# Techniques for Embedding Postfix Languages in Haskell

Chris Okasaki
United States Military Academy*
West Point, New York
Christopher.Okasaki@usma.edu

## Abstract

One popular use for Haskell in recent years has been as a host language for domain-specific embedded languages. But how can one embed a postfix language in Haskell, given that Haskell only supports prefix and infix syntax? This paper describes several such embeddings for increasingly complex postfix languages.

## Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

## General Terms

Languages

## Keywords

Postfix notation, domain-specific embedded languages, Haskell

## 1 Introduction

Here is a programming puzzle. Find Haskell definitions for `begin`, `push`, `add`, and `end` such that the RPN-like expressions

```
begin push 5 push 6 add end
```

and

```
begin push 5 push 6 push 7 add add end
```

evaluate to 11 and 18 respectively.

Besides being a fun exercise, this puzzle also demonstrates that Haskell is capable of simulating postfix syntax using only prefix function application. But how far can we push this? Can we use these techniques to embed full-blown postfix languages like Forth or Postscript in Haskell? This paper shows that we can.

This result further strengthens the position of Haskell as the host language of choice for domain-specific embedded languages (DSELs). In recent years, hordes of DSELs have emerged that successfully employ Haskell as a host language, in such diverse areas as animation [4], robotics [11], music composition [6], circuit design [2], and parsing [7]. However, most of the current DSELs for Haskell have been designed from scratch, and the designers have carefully tailored the syntax of each to be compatible with Haskell. By showing how to simulate postfix syntax in Haskell, we open the door to embedding existing postfix DSLs without butchering their "look and feel".

Section 2 describes a solution to the puzzle, introducing the basic techniques for supporting postfix syntax. Sections 3 and 4 introduce a representation of heterogeneous stacks as nested pairs and adapt the puzzle solution to use these stacks. Section 5 shows how to extend the basic postfix language to support procedural abstraction, control constructs, and imperative features. Section 6 describes how to define recursive postfix procedures. Section 7 concludes.

Our techniques are presented in the context of Haskell, but many other functional languages would work as well. Polymorphism and higher-order functions play crucial roles in our techniques, but lazy evaluation and type classes do not.

## 2 A Solution to the Puzzle

Figure 1, on the next page, shows a basic solution to the puzzle, based on the flattening combinators of Okasaki [10].[1] It threads a stack of type `[Int]` through the computation, beginning with the empty stack. Each intermediate command takes the current stack, performs some action on it, and passes the resulting stack to the next command. The key is that each command takes the next command as an argument.

The types of the four commands are

```
begin ::                   ([Int] -> a) -> a
push  :: [Int] -> Int -> ([Int] -> a) -> a
add   :: [Int] ->         ([Int] -> a) -> a
end   :: [Int] -> Int
```

Notice that the type of every command except `begin` starts with

```
[Int] -> ...
```

and the type of every command except `end` finishes with

* The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Military Academy, the Department of the Army, the Department of Defense, or the U.S. Government.

[1] Mayer Goldberg invented a similar set of combinators, but never published them.


*Haskell'02*, October 3, 2002, Pittsburgh, Pennsylvania, USA.
1-58113-605-6/02/0010

```
begin k = k []
push s x k = k (x:s)
add (x:y:s) k = k (y+x : s)
end [x] = x
```
**Figure 1. A solution to the puzzle.**

```
... -> ([Int] -> a) -> a
```

In other words, every command except `begin` takes the current stack as its first argument, and every command except `end` takes the next command as its last argument. The last action of every non-terminal command is to pass the current stack to the next command.

