Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 193 (2007) 29–45
www.elsevier.com/locate/entcs

Plugging a Space Leak with an Arrow

Hai Liu and Paul Hudak[1]

Department of Computer Science
Yale University
New Haven, CT 06511, U.S.A.

Abstract

The implementation of conceptually continuous *signals* in functional reactive programming (FRP) is studied in detail. We show that recursive signals in standard implementations using streams and continuations lead to potentially serious time and space leaks under conventional call-by-need evaluation. However, by moving to the level of *signal functions*, and structuring the design around *arrows*, this class of time and space leaks can be avoided. We further show that the use of *optimal reduction* can also avoid the problem, at the expense of a much more complex evaluator.

Keywords: programming languages, functional programming, arrows, Haskell

1 Introduction

Functional Reactive Programming, or FRP, is an approach to programming hybrid systems in a declarative style, using two particular abstractions: a continuous (functional) modeling of time-varying behaviors, and a discrete (reactive) calculus of user and process interaction. FRP has been used in a variety of applications, including computer animation [10], mobile robotics [24,25], humanoid robotics [13], real-time systems [31], parallel processing [12], and graphical user interfaces [7].

In this paper we focus on the continuous nature of FRP, and ignore its reactive component. Since the continuous nature of FRP is only an ideal, it must be approximated in a real implementation. The original implementations of FRP used *time-ordered streams* of values for this approximation [9,10].

[1] Email: hai.liu@yale.edu and paul.hudak@yale.edu

doi:10.1016/j.entcs.2007.10.006

Later implementations used a simple kind of *continuation*, and furthermore were structured using *arrows* [16,22,15].

Although FRP has been used successfully in a number of applications, most of the implementations have suffered from varying degrees of *space leaks*. Interestingly, a noticeable improvement (i.e. reduction) in the degree of space leaks has been observed in the most recent incarnation of FRP that we call *Yampa* [8]. Yampa's implementation uses continuations and arrows, yet space leaks were not the original motivation for this design decision. The reasons for the improvement in space utilization are quite subtle, and up until now have been mostly anecdotal. Indeed, the primary purpose of this paper is to describe a particular class of space leaks in FRP, show why they occur, and explain precisely how it is that they are avoided in Yampa.

In the remainder of this paper we first describe two standard non-arrow-based implementations of FRP, and show that they are both susceptible to serious space leaks. We then describe arrows in Section 3, and use them to design a new implementation of FRP in Section 4 that is similar to that of Yampa. We show in Section 5 that this new implementation does not suffer from the same space leak problem as the standard implementation. In Section 6 we discuss some alternative approaches to solving the space leak problem. We assume familiarity with Haskell [26] and basic functional programming concepts [14].

2 Two Standard Implementations

Conceptually, continuous values in FRP, which are called *signals*, are time-varying values that can be thought of as functions of time:

$$\texttt{Signal}\ \alpha\ \approx\ \texttt{Time} \rightarrow \alpha$$

The power of FRP lies in the fact that programming is done at the level of signals. For example, two signals `s1` and `s2` may be added together, as in `s1+s2`, which is conceptually the point-wise sum of the functions representing `s1` and `s2`. (And Haskell's flexible overloading mechanism allows us to write this style of expression for all of the arithmetic operators.)

More importantly, stateful computations such as integration and differentiation may be performed on signals. For example, the integral of signal `s1` is simply `integral s1`. In this way it is easy to write integral or differential equations that are commonly used to describe dynamic systems – these equations are then directly executable.

Despite the appealing nature of continuous signals, and their elegant representation as functions of time, in practice we are interested in computing a

```
newtype S a = S ([DTime] → [a])
type DTime = Double

integralS :: Double → S Double → S Double
integralS i (S f) = S (λdts → scanl (+) i (zipWith (*) dts (f dts)))

runS :: S Double → [Double]
runS (S f) = f (repeat dt)
```

Fig. 1. Stream-Based FRP

```
newtype C a = C (a, DTime → C a)

integralC :: Double → C Double → C Double
integralC i (C p) = C (i, λdt→integralC (i + fst p * dt) (snd p dt))

runC :: C Double → [Double]
runC (C p) = fst p : runC (snd p dt)
```

Fig. 2. Continuation-Based FRP

continuous stream of these values on a digital computer, and thus the functional implementation implied by the above representation is impractical. In what follows we describe two of the simplest implementations that we have used, and that are adequate in demonstrating the space leak properties that we are interested in.

