_2 v)\; M_3 \end{aligned}$$

It is straightforward to verify that this semantics satisfies safety and progress.

[2]Our results can be extended to streams $S\alpha$ of arbitrary type, but we do not need the additional generality here.

[3]Remember that *unit*, in the context of mPCF, means the computation has terminated with some result.

6.4 Adequacy

Now we prove the adequacy of our operational semantics with respect to the lift semantics, L. The main result is theorem 4. It states that a closed term M reduces to a value in the operational semantics if and only if M does not denote $\bot$. This property is of particular interest when M is of an observable type; we can use it to show that these two semantics agree on the meaning of terms whose values are observable. Our presentation follows that of Winskel [22].

Assume M is a closed mPCF term. Define operational convergence (1) for M and denotational convergence (2) for M as follows:

$$\begin{aligned}
M\downarrow &\iff \exists v.(M \to^* v) && (1)\\
M\Downarrow &\iff \vdash M:\sigma && (2)\\
&\quad\vee\ \ \vdash M:\tau_1\to\tau_2 &&\\
&\quad\vee\ \ \vdash M: \mathit{Snat} &&\\
&\quad\vee\ \ \vdash M: L\tau\ \wedge\ \exists d\in\tau^L.M^L=\mathit{lift}(d) &&
\end{aligned}$$

Theorem 4 (Adequacy). *If M is a closed mPCF term, then $M\downarrow \iff M\Downarrow$*

Proof:

- $\Longrightarrow$: The forward implication follows directly from Lemma 4 below. □
- $\Longleftarrow$: The reverse implication follows from Lemma 6 below and considering only closed terms. □

■

Note that if M is not of computation type, then $M\Downarrow$, and hence M reduces to a value v.

Lemma 4. *For closed mPCF terms M, we have*

$$M\to^* v \implies M^L = v^L$$

Proof: By induction on the length of the reduction sequence. ■

In order to prove the converse, we define an approximation relation $\le^o_\tau$ between denotations and closed values for each type τ. In order to express that a denotation approximates a closed term, we use $\le_\tau$. Define $(\le^o_\tau)\subseteq(\tau^L\times V^\tau)$ and $(\le_\tau)\subseteq(\tau^L\times M^\tau)$ by induction on τ:

$$\begin{array}{lll}
d\le^o_\sigma & v & \iff d\in D^\sigma\ \wedge\ d=v^L\\
d\le^o_{\tau_1\to\tau_2} & v & \iff \forall(d',M')\in(\le_{\tau_1}).d(d')\le_{\tau_2}(v\ M')\\
d\le^o_{L\tau} & \mathit{unit}(M) & \iff d=\bot\ \vee\ d=\mathit{lift}(d')\wedge d'\le_\tau M\\
d\le^o_{S(nat)} & v & \iff d=\langle\rangle \text{ and } v=\mathit{nil}\\
 & & \quad\ \vee\ \ d=\langle a\rangle\ \hat{}\ b \text{ and } v=\mathit{cons}\ v'\ M \text{ s.t.}\\
 & & \qquad a\le^o_{nat} v' \text{ and } b\le_{L(S\,nat)} M\\
d\le_\tau & M & \iff \exists v.(M\to^* v\wedge d\le^o_\tau v)
\end{array}$$

The recursion in the definition of $\le^o_{Snat}$ and $\le_{Snat}$ is to be interpreted coinductively. That is, $\le^o_{Snat}$ and $\le_{Snat}$ are defined to be the *largest* relations satisfying the given conditions. This allows $d\le^o_{Snat} v$ to hold even in cases when d is an infinite stream.

We note in Lemma 5 that the approximation relation is closed under limits of ω-chains on the left.