The type `([Int] -> a) -> a` and the treatment of the "next command" screams *continuations* to anyone familiar with the concept. However, continuations typically represent the rest of the computation, and these continuations represent only the very next command. Thus, our continuations are actually *partial continuations* [8]. To see how these partial continuations work, it is helpful to step through the evaluation of a sample expression.

```
   begin push 5 push 6 add end
⇒  push [] 5 push 6 add end
⇒  push [5] 6 add end
⇒  add [6,5] end
⇒  end [11]
⇒  11
```

Notice that polymorphism plays a crucial role here. For example, in the expression

```
begin push 5 push 6 add end
```

the second `push` is used at type

```
[Int] -> Int
      -> ([Int] -> ([Int] -> Int) -> Int)
      -> ([Int] -> Int) -> Int
```

and the first `push` is used at type

```
[Int] -> Int
      -> ([Int] -> Int
                -> ([Int] -> ([Int] -> Int) -> Int)
                -> ([Int] -> Int) -> Int)
      -> Int
      -> ([Int] -> ([Int] -> Int) -> Int)
      -> ([Int] -> Int) -> Int
```

This combination of polymorphism and continuations to produce functions that take greater and greater numbers of arguments is reminiscent of the functional unparsers of Danvy [3].

# 3 Heterogeneous Stacks

Integer lists are a poor representation for the internal stacks of a postfix language. They suffer from two major deficiencies. First, we usually want an embedded language to be able to manipulate more than just integers. Indeed, we would prefer to have access to the full gamut of Haskell types. Second, we would like the type checker to catch errors related to the size of the stack, such as calling `add` with only a single integer on the stack or ending with two or more values on the stack.

```
module Stack where

data Empty = Empty -- the empty stack

push  :: a -> s -> (s,a)
pop   :: (s,a) -> s
dup   :: (s,a) -> ((s,a),a)
exch  :: ((s,a),b) -> ((s,b),a)    -- swap

add   :: ((s,Int),Int) -> (s,Int)
sub   :: ((s,Int),Int) -> (s,Int)  -- subtract
mul   :: ((s,Int),Int) -> (s,Int)  -- multiply
eq    :: ((s,Int),Int) -> (s,Bool) -- equals
lt    :: ((s,Int),Int) -> (s,Bool) -- less than

nil   :: s -> (s,[a])
cons  :: ((s,a),[a]) -> (s,[a])

only  :: (Empty,a) -> a

smap  :: (a -> b) -> (s,a) -> (s,b)
smap2 :: (a -> b -> c) -> ((s,a),b) -> (s,c)
```
**Figure 2. Signature for a module of heterogeneous stacks. The implementations are entirely straightforward given the types.**

Taking both of these issues into account, we choose to implement stacks as nested pairs. For example,

```
(((Empty,1),True),"hello")
```

is a stack of size 3, with type

```
(((Empty,Int),Bool),String)
```

The `Empty` value and type constructors are defined in Figure 2, along with a number of utility functions on stacks that will be used throughout the rest of the paper. Note that these stacks grow to the right, which is backward from the usual treatment in functional languages but is consistent with stack diagrams in postfix languages such as Forth and Postscript.

The types of stacks now encode their exact size and layout. For example, the type of the `only` function from Figure 2

```
only :: (Empty,a) -> a
```

now guarantees that it will only be called on stacks containing exactly one element. Most operations, however, do not need to constrain the entire stack, merely the top few elements. Such constraints are elegantly captured as polymorphic types. For example, the type of `add`

```
add :: ((s,Int),Int) -> (s,Int)
```

guarantees that the top two elements of the input stack and the top element of the output stack are all integers.

In a real application, we would probably give `add` the more general type

```
add :: Num n => ((s,n),n) -> (s,n)
```

but type classes have no direct bearing on the use of postfix syntax, so we will eschew these more complex types.

The primary disadvantage of using nested pairs instead of lists is that it becomes more difficult to handle situations in which varying numbers of items might be on the stack. However, such situations can usually be resolved by allowing one or more elements of the stack to themselves be lists. It is just as easy to manipulate lists on the stack as it is to manipulate integers on the stack. For example, Figure 2 contains the list operations `nil` and `cons`:

```
nil  :: s -> (s,[a])
cons :: ((s,a),[a]) -> (s,[a])
```

# 4 The Postfix Transformation

Given the stack operations in Figure 2, we can express the addition $5+6$ as

```
only (add (push 6 (push 5 Empty)))
```

