Relating Models of Backtracking

Mitchell Wand
wand@ccs.neu.edu

Dale Vaillancourt
dalev@ccs.neu.edu

College of Computer and Information Science
Northeastern University
360 Huntington Ave #161 CN
Boston, MA 02115

ABSTRACT

Past attempts to relate two well-known models of backtracking computataion 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.

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 with a stream of answers, and the two-continuation model uses a success continuation and a failure continuation. Hughes [7] 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. [3]. 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 modeling a simply-typed call-by-value PCF that permits backtracking computation (consider a higher-order variant of Icon or Snobol). Such a language has types

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

and terms

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

where $c_0, c_1, c_2, \cdots$ are constants of type *nat*. We allow recursive functions with $rec\ f.(\lambda x.t)$, 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 for t_2. The canonical toy example in this 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.

$$\begin{aligned} \textit{nats-from} &= rec\, f.(\lambda n.n \vee f(succ\ n)) \\ \textit{nats} &= \textit{nats-from}(c_0) \end{aligned}$$

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). A 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 simply 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:

$$\begin{aligned} e \;::=&\; x \;\mid\; (\lambda x.e) \;\mid\; e_1\; e_2 \;\mid\; succ \;\mid\; c_n \\ \mid&\; unit_\alpha \;\mid\; bind_{\beta\alpha} \end{aligned}$$

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

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

for each pair of types α and β. $unit_\alpha$ takes a value v of type α to a computation which simply produces v when run. $bind_{\beta\alpha}\ c\ f$ runs c and produces a new computation 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 and the standard monad laws:

$$\begin{array}{l} bind\ (unit\ e)\ f = (f\ e) \\ bind\ e\ \ unit = e \\ bind\ (bind\ c\ f)\ g = bind\ c\ (\lambda v.\, bind\ (f\ v)\ g) \end{array}$$

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 [7] 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:

$$\begin{array}{l} fail_\alpha\ : T\alpha \\ disj_\alpha : T\alpha \to T\alpha \to T\alpha \end{array}$$

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

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:

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

We will consider an 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, and we write α^M for the interpretation of type α in a monad M. Scalar types map to discrete orders (i.e., sets whose order is given by an equivalence relation).[1] Function types map to continuous function spaces.

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

Note that cpos need not have a bottom element, and the function space $[\alpha^M \to \beta^M]$ has a bottom element when β^M has one. We write $\sqsubseteq_{\alpha^M}$ for the ordering relation on the domain α^M.

We write e^M for the interpretation of an expression e in the monad M. If $\Gamma \vdash e : \alpha$, then $e^M \in [Env \to \alpha^M]$, and $Env = [Var \to \bigcup_\alpha \alpha^M]$. We say that an environment ρ is Γ-consistent when $dom(\rho) \subseteq dom(\Gamma)$ and $\forall x \in dom(\rho).(\rho(x) : \Gamma(x))$. When e is closed, we take the liberty of writing e^M in place of $e^M\rho$ (where $\rho \in Env$).

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

Our interpretations for computation types will always have a bottom element, and this is why we type *fix* as given above.

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 apply the call-by-value translation [11] into the metalanguage. This translation maps object language functions to metalanguage functions that consume values and produce computations.[2] 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:

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

and the translation on terms is as follows:

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

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

[1] For example, $nat^M = D^{nat} = \mathbb{N}$, and the ordering is the identity relation.

[2] The call-by-name translation maps object language functions to metalanguage functions that both consume and produce computations.

MML's interpretation:

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

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

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

where $\hat{}$ is stream append (with $w_1 \mathbin{\hat{}} w_2 = w_1$ 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 $\langle a_1, \ldots, a_n \rangle \mathbin{\hat{}} \bot$ 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 $s \in S(A)$:

$$\bot \quad \sqsubseteq_{S(A)} \quad s \tag{1}$$

$$
\begin{array}{ccc}
\langle a_1, a_2, \ldots, a_n \rangle & \sqsubseteq_{S(A)} & \langle b_1, b_2, \ldots, b_m \rangle \\
& \Longleftrightarrow & \\
n = m & \wedge & a_i \sqsubseteq_A b_i \quad (1 \leq i \leq n)
\end{array} \tag{2}
$$

$$
\begin{array}{ccc}
\langle a_1, a_2, \ldots, a_n \rangle \mathbin{\hat{}} \bot & \sqsubseteq_{S(A)} & \langle b_1, b_2, \ldots, b_n \rangle \mathbin{\hat{}} s \\
& \Longleftrightarrow & \\
& a_i \sqsubseteq_A b_i \quad (1 \leq i \leq n) &
\end{array} \tag{3}
$$

$$
\begin{array}{ccc}
\langle a_1, a_2, \ldots \rangle & \sqsubseteq_{S(A)} & \langle b_1, b_2, \ldots \rangle \\
& \Longleftrightarrow & \\
& a_i \sqsubseteq_A b_i \quad (\forall i \in \omega) &
\end{array} \tag{4}
$$