2.1 Stream-Based FRP

In the book *The Haskell School of Expression* [14] continuous signals are called *behaviors* and are defined as a function from a list of discrete time samples to a list of values. A simplified version of this is given in Figure 1 where, instead of time samples, we use time *intervals* (which we call "delta times" and are represented by the type `DTime`). Also included is a definition of an integral function, which will play a key role in our example of a space leak. `integralS` takes an initial value and returns a signal that is the numerical integration of the input signal using Euler's rule.

2.2 Continuation-Based FRP

An alternative approach to implementing FRP is to view a signal as a pair, consisting of its current value and a simple continuation that depends only on the time interval that gives rise to its future values. The full definition is given in Figure 2.

Note that both `runS` and `runC` assume a fixed delta time `dt`. We do this for convenience here, but in practice the delta time varies during the course of program execution, and depends on processor speed, computational load, interrupts, and so on.

2.3 A Space Leak

Despite the heralded advantages of functional languages, perhaps their biggest drawback is their sometimes poor and often unpredictable consumption of space, especially for non-strict (lazy) languages such as Haskell. A number of optimization techniques have been proposed, including tail-call optimization, CPS transformation, garbage collection, strictness analysis, deforestation, and so on [6,1,29,30,28,21]. Many of these techniques are now standard fare in modern day compilers such as the Glasgow Haskell Compiler (GHC). Not all optimization techniques are effective at all times, however, and in certain cases may result in worse behavior rather than better [11]. There has also been work on *relative leakiness* [5,11,4], where the space behavior of different optimization techniques or abstract machines are studied and compared.

In fact, both of the above FRP implementations, with their innocent-looking definitions, can lead to space leaks. In particular, suppose we define a recursive signal such as this definition of the exponential value `e`, which directly reflects its mathematical formulation:

```
e = integralC 1 e
```

Our intuition tells us that unfolding `e` should be linear in time and constant in space. Yet in reality, the time complexity of computing the n^{th} value of `e` is $O(n^2)$ and the space complexity is $O(n)$. Thus evaluating successive values of `e` will soon blow up in any standard Haskell compiler, eating up memory and taking successively longer and longer to compute each value. (The same problem arises if we use `integralS` instead of `integralC`.)

To see where the leak occurs, let's perform a step-by-step unfolding of the computation using call-by-need evaluation. Lack of formality aside, we adopt a familiar style of using `let`-expressions to denote sharing of terms [18,2,20]. The unfolding of `runC e`, where `e = integralC 1 e`, is shown in Figure 3.

The problem here is that the standard call-by-need evaluation rules are unable to recognize that the function:

```
f = λdt → integralC (1 + dt) (f dt)
```

is the same as:

```
f = λdt → let x = integralC (1 + dt) x
          in x
```

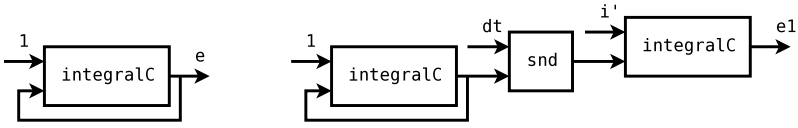
The former definition causes work to be repeated in the recursive call to `f`, whereas in the latter case the computation is shared. This leads to $O(n)$ space and $O(n^2)$ time to compute a stream of n values, rather than $O(1)$ space and $O(n)$ time that we would like.

```
e = integralC 1 e
↪ let p = (1, λdt → integralC (1 + fst p * dt) (snd p dt))
   in C p
↪ let p = (1, f)
       f = λdt → integralC (1 + 1 * dt) (f dt)
   in C p
↪ let f = λdt → integralC (1 + 1 * dt) (f dt)
   in C (1, f)

runC e
↪ runC (C (1, f))
↪ 1 : runC (f dt)
↪ 1 : runC (integralC (1 + 1 * dt) (f dt))
↪ 1 : runC (let i' = 1 + 1 * dt
                f = λdt → integralC i' (f dt)
                g = λdt → integralC (i' + i' * dt) (snd (f dt) dt)
            in C (i', g))
↪ λdots
```

Fig. 3. Unfolding `runC e`

Figure 4 shows graphically the signal `e` and `e1 = snd e dt`, where `e1` clearly has grown in size. Further unfolding of `runC e` will result in more growth and repeated sub-structures. Ideally what we want is the equivalent of `e1 = integralC i' e1`, but call-by-need evaluation won't give us that result.