Obviously, this expression is not written in postfix notation, but we can simulate postfix notation by defining the symbol `#` as left-associative reverse function application (i.e., `x # f = f x`). Then we can rewrite the above expression as

```
Empty # push 5 # push 6 # add # only
```

To abstract away from the details of the stack representation, we define two helper commands `begin` and `end`

```
begin = Empty
end = only
```

so that we can rewrite the addition as

```
begin # push 5 # push 6 # add # end
```

If we want to use postfix notation directly, without the infix `#` symbols, we can adapt the combinators from Section 2 to use the nested-pair representation of stacks. However, rather than doing so individually, we define a few general functions to convert ordinary stack operations into postfix stack operations, which we will call *commands*. The type of a typical postfix command is

```
type Cmd s s' = forall a. s -> (s' -> a) -> a
```

where the command takes a stack of type `s` and produces a stack of type `s'`. The `forall` indicates that every command is polymorphic in the result type of the *next* command. The `forall` construct is not officially part of Haskell, but it is supported by GHC. We could get by without it, but many of the types would be messier.

To convert an ordinary stack operation into postfix form, we use the function `post`:

```
post :: (s -> s') -> Cmd s s'
post f s = next (f s)
```

where

```
next :: s -> (s -> a) -> a
next s k = k s
```

Now we can define postfix stack operations such as

```
add = post Stack.add
dup = post Stack.dup
...
```

Some operations, such as `push`, take an argument directly from the instruction stream rather than from the stack. We implement such operations in a similar fashion:

```
type Cmd1 x s s' =
  forall a. s -> x -> (s' -> a) -> a

post1 :: (x -> s -> s') -> Cmd1 x s s'
post1 f s x = next (f x s)

push = post1 Stack.push
apply = post1 Stack.smap
...
```

The `begin` and `end` operations are simply

```
begin :: (Empty -> a) -> a
begin = next Empty

end :: (Empty,a) -> a
end = Stack.only
```

# 5 Extending the Postfix Language

If you want to use postfix notation for more than simple arithmetic expressions, you quickly discover that you need more language features. We gradually extend our basic postfix language with procedural abstraction, control constructs, and imperative features.

## 5.1 Procedural Abstraction

We would like to be able to define new commands from old ones using postfix syntax, as in

```
incr = begindef push 1 add enddef
```

where `begindef` and `enddef` are new commands. We could then write programs like

```
begin push 5 incr incr end
```

But what stack should `begindef` pass to `push` in the definition of `incr`? We don't yet have our hands on an appropriate stack—in fact, there isn't one such stack since `incr` is called twice. Instead, we need `incr` to somehow take the stack on which it will operate. To make this work, we change all postfix commands to pass around functions from stacks to stacks instead of plain stacks. This function is the composition of all the operations from the most recent `begin` or `begindef` to the current point. When we hit the `end` command, the function is the composition of all the operations in the entire postfix program, and we run the program by applying the function to `Empty`.

We change the types of postfix commands to

```
type Cmd s s' = forall s0 a.
   (s0 -> s) -> ((s0 -> s') -> a) -> a
```

The `post` function then composes the new function with the old one rather than applying the function to a stack.

```
post :: (s -> s') -> Cmd s s'
post f ss = next (f . ss)
```

The definitions of the basic postfix commands remain

```
add = post Stack.add
dup = post Stack.dup
...
```

`Cmd1` and `post1` are redefined similarly, and are again used to implement `push`.

```
type Cmd1 x s s' = forall s0 a.
  (s0 -> s) -> x -> ((s0 -> s') -> a) -> a

post1 :: (x -> s -> s') -> Cmd1 x s s'
post1 f ss x = next (f x . ss)

push = post1 Stack.push
```

The `begin` and `end` commands become

```
begin :: ((Empty -> Empty) -> a) -> a
begin = next id

end :: (Empty -> (Empty,a)) -> a
end ss = Stack.only (ss Empty)
```

Revisiting the example reduction sequence from Section 2, we now get

```
   begin push 5 push 6 add end
⇒  push id 5 push 6 add end
⇒  push (S.push 5 . id) 6 add end
⇒  add (S.push 6 . S.push 5 . id) end
⇒  end (S.add . S.push 6 . S.push 5 . id)
⇒  S.only ((S.add . S.push 6 . S.push 5 . id) Empty)
⇒  ...
⇒  S.only (Empty,11)
⇒  11
```

where the `Stack` module is abbreviated as `S`.