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.[3]

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:

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

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

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

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 the interpretation of some scalar type σ^K, and let $O_\bot$ be O with a bottom element adjoined. We refer to values

[3]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: $\langle 1, 2 \rangle \mathbin{\hat{}} \bot \sqsubseteq_{S(\mathbb{N})} \langle 1, 2, 3 \rangle$.

of type O as "observables" (as opposed to values of higher-order type). Now we define

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

The domain $[\mathbb{1} \to O_\perp]$ is the domain of failure continuations, and $[A \to [\mathbb{1} \to O_\perp] \to O_\perp]$ 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:

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

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 $\mathbb{1}$.

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.

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

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 [12], though he credits Troelstra [14]. Mitchell [10] provides a thorough account of logical relations.

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:
   - $(\perp, \perp) \in R_{T\alpha}$.
   - $(\langle\rangle, \lambda\kappa.\lambda\phi.\phi()) \in R_{T\alpha}$.
   - if $(a_1, a_1'), \ldots, (a_n, a_n') \in R_\alpha$, then
     - $(\langle a_1, \ldots, a_n\rangle,$ $(\lambda k.(k\ a_1') \star (k\ a_2') \star \cdots \star (k\ a_n'))) \in R_{T\alpha}$ $(n \geq 1)$
     - $(\langle a_1, \ldots, a_n\rangle\ \hat{}\ \perp,$ $(\lambda k.(k\ a_1') \star (k\ a_2') \star \cdots \star (k\ a_n') \star \perp)) \in R_{T\alpha}$
   - $R_{T\alpha}$ is closed under limits of ω-chains in $(T\alpha)^S \times (T\alpha)^K$.

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

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

and is right-associative. We use $\star$ to "compose" two functions that each consume a failure continuation:

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

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. ■

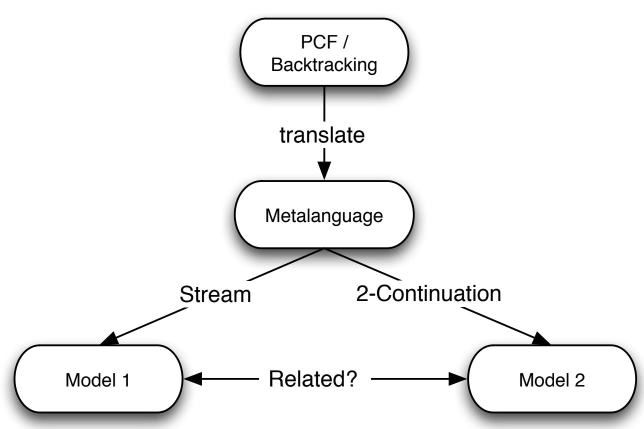

Figure 1: A logical relation relates the models of two semantics.

LEMMA 2. *R is a logical relation.*

Proof: The proof is by consideration of each of the constants. The case for *disj* uses Lemma 1. ■

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:

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

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 [10]. ■

THEOREM 2. *Let $\vdash e : T(\sigma)$ be a closed backtracking MML term. Then $e^S = e^K\ cons\ (\lambda().\langle\rangle)$, where $cons = \lambda a\phi.\langle a\rangle\ \hat{}\ \phi()$.*

Proof: 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

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

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 $[1 \to O_\perp]$ by failure computations $O_\perp$, so $K'(A) = (A \to O_\perp \to O_\perp) \to O_\perp \to O_\perp$, and

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

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. We proceed by providing an 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. Its types[4] and terms are:

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

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:

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

The interpretations for variables, abstraction, application, *succ*, c_n, and *fix* remain as they are in S and K (the details are in section 3). In L, *unit* takes a value to a computation which terminates with that value, and *bind* encodes strict application.