Lemma 5. *Let $\vdash M:\tau$. If $d_0\sqsubseteq d_1\sqsubseteq\ldots\sqsubseteq d_n\sqsubseteq\ldots$ is an ω-chain in τ^L and $\forall n\in\omega.d_n\le_\tau M$, then $\bigsqcup_{n\in\omega} d_n\le_\tau M$.*

Proof: By structural induction on τ and coinduction at *Snat*. ■

Lemma 6 is the main lemma for the adequacy proof; it is used to show the reverse implication in the adequacy theorem. Let θ range over substitutions of closed terms for variables, and define $\rho\le_\Gamma\theta \iff \forall x\in \mathit{dom}(\Gamma).\rho(x)\le_{\Gamma(x)}\theta(x)$. Let $M\cdot\theta$ denote the simultaneous, capture-free substitution of the free variables in M for their values in θ.

Lemma 6. *If $\Gamma\vdash M:\tau$ and $\rho\le_\Gamma\theta$, then $M^L\rho\le_\tau M\cdot\theta$.*

Proof: By induction on the structure of M.

- $M\equiv x$: $x^L\rho=\rho(x)\le_\tau\theta(x)=x\cdot\theta$ holds since $\rho\le_\Gamma\theta$. □

- $M\equiv\lambda x.M_0$:
  We must show
  $$(\lambda x.M_0)^L\rho\le^o_{\tau_1\to\tau_2}(\lambda x.M_0)\cdot\theta$$
  Fix an arbitrary $(d',e')\in(\le_{\tau_1})$. By definition of $\le^o_{\tau_1\to\tau_2}$, we must demonstrate
  $$\begin{aligned}
  &((\lambda x.M_0)^L\rho)(d')\le_{\tau_2}(((\lambda x.M_0)\cdot\theta)e')\\
  \iff\ &M_0^L\rho[d'/x]\le_{\tau_2}(((\lambda x.M_0)\cdot\theta)e')\\
  \iff\ &\exists w.(((\lambda x.M_0\cdot\theta)\ e')\to^* w\wedge M_0^L\rho[d'/x]\le^o_{\tau_2} w)
  \end{aligned}$$
  Following the definition of $\to$, the application on the left-hand side takes a step and the goal becomes:
  $$\exists w.(M_0\cdot\theta[e'/x]\to^* w\wedge M_0^L\rho[d'/x]\le^o_{\tau_2} w)$$
  We have that $\Gamma,x:\tau_1\vdash M_0:\tau_2$ and $\rho[d'/x]\le_{\Gamma,x:\tau_1}\theta[e'/x]$, and so, inductively, we can assume
  $$M_0^L\rho[d'/x]\le_{\tau_2} M_0\cdot\theta[e'/x]$$
  Unfolding the definition of $\le_{\tau_2}$ completes the proof:
  $$\exists w.(M_0\cdot\theta[e'/x]\to^* w\wedge M_0^L\rho[d'/x]\le^o_{\tau_2} w)$$
  □

- $M\equiv M_1\ M_2$:
  Let $\Gamma\vdash M_1:\tau_1\to\tau_2$, $\Gamma\vdash M_2:\tau_1$, and $\rho\le_\Gamma\theta$. Inductively, assume:
  $$\begin{aligned}
  M_1^L\rho&\le_{\tau_1\to\tau_2} M_1\cdot\theta && (3)\\
  M_2^L\rho&\le_{\tau_1}\qquad M_2\cdot\theta && (4)
  \end{aligned}$$
  (3) implies there is a closed value v such that
  $$M_1\cdot\theta\to^* v\wedge(M_1^L\rho)\le^o_{\tau_1\to\tau_2} v$$
  and unwinding the definition of $\le^o_{\tau_1\to\tau_2}$ gives
  $$\forall(d',M')\in(\le_{\tau_1}).(M_1^L\rho)(d')\le_{\tau_2}(v\ M')$$
  Applying (4) we infer
  $$\begin{aligned}
  &(M_1^L\rho)(M_2^L\rho)\quad\le_{\tau_2}\quad(v\ M_2\cdot\theta)\\
  \equiv\ &(M_1M_2)^L\rho\quad\le_{\tau_2}\quad(v\ M_2\cdot\theta) && (5)
  \end{aligned}$$
  By (5):
  $$\exists w.(v\ M_2\cdot\theta)\to^* w\wedge(M_1M_2)^L\rho\le^o_{\tau_2} w$$
  But $M_1\to^* v$ and $(v\ M_2\cdot\theta)\to^* w$ implies
  $$M_1M_2\cdot\theta\to^* w$$

and therefore

$$(M_1 M_2)^L \rho \leq_{\tau_2} M_1 M_2 \cdot \theta$$

$\square$

- $M \equiv c_n$:
  Follows trivially from the definitions of $(\leq_{nat})$ and $(\leq^o_{nat})$. $\square$