In this new setup, the `begindef` and `enddef` commands turn out to be surprisingly simple.

```
begindef :: ((s -> s) -> a) -> a
begindef = next id

enddef :: (s -> s') -> Cmd s s'
enddef = post
```

The `begindef` command is just like `begin`, but with a more general type, and the `enddef` command turns out to be identical to `post`! Now the definition

```
incr = begindef push 1 add enddef
```

has the desired type `Cmd (s,Int) (s,Int)`.

## 5.2 Basic Control Constructs

The next thing we want to add to the embedded language is control structures. We will describe conditionals in detail, but various flavors of loops can be implemented in the same fashion. Forth and Postscript take significantly different approaches to control structures. We will follow the style of Forth, because it leads to more interesting implementations.

In Forth, a conditional is written

```
...condition... IF
...then-part... ELSE
...else-part... THEN
```

The reason the `ELSE` and `THEN` keywords are reversed is that the `ELSE` is optional, so `THEN` marks the end of the `IF`. In Haskell, we will avoid this peculiarity by making both branches mandatory and write

```
...condition... if_
...then-part... then_
...else-part... else_
```

For example, the following code would implement the absolute-value function:

```
absval =
  begindef
    dup push 0 lt if_       -- is the # < 0?
    push 0 exch sub then_   -- negate the #
    else_                   -- do nothing
  enddef
```

Notice that, although the `else_` command is mandatory, the code part between the `then_` and the `else_` may be empty.

To implement control structures in this style, we extend all our commands to pass around a control stack along with the stack-to-stack function. Most commands ignore the control stack. We redefine the `Cmd` type as

```
type Cmd s s' = forall s0 c a.
  (s0 -> s,c) -> ((s0 -> s',c) -> a) -> a
```

Then `post` becomes

```
post :: (s -> s') -> Cmd s s'
post f (ss,c) = next (f . ss,c)
```

while the standard stack commands remain

```
add = post Stack.add
dup = post Stack.dup
...
```

`Cmd1`, `post1`, and `push` are redefined similarly

```
type Cmd1 x s s' = forall s0 c a.
  (s0 -> s,c) -> x -> ((s0 -> s',c) -> a) -> a

post1 :: (x -> s -> s') -> Cmd1 x s s'
post1 f (ss,c) x = next (f x . ss,c)

push = post1 Stack.push
```

Control operators like `if_`, `then_`, and `else_` pass control information around on the control stack. In particular, `if_` and `then_` store stack-to-stack functions on the control stack, and `else_` takes these functions from the control stack and composes them appropriately. We represent control stacks using the following types:

```
data BEGIN = BEGIN  -- the empty control stack
data IF s0 s c = IF (s0 -> (s,Bool)) c
data IFTHEN s0 s s' c =
       IFTHEN (s0 -> (s,Bool)) (s -> s') c
```

Then the control operators are defined as

```
if_ :: (s0 -> (s,Bool),c) ->
         ((s -> s,IF s0 s c) -> a) -> a
if_ (ss,c) = next (id, IF ss c)
```

```
then_ :: (s -> s',IF s0 s c) ->
           ((s -> s,IFTHEN s0 s s' c) -> a) -> a
then_ (ssThen, IF ssIf c) =
    next (id,IFTHEN ssIf ssThen c)

else_ :: (s -> s',IFTHEN s0 s s' c) ->
           ((s0 -> s',c) -> a) -> a
else_ (ssElse, IFTHEN ssIf ssThen c) =
    next (ssIfThenElse,c)
  where ssIfThenElse s0 = ssTaken s
          where (s,cond) = ssIf s0
                ssTaken = if cond then ssThen
                                  else ssElse
```