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

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

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

[4]We have renamed the type constructor T to L to avoid confusion between backtracking MML and mPCF computation types.

$$
\begin{array}{lcl}
\lceil unit \rceil & = & (\lambda a.(\lambda \kappa \phi.\kappa a \phi)) \\
\lceil bind \rceil & = & (\lambda c f.(\lambda \kappa \phi.c\ (\lambda b.\lambda \phi.f b \kappa \phi)\ \phi)) \\
\lceil disj \rceil & = & (\lambda c_1 c_2.c_1\ \kappa\ (\lambda ().c_2\ \kappa\ \phi)) \\
\lceil fail \rceil & = & (\lambda \kappa \phi.\phi()) \\
\lceil succ \rceil & = & succ \\
\lceil c_n \rceil & = & c_n \\
\lceil (\lambda x.e) \rceil & = & (\lambda x.\lceil e \rceil) \\
\lceil e_1\ e_2 \rceil & = & \lceil e_1 \rceil\ \lceil e_2 \rceil
\end{array}
$$

Note two interesting facts: (1) this translation never produces any *bind* terms in mPCF, and (2) *unit* will only appear in the initial continuation. This has to do with the fact that the K semantics is a continuation passing semantics, and so computation does not build control context.

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 values are given by V, and the evaluation contexts are specified by E. We will write V^τ to refer to the set of values of type τ. Reduction halts if the term reaches any constant, an abstraction, or *unit* of a some term.[5] 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.

$$
\begin{array}{lcl}
V & ::= & c_n \mid (\lambda x.M) \mid unit\ M \\
  & \mid & succ \mid unit \mid bind \mid fix \\
E & ::= & [] \mid EM \mid bind\ E \mid succ\ E
\end{array}
$$

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.

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

The reduction and context for *bind* encodes strict application.

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

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 (such as *nat* or $L\,nat$); 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 [18].

Assume M is a closed term of either scalar or computation type. Define (5) operational convergence for M and (6) denotational convergence for M as follows:

$$
\begin{array}{lcll}
M\downarrow & \iff & \exists v.(M \to^* v) & (5) \\
M\Downarrow & \iff & \vdash M : \sigma & (6) \\
 & \vee & \vdash M : L\tau\ \wedge\ \exists d \in \tau^L.M^L\rho = lift(d) &
\end{array}
$$

THEOREM 4 (ADEQUACY). $M\downarrow \iff M\Downarrow$

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. □

■

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. ■

Next, we define an approximation relation $\leq^o_\tau$ between denotations and closed values for each type τ. We extend this to $\leq_\tau$ to relate denotations to closed terms. Define $(\leq^o_\tau) \subseteq (\tau^L \times V^\tau)$ by induction on τ:

$$
\begin{array}{lll}
d \leq^o_\sigma\ v & \iff & d \in D^\sigma\ \wedge\ d = v^L \\
d \leq^o_{\tau_1 \to \tau_2}\ v & \iff & \forall (d', M') \in (\leq_{\tau_1}).d(d') \leq_{\tau_2} (vM') \\
d \leq^o_{L\tau}\ unit(M) & \iff & d = \bot\ \vee\ d = lift(d') \wedge d' \leq_\tau M
\end{array}
$$

and define $d \leq_\tau M \iff \exists v.(M \to^* v \wedge d \leq^o_\tau v)$.

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* $d_0 \sqsubseteq_{\tau^L} d_1 \sqsubseteq_{\tau^L} \ldots \sqsubseteq_{\tau^L} d_n \sqsubseteq_{\tau^L} \ldots$ *is an* ω*-chain in* τ^L *and* $\forall n \in \omega . d_n \leq_\tau M$, *then* $\bigsqcup_{n \in \omega} d_n \leq_\tau M$.

Proof: By structural induction on τ. ■

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 $\rho \leq_\Gamma \theta \iff \forall x \in dom(\Gamma) . \rho(x) \leq_{\Gamma(x)} \theta(x)$.

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

Proof: By induction on the structure of M.

