# Putting Failure in Context: Abstract Data Types of Computations and Contexts

Mitchell Wand*
College of Computer and Information Science
Northeastern University
Boston, MA 02115, USA
wand@ccs.neu.edu

## ABSTRACT

There are two standard models of backtracking programs: a model in which the many possible answers are represented as a stream, and a model in which the action of the backtracking program is represented by two continuations: a success continuation and a failure continuation. These two models, both intuitively plausible, have never been (to our knowledge) been formally related. We show how the two-continuation model can be derived from the stream model.

We do this by treating computations and contexts as abstract data types. We start by representing computations as elements in a term algebra. We derive an operational semantics for this term algebra by representing it in an extension of call-by-value PCF. This allows us to build an algebra of contexts, in which the usual representation of contexts as terms with holes is an initial algebra. We then represent these contexts as a final algebra using two observers, one representing the action of the context on success, and one representing its action on failure. Finally, we regain a compositional semantics by representing computations by their action on contexts.

## 1. INTRODUCTION

The semantic treatment of backtracking programs generally follows one of two paths. One path treats a program as producing a stream of values corresponding to the choices the program makes. This may be roughly represented as a monad `S`, where

```
S(A) = Str(A)

unit a = cons a nil
bind c f = mapappend f c
disj c1 c2 = append c1 c2
fail = nil
```

*Work supported by the National Science Foundation under grant number CCR-0097740.

where `mapappend : (A -> Str(B)) -> (Str(A) -> Str(B))` is the function such that `(mapappend f w)` concatenates the results of `f` operating on each of the elements of `w`.[1]

In the application to logic programming, this monad is augmented with a state containing the current substitution. Conjunction is represented by `bind`, and disjunction, represented by `disj`, saves the state. Furthermore, computations may produce infinite streams or be non-terminating. Our treatment of the backtracking monad below will incorporate all these attributes.

This model is due to [1, 15]; the idea of formulating it as a monad was first proposed in [14] and worked out in more detail in [10, 2].

A second model for backtracking programs may be formulated using two continuations: a *success continuation* and a *failure continuation*. In this model, we set up the domains

```
KF = () -> Ans
KS(A) = A -> S -> KF -> Ans
```

Then the monad is given by

```
U(A) = KS(A) -> S -> KF -> Ans

unit a     = λ k s kf. k a s kf
bind c f   = λ k s kf.
                 c (λ a s kf. f a k s kf) s kf
disj c1 c2 = λ k s kf. c1 k s (λ _. c2 k s kf)
fail       = λ k s kf. kf ()
```

So far as we know, this model first appeared in [5], and is presented in [1] as a cps implementation of the stream model. However, we know of no formal exploration of the relationship between these two models.

[1]We will write most of our definitions in a mostly-ascii pseudocode. We use left-associative concatenation for application, and parentheses for grouping as needed or for clarity. We use `()` to denote the unique element of the singleton type, and `λ _` to denote the binder of a procedure that expects that as its argument (corresponding to a 0-ary procedure in Scheme). Hence `(e)` is identical to `e`, but `e()` denotes the application of `e` to no arguments. The remainder of the notation will be self-explanatory (we hope). We are not entirely happy with this typesetting arrangement, and we hope to improve it for the final version (reactions are welcomed!).

Our goal is to show a connection between these two denotational models. We do this by showing that they share a common operational semantics.

We start with the stream monad. We give it an operational semantics as follows: first, we formulate the computations in the monad as an abstract data type, and represent them as higher-order terms. We get an operational semantics for these terms by interpreting them in an extension of call-by-value PCF, for which we have an adequate operational semantics. By simulating the reduction of the translated terms, we can recover an adequate operational semantics for the terms in our original algebra of computations.

We can formulate that operational semantics as a context machine, in the style of Felleisen [6]. This machine has two components: a computation and a context in which it is to be evaluated. We then formulate the contexts as an abstract data type, for which the usual representation is as terms with holes is an initial algebra.

We then move from initial to final algebras: we represent contexts by their actions under two observers (success and failure), thus getting continuations. Finally, we represent computations by their actions on contexts. The result is the desired compositional two-continuation semantics.

The remainder of the paper is organized as follows:

- In section 2, we warm up by applying these techniques to a simpler example: the exception monad.
- In section 3, we present the semantics of streams and the extension of $\mathrm{PCF}_V$ to streams in some detail.
- Section 4 defines the stream-based backtracking monad, and section 5 derives its operational semantics.
- Section 6 presents the algebra of contexts and concludes with the derivation of the two-continuation model.
- Three appendices give short treatments of initial and final algebras, the adequacy theorem for $\mathrm{PCF}_V$ with streams, and extending $\mathrm{PCF}_V$ with types for computations as terms.

## 2. WARM-UP: THE EXCEPTION MONAD

We begin with the exception monad. The exception monad is given by

```
T(A) = A + 1

unit_A a = inL(a)
bind_AB c f = case c of
                inL(a) => f a
              | inR() => inR()
raise_A = inR()
handle_A c1 c2 = case c1 of
                   inL(a) => inL(a)
                 | inR() => c2
```

Normal values are represented in the left summand of `A+1`; an error is represented by the right summand. Here `unit` returns a normal value by injecting it into the left-hand summand. `bind` does the usual sequencing on normal values but propagates error values. `raise` signals an error by returning a value in right summand. In `(handle c1 c2)`, `c1` is evaluated. If it returns a normal value, that value is propagated; if it returns an error, the error is caught and `c2` is evaluated instead.