(a) `e` (b) `e1`

Fig. 4. Diagram of `e` and `e1`

2.4 An Analogy

To better understand the problem, it might help to describe a simpler but analogous example. Suppose we wish to define a function that repeats its argument indefinitely:

```
repeat x = x : repeat x
```

or, in lambdas:

```
repeat = λx → x : repeat x
```

This requires $O(n)$ space. But we can achieve $O(1)$ space by writing instead:

```
repeat = λx → let xs = x : xs
              in xs
```

The time and space complexity of this example, however, is still a factor of n away from that exhibited by e above. To mimic the $O(n^2)$ time behavior, suppose that instead of repeating a number, we wish to increment it indefinitely. One way to do this is as follows:

```
successors n = n : map (+1) (successors n)
```

Unfortunately, this takes $O(n^2)$ steps to compute the nth value. To fix it, we can do the following instead:

```
successors n = let ns = n : map (+1) ns
               in ns
```

It is worth noting that if the delta times were fixed to a constant `dt`, we could redesign the implementation as follows:

```
newtype C a = C (a, C a)

integralC :: Double → C Double → C Double
integralC i (C p) = C (i, integralC (i + fst p * dt) (snd p))
```

Now note that `C` is isomorphic to Haskell's list data type, and `e` will run in linear time and constant space. (A similar simplification can be done for the stream implementation.)

But in fact, as mentioned earlier, in our real implementations of FRP we cannot assume a fixed delta time (even though in our simple implementations of `runS` and `runC` it is fixed), and thus we cannot eliminate the function types.

Is there a better solution?

3 A Brief Introduction to Arrows

Arrows [16,15] are a generalization of monads that relax the stringent linearity imposed by monads, while retaining a disciplined style of composition. This discipline is enforced by requiring that composition be done in a "point-free" style – i.e. combinators are used to compose functions without making direct reference to the functions' values. These combinators are captured in the `Arrow` type class:

```
class Arrow a where
  arr   :: (b → c) → a b c
  (>>>) :: a b c → a c d → a b d
  first :: a b c → a (b,d) (c,d)
```

`arr` lifts a function to a "pure" arrow computation; i.e., the output entirely depends on the input (it is analogous to `return` in the `Monad` class). `(>>>)` composes two arrow computations by connecting the output of the first to the input of the second (and is analogous to bind (`(>>=)`) in the `Monad` class). But in addition to composing arrows linearly, it is desirable to compose them in parallel – i.e. to allow "branching" and "merging" of inputs and outputs. There are several ways to do this, but by simply defining the `first` combinator in the `Arrow` class, all other combinators can be defined. `first` converts an arrow computation taking one input and one result, into an arrow computation taking two inputs and two results. The original arrow is applied to the first part of the input, and the result becomes the first part of the output. The second part of the input is fed directly to the second part of the output.

Other combinators can be defined using these three primitives. For example, the dual of `first` can be defined as:

```
second :: (Arrow a) ⇒ a b c → a (d,b) (d,c)
second f = let swapA = arr (λ(a,b) → (b,a))
           in swapA >>> first f >>> swapA
```

Finally, it is sometimes desirable to write arrows that "loop", such as the exponential value `e` defined earlier, or a signal processing application with feedback. For this purpose, an extra combinator (not derivable from the three base combinators) is needed, and is captured in the `ArrowLoop` class:

```
class ArrowLoop a where
   loop :: a (b,d) (c,d) → a b c
```

We find that arrows are best viewed pictorially, especially for the applications commonly used with FRP. Figure 5 shows the basic combinators in this manner, including `loop`.

