Relating Models of Backtracking

Mitchell Wand and Dale Vaillancourt

Northeastern University
Outline

- Introduce backtracking computation.
  - Two well-known models.
  - A monadic framework for backtracking.

- How are the monads related?
  - Past attempts.
  - Our solution.

- Relate an implementation of one model to both models.

- Conclude.
Backtracking computation

- Simply-typed $\lambda$-calculus with backtracking (and constants).
- Might write something like:

\[
\text{nats-from } n = n \lor \text{nats-from}(n + 1)
\]
\[
\text{nats} = \text{nats-from } 0
\]

- How might we model such a language?
Two models

- Represent a backtracking computation as a stream of answers.

\[
\begin{align*}
\text{fail}^S & = \langle \rangle \\
nats^S & = \langle 0, 1, 2, \ldots \rangle
\end{align*}
\]
Two models

- Represent a backtracking computation as a stream of answers.

  $$\text{fail}^S = \langle \rangle$$
  $$\text{nats}^S = \langle 0, 1, 2, \ldots \rangle$$

- Represent a backtracking computation as a procedure that consumes two values.

  $$\text{fail}^K = \lambda \kappa \phi. \phi$$
  $$\text{nats}^K = \lambda \kappa \phi. \kappa 0 ((\text{nats-from} 1)^K \kappa \phi)$$
Two models

- Represent a backtracking computation as a stream of answers.

\[
\begin{align*}
\text{fail}^S &= \langle \rangle \\
\text{nats}^S &= \langle 0, 1, 2, \ldots \rangle
\end{align*}
\]

- Represent a backtracking computation as a procedure that consumes two values.

\[
\begin{align*}
\text{fail}^K &= \lambda \kappa \phi. \phi \\
\text{nats}^K &= \lambda \kappa \phi. \kappa \ 0 \ ((\text{nats-from} \ 1)^K \ \kappa \ \phi)
\end{align*}
\]

- How are these models related?
The models’ relationship

Maybe the two-continuation terms arise by …

- Church-encoding the streams?
- Scott-encoding the streams?
- Final algebra-encoding the streams?
They’re backtracking monads

Hughes ’95 defines a backtracking monad and notes that both models are backtracking monads.

\[\text{unit} : \alpha \rightarrow T\alpha\]
\[\text{bind} : T\beta \rightarrow (\beta \rightarrow T\alpha) \rightarrow T\alpha\]
\[\text{disj} : T\alpha \rightarrow T\alpha \rightarrow T\alpha\]
\[\text{fail} : T\alpha\]
They’re backtracking monads

We have some new monad laws:

\[ \text{disj } M \text{ fail } = M \]
\[ \text{disj fail } M = M \]
They’re backtracking monads

We have some new monad laws:

\[ \text{disj } M \text{ fail } = M \]

\[ \text{disj } \text{ fail } M = M \]

\[ \text{disj } (\text{disj } M_1 M_2) M_3 = \text{disj } M_1 (\text{disj } M_2 M_3) \]
They’re backtracking monads

We have some new monad laws:

\[ \text{disj } M \text{ fail } = M \]
\[ \text{disj fail } M = M \]
\[ \text{disj } (\text{disj } M_1 \ M_2) \ M_3 = \text{disj } M_1 \ (\text{disj } M_2 \ M_3) \]
\[ \text{bind } (\text{disj } M_1 \ M_2) \ M_3 = \text{disj } (\text{bind } M_1 \ M_3) \]
\[ (\text{bind } M_2 \ M_3) \]
They’re backtracking monads

We have some new monad laws:

\[ \text{disj } M \text{ fail } = M \]
\[ \text{disj } \text{ fail } M = M \]
\[ \text{disj } (\text{disj } M_1 M_2) M_3 = \text{disj } M_1 (\text{disj } M_2 M_3) \]
\[ \text{bind } (\text{disj } M_1 M_2) M_3 = \text{disj } (\text{bind } M_1 M_3) \]
\[ \quad (\text{bind } M_2 M_3) \]
\[ \text{bind } \text{ fail } M = \text{fail} \]
Monadic Semantics

Object Language

Translation

Monadic Metalanguage

$S$ Semantics $K$ Semantics

Stream Model

Two-Continuation Model
Two Monads

The stream monad $S$:

$$ \text{unit}^S v = \langle v \rangle $$

$$ \text{bind}^S c f = \text{flatten}(\text{map} f c) $$

$$ \text{disj}^S s t = s \hat{\land} t $$

$$ \text{fail}^S = \langle \rangle $$
Two Monads

The two-continuation monad $K$:

$\text{unit}^K v = \lambda k \phi. k v \phi$

$\text{bind}^K c f = \lambda k \phi. c (\lambda a \phi. (f a) k \phi) \phi$

$\text{disj}^K c d = \lambda k \phi. c k (d k \phi)$

$\text{fail}^K = \lambda k \phi. \phi$
Even more monads . . .

- Backtracking Monad Transformers (Hinze 1999).
- The Algebra of Logic Programming, Embedding Prolog into Haskell (Seres & Spivey 1999).
- Typed Logical Variables in Haskell (Claessen et al. 2000).
Hughes derives the two-continuation model as an “optimized” representation of streams.

Unfortunately the derivation only works in one direction.