We reformulate this as an abstract data type. Our data type will have sorts `Comp(A)` for each type `A`, and 5 families of operators

```
unit_A : A -> Comp(A)
bind_AB : Comp(A) -> (A -> Comp(B)) -> Comp(B)
raise_A : Comp(A)
handle_A : Comp(A) -> Comp(A) -> Comp(A)
eval_A : Comp(A) -> A+1
```

for each type `A` and `B`, subject to the axioms:

```
eval_A (unit_A a) = inL(a)
eval_A (bind_BA c f) = case (eval_B c) of
                         inL(b) => (eval_A (f b))
                       | inR() => inR()
eval_A (raise_A) = inR()
eval_A (handle_A c1 c2) = case (eval_A c1) of
                            inL(a) => inL(a)
                          | inR() => (eval_A c2)
```

The standard monad provides such an algebra, with `Comp(A)` $=$ `A+1`, with `unit`, `bind`, `raise`, and `handle` as defined above, and with `eval` interpreted as the identity.

We would like to provide something more like an initial algebra for this data type. This would interpret `Comp(A)` as a (higher-order) term algebra with terms given by

```
Comp(A) = unit_A of A
        | (all B) . bind_BA of
                      Comp(B) * (B -> Comp(A))
        | raise_A
        | handle_A of Comp(A) * Comp(A)
```

In this algebra the constructors are the term constructors. Note that this data type is an infinite sum, since there is a constructor `bind_BA` for each type `B`.

But how is `eval` to be defined? The axioms do not provide an immediate answer: because of the `(eval_A (f a))`, they do not constitute a structural induction.

To find a term algebra for computations, we provide a compositional translation of the language into a target language for which there is an adequate operational semantics, and simulate the reductions of the resulting terms.

We choose to code our language in an extension of $\mathrm{PCF}_V$, the call-by-value version of PCF. In appendix C, we show how we can extend $\mathrm{PCF}_V$ with types `Comp(A)` (at least for types `A` that do not involve `Comp`), while preserving the adequacy of the usual reduction-style (small-step) operational semantics. Our language will have

```
<bind c f, E> -> <c, <bindE f E>>

<handle c1 c2, E> -> <c1, <handleE c2 E>>

<unit a, []> -> inL(a)      (a final value)

<unit a,  E[case [] of inL(a) => (eval (f a)) | inR() => inR()]> -> <(f a), E>

<unit a, E[case [] of inL(a) => inL(a) | inR() => (eval c2)]> -> <unit a, E>

<raise, []> -> inR()     (an unhandled error)

<raise,  E[case [] of inL(a) => (eval (f a)) | inR() => inR()]>  -> <raise, E>

<raise, E[case [] of inL(a) => inL(a) | inR() => (eval c2)]> -> <c2, E>
```

**Figure 1: Context machine for the exception monad**

constructors that evaluate all their arguments, and a `case` construct that evaluates its first argument and chooses a branch appropriately, as suggested by the use of `case` above. Then `eval` can be defined using the fixed-point operator of $PCF_V$, so long as the initial program involves evaluation at only finitely many types. This will be true for any closed computation term.

Since we have an adequate operational semantics for $PCF_V$, we can see what this reduction semantics does for terms of the form `(eval c)` for each closed term `x` of type `Comp(A)`. In general, we will need to consider the reduction of terms of the form `E[eval c]` for any $PCF_V$ evaluation context `E`. We write `E_1[E_2]` for the evaluation context created by plugging `E_2` into the hole of `E_1`. Now we can watch the reduction of `E[eval c]` for each possible `c`:

```
E[(eval_A (unit_A a))] -> E[(inL a)]

E[(eval_A (bind_BA c f))]
-> E[case (eval_B c) of
       inL(b) => (eval_A (f b))
     | inR() => inR()]
=  E[case [] of
       inL(b) => (eval_A (f b))
     | inR() => inR()] [(eval_B c)]

E[(eval_A raise_A)] -> E[(inR ())]

E[(eval_A (handle_A c1 c2))]
-> E[case (eval_A c1) of
       inL(a) => inL(a)]
     | inR() => (eval_A c2) ]
=  E[case [] of
       inL(a) => inL(a)]
     | inR() => (eval_A c2)] [(eval_A c1)]
```

From this we see that the evaluation of `E[eval_A c]` gives rise to contexts that grow in only two ways, and are given by the grammar

```
E ::= []
   | E[case [] of
         inL(b) => (eval_A (f b))
       | inR() => inR()]
   | E[case [] of
         inL(a) => inL(a)
       | inR() => (eval_A c2)]
```

We can simulate the behavior of `E[(eval c)]` by a machine that takes a configuration `<c, E>` and simulates the operational semantics of `E[(eval c)]`. The machine is shown in figure 1. We omit the subscripts on the operators, since they do not affect the operation of the machine. For `bind` and `handle`, the action is obvious. For `unit` and `raise`, we need to consider the form of `E`. For `(unit a)`, we plug `inL(a)` into `E` and simulate the result. Similarly for raise, we consider the result of putting `inR()` in the hole for each kind of evaluation context E.