- $M \equiv x$: $x^L \rho = \rho(x) \leq_\tau \theta(x) = x \cdot \theta$ holds since $\rho \leq_\Gamma \theta$. □
- $M \equiv (\lambda x . M_0)$:
We must show
$$(\lambda x . M_0)^L \rho \leq^o_{\tau_1 \to \tau_2} (\lambda x . M_0) \cdot \theta$$
Fix an arbitrary $(d', e') \in (\leq_{\tau_1})$. By definition of $\leq^o_{\tau_1 \to \tau_2}$, we must demonstrate
$$\begin{aligned} & ((\lambda x . M_0)^L \rho)(d') \leq_{\tau_2} (((\lambda x . M_0) \cdot \theta) e') \\ \equiv\ & M_0^L \rho[d'/x] \leq_{\tau_2} (((\lambda x . M_0) \cdot \theta) e') \\ \equiv\ & \exists w . ((((\lambda x . M_0) \cdot \theta)\ e') \to^* w \wedge M_0^L \rho[d'/x] \leq^o_{\tau_2} w) \end{aligned}$$
Following the definition of $\to$, we choose $w = M_0 \cdot \theta[e'/x]$ and require
$$\begin{aligned} & M_0^L \rho[d'/x] \leq^o_\beta M_0 \cdot \theta[e'/x] \\ \equiv\ & M_0^L \rho[d'/x] \leq_\beta M_0 \cdot \theta[e'/x] \end{aligned}$$
Since $d' \leq_{\tau_1} e'$, we infer $\rho[d'/x] \leq_{\Gamma \cup \{x : \tau_1\}} \theta[e'/x]$, and we apply our induction hypothesis at M_0 to conclude. □
- $M \equiv M_1\ M_2$:
Let $\Gamma \vdash M_1 : \tau_1 \to \tau_2$, $\Gamma \vdash M_2 : \tau_1$, and $\rho \leq_\Gamma \theta$. Inductively, assume:
$$M_1^L \rho \leq_{\tau_1 \to \tau_2} M_1 \cdot \theta \qquad (7)$$
$$M_2^L \rho \leq_{\tau_1} M_2 \cdot \theta \qquad (8)$$
(7) implies
$$\exists v . (M_1 \to^* v \wedge (M_1^L \rho) \leq^o_{\tau_1 \to \tau_2} v)$$
and combined with (8) we have
$$\begin{aligned} & (M_1^L \rho)(M_2^L \rho) \leq_{\tau_2} v M_2 \cdot \theta \\ \equiv\ & (M_1 M_2)^L \rho \leq_{\tau_2} v M_2 \cdot \theta \end{aligned} \qquad (9)$$
By (9):
$$\exists w . v M_2 \cdot \theta \to^* w \wedge (M_1 M_2)^L \rho \leq^o_{\tau_2} w$$
But $v M_2 \cdot \theta \to^* w$ and $M_1 \to^* v$ implies
$$M_1 M_2 \cdot \theta \to^* w$$
and therefore
$$(M_1 M_2)^L \rho \leq_{\tau_2} M_1 M_2 \cdot \theta$$
□
- $M \equiv succ$:
Assume $M_0^L \rho \leq_{nat} M_0 \cdot \theta$. Then
$$\exists v . M_0 \cdot \theta \to^* v \wedge M_0^L \rho \leq^o_{nat} v$$
Letting $n = M_0^L \rho$, it is clear that $v = c_n$ by definition of $\leq^o_{nat}$.
We must show
$$\begin{aligned} & (succ\ M_0)^L \rho \leq_{nat} succ\ M_0 \cdot \theta \\ \equiv\ & (n+1) \leq_{nat} succ\ M_0 \cdot \theta \\ \equiv\ & \exists w . (succ\ M_0 \cdot \theta \to^* w \wedge (n+1) \leq^o_{nat} w) \end{aligned}$$
But $M_0 \cdot \theta \to^* v$, so we need only show:
$$\exists w .\ succ\ v \to^* w \wedge (n+1) \leq^o_{nat} w$$
Evidently $w = c_{n+1}$ by definition of $\leq^o_{nat}$, and $succ\ v \to w$ by the operational semantics. □
- $M \equiv unit$:
Assume $M_0^L \rho \leq_\tau M_0 \cdot \theta$. We must show
$$\begin{aligned} & (unit\ M_0)^L \rho \leq^o_{L\tau} unit\ M_0 \cdot \theta \\ \equiv\ & lift(M_0^L \rho) \leq_{L\tau} unit\ M_0 \cdot \theta \\ \equiv\ & \exists w .\ unit\ M_0 \cdot \theta \to^* w \wedge lift(M_0^L \rho) \leq^o_{L\tau} w \end{aligned}$$
But since $lift(M_0^L \rho) \neq \bot$, it suffices to let $w = unit\ M_0 \cdot \theta$ and show:
$$M_0^L \rho \leq_\tau M_0 \cdot \theta$$
But this is just our inductive hypothesis. □
- $M \equiv bind$:
We show
$$bind^L\ \rho \leq_{L\tau_2 \to (\tau_2 \to L\tau_1) \to L\tau_1} bind \cdot \theta$$
by proving
$$(bind\ M_1\ M_2)^L \rho \leq_{L\tau_1} bind\ M_1\ M_2 \cdot \theta$$
whenever
$$\begin{aligned} M_1^L \rho &\leq_{L\tau_2} M_1 \cdot \theta \ \text{ and} \\ M_2^L \rho &\leq_{\tau_2 \to L\tau_1} M_2 \cdot \theta \end{aligned}$$
We proceed by cases on $M_1^L \rho$.
  - $M_1^L \rho \equiv \bot$:
$bind^L\ \bot\ M_2^L \rho = \bot$, and so this case holds by the definitions of $\leq_{L\tau_1}$ and $\leq^o_{L\tau_1}$.
  - $M_1^L \rho \equiv lift(d)$:
Unwinding the definition of $bind^L$, we must show
$$M_2(d) \leq_{L\tau_1} bind\ (unit\ M)\ M_2 \cdot \theta$$
By definition of $\leq_{L\tau_1}$ and the operational semantics, the goal becomes
$$M_2(d) \leq^o_{L\tau_1} (M_2\ M) \cdot \theta$$
The inductive hypotheses are
$$lift(d) \leq_{L\tau_2} unit\ M \cdot \theta \qquad (10)$$
$$M_2^L \rho \leq_{\tau_2 \to L\tau_1} M_2 \cdot \theta \qquad (11)$$