- $M \equiv succ$:
  Assume $M_0^L = n \leq_{nat} M_0$. Then

  $$\exists v. M_0 \rightarrow^* v \wedge n \leq^o_{nat} v$$

  By definition of $\leq^o_{nat}$, $v = c_n$.

  We must show

  $$\begin{aligned} & (succ\, M_0)^L = (n+1) \leq_{nat} succ\, M_0 \\ \iff & \exists w.(succ\, M_0 \rightarrow^* w \wedge (n+1) \leq^o_{nat} w) \end{aligned}$$

  Since $M_0 \rightarrow^* c_n$, we know $succ\, M_0 \rightarrow^* c_{n+1}$. Moreover, $(n+1) \leq^o_{nat} c_{n+1}$, and so choosing $w = c_{n+1}$ completes the proof. $\square$

- $M \equiv unit$:
  Assume $M_0^L \leq_\tau M_0$. We must show

  $$\begin{aligned} & (unit\, M_0)^L = lift(M_0^L) \leq^o_{L\tau} unit\, M_0 \\ \iff & \exists w.\, unit\, M_0 \rightarrow^* w \wedge lift(M_0^L) \leq^o_{L\tau} w \end{aligned}$$

  But since $lift(M_0^L) \neq \perp$, it suffices to choose $w = unit\, M_0$. $\square$

- $M \equiv bind$:
  We show

  $$bind^L \leq_{L\tau_2 \rightarrow (\tau_2 \rightarrow L\tau_1) \rightarrow L\tau_1} bind$$

  by proving

  $$(bind\ M_1\ M_2)^L \leq_{L\tau_1} bind\ M_1\ M_2$$

  whenever

  $$\begin{aligned} M_1^L &\leq_{L\tau_2} && M_1 \quad \text{and} \\ M_2^L &\leq_{\tau_2 \rightarrow L\tau_1} && M_2 \end{aligned}$$

  We proceed by cases on M_1^L.

  - $M_1^L \equiv \perp$:
    $bind^L \perp M_2^L = \perp$, and so this case holds by the definitions of $\leq_{L\tau_1}$ and $\leq^o_{L\tau_1}$.
  - $M_1^L \equiv lift(d)$:
    Unwinding the definition of $bind^L$, we must show

    $$M_2^L(d) \leq_{L\tau_1} bind\ (unit\ M)\ M_2$$

    By definition of $\leq_{L\tau_1}$ and the operational semantics, the goal becomes

    $$M_2^L(d) \leq^o_{L\tau_1} (M_2\ M)$$

    The inductive hypotheses are

    $$lift(d) \leq_{L\tau_2} \quad unit\ M \qquad (6)$$
    $$M_2^L \leq_{\tau_2 \rightarrow L\tau_1} M_2 \qquad (7)$$

    (6) gives us $d \leq_{\tau_2} M$, and applying (7) yields the goal.

$\square$

- $M \equiv fix$:
  We must show

  $$fix^L \leq_{((\tau_1 \rightarrow L\tau_2) \rightarrow (\tau_1 \rightarrow L\tau_2)) \rightarrow (\tau_1 \rightarrow L\tau_2)} fix$$

  It suffices to assume

  $$(\lambda f.M)^L \leq_{(\tau_1 \rightarrow L\tau_2) \rightarrow (\tau_1 \rightarrow L\tau_2)} \lambda f.M$$

  and show

  $$\begin{aligned} & (fix\ \lambda f.M)^L && \leq_{\tau_1 \rightarrow L\tau_2} && (fix\ \lambda f.M) \\ \iff & \bigsqcup_{n \in \omega} g^{(n)} && \leq_{\tau_1 \rightarrow L\tau_2} && (fix\ \lambda f.M) \end{aligned} \qquad (8)$$

  where $g = (\lambda f.M)^L = \lambda \psi. M^L[\psi/f]$. (8) follows from

  $$\forall n \in \omega. g^{(n)} \leq_{\tau_1 \rightarrow L\tau_2} (fix\ \lambda f.M) \qquad (9)$$

  and Lemma 5.