Because this reduction semantics simulates the reduction semantics in $PCF_V$, and the reduction of $PCF_V$ is sound and adequate we know that if the reduction of `<c, E>` terminates, it yields a representation of the meaning of `E[eval c]`, and furthermore, if the reduction fails to terminate, then the reduction of `E[eval c]` in $PCF_V$ fails to terminate, and hence `E[eval c]` denotes the bottom element.

We'd like to analyze the contexts that are generated by this semantics as an abstract data type. Our data type will have a doubly-indexed family of sorts `E_AB`, where `E_AB` represents the contexts with hole of type `A` and result of type `B+1`, and families of operators

```
idE_A : E_AA
bindE_ABC : (A -> Comp(B)) -> E_BC -> E_AC
handleE_AB : Comp(A) -> E_AB -> E_AB
run_AC : Comp(A) -> E_AC -> C+1
```

for each type `A`, `B`, and `C`, subject to the equations

```
run_AC (bind_BA c f) E = run_BC c (bindE_BAC f E)
run_AC (handle_AC c1 c2) E = run_AC c1 (handleE_AC c2 E)
run_AA (unit_A a) idE_A = inL(a)
run_AC (unit_A a) (bindE_ABC f E) = run_BC (f a) E
run_AC (unit_A a) (handleE_AC c E) = run_AC (unit_A a) E
```

```
run_AA (raise_A) idE_AA = inR()
run_AC (raise_A) (bindE_ABC f E) = run_BC (raise_B) E
run_AC (raise_A) (handleE_AC c E) = run_AC c E
```

We can represent the contexts by a term algebra, in which the constructors build context terms, and run causes the machine to run to completion (an induction on the number of steps). This would be an initial algebra representation. Instead, we use a *final algebra* representation, in which each element of the algebra is represented by its action on the observer or observers. Appendix A gives a short introduction to final algebras.

If we choose to represent a context by its action on a the single observer of substitution, we get the usual continuations [6, 7, 3, 4].

Instead, we choose to have two observers: one for the action of the context on success, and one for its action on failure. We represent a context `E : E_AC` by the pair of functions

```
<λ a. run_AC (unit_A a) E, λ _. run_AC raise_A E>
   : ((A -> C+1) * (() -> C+1))
```

The constructors take the old behavior and produce the new behavior. Observing the equations above, we can define

```
idE_A                   = <inL, inR>
bindE_ABC f <ks,kf>  = <λ a. run_BC (f a) <ks, kf>,
                          kf>
handleE_AC c <ks, kf> = <ks, λ _. run_AC c <ks, kf>>
```

With this representation, we can define `run` as:

```
run_AC (bind_BA c f) E = run_BC c (bindE_BAC f E)
run_AC (handle_A c1 c2) E = run_A c1 (handleE_AC c2 E)
run_AC (unit_A a) E = (E2ks E) a
run_AC (raise_A) E = (E2kf E) ()
```

where `E2ks` and `E2kf` select the appropriate component of the context.

Last, we represent the algebra of computations by its action on contexts. That is, we represent a computation `c` of type `A` by the function

```
Λ C. λ E : E_AC. run_AC c E
```

and the observer `run_AC` becomes

```
λ c:Comp(A) λ E:E_AC. (c E).
```

We can now work out the representations of the computation constructors in this model. Define the types `K_AC = A -> C+1` and `F_C = () -> C+1`. Unwinding all the definitions, we get:

```
unit_A a        = Λ C. λ ks:K_AC kf:F_C. ks a
bind_BA c f     = Λ C. λ ks:K_AC kf:F_C.
                      c (λ a:A. f a ks kf) kf
raise_A         = Λ C. λ ks:K_AC kf:F_C. kf ()
handle_A c1 c2 = Λ C λ ks:K_AC kf:F_C.
                      c1 ks (λ _. c2 ks kf)
```

which is the usual two-continuation model of exception handling in the polymorphic typed lambda calculus.

## 3. EXTENDING PCF$_V$ TO STREAMS

Our goal is to do the same thing for the backtracking monad. In the backtracking monad, the result of a backtracking program is considered to be a stream of possible answers, and the backtracking operators manipulate these streams.

Following the plan above, we start by considering the semantics of streams and their representation in an extension of PCF.

We give a denotational semantics for this language following Winskel's presentation in [22]. For each type $\sigma$, we define a cpo (not necessarily with bottom) $D^\sigma$ of semantic values of type $\sigma$. Environments are populated with elements of $D^\sigma$, but an expression of type $\sigma$, given a suitable environment, produces a result which is an element of $D^\sigma_\perp$, where the additional bottom element signifies non-termination. In this scheme, the $D^\sigma$ are defined as follows:

$$D^{\mathtt{int}} = \{0,1,2,\dots\}$$
$$D^{\mathtt{bool}} = \{\mathit{true},\mathit{false}\}$$
$$D^{(\sigma\to\tau)} = [D^\sigma \to D^\tau_\perp]$$

where $D^{\mathtt{int}}$ and $D^{\mathtt{bool}}$ have the discrete ordering, and $[D \to E]$ denotes all continuous functions from $D$ to $E$. We also assume that we have binary sums and and finite products, including the empty product.