The remaining begin/end commands all assume that the control stack is empty.

```
begin :: ((Empty -> Empty,BEGIN) -> a) -> a
begin = next (id,BEGIN)

end :: (Empty -> (Empty,a),BEGIN) -> a
end (ss,BEGIN) = Stack.only (ss Empty)

begindef :: ((s -> s,BEGIN) -> a) -> a
begindef = next (id,BEGIN)

enddef :: (s -> s',BEGIN) -> Cmd s s'
enddef (ss,BEGIN) = post ss
```

One interesting feature of this design is that syntactic constraints on conditionals are enforced by the typechecker rather than the parser. For example, the following malformed expression

```
begin push True if_ push 5 then_ push 6 then_ end
```

would generate a type error, not a syntax error.

## 5.3 Separable Control Constructs

The types of begindef and enddef guarantee that user-defined procedures have no effect on the control stack. In other words, the multiple parts of a control construct must always appear together. However, there are times when it is useful to spread the parts of a control construct across several user-defined procedures. Doing so allows the user to create customized control constructs.

For example, suppose we want a conditional command fi_ that terminates an if_ right after the then-part, doing nothing if the condition is false. We could then rewrite absval as

```
absval =
  begindef
    dup push 0 lt if_     -- is the # < 0?
    push 0 exch sub fi_   -- negate the #
  enddef
```

Rather than defining fi_ from scratch, we would like the user to be able to write

```
fi_ = begindef then_ else_ enddef
```

In implementing this, we run into the same problem we encountered in Section 5.1. Back then, we didn't have our hands on the right data stack at the time of the begindef. Now, we don't have our hands on the right control stack. The solution is again to pass around functions from stacks to stacks instead of just stacks, but now with control stacks instead of data stacks. However, our new functions must also pass around the data stack-to-stack functions developed earlier. Altogether, the type being passed around between commands now has the form

```
(s0 -> s1,c0) -> (s2 -> s3,c1)
```

The details are shown in Figure 3, on the next page. Note that the definitions of the control stack types (BEGIN, IF, IFTHEN) do not change.

```
module IntState where

newtype M a = M (Int -> (a,Int))

instance Monad M where
  return x = M $ \n -> (x,n)
  M f >>= k = M $ \n -> let (x,n') = f n
                             M g = k x
                         in g n'

instance Functor M where
  fmap f m = m >>= \x -> return (f x)

run    :: M a -> a
mread  :: M Int
mwrite :: Int -> M ()

run (M f) = fst (f 0)
mread = M $ \n -> (n,n)
mwrite n = M $ \_ -> ((),n)
```

**Figure 4. An integer state monad.**

## 5.4 Imperative Features

Many postfix languages support imperative features such as assignment or IO. These features are typically implemented in Haskell as monads, and we will follow this tradition. Any of the previous implementations can be extended with monadic operations simply by replacing every occurrence of a stack with a monadic computation producing a stack. For example, the Cmd type from Section 5.1

```
type Cmd s s' = forall s0 a.
  (s0 -> s) -> ((s0 -> s') -> a) -> a
```

becomes

```
type Cmd s s' = forall s0 a.
  (M s0 -> M s) -> ((M s0 -> M s') -> a) -> a
```

where M is the monad in question.

We illustrate by extending the core language of Section 5.1 with operations that read and write an integer state. The underlying state monad is shown in Figure 4. Figure 5 shows the changes to the major postfix commands.

# 6 Recursion

Now that we can define procedures and conditionals, it is natural to want to use recursion. But doing so turns out to be suprisingly difficult, in much the same way that recursion is difficult to combine with monads [5].