4 Yampa: Arrow-Based FRP

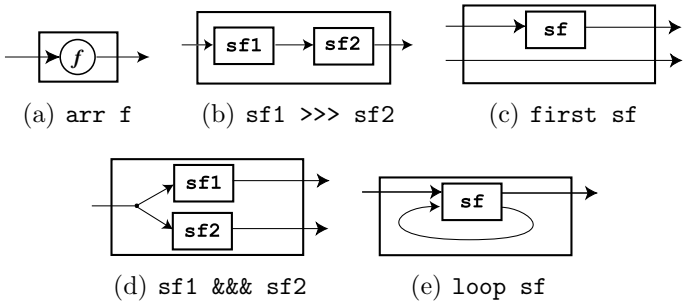
Yampa, the latest variation in FRP implementations, makes use of the Arrow class as an abstraction for *signal functions* [15], which conceptually can be viewed as:

$$\texttt{SF}\ \alpha\ \beta\ \approx\ \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$$

Programming at the level of signal functions instead of at the level of signals has certain advantages with respect to modularity and input/output. But in addition, as we shall see, it results in generally fewer space leaks.

The above conceptual realization of signal functions is not of much use in an implementation. Pragmatically, following the continuation style, a signal

```
arr    :: Arrow a ⇒ (b → c) → a b c
(>>>)  :: Arrow a ⇒ a b c → a c d → a b d
(<<<)  :: Arrow a ⇒ a c d → a b c → a b d
first  :: Arrow a ⇒ a b c → a (b,d) (c,d)
second :: Arrow a ⇒ a b c → a (d,b) (d,c)
(***)  :: Arrow a ⇒ a b c → a b' c' → a (b,b') (c,c')
(&&&)  :: Arrow a ⇒ a b c → a b c' → a b (c,c')
loop   :: Arrow a ⇒ a (b,d) (c,d) → a b c
```



(a) `arr f` (b) `sf1 >>> sf2` (c) `first sf`

(d) `sf1 &&& sf2` (e) `loop sf`

Fig. 5. Commonly Used Arrow Combinators

function is a function that, given the current input, produces a pair consisting of its current output and a continuation:

```
newtype SF a b = SF (a → (b, DTime → SF a b))
```

The full definition of `SF` as an arrow, a definition of an integral function, and the definition of a run function, are given in Figure 6. (Omitted is an alternative definition of arrowed-based FRP in the stream style, which works equally well.)

A downside of programming with signal functions is that they must be combined in a point-free style using the `Arrow` class combinators, which can lead to unwieldy programs. Fortunately, a convenient syntax for arrows has recently become popular that makes such programs easier to write and, in the case of FRP, strengthens the signal-processing intuition [23]. For example, our running example of an exponential signal can be defined as a signal function using arrow syntax as follows:

```
eSF :: SF () Double
eSF = proc () → do
  rec
    e ← integralSF 1 -≺ e
  returnA -≺ e
```

```
newtype SF a b = SF (a → (b, DTime → SF a b))

instance Arrow SF where
  arr f    = SF (λx → (f x, λdt → arr f))
  first (SF f) = SF (λ(x, z) → let (y, f') = f x
                                in ((y, z), first ∘ f'))
  SF f >>> SF g = SF (λx → let (y, f') = f x
                               (z, g') = g y
                           in (z, λdt → f' dt >>> g' dt))

instance ArrowLoop SF where
  loop (SF f) = SF (λx → let ((y, z), f') = f (x, z)
                         in (y, loop ∘ f'))

integralSF :: Double → SF Double Double
integralSF i = SF (λx → (i, λdt → integralSF (i + dt * x)))

runSF :: SF () Double → [Double]
runSF (SF f) = v : runSF (c dt)
               where (v, c) = f ()
```

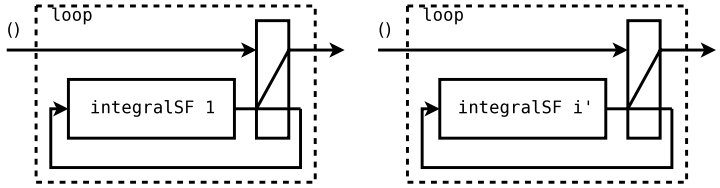
Fig. 6. Arrow-Based FRP

Note that the input (on the right) and output (on the left) to the signal function `integral 1` is the same (namely `e`), and thus this is a circular signal. This program expands into suitable calls to the `Arrow` class combinators, as well as to the `loop` combinator in the `ArrowLoop` class (because of the recursive nature of `e`), as shown in the Appendix.