  We prove (9) by induction on n.

  When $n = 0$,

  $$\perp \leq_{\tau_1 \rightarrow L\tau_2} (fix\ \lambda f.M)$$

  follows from the definitions of $\leq_{\tau_1 \rightarrow L\tau_2}$, $\leq^o_{\tau_1 \rightarrow L\tau_2}$, $\leq_{L\tau_2}$, and $\leq^o_{L\tau_2}$.

  Now, inductively, we assume

  $$g^{(n)} \leq_{\tau_1 \rightarrow L\tau_2} (fix\ \lambda f.M)$$

  and must show

  $$g^{(n+1)} \leq_{\tau_1 \rightarrow L\tau_2} (fix\ \lambda f.M) \qquad (10)$$

  Expanding the definition of $\leq_{\tau_1 \rightarrow L\tau_2}$, the goal becomes:

  $$\exists v.(fix\ \lambda f.M) \rightarrow^* v \ \wedge\ g^{(n+1)} \leq^o_{\tau_1 \rightarrow L\tau_2} v$$

  By the operational semantics

  $$\begin{aligned} (fix\ \lambda f.M) &\rightarrow (\lambda f.M)(fix\ \lambda f.M) \\ &\rightarrow M \cdot [(fix\ \lambda f.M)/f] \end{aligned}$$

  We choose v to be
  $M \cdot [(fix\ \lambda f.M)/f]$, and we know

  $$g^{(n+1)} = M^L[g^{(n)}/f]$$

  Therefore our goal becomes

  $$M^L[g^{(n)}/f] \leq^o_{\tau_1 \rightarrow L\tau_2} M \cdot [(fix\ \lambda f.M)/f]$$

  We note that $[g^{(n)}/f] \leq_\Gamma [(fix\ \lambda f.M)/f]$, and therefore by structural induction we have

  $$M^L[g^{(n)}/f] \leq_{\tau_1 \rightarrow L\tau_2} M \cdot [(fix\ \lambda f.M)/f]$$

  Unwinding the definition of $\leq_{\tau_1 \rightarrow L\tau_2}$ satisfies our goal. $\square$

- $M \equiv nil$:
  We must show $nil^L = \langle\rangle \leq_{S(nat)} nil$, but this is immediate since nil is a value and $\langle\rangle \leq^o_{S(nat)} nil$. $\square$

- $M \equiv cons$:
  Assume $n \leq_{nat} N$ and $d \leq_{S(nat)} M_0$. We must show

  $$cons^L\ n\ d \leq_{S(nat)} cons\ N\ M_0$$

We know (1) $cons\ N\ M_0 \to^* cons\ c_n\ M_0$, and (2) $n \leq^o_{nat} c_n$ by the first inductive hypothesis. Combining (2) with the second inductive hypothesis gives us

$$cons^L\ n\ d \leq^o_{S(nat)} cons\ c_n\ M_0$$

Thus, in conjunction with (1), we deduce

$$\exists v.\ cons\ N\ M_0 \to^* v \wedge cons^L\ n\ d \leq^o_{S(nat)} v$$

and the goal immediately follows. □

■

There are a number of notable corollaries of the adequacy theorem. The first says that the operational semantics and the L semantics agree on programs of scalar, function, and stream-of-scalar type:

Theorem 5. *Let M be a closed mPCF term of scalar, function, or stream-of-scalar type. Then*

$$M \to^* v \iff M^L = v^L$$

The second says that the operational semantics and the L semantics agree on programs of computation type:

Theorem 6. *Let $\vdash M : L\tau$ for some mPCF term M. Then:*

$$M \to^* (unit\ M_0) \iff \exists d \in \tau^L.(M^L = lift(d) \wedge M_0^L = d)$$

Proof: Both theorems 5 and 6 are immediate via adequacy.