We extend this model by adding new types $Str(\tau)$ for each type $\tau$, and defining

$$\begin{aligned} D^{Str(\tau)} = \{\langle\rangle\} & \\ & \cup \{\langle v_1,\dots,v_n\rangle \mid n \geq 1, v_i \in D^\tau\} \\ & \cup \{\langle v_1,\dots,v_n \,.\, \perp\rangle \mid n \geq 1, v_i \in D^\tau\} \\ & \cup \{\langle v_1,\dots\rangle \mid v_i \in D^\tau\} \end{aligned}$$

As usual, a stream is either a finite stream, an incomplete stream, or an infinite stream. Each element of a stream is a value. We use $w$ to range over streams. We write $v.w$ for value $v$ prefixed to stream $w$, and infix @ for the concatenation of streams. If $w$ is an incomplete or infinite stream, then $w \mathbin{@} w' = w$. Note that @ is associative.

The ordering on $D^{Str(\tau)}_\perp$ is the smallest reflexive, transitive relation such that

$$\perp \leq w \qquad \frac{v \leq v' \qquad w \leq w'}{v.w \leq v'.w'}$$

The ordering on $D^{Str(\tau)}$ is given by its inclusion in $D^{Str(\tau)}_\perp$.

Having defined the domains $D^{Str(\tau)}$, we now add constants to the language to manipulate these vaues. We add new families of constants `nil`, `cons`, `car`, `cdr`, and `append` to the language. As usual for PCF, we equip these with subscripts to indicate their type (for example, $\mathtt{car}_\tau$ will have type $Str(\tau) \to \tau$), and then we omit the subscripts since they are always obvious from context. We equip each of these constants with the evident denotational semantics.

The operational semantics of $\mathrm{PCF}_{VS}$ is given in a small-step style, using reduction contexts in the manner of Felleisen. To deal with streams, we add new syntactic values `V`, reduction contexts `E`, and reduction rules as follows:

```
V ::= ... | nil | cons V M
```

Thus a stream-valued expression terminates as soon as it is known that either the stream is empty, or its first element is known. Subsequent elements of the stream can be extracted if the stream appears in a suitable reduction context. The reduction contexts are extended by

```
E ::= ... | cons E M | car E | cdr E | append E M
```

and we add new rewrite rules:

```
car (cons V M)  -> V
cdr (cons V M) -> M
append (cons V M) N -> cons V (append M N)
append nil M -> M
```

This formulation makes a stream-valued expression converge as soon as its first element converges; thus streams are eager in the their first element and lazy in their tails. `append` is lazy in its second argument; the second argument is not evaluated until the first argument is exhausted.

The subject-reduction and progress theorems hold evidently. Furthermore, this operational semantics is adequate; some of the details are presented in appendix B.

This is not intended as a general-purpose embedding of streams in $\mathrm{PCF}_V$. For example, all user-defined procedures will be strict in all their arguments; the laziness introduced by `append` and `cons` is hidden by any lambda binding. Nevertheless, it is sufficient for our purposes.

## 4. THE BACKTRACKING MONAD

We are interested in modelling programs with a state that can be reset by backtracking. For example, in logic programming, the state will contain the current substitution. Backtracking programs may also contain non-terminating computations.

To build a monad with these effects, we first define a $\mathrm{PCF}_{VS}$ term

$$\mathtt{lift} : (\sigma \to Str(\tau)) \to (Str(\sigma) \to Str(\tau))$$

such that `(lift f w)` is obtained by concatenating the output of `f` on each of the elements of stream `w`. `lift` satisfies the equations

```
lift f nil = nil
lift f (cons V M) = f(V) @ (lift f M)
```

so it can be defined as a fixed point. Furthermore, `lift` also satisfies

```
lift f (w @ w') = (lift f w) @ (lift f w')
```

With this preparation, we can express this model as a monad $T$ where

$$T(A) = S \to Str(A \times S)_\perp$$

and `S` is the set of states. Since procedure types in $\mathrm{PCF}_{VS}$ automatically include bottom in their codomains, this means we can interpret $T(A)$ as the $\mathrm{PCF}_{VS}$ type $S \to Str(A \times S)$

The monad operations are given by

```
unit_A a = λ s. cons (a,s) nil
bind_AB c f = λ s. f^ (c s)
disj_A c1 c2 = λ s. (c1 s) @ (c2 s)
fail_A = λ s. nil
```

where `f^` is defined as `lift (λ (a,s).(f a s))`.

When this monad is applied to logic programming, the state typically contains a substitution, and conjunction is modelled by bind. An essentially identical monad is proposed in [2].

## 5. OPERATIONAL SEMANTICS FOR THE MONAD

As before, we turn the monad into an algebra `Comp(A)` with 5 operations such that:

```
eval_A (unit_A a) s = cons (a,s) nil
eval_A (bind_BA c f) s =  (eval_A . f)^ (eval_B c s)
eval_A (disj_A c1 c2) s = (eval_A c1 s) @ (eval_A c2 s)
eval_A (fail_A) s =  nil
```

where `(eval_A . f)` denotes the composition of `eval_A` and `f`.

If we define `Comp(A)` as `T(A)`, the constructors as at the end of the last section, and `eval` as the identity, then we get a model of this algebra.

As before, we would like to present this algebra as a term algebra, with terms defined by

```
Comp(A) = unit_A of A
        | (all B) . bind_AB of
                      Comp(B) * (B -> Comp(A))
        | handle_A of Comp(A) * Comp(A)
        | fail_A
```

We can add such values as types in $\mathrm{PCF}_{VS}$, just as we did for $\mathrm{PCF}_V$ in Appendix C, and then define the `eval_A` using the least-fixpoint operator, again using the fact that a closed program can involve only finitely many types. This gets us the desired presentation of the algebra.