5 Leak Analysis

Perhaps surprisingly, `runSF eSF` does not have a time or space leak – it runs in linear time and constant space. It behaves this way because the computation never needs to share function application results. Diagrammatically we show `eSF` and `eSF1 = snd (eSF () dt` in Figure 7 – note that they are the same size, and differ only in the initial values (1 and `i'`). Their lexical unfoldings, considerably more tedious, are given in the Appendix.

Comparing the definition of `eSF` and `e` reveals that the primary difference is in the fixed-point operators they use. `e` uses Haskell's built-in fixed-point operator, which is equivalent to the standard:



(a) eSF (b) eSF1

Fig. 7. Diagram of eSF and eSF1

```
fix f = f (fix f)
```

eSF, on the other hand, is defined in terms of the loop combinator, which *ties the loop tighter* than the standard fixed-point operator. In particular, note in Figure 6 that loop computes the value-level fixed point as z, but re-uses itself in the continuation part. This is the key to avoiding the space leak.

Indeed, all of the signal function combinators defined in Figure 6 share a common characteristic. Namely, their continuations at the next time step are identical to the current time step except for the parameters, and hence they help to preserve the structure of eSF at each unfolding.

Note that the data structures SF and C are similar: both are continuation based, and both consist of a value and a function. Both e and eSF are the fixed point of some higher-order function since the integral functions are already recursively defined. Having to compute the fixed point of recursively defined higher-order functions, and the inability of the standard call-by-need evaluation to properly detect emerging vertical sharing, are the reasons for the time and space leak in the first two FRP implementations.

To highlight the importance of the method for computing the fixed point, we note that there is another valid way to define the exponential function, namely:

```
eSF = eSF >>> integralSF 1
```

Here we rely on Haskell's default fixed-point operator, rather than the arrow loop combinator, to capture the recursion. Unfortunately, this definition suffers from the same time and space leak problem that we saw previously.

Indeed, we note that the standard arrow combinators aren't really needed at all for the exponential signal function. Suppose we define a special fixed-point operator:

```
fixSF :: SF a a → SF () a
fixSF (SF f) =
  SF (λ() → let (y, c) = f y
            in (y, λdt → fixSF (c dt)))
```

and redefine the exponential as:

```
eSF = fixSF (integralSF 1)
```

This has no space leak.

6 Alternative Approaches

It may seem that evaluating the fixed point of a higher-order function is the root of all evil in a call-by-need language, and it is reasonable to ask whether we can solve this problem independently of the FRP setting. In Figure 8 we show that in fact with the help of equational reasoning it is possible to obtain a leak-free version of `e` by term rewriting. The result of this transformation becomes a single fixed point, and unfolding `g` does not explode the closure; rather it re-uses the same `g` with a different `i`. Therefore it also avoids the space leak problem in a way that is similar to what the Arrow loop combinator does.

Trying to generalize this kind of clever transformation as rewrite rules, however, is difficult. The reason it works for FRP is that the structure of such recursively defined signals all share a common characteristic, namely that their future values retain the same kind of structure. But this is not necessarily the case in general.

Alternatively, instead of rewriting the source term to reveal its recurrent structure, we may recover the loss of sharing at runtime by using an evaluation strategy that is more clever than call-by-need. Levy [19] introduced the notion of *optimal reduction* in 1990, and Lamping [17] was the first to invent an optimal reducer for the lambda calculus. Asperti and Guerrini [3] summarize optimal reduction as:

lambda calculus = linear lambda calculus + sharing

The purpose of optimal reduction is to carefully keep track of all shared structures so that redundant reductions never occur.