The stream semantics is also related to the operational semantics. By varying the type of observables, we can deduce different relations between them. First, let e be a backtracking term of type $T\,nat$. We show that the first answer produced is related to the first element of the stream e^S. Let our observable domain O be $\mathbb{N}$. Encode a stream whose head is n as the natural number $n+1$, and encode the empty stream as 0.

Theorem 7. *Let e be a closed backtracking MML term of type $T\,nat$, and define*

$$\begin{aligned}\kappa_0 &= (\lambda n.\lambda\phi.(unit(succ\ n)))\\ \phi_0 &= \lambda().\ unit\ c_0\end{aligned}$$

Then $\lceil e \rceil\ \kappa_0\ \phi_0 \to^* (unit\,M)$ iff either

$$\begin{aligned}&(1)\ e^S = \langle\rangle \text{ and } M^L = 0\\ &(2)\ e^S = \langle n\rangle\ \hat{}\ w \text{ and } M^L = n+1\end{aligned}$$

Proof: From Theorem 1 we know that

$$(e^S, e^K) \in R_{T\alpha} \tag{11}$$

and by Theorem 3 we know

$$e^K = \lceil e \rceil^L$$

We consider each case of the shape of the stream e^S:

- case $e^S = \bot$:
  By (11), $e^K = \bot$. Hence $\lceil e \rceil^L\ \kappa_0^L\ \phi_0^L = e^K\ \kappa_0^L\ \phi_0^L = \bot$. So, by Theorem 6, $\lceil e \rceil\ \kappa_0\ \phi_0$ does not reduce to a value.
- case $e^S = \langle\rangle$:
  By (11), $e^K = \lambda\kappa.\lambda\phi.\phi()$. Hence, $\lceil e \rceil^L\ \kappa_0^L\ \phi_0^L = \phi_0^L() = lift(0)$. So, by Theorem 6, $\lceil e \rceil\ \kappa_0\ \phi_0 \to^* unit(M)$, where $M^L = 0$.
- case $e^S = \langle n\rangle\ \hat{}\ w$:

  By (11), $e^K = \lambda\kappa.(\kappa n) \star w'$, where $(w, w') \in R_{T\alpha}$. Hence
  $$\begin{aligned}\lceil e \rceil^L\ \kappa_0^L\ \phi_0^L &= e^K\ \kappa_0^L\ \phi_0^L\\ &= (\kappa_0^L n)(\lambda().(w'\phi_0^L))\\ &= lift(n+1)\end{aligned}$$
  By Theorem 6, $\lceil e \rceil\ \kappa_0\ \phi_0 \to^* unit(c_{n+1})$.

■

We can derive a stronger relationship between the stream semantics and the operational semantics by choosing a different observable domain.

Theorem 8. *Let e be a closed backtracking MML term of type $T\,nat$, let $\Omega = Snat$, and define*

$$\begin{aligned}\kappa_0 &= \lambda a\phi.\ unit(cons\ a\ \phi())\\ \phi_0 &= \lambda().\ unit(nil)\end{aligned}$$

Then $\lceil e \rceil\ \kappa_0\ \phi_0 \to^* unit(M)$ iff either

$$\begin{aligned}&(1)\ e^S = \langle\rangle \text{ and } M = nil\\ &(2)\ e^S = \langle n\rangle\ \hat{}\ w \text{ and } M = cons\ c_n\ M_0 \text{ and } M_0^L = w\end{aligned}$$

Proof: If $\Omega = Snat$, then $O_\bot = S(\mathbb{N})$. Note that κ_0^L and ϕ_0^L are just *cons* and *nil* as defined in Theorem 2. From Theorem 1 we know that

$$(e^S, e^K) \in R_{T\alpha} \tag{12}$$

and by Theorem 3 we know

$$e^K = \lceil e \rceil^L$$

We consider each case of the shape of the stream e^S:

- case $e^S \equiv \bot$:
  By (12), $e^K = \bot$. Hence $e^K\ \kappa_0^L\ \phi_0^L = \lceil e \rceil^L\ \kappa_0^L\ \phi_0^L = \bot$. So, by Theorem 6, $\lceil e \rceil\ \kappa_0\ \phi_0$ does not reduce to a value.
- case $e^S \equiv \langle\rangle$:
  By (12), $e^K = \lambda\kappa\phi.\phi()$. Hence
  $$\begin{aligned}\lceil e \rceil^L\ \kappa_0^L\ \phi_0^L &= e^K\ \kappa_0^L\ \phi_0^L\\ &= \phi_0^L()\\ &= lift(\langle\rangle)\end{aligned}$$
  So, by Theorem 6, $\lceil e \rceil\ \kappa_0\ \phi_0 \to^* unit(nil)$.
- case $e^S \equiv \langle n\rangle\ \hat{}\ w$:
  By (12), $e^K = \lambda\kappa.(\kappa n) \star w'$, where $(w, w') \in R_{T(nat)}$. Hence
  $$\begin{aligned}\lceil e \rceil^L\ \kappa_0^L\ \phi_0^L &= e^K\ \kappa_0^L\ \phi_0^L\\ &= (\kappa_0^L n)(\lambda().(w'\phi_0^L)))\\ &= lift(\langle n\rangle\ \hat{}\ (w'\phi_0^L))\end{aligned}$$
  So, by Theorem 6, $\lceil e \rceil\ \kappa_0\ \phi_0 \to^* unit(cons\ c_n\ M)$, where $M^L = (w'\phi_0^L)$.

■

7. RELATED WORK

Both the stream and two-continuation models have a long history in Prolog implementations. The stream model was invented independently by Abelson and Sussman [1], Kahn [9], and Wadler [19]. The two-continuation model appears in Federhen [5] though it is undoubtedly older. Kahn [9] discusses "upward failure continuations" versus "downward success continuations" as strategies for embedding the two-continuation model in a stack architecture.

It is worth noting that the computational model of Prolog is a stripped-down version of the backtracking monad. Prolog programs do not pass any values; all communication happens by modifying the current substitution. In this sense, Prolog programs are actually *imperative* programs in the backtracking monad. This means that the parameter of the computation type is fixed. Rather than dealing with $T\alpha$ for arbitrary α, Prolog models need only consider $T(\mathit{Subst})$, where *Subst* is the type of substitutions. This simplified form of computation makes the relationship between $T(\mathit{Subst})^S$ and $T(\mathit{Subst})^K$ far less complicated than in the general case that we explore.

The definition of a backtracking monad comes, *inter alia*, from Hughes [8]. Hughes also presented a derivation of the two-continuation model from the axioms via fold-unfold transformations [2]. This derivation is flawed in two ways: first, it depends on an induction hypothesis that is at best problematical. We will discuss this issue below. Furthermore, the derivation shows only that if $e = v$ is deducible in the derived system, then it is also deducible in the original system. It does not show that if $e = v$ is deducible in the original system, it is also deducible in the derived system. That is, the derived system is sound but not adequate with respect to the original. This is a standard problem with fold-unfold transformations [14] that must be addressed if we are to deal correctly with possibly non-terminating computations.

Hinze [7] extended Hughes' result from monads to monad transformers. His derivation followed that of Hughes, with the same difficulties.

The problem in establishing even the soundness of these derivations lies in writing down an induction hypothesis that is strong enough to support the proof. Both of the proofs represent computations by higher-order abstract syntax, so that they are envisioned as trees, which might be expressed in Haskell as:

```
data Comp a = Unit a
            | forall b.Bind (Comp b) (b -> Comp a)
```