As we did before, we can consider the reduction of terms `E[eval_A c]` where `c` is a closed term of type `Comp(A)` and `E` is a reduction context in $\mathrm{PCF}_{VS}$ +terms.

Observing the behavior of `eval`, we get the following reductions:

```
E[(eval_A (unit_A a) s)]
-> E[ cons (a,s) nil]

E[(eval_A (bind_BA c f) s)]
-> E[((eval_A . f)^ [])] [(eval_B c s)]

E[(eval_A (disj_A c1 c2) s)]
-> E[([]@(eval_A c2 s))][(eval_A c1 s)]

E[(eval_A (fail_A) s)]
-> E[nil]
```

This gives us three kinds of contexts:

```
E ::= []
   |  E[((eval_A . f)^ [])]
   |  E[([]@(eval_A c2 s))]
```

In order to derive an operational semantics, we show that every context constructed in this way can be represented in a restricted form.

LEMMA 1. *Every reduction context E is denotationally equal to one in the form K@F, where*

```
K ::= [] | K[((eval_A . f)^ [])]
F ::= nil | K[(eval_A c s)] @ F
```

PROOF. First, observe that the `K`-contexts are distributive, that is for any `K` and any streams `w1` and `w2`, `K[w1 @ w2] = K[w1] @ K[w2]`.

Then we have:

```
[] = ([] @ nil)

(K @ F)[((eval_A . f)^ [])]
= K[((eval_A . f)^ [])] @ F

(K @ F)[([]@(eval_A c2 s))]
= K[([]@(eval_A c2 s))] @ F
= (K[] @ K[(eval_A c2 s)]) @ F
= K[] @ (K[(eval_A c2 s)] @ F)
```

$\blacksquare$

Now every reduction context is of the form `E[(eval c s)] = (K@F)[(eval c s)]`, so we can write down an operational semantics for a context machine `<c, s, K, F>`. This machine is shown in figure 2. Since the behavior of `eval`, `unit`, `bind`, etc. is uniform in their subscript, we omit the type subscripts as before. The soundness of each rule can be checked by observing that each step corresponds to a reduction of `(K@F)[(eval c s)]` in $\mathrm{PCF}_{VS}$, with the reduction context represented as `K@F`. For the failure rule, observe that for any `K`, `K[nil] -> nil`.

## 6. THE ALGEBRA OF CONTEXTS

Now we proceed as we did before. We first formulate the algebra of success and failure contexts. The algebra will have two families of sorts: `K_AC`, representing success contexts with a hole of type `A` and with a result of type `Str(C)` and `F_C`, representing failure contexts with a result of type `Str(C)`. We will have families of operations

```
idK_A : K_AA
bindK_ABC : (B -> Comp(A)) -> K_AC -> K_BC
emptyF_A : F_A
extF_AC : K_AC -> Comp(A) -> S -> F_C -> F_C
run_AC : Comp(A) -> K_AC -> S -> F_C -> Str(C)
```

for each type `A`, `B`, and `C`, subject to the equations

```
run_AC (unit_A a) (bindK f K) s F
  = run_B (f a) K s F
run_AA (unit_A a) idK_AA s F = cons (a, s) F
run_AC (bind_BA c f) K s F
  = run_BC c (bindK_BAC f K) s F
run_AC (disj_A c1 c2) K s F
  = run_AC c1 K s (extF_AC K c2 s F)
run_AC fail_A K s (extF_AC K' c' s' F)
  = run_AC c' K' s' F'
run_AC fail_A K s emptyF_C = nil
```

As before, representing the computations as higher-order terms, contexts as terms with holes and `run` as the action of the machine yields a presentation of this algebra.

Again as before, we adopt a final-algebra representation. We represent contexts as a pair of observers; one representing the action of the context on `(unit a)`, and one representing its actionn on failure.

We begin with the failure action. The action of the context `K@F` on failure depends only on `F`, so we represent `F : F_C` as a function of no arguments, as `λ _.F : () -> Str(C)`

```
emptyF_A           = λ _. nil
extF_AC K' c' s' F = λ _. run_AC c' K' s' F
```

The action of the context `K@F` on `(unit a)` depends on `s` and `F` as well as `K`, so we represent it as a function of `a`, `s`, and `F`:

```
<unit a, s, K[((eval.f)^ [])], F> -> <(f a), s, K, F>

<unit a, s, [], F> -> cons (a,s) F    (a syntactic value)

<bind c f, s, K, F> -> <c, s, K[((eval.f)^ [])], F>

<disj c1 c2, s, K, F> -> <c1, s, K, K[eval c2 s]@F>

<fail, s, K, K'[eval c' s']@F> -> <c', s', K', F>

<fail, s, K, nil> -> nil    (a syntactic value)
```

**Figure 2: Context machine for the backtracking monad**

```
idK_A         = λ a s F. cons (a,s) F
bindK_ABC f K = λ a s F. run_B (f a) K s F
```

Thus `K_AC` is represented by functions in `A -> S -> F_C -> Str(C)`.