Hinze’s extension to monad transformers suffers the same difficulty.
Their insight is that we really want:

\[ M^K \text{ cons nil} = M^S \]

- They connect a stream with its encoding using a monad morphism.
- They claim to use a Church encoding.
Our insight is that they encode:

\[ \langle a_1, a_2, \ldots, a_n \rangle \]

as

\[ \lambda \kappa. (\kappa a_1) \circ (\kappa a_2) \circ \ldots \circ (\kappa a_n) \]
Two Issues

- Let $M = \text{unit}(\lambda x. (\text{unit} \ 42))$. Then

  $$M^K \ \text{cons} \ \text{nil} = \langle \lambda x. (\lambda \kappa. \kappa 42) \rangle$$

  $$M^S = \langle \lambda x. \langle 42 \rangle \rangle$$

  When $M$ involves higher-order quantities,

  $$M^K \ \text{cons} \ \text{nil} \neq M^S$$

- They do not consider recursion.
We need $R_{\alpha} \subseteq \alpha^S \times \alpha^K$ for each type $\alpha$ such that:

$$f \ R_{\alpha \rightarrow \beta} \ g \ \text{iff} \ a \ R_{\alpha} \ b \ \implies \ (fa) \ R_{\beta} \ (gb)$$

A type-indexed family of relations satisfying this property is called a **logical relation**.
Our Logical Relation

Define $R_\alpha \subseteq \alpha^S \times \alpha^K$:

- At scalar type $\sigma$, $R_\sigma$ is the identity.
Our Logical Relation

Define $R_\alpha \subseteq \alpha^S \times \alpha^K$:

- At scalar type $\sigma$, $R_\sigma$ is the identity.
- $f \ R_\alpha \rightarrow_\beta \ g$ iff $a \ R_\alpha \ b \implies (fa) \ R_\beta \ (gb)$
Our Logical Relation

Define $R_\alpha \subseteq \alpha^S \times \alpha^K$:

- At scalar type $\sigma$, $R_\sigma$ is the identity.
- $f \ R_{\alpha \rightarrow \beta} \ g$ iff $a \ R_\alpha \ b \implies (fa) \ R_\beta \ (gb)$
- At computation type $T_\alpha$:

\[
\frac{a_1 \ R_\alpha \ b_1, \ldots, \ a_n \ R_\alpha \ b_n}{\langle a_1, \ldots, a_n \rangle \ R_{T_\alpha} \ \lambda \kappa. (\kappa b_1) \circ \cdots \circ (\kappa b_n)}
\]
Relating Infinite Computations

- Require $\perp_{T\alpha^S} R_{T\alpha} \perp_{T\alpha^K}$.
- Close $R_{T\alpha}$ under limits of $\omega$-chains in $T\alpha^S \times T\alpha^K$. 
Theorem. If $M : \alpha$, then $M^S R_{\alpha} M^K$.

Proof: For each constant $c : \alpha$, we must show $c^S R_{\alpha} c^K$.

Then the result follows from the fundamental theorem of logical relations.
A Consequence

We recover Danvy, Grobauer, and Rhiger’s result:
If $M : T\sigma$, then $M^K \text{ cons } \text{nil} = M^S$

$$M^K = \lambda \kappa.(\kappa 0) \circ (\kappa 1) \circ (\kappa 2) \circ \ldots$$

$$M^S = \langle 0, 1, 2, \ldots \rangle$$
We recover Danvy, Grobauer, and Rhiger’s result:
If $M : T\sigma$, then $M^K \text{ cons } nil = M^S$

\[ M^K = \lambda \kappa. (\kappa \ 0) \circ (\kappa \ 1) \circ (\kappa \ 2) \circ \ldots \]

$R_\sigma$

\[ M^S = \langle 0, 1, 2, \ldots \rangle \]
A step further...

We relate the stream model to an operational semantics that implements it.

- Translate the monadic metalanguage terms into a lazy PCF called mPCF.
- Extend a standard adequacy theorem for the semantics of lazy PCF.
- Use both the logical relation and adequacy to relate the $S$ semantics with the operational semantics.
mPCF extends lazy PCF

- *unit* and *bind* to build and sequence possibly nonterminating computations.
- A fixed-point operator.
- Include *cons* and *nil* to represent initial continuations.
The translation looks much like the $K$ semantics:

\[
\begin{align*}
[\text{unit}] &= \lambda a.(\lambda \kappa \phi. \kappa a \phi) \\
[\text{disj}] &= \lambda ab.(\lambda \kappa \phi. a \kappa (b \kappa \phi)) \\
&\vdots
\end{align*}
\]
mPCF Semantics

- A call-by-name operational semantics.
- The denotational semantics $L$ interprets terms in the lifting monad.

\[
\begin{align*}
\text{unit}^L a &= \text{lift}(a) \\
\text{bind}^L c f &= \text{case } c \text{ of } \bot \rightarrow \bot \\
&\quad \mid \text{lift}(d) \rightarrow f(d)
\end{align*}
\]
Implementing the streams
Adequacy

**Theorem.** $M$ reduces to a value iff either
- $M$ is of value type.
- $M$ is of computation type and denotes a non-$\bot$ element.

**Proof:** We adapt a standard proof technique (Winskel 1993). The reverse direction requires coinduction to relate infinite streams.
Assume $M : T\sigma$ and $M^S = \langle d \rangle^s$. Then

$$[M]^L \text{ cons}^L \text{ nil}^L = M^K \text{ cons}^L \text{ nil}^L$$

$$= M^S$$

$$= \langle d \rangle^s$$

By adequacy, $[M] \text{ cons} \text{ nil} \rightarrow^* \text{ cons} \nu M'$, where

$$d = \nu^L$$

$$s = M'^L$$
Contributions

- We have related two models of backtracking computation.
- We accommodate higher-order quantities and infinite streams.
- We related the stream model to an operational semantics that implements it.
Contributions

- We have related two models of backtracking computation.
- We accommodate higher-order quantities and infinite streams.
- We related the stream model to an operational semantics that implements it.

Thank you!