To illustrate the difficulty, consider the calculation near the end of Section 5.2 of [8]:

$$\begin{aligned}
& (m \text{ ‘}\mathit{bind}\text{‘ } f)\ (\mathit{ValueBind}(\mathit{value}\ .\ k)) \\
= \;& (m \text{ ‘}\mathit{bind}\text{‘ } f) \text{ ‘}\mathit{bind}\text{‘ } k) && (13)\\
= \;& m \text{ ‘}\mathit{bind}\text{‘ } \lambda x \to (fx \text{ ‘}\mathit{bind}\text{‘ } k)) && (14)\\
= \;& m(\mathit{ValueBind}(\lambda x \to \mathit{value}(fx \text{ ‘}\mathit{bind}\text{‘ } k))) && (15)\\
= \;& m(\mathit{ValueBind}(\lambda x \to fx(\mathit{ValueBind}(\mathit{value}\ .\ k)))) && (16)
\end{aligned}$$

Here, (13), (15) and (16) are instances of the induction hypothesis

$$m(\mathit{ValueBind}(\mathit{value}\ .\ k)) = \mathit{value}(m \text{ ‘}\mathit{bind}\text{‘ } k)$$

and the conclusion is an instance of the same equation, with m replaced by m ‘*bind*‘ f. But what is the induction measure here? (1) and (3) use the induction hypothesis at m, which is certainly smaller than m ‘*bind*‘ f. But (16) uses it at (fx), which might be larger than m ‘*bind*‘ f.

Hinze [personal communication] has suggested the following induction principle: to prove a property P for all elements of `Comp a`, prove the following (where here a, b range over values, not types):

1. $P($`Unit a`$)$
2. If $P(m)$ and for all b, $P(f(b))$, then $P($`Bind` $m\ f)$

This would work fine if the elements of `Comp a` were B-way trees of finite depth for any fixed B (e.g., $B = \mathit{Subst}$ as suggested above for Prolog); the induction principle would be justified by induction on the height of the tree. Trees of infinite depth, as generated by *fix*, would be handled by appropriate continuity arguments.

Unfortunately, the elements of `Comp a` are not B-way trees of finite depth. Choose `a` to be `int` and `b` to be `(Comp int)`. Then `Bind` at `Comp int` is an injection from

```
(Comp (Comp int)) * (Comp int -> Comp int)
```

to `(Comp int)`. Therefore `(Comp int)` is required to contain its own function space as a subset. Thus, `(Comp int)` does not meet our expectation that it look like a set of trees, and the induction principle suggested above is problematical at best. It is possible that this induction principle is fixable, but it would clearly involve order-theoretical niceties.

We avoid this dilemma by treating the metalanguage as an ordinary λ-calculus, without the complications of higher-order abstract syntax. Haskell implementations essentially do the same thing: the second component to `Bind` is represented not as a function, but as a closure. Since Haskell does not allow testing for equality between higher-order values, there is no way to detect the difference inside the language.

Another approach to this problem is to try to describe the two-continuation model as a representation of the operational semantics of the stream model. We have explored final algebras for this purpose [21, 10, 6]. In this approach, the stream is represented by its actions on success (*unit*) and failure (*fail*). This also gives soundness but not adequacy. Other alternatives are to represent the stream by its `case` function (sometimes called the *Scott representation* [20]) or by its `fold` function (corresponding to the Church representation); neither of these representations yields the two-continuation model. Our representation is similar to a Church representation, except that the first argument to the fold function (the action on "cons") is required to be distributive.

The approach closest to ours is that of Danvy et al. [4], who formulate the problem in terms of monad morphisms. They deal only with a first-order object-language, and the behavior of their proposal under fixed-points is not clear. Although they do not set out their representation explicitly as we do in case 3 of the definition of our logical relation, their representation coincides with ours for first-order quantities.

Seres, Spivey, and Hoare [16, 15] also explore the relationship between backtracking monads. They are interested in monads that capture search strategies for logic programs. They formulate three monads: the first captures depth-first search, the second breadth-first, and the third allows both. They define monad morphisms from the third one to both the first and second. Both monads in this paper implement depth-first search.

Claessen and Ljunglöf extend Seres' and Spivey's embedding of Prolog into Haskell [16] by using more sophisticated types. Certain Prolog programs which would normally just fail at run-time are considered ill-typed in their system. To accomplish this, they generalize their type of substitutions by taking advantage of extensions provided by the GHC implementation of Haskell.