(10) gives us $d \le_{\tau_2} M \cdot \theta$, and applying (11) yields the goal.

□

- $M \equiv \mathit{fix}$:
  We must show

$$\mathit{fix}^L\, \rho \le_{((\tau_1 \to L\tau_2) \to (\tau_1 \to L\tau_2)) \to (\tau_1 \to L\tau_2)} \mathit{fix} \cdot \theta$$

It suffices to assume

$$(\lambda f.M)^L \rho \le_{(\tau_1 \to L\tau_2) \to (\tau_1 \to L\tau_2)} (\lambda f.M) \cdot \theta$$

and show

$$\begin{array}{lll} (\mathit{fix}\ (\lambda f.M))^L \rho & \le_{\tau_1 \to L\tau_2} & (\mathit{fix}\ (\lambda f.M)) \cdot \theta \\ \equiv \bigsqcup_{n \in \omega} g^{(n)} & \le_{\tau_1 \to L\tau_2} & (\mathit{fix}\ (\lambda f.M)) \cdot \theta \quad (12) \end{array}$$

where $g = (\lambda f.M)^L \rho = (\lambda \psi. M^L \rho[\psi/f])$. (12) follows from

$$\forall n \in \omega. g^{(n)} \le_{\tau_1 \to L\tau_2} (\mathit{fix}\ (\lambda f.M)) \cdot \theta \qquad (13)$$

and Lemma 5.

We prove (13) by induction on n.

When $n = 0$,

$$\bot \le_{\tau_1 \to L\tau_2} (\mathit{fix}\ (\lambda f.M)) \cdot \theta$$

follows from the definitions of $\le_{\tau_1 \to L\tau_2}$, $\le^o_{\tau_1 \to L\tau_2}$, $\le_{L\tau_2}$, and $\le^o_{L\tau_2}$.

Now, inductively, we assume

$$g^{(n)} \le_{\tau_1 \to L\tau_2} (\mathit{fix}\ (\lambda f.M)) \cdot \theta$$

and must show

$$g^{(n+1)} \le_{\tau_1 \to L\tau_2} (\mathit{fix}\ (\lambda f.M)) \cdot \theta \qquad (14)$$

Expanding the definition of $\le_{\tau_1 \to L\tau_2}$, the goal becomes:

$$\exists v.(\mathit{fix}\ (\lambda f.M)) \cdot \theta \to^* v \ \wedge\ g^{(n+1)} \le^o_{\tau_1 \to L\tau_2} v$$

By the operational semantics

$$\begin{array}{rl} (\mathit{fix}\ (\lambda f.M)) \cdot \theta & = (\mathit{fix}\ (\lambda f.M \cdot \theta)) \\ & \to (\lambda f.M \cdot \theta)(\mathit{fix}(\lambda f.M \cdot \theta)) \\ & \to M \cdot \theta[(\mathit{fix}(\lambda f.M \cdot \theta))/f] \\ & = M[(\mathit{fix}(\lambda f.M))/f] \cdot \theta \end{array}$$