As an example, suppose we wish to write the usual recursive facto-

```haskell
type Cmd s s' = forall ssc s0 c a. (ssc -> (s0 -> s,c)) -> ((ssc -> (s0 -> s',c)) -> a) -> a

post :: (s -> s') -> Cmd s s'
post f sscssc = next (extendSS f sscssc)

extendSS :: (s -> s') -> (ssc -> (s0 -> s,c)) -> (ssc -> (s0 -> s',c))
extendSS f sscssc ssc = (f . ss,c)
  where (ss,c) = sscssc ssc

add = post Stack.add
dup = post Stack.dup
...

type Cmd1 x s s' = forall ssc s0 c a. (ssc -> (s0 -> s,c)) -> x -> ((ssc -> (s0 -> s',c)) -> a) -> a

post1 :: (x -> s -> s') -> Cmd1 x s s'
post1 f sscssc x = next (extendSS (f x) sscssc)

push   = post1 Stack.push
...

begin :: (((Empty -> Empty,BEGIN) -> (Empty -> Empty,BEGIN)) -> a) -> a
begin = next id

end :: ((Empty -> Empty,BEGIN) -> (Empty -> (Empty,answer),BEGIN)) -> answer
end sscssc = Stack.only (ss Empty)
  where (ss,BEGIN) = sscssc (id,BEGIN)

begindef :: ((ssc -> ssc) -> a) -> a
begindef = next id

enddef   :: (ssc' -> ssc'') -> (ssc -> ssc') -> ((ssc -> ssc'') -> a) -> a
enddef sscssc sscssc' = next (sscssc . sscssc')

if_   :: (ssc -> (s -> (s',Bool),c)) -> ((ssc -> (s' -> s',IF s s' c)) -> a) -> a
then_ :: (ssc -> (s' -> s'',IF s s' c)) -> ((ssc -> (s' -> s',IFTHEN s s' s'' c)) -> a) -> a
else_ :: (ssc -> (s' -> s'',IFTHEN s s' s'' c)) -> ((ssc -> (s -> s'',c)) -> a) -> a

if_ sscssc = next sscsscIf
  where sscsscIf ssc = (id,IF ss c)
          where (ss,c) = sscssc ssc

then_ sscssc = next sscsscThen
  where sscsscThen ssc = (id,IFTHEN ssIf ssThen c)
          where (ssThen,IF ssIf c) = sscssc ssc

else_ sscssc = next sscsscElse
  where sscsscElse ssc = (ss,c)
          where (ssElse,IFTHEN ssIf ssThen c) = sscssc ssc
                ss s = ssTaken s'
                  where (s',cond) = ssIf s
                        ssTaken = if cond then ssThen else ssElse
```

**Figure 3. The details of implementing separable control constructs.**

```
type Cmd s s' = forall s0 a. (M s0 -> M s) -> ((M s0 -> M s') -> a) -> a

post :: (s -> s') -> Cmd s s'
post f ss = next (fmap f . ss)

add = post Stack.add
dup = post Stack.dup
...

type Cmd1 x s s' = forall s0 a. (M s0 -> M s) -> x -> ((M s0 -> M s') -> a) -> a

post1 :: (x -> s -> s') -> Cmd1 x s s'
post1 f ss x = next (fmap (f x) . ss)

push = post1 Stack.push
...

begin :: ((M Empty -> M Empty) -> a) -> a
begin = next id

end :: (M Empty -> M (Empty,a)) -> a
end ss = Stack.only (run (ss (return Empty)))

begindef :: ((M s -> M s) -> a) -> a
begindef = next id

enddef :: (M s -> M s') -> Cmd s s'
enddef ss ss' = next (ss . ss')

mread  :: Cmd s (s,Int)
mwrite :: Cmd (s,Int) s

-- The code for mread and mwrite in the printed proceedings was incorrect.
-- These are the corrected versions.
mread ss = next $ (\m -> do {s <- m; n <- IntState.mread; return (s,n)}) . ss
mwrite ss = next $ (\m -> do {(s,n) <- m; IntState.mwrite n; return s}) . ss
```

**Figure 5. Implementing monadic postfix commands.**

rial function. We would like to be able to write

```
fact = begindef
         dup push 0 eq if_
         pop push 1 then_
         dup push 1 sub fact mul else_
       enddef
```

but this fails to typecheck. In particular, it fails the occurs check because it needs polymorphic recursion—the inner `fact` is called with one more integer on the stack than the outer `fact`. We can fix this by adding the type signature

```
fact :: Cmd (s,Int) (s,Int)
```