Thielecke [17] investigates the relationship between the answer type for CPS terms and the control effects exhibited by those terms. He employs a type-and-effect system to reason about control effects. This work could be useful in refining our formulation of observable types in backtracking programs.

Gordon and Crole [3] have developed a technique for factoring adequacy proofs for languages whose semantics is organized monadically. Essentially, they prove adequacy of an object-language operational semantics by relating it to an adequate metalanguage operational semantics.

8. FUTURE WORK

We intend to provide operational semantics for the the backtracking metalanguage, MML, and also one for the object language. We would like to extend our adequacy result to these semantics. This goal exists in the context of a larger goal: we would like to explore how program analyses of metalanguage programs relate to analyses of corresponding object-language programs. For example, if we perform a flow analysis on a mPCF program, does this induce a useful flow analysis for the corresponding MML and object-language programs? Finally, we believe that it is desirable to automate these techniques by formalizing them in a theorem-prover or a metalogic.

9. SUMMING UP

We attack the long-standing question of the relationship between two well-known models of backtracking computation. We are able to relate the two models, and we accommodate higher-order quantities and infinite computations using a logical relation. Since the models share a representation of observable values, we obtain a denotational equivalence at observable types. This means that an operational semantics that is adequate for one model is adequate for the other.

We provide an operational semantics which is adequate for mPCF, a variant of PCF that can express non-terminating computations. By giving a translation from MML to mPCF, we obtain an evaluator for MML programs. Since the translation preserves semantics, the evaluator for MML programs is adequate for both the two-continuation and stream semantics.

10. ACKNOWLEDGEMENTS

We thank David Herman for his comments and for identifying a bug in the adequacy proof. We also express our gratitude to the anonymous referees for their insightful comments.

11. REFERENCES

[1] Hal Abelson and Gerald Jay Sussman. *The Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[2] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[3] R. L. Crole and A. D. Gordon. Factoring an Adequacy Proof. In C. J. van Rijsbergen, editor, *FP'93 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 9–25. Springer-Verlag, 1994.

[4] Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20:53–73, 2002.

[5] Scott Federhen. A mathematical semantics for PLANNER. Master's thesis, University of Maryland, 1980.

[6] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce Duba. Abstract continuations: A mathematical semantics for handling functional jumps. In *Proceedings of the 1988 ACM Symposium on LISP and Functional Programming*, Salt Lake City, Utah, July 1988.

[7] Ralf Hinze. Deriving backtracking monad transformers. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 186–197, 2000.

[8] John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925. Springer Verlag, 1995.

[9] K. M. Kahn and M. Carlsson. How to implement prolog on a LISP machine. In J. A. Campbell, editor, *Implementations of Prolog*, pages 119–134. Chichester, 1984.

[10] Samuel Kamin. Final data type specifications: A new data type specification method. In *Conf. Rec. 7th ACM Symposium on Principles of Programming Languages*, pages 131–138, 1980.

[11] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, MA, 1996.

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

[13] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[14] David Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[15] Silvija Seres, J. Michael Spivey, and C. A. R. Hoare. Algebra of logic programming. In *International Conference on Logic Programming*, pages 184–199, 1999.

[16] Silvija Seres and Michael J. Spivey. Embedding prolog into haskell. In *Haskell Workshop '99*, 1999.

[17] Hayo Thielecke. From control effects to typed continuation passing. In *Conf. Rec. 30th ACM Symposium on Principles of Programming Languages*, pages 139–149. ACM Press, 2003.

[18] A. S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and*

Analysis, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, Heidelberg, and New York, 1973.

[19] P. L. Wadler. How to replace failure by a list of successes. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer Verlag, September 1985.

[20] Christopher P. Wadsworth. Some unusual λ-calculus numeral systems. In J. R. Seldin and J. P. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 215–230. Academic Press, New York and London, 1980.

[21] Mitchell Wand. Final algebra semantics and data type extensions. *Journal of Computer and Systems Science*, 19:27–44, 1979.

[22] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, 1993.