Note that it is okay to shuffle θ around as necessary because it does not bind f. We choose v to be $M[(\mathit{fix}(\lambda f.M))/f] \cdot \theta$, and we know

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

Therefore our goal becomes

$$M^L \rho[g^{(n)}/f] \le^o_{\tau_1 \to L\tau_2} M \cdot [(\mathit{fix}(\lambda f.M))/f]\theta$$

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

$$M^L \rho[g^{(n)}/f] \le_{\tau_1 \to L\tau_2} M \cdot [(\mathit{fix}(\lambda f.M))/f]\theta$$

Unwinding the definition of $\le_{\tau_1 \to L\tau_2}$ satisfies our goal.
□

■

There are number of notable corollaries of the adequacy proof. The first says that the operational semantics and the L semantics agree on programs of scalar type:

Theorem 5. *Let $\vdash M : \sigma$ for some mPCF term M. 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:*

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

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

The stream semantics is also related to the operational semantics. Let e be a backtracking mPCF term of computation type. We show that the first answer produced is related to the first element of the stream that e^S denotes.

Theorem 7. *Let e be a closed backtracking MML term and assume $\vdash e : T\alpha$. Without loss of generality, assume our observable type $O = \mathit{nat}$. Encode a stream whose head is n as the natural number $n+1$, and encode the empty stream as 0. Then:*

$$\begin{array}{c} \lceil e \rceil (\lambda n.\lambda \phi.(\mathit{unit}(\mathit{succ}\ n)))(\lambda ().\ \mathit{unit}\ c_0) \to^* (\mathit{unit}\ M) \\ \iff \\ (e^S = \langle\rangle \ \wedge\ M^L = 0 \\ \vee \\ (e^S = \langle n \rangle \,\hat{}\, w \ \wedge\ M^L = n+1)) \end{array}$$

Proof: From theorem 1 we know that

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

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 $e^S = \bot$:
  Then $e^K = \bot$, and therefore, by adequacy, $\lceil e \rceil$ in the initial continuations does not reduce to a value.

- case $e^S = \langle\rangle$:
  The logical relation tells us that $e^K = \lambda\kappa.\lambda\phi.\phi()$. The operational semantics then tells us that $\lceil e \rceil$ in the initial continuations will reduce to $(\mathit{unit}\, c_0)$ and $c_0^L = 0$.

- case $e^S = \langle n \rangle \,\hat{}\, w$:
  We appeal to the logical relation and our encoding of observations to see that $e^K = \lambda\kappa.(\kappa(n+1)) \star \ldots$. The adequacy theorem tells us that $\lceil e \rceil$ in the initial continuations will reduce to $(unit\, M)$ where $M^L = n+1$.

■

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 [8], and Wadler [15]. The two-continuation model appears in Federhen [4] though it is undoubtedly older. Kahn [8] discusses "upward failure continuations" versus "downward success continuations" as strategies for embedding the two-continuation model in a stack architecture.

The definition of a backtracking monad comes, *inter alia*, from [7]. 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 [13] that must be addressed if we are to deal correctly with possibly non-terminating computations.

Hinze [6] extended Hughes's 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 [7]:

$$\begin{aligned}
& (m \text{ ` } bind \text{ ` } f)\ (ValueBind(value\ .\ k)) \\
= \;& (m \text{ ` } bind \text{ ` } f) \text{ ` } bind \text{ ` } k) && (15)\\
= \;& m \text{ ` } bind \text{ ` } \lambda x \to (fx \text{ ` } bind \text{ ` } k)) && (16)\\
= \;& m(ValueBind(\lambda x \to value(fx \text{ ` } bind \text{ ` } k))) && (17)\\
= \;& m(ValueBind(\lambda x \to fx(ValueBind(value\ .\ k)))) && (18)
\end{aligned}$$

Here, (15), (17) and (18) are instances of the induction hypothesis

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

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 (18) 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; the induction principle would be justified by induction on the height of the tree.

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. We have not surveyed the HOAS literature to see how widespread this bug is. 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 lambda-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 [17, 9, 5]. 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* [16]) or by its `fold` function (corresponding to the Church representation); neither of these representations yield 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. [3], 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.

8. 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.

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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