With this representation for `K` and `F`, we can implement the operations `run_AC` by:

```
run_AC (unit_A a) K s F     = K a s F
run_AC (bind_BA c f) K s F  = run_BC c (bindK f K) s F
run_AC (disj_A c1 c2) K s F = run_AC c1 K s
                                 (extF K c2 s F)
run_AC (fail_A) K s F       = F ()
```

Last, we represent `Comp(A)` by its action on `K`, `K`, and `F`, so that

```
run_AC c K s F = c K s F
```

and

```
Comp(A) = (∀ C). K_AC -> S -> F_C -> Str(C)
```

Hence we have

```
unit_A a     = Λ C. λ K:K_AC s F:F_C. K a s F
bind_AB c f  = Λ C. λ K:K_BC s F:F_C.
                 c (λ a s F:F_C. (f a) K s F) s F
disj_A c1 c2 = Λ C. λ K:K_AC s F:F_C.
                 c1 K s (λ _.c2 K s F)
fail_A       = Λ C. λ K:K_AC s F:F_C. F ()
```

which is the standard two-continuation model of backtracking, as above, with type annotations. Note also that `unit` and `bind` are the same as in the usual continuation monad, up to $\eta$-conversion.

## 7. RELATED WORK

The standard definition of the operational semantics of Prolog is [21].

The relationship between contexts and continuations was introduced in [6]. The idea that continuations were a final algebra representation of contexts was introduced in [7]. Danvy and Nielson, in [3, 4], suggest the opposite direction: that contexts are defunctionalized continuations.

Hinze [10] shows how to derive a monad transformer that adds backtracking to an arbitrary monad. Although his construction works over an arbitrary monad, he derives the operational semantics as the workings of a simplifier (the "observer" $\omega$), rather than from an adequacy theorem.

The idea of getting an adequate operational semantics by providing a compositional translation of the language into a target language for which there is an adequate operational semantics, and simulating the reductions of the resulting terms, is an old one. This strategy was implicit in our early work on compiler correctness [17, 18] and in work on self-interpreters for the lambda calculus [12, 19, 20].

Plotkin and Power [13] provide a technique for deriving an operational semantics for a certain class monads. Their techniques are quite different from ours; we do not know at present to what extent theirs applies to the cases at hand.

Final algebras are due to [9, 16, 11].

## 8. FURTHER WORK

The techniques in this paper ought to extend to handle the cut operator, for which [15] provides a stream-based model, but we have not yet succeeded in doing so.

Our model is depth-first, but Spivey and Seres [14] show how to obtain other search strategies by minimal modifications of the stream model. It would be interesting to see if these techniques can be applied to these alternate search strategies.

Danvy and Nielson's work also connects these algebraic models of contexts with one-pass cps transformers; similar tricks show up in [10]. Our work is formulated strictly in terms of interpretation; it would be interesting to see if it could be extended to yield one-pass translation or compilation algorithms.

## 9. CONCLUSIONS

We have presented the first derivation of the two-continuation model of backtracking from the stream model. We do this by showing that they share a common operational semantics.

The derivation depends on the representation of computations and contexts as abstract data types. For appropriate implementations of these data types, we get traditional interpreters, stream semantics, or continuation semantics. We believe that these techniques may have wider applicability.

## Acknowledgements

Our interest in this problem was sparked by discussions with Dan Friedman. Shriram Krishnamurthi made some key suggestions.

# APPENDIX

## A. INITIAL AND FINAL ALGEBRA REPRESENTATIONS OF ABSTRACT DATA TYPES

Consider a data type $T$ with two constructors and a single observer:

$$\begin{array}{lll} \textbf{Constructors}: & f: & A \to T \\ & g: & B \times T \to T \\ \textbf{Observer}: & o: & T \times C \to D \end{array}$$

Assume that $f$, $g$, and $o$ are related by a fold

$$\begin{aligned} o(f(a),c) &= j(a,c) \\ o(g(b,t),c) &= h(b,o(t,c)) \end{aligned}$$

for some functions $j$ and $h$.

In the initial algebra representation $T_\alpha$, we represent $T$ by trees, defined by

$$T = A + (B \times T)$$

$f$ and $g$ construct trees, and $o$ observes them, using the fold to recur on the tree structure:

$$\begin{aligned} f(a) &= a \\ g(b,t) &= (b,t) \\ \\ o(a,c) &= j(a,c) \\ o((b,t),c) &= h(b,o(t,c)) \end{aligned}$$

This is said to be *initial* because for any representation $M$ of this ADT, there is a unique homomorphism from $T_\alpha$ to $M$.

In the final algebra representation $T_\omega$, we represent $T$ by its $o$-action on $C$:

$$T = C \to D$$

Then we can define:

$$\begin{aligned} o(t,c) &= t(c) \\ \\ f(a) &= \lambda c.j(a,c) \\ \\ g(b,t) &= \lambda c.o(g(b,t),c) && \text{representation of } g(b,t) \\ &= \lambda c.h(b,o(c,t)) && \text{equation 2} \\ &= \lambda c.h(b,t(c)) && \text{representation of } t \end{aligned}$$

This is sometimes called the "Church encoding" [3] or "procedural representation" [8]. It is called a final algebra because for any representation $M$ of this algebra, there is a unique homomorphism from $M$ to $T_\omega$.

This covers the case of a single observer. If there are multiple observers, the final algebra representation is obtained by representing $T$ by a tuple of actions

$$(C_1 \to D_1) \times (C_2 \to D_2) \ldots$$

one for each observer.