Now the program typechecks, but when we try to use it, we immediately fall into a blackhole. In particular, the outer `fact` cannot be evaluated without evaluating the inner `fact`. A moment's reflection reveals that every command is strict in the following command, so what we need is a way to delay commands that we wish to call recursively. We add a new `call` command for this purpose, and write

```
fact = begindef
         dup push 0 eq if_
         pop push 1 then_
         dup push 1 sub call fact mul else_
       enddef
```

Like `push`, the `call` command takes its main argument from the instruction stream rather than the stack. Assuming we are using the types from Section 5.2, `call` is defined as

```
call :: Cmd1 (Cmd s s') s s'
call (ss,c) cmd = next (cmd (id,BEGIN) fst . ss,c)
```

Notice that the expression

```
cmd (id,BEGIN) fst
```

converts a command of type `Cmd s s'` to a function of type `s -> s'`. In other words, it is essentially the inverse of the `post` function.

Unfortunately, we now trip over limitations in GHC's treatment of the `forall`s hidden inside `Cmd` and `Cmd1`, which forbid this kind of nesting.[2] To get around these limitations, we expand the type of `call` to

```
call :: Cmd1 ((s -> s,BEGIN) ->
                ((s -> s',BEGIN) -> (s->s')) ->
                (s -> s'))
             s s'
```

We can finally run

```
begin push 5 fact end
```

to get the answer `120`.

# 7 Conclusions

Other researchers have considered the relationship between postfix languages and functional languages. For example, the purely functional language Joy [12] uses postfix syntax and celebrates its ties to Forth. Linear Lisp [1] uses ordinary Lisp syntax but is implemented as a Forth-like stack machine. Koopman and Lee [9] implement combinator graph reduction using a Forth-inspired threaded interpretive engine. However, this paper is the first to seriously consider embedding a postfix language in a language like Haskell, building on our previous work on flattening combinators [10], which allow arbitrary combinator expressions to be written without parentheses.

We have addressed the major theoretical concerns of such an embedding, but several practical concerns remain. First, the types in these kinds of programs are huge, frequently making type-error messages unreadable. Second, the combination of huge types and functions with dozens of arguments make compilation slow. Today's compilers do not expect such large arities and appear in incorporate algorithms that are quadratic (or worse!) in the number of arguments. For example, the factorial function in Section 6 uses `begindef` with 18 arguments and makes GHC noticeably sluggish. The recursive Fibonacci function

```
fib :: Cmd (s,Int) (s,Int)
fib = begindef
        dup push 2 lt if_
        pop push 1 then_
        dup push 1 sub call fib
        exch push 2 sub call fib add else_
      enddef
```

uses `begindef` with 24 arguments, and crashes the compiler after a long wait.

[2]Never bet against the GHC folks! Shortly after this paper was written, they released GHC 5.04, which does support nested `forall`s.

# 8 References


[1] Henry G. Baker. Linear logic and permutation stackts—the Forth shall be first. *Computer Architecture News*, 22(1):34–43, March 1994.

[2] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *ACM SIGPLAN International Conference on Functional Programming*, pages 174–184, September 1998.

[3] Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, November 1998.

[4] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, June 1997.

[5] Levent Erkök and John Launchbury. Recursive monadic bindings. In *ACM SIGPLAN International Conference on Functional Programming*, pages 174–185, September 2000.

[6] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.

[7] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[8] Gregory F. Johnson. GL—a denotational testbed with continuations and partial continuations as first-class objects. In *Symposium on Interpreters and Interpretive Techniques*, pages 165–176, June 1987.

[9] Philip Koopman and Peter Lee. A fresh look at combinator graph reduction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 110–119, June 1989.

[10] Chris Okasaki. Flattening combinators: Surviving without

parentheses. *Journal of Functional Programming*, 2002. To appear.

[11] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages*, pages 91–105, January 1999.

[12] Manfred von Thun. Joy: Forth's functional cousin. In *Euro-Forth*, 2001.