## B. ADDING STREAMS TO PCF$_V$

In this section we sketch a proof of adequacy for PCF$_V$ with streams as described above. We follow the presentation of Winskel [22]. This approach begins by defining, for each type $\tau$, a relation

$$(\leq_o^\tau) \subseteq (D^\tau \times V^\tau)$$

between the semantic values of type $\tau$ and closed syntactic values of type $\tau$, with the following properties:

- if $d \leq_o^\tau V$, then $d \sqsubseteq [\![V]\!]$
- if $\{d_i\}$ and $\{V_i\}$ are $\omega$-chains such that $d_i \leq_o^\tau V_i$ for each $i$, then $\bigsqcup d_i \leq_o^\tau \bigsqcup [\![V_i]\!]$.
- $d \leq_o^{(\sigma\to\tau)} V$ iff for all $d'$ and $W$ such that $d' \leq^\sigma W$, if $d(d') \neq \perp$ then $VW$ reduces to a value $Z$ such that $d(d') \leq_o^\tau Z$.

Given such a relation $\leq_o^\tau$, we extend it to a relation $(\leq^\tau) \subseteq (D_\perp^\tau \times \text{Terms}_0^t)$ such that if $d \in D_\perp^\tau$ and $M$ is a closed term of type $\tau$, then

$$d \leq^\tau M \iff (d \neq \perp \Rightarrow (\exists V)(M \to^* V \wedge d \leq_o^\tau V))$$

Thus $d \leq^\tau M$ means that if $d \neq \perp$, then $M$ converges to a value at least as large as $d$.

Given these relations, we then wish to prove

THEOREM 1 (ADEQUACY LEMMA). *If*

- *$M$ is a term with free variables $\{x_1, \ldots, x_k\}$, and*
- *$\Gamma$ is a type environment such that $\Gamma \vdash M : \tau$, and*
- *$\rho$ is an environment for $\Gamma$, and*
- *$N_1, \ldots, N_k$ are closed terms of types $\Gamma(x_1), \ldots \Gamma(x_k)$ respectively, and*
- *for each $i$, $\rho(x_i) \leq^{\Gamma(x_i)} N_i$,*

*then*

$$[\![M]\!]\rho \leq^\tau M[N_i/x_i \mid i = 1, \ldots, k]$$

Specializing to closed terms, we deduce:

THEOREM 2 (ADEQUACY). *If $M$ is a closed term of type $\tau$, then $[\![M]\!] \leq^\tau M$. In particular, if $[\![M]\!] \neq \perp$, then $M$ converges.*

Note that since we are in a call-by-value universe, we have adequacy at every type, not just at ground types. For ordinary PCF$_V$ with product and sum types, this theorem was proved in [22].

To extend this theorem to stream types, we need to define $\leq_o^{Str(\tau)}$ and then show that theorem holds for the new constants `nil`, `cons`, and `append`.

We define $\leq_o^{Str(\tau)}$ coinductively:

DEFINITION 1. $\leq_o^{Str(\tau)}$ *is the largest relation such that:*

- $d \leq_o^{Str(\tau)}$ `nil` *iff* $d = \langle\rangle$
- $d \leq_o^{Str(\tau)}$ (`cons` $V$ $M$) *iff* $d = v.w$ *(where* $w \in D_\perp^{Str(\tau)}$*) and if* $w \neq \perp$ *then* $M$ *reduces to a value* $W$ *such that* $w \leq_o^{Str(\tau)} W$.

This is a coinductive definition, depending only on the definition of $\leq_o^\tau$, so it is an acceptable clause to add to the definition of the family $\{\leq_o^\sigma\}$. To fill in the rest of the proof, we must show:

LEMMA 2. *1. if* $w \leq_o^{Str(\tau)} V$*, then* $w \sqsubseteq [\![V]\!]$.

*2.* $\leq_o^{Str(\tau)}$ *is closed under* ω*-chains.*

*3.* $[\![\mathtt{nil}]\!] \leq_o^{Str(\tau)}$ `nil`

*4.* $[\![\mathtt{cons}]\!] \leq_o^{\tau \to Str(\tau) \to Str(\tau)}$ `cons`

*5.* $[\![\mathtt{append}]\!] \leq_o^{Str(\tau) \to Str(\tau) \to Str(\tau)}$ `append`

PROOF. (1) is trivial; (2) proceeds by straightforward induction, observing that the only ω-chains are those created either by an ω-chain of values at some position in the stream, or by an infinite sequence of incomplete streams whose limit is an infinite stream. (3) is immediate.

For (4), we must show that if $v \leq^\tau M$ and $w \leq^{Str(\tau)} N$, then $v.w \leq^{Str(\tau)}$ `cons` $M$ $N$. To show the latter, assume that $v.w \neq \perp$. Then $v \neq \perp$, so $M \to^* V$ such that $v \leq_o^\tau V$. Hence, under the operational semantics of $\mathrm{PCF}_{VS}$, `cons` $M$ $N \to^*$ `cons` $V$ $N$, and $v.w \leq^{Str(\tau)}$ `cons` $V$ $N$, as desired.

For (5), the argument is similar. If the first argument to `append` denotes a stream that is either incomplete or finite, we proceed by induction on its length. If the first argument denotes an infinite stream, it is a limit of a sequence of incomplete streams, and the result follows by closure under ω-sequences. ∎

This completes the extension of the adequacy theorem to streams.

## C. EXTENDING $\mathrm{PCF}_V$ TO HIGHER-ORDER TERM ALGEBRAS

Extending $\mathrm{PCF}_V$ to term representations of computations is somewhat more challenging. For simplicity, we consider only unit and bind:

```
Comp(σ) = unit_σ of σ
        | (all τ) . bind_(τ,σ) of
                       Comp(τ) * (τ -> Comp(σ))
```

There are two difficulties in embedding these types in $\mathrm{PCF}_V$. First, the quantifier in the second clause implies an impredicative model. We circumvent this by restricting σ (and hence τ) to types that do not themselves contain *Comp*.

The second is that these term algebras should be coinductive rather than inductive. Consider the term

```
fix (λ f. λ n. disj (unit n) (f (+1 n)) 0
```

Applied to an initial state $s$, this should denote the infinite stream $\langle(0,s),(1,s),\ldots\rangle$. If computation terms were finite, they could produce only finite answers. Hence they must be infinite.

Our plan is to treat computation terms as stream-like objects, except that each term of the form $\mathtt{bind}_{\sigma\tau}$ will have a successor for its body (`c` in `bind c f`), and a successor for each value of $D^\tau$ (corresponding to the computation `(f a)` for each value of `a`).

More precisely, we define computation terms as infinite trees over an alphabet of nodes labelled $\mathtt{unit}_\sigma$ and $\mathtt{bind}_{\sigma\tau}$. Each node that is labelled $\mathtt{unit}_\sigma$ also contains a value that is an element of $D^\sigma$, and has no daughters. Each node that is labelled $\mathtt{bind}_{\sigma\tau}$ has a distinguished daughter (which we call its body), and a daughter for every element of $D^\tau$. In addition, there is a distinguished node called $\perp$.

We define sets of trees $Comp(\sigma)$ for each σ. We do this simultaneously for all σ by coinduction. That is, the family $\{Comp(\sigma)\}$ is the largest family such that:

1. every node labelled $\mathtt{unit}_\sigma$ is in $Comp(\sigma)$.
2. if a tree whose root is labelled $\mathtt{bind}_{\sigma\tau}$ is in $Comp(\sigma')$ , then $\sigma' = \sigma$, its body is in $Comp(\tau)$, and each of its other daughters is either in $Comp(\sigma)$ or is $\perp$.

This establishes a coinductive assignment of types to trees that is consistent with the signature above.

We order the trees in $Comp(\sigma)_\perp$ by making $\perp$ the smallest element, making $\mathtt{unit}_\sigma(a) \leq \mathtt{unit}_\sigma(a')$ iff $a \leq a'$, and proceeding coinductively.

Last, we define $D^{Comp(\sigma)}$ to be those trees in $Comp(\sigma)$ such that for each `bind` node, the induced function $D^\tau \to Comp(\sigma)_\perp$ is continuous.

With this model, the unit and bind constructors are clearly continuous and strict, so that the usual eager operational semantics

```
E ::= ... | unit E | bind E M | bind V M
```

in which `unit` and `bind` evaluate all of their arguments, is adequate. The argument is like that for streams, except that here each $\mathtt{bind}_{\sigma\tau}$ node has $|D^\tau|+1$ successors. Furthermore, since the non-strict successors are hidden under an abstraction, we do not need any of the special rules like the ones for `cons`.

## D. REFERENCES


[1] H. Abelson and G. J. Sussman. *The Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[2] K. Claessen and P. Ljunglof. Typed logical variables in haskell. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.

[3] O. Danvy and L. R. Nielsen. Defunctionalization at work. Technical Report BRICS RS-01-23, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, July 2001. Extended version of an article to appear in the proceedings of the Third International Conference on Principles and Practice of Declarative Programming (PPDP 2001), Firenze, Italy, September 5-7, 2001.

[4] O. Danvy and L. R. Nielsen. Syntactic theories in practice. Technical Report BRICS RS-01-31, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, July 2001. Extended version of an article to appear in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Firenze, Italy, September 4, 2001.

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

[6] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[7] M. Felleisen, M. Wand, D. P. Friedman, and B. 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.

[8] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, second edition, 2001.

[9] V. Giarratana, F. Gimona, and U. Montanari. Observability concepts in abstract data type specifications. In A. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science 1976*, volume 45 of *Lecture Notes in Computer Science*, pages 576–587. Springer-Verlag, Berlin, Heidelberg, New York, 1976.

[10] R. Hinze. Deriving backtracking monad transformers. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 186–197, N.Y., Sept. 18–21 2000. ACM Press.

[11] S. 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.

[12] T. A. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In C. Consel, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–121, 1992.

[13] G. Plotkin and J. Power. Adequacy for algebraic effects. *Lecture Notes in Computer Science*, 2030, 2001.

[14] J. Spivey and S. Seres. Embedding prolog in haskell, 1999.

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

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

[17] M. Wand. Semantics-directed machine architecture. In *Conf. Rec. 9th ACM Symposium on Principles of Programming Languages*, pages 234–241, 1982.

[18] M. Wand. Loops in combinator-based compilers. *Information and Control*, 57(2–3):148–164, May/June 1983.

[19] M. Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993. preliminary version appeared in *Conf. Rec. 20th ACM Symp. on Principles of Prog. Lang.* (1993), 137–143.

[20] M. Wand. The theory of Fexprs is trivial. *Lisp and Symbolic Computation*, 10:189–199, 1998.

[21] D. H. D. Warren. An Abstract PROLOG Instruction Set. Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Menlo Park, CA, Oct. 1983.

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