In fact, optimal reduction is able to recover all forms of sharing as long as it is encoded in the original expression, including the aforementioned emerging vertical sharing problem. Verified by our implementation of both Lambdascope [27], an optimal algorithm, and the standard call-by-need algorithm using Interaction Nets, we present the comparison of the number of beta reductions and arithmetic operations[2] during the unfolding of e_n = `(runC e)!!n` in Figure 9. The data confirms that the time complexity of unfolding e_n un-

[2] We count *next dt i j* = $dt \times i + j$ as 3 operations, because there are 3 redices in the term *next dt i j*.

```
 e
↪ integralC 1 e
↪ fix (integralC 1)
-- rewrite e using fix
↪ let g i = fix (integralC i)
   in g 1                                                    -- introduce g
↪ let g i = integralC i (g i)
   in  g 1                                                   -- unfold fix
↪ let g i = let f = integralC i f
            in f
   in g 1                                                    -- introduce f
↪ let g i = let f = (i, λdt →
                        integralC (i + dt * fst f) (snd f dt))
            in C f
   in g 1                                                    -- unfold integralC
↪ let g i = let f = (i, λdt →
                        let h = integralC (i + dt * i) (snd f dt)
                        in h)
            in C f
   in g 1
-- reduce (fst f), introduce h
↪ let g i = let f = (i, λdt →
                        let h = integralC (i + dt * i) h
                        in h)
            in C f
   in g 1
-- fold (snd f dt) as h
↪ let g i = let f = (i, λdt → fix (integralC (i + dt * i)))
            in C f
   in g 1
-- rewrite h using fix
↪ let g i = let f = (i, λdt → g (i + dt * i))
            in C f
   in g
-- fold (fix ∘ integralC) as g
↪ let g i = C (i, λdt → g (i + dt * i))
   in g 1                                                    -- eliminate f
```

Fig. 8. Rewriting `e`

Exp	Call-by-need		Optimal	
	beta	arithmetic	beta	arithmetic
e_1	13	3	11	3
e_2	28	9	16	6
e_3	50	18	21	9
e_4	79	30	26	12
e_5	115	45	31	15
e_6	158	63	36	18
e_7	208	84	41	21

Fig. 9. Reduction Steps of unfolding `runC e`

der call-by-need is quadratic instead of exponential, because it redundantly re-evaluates e_{n-1}. In fact we have:

$$\text{steps}_{cbn}(n) \approx \text{steps}_{opt}(n) + \text{steps}_{cbn}(n-1)$$

where $\text{steps}_{opt}(n)$ is linear in n.

On the other hand, being optimal does not necessarily imply being the most efficient. The extra book-keeping of sharing analysis during optimal evaluation incurs a large operational overhead of both time and space. Compared to the relatively well-developed call-by-need compilation techniques, optimal evaluation is far less explored, and no truly practical implementations yet exist.

7 Conclusion

We have described two standard (albeit simplified) implementations of continuous signals in FRP, one based on streams, the other on continuations. Unfortunately, recursive signals expressed using both of these implementations have serious space and time leaks when using conventional call-by-need evaluation. The source of the problem is the failure to recognize sharing that arises inside of a recursive lambda expression.

If instead we move to the level of *signal functions*, which naturally leads to a design based on *arrows*, the leak can be eliminated. This is because a tighter computation of the fixed point of the recursive signal is achieved, in which sharing is restored. The tighter fixed point can be achieved via a suitable definition of the `loop` combinator in the arrow framework, or through the design of a special-purpose fixed-point operator.

We further show that the use of *optimal reduction* can also avoid the leak, at the expense of a much more complex evaluator. An optimal reducer is able to recognize the sharing under the recursive lambda, thus avoiding redundant computation.

8 Acknowledgements

This research was supported in part by a subcontract from Galois Connections under DARPA STTR contract 06ST1-0016, and by Yale University under a Kempner Fund Fellowship. Thanks also to the IFIP WG2.8 Working Group on Functional Programming (especially Simon Peyton Jones) for comments received during a preliminary presentation of the research.

References

[1] Andrew W. Appel. *Compiling with Continuation.* Cambridge University Press, New York, NY, USA, 1992.

[2] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. In *POPL*, pages 233–246, 1995.

[3] Andrea Asperti and Stefano Guerrini. *The optimal implementation of functional programming languages*. Cambridge University Press, New York, NY, USA, 1998.

[4] Adam Bakewell. *An Operational Theory of Relative Space Efficiency*. PhD thesis, University of York, December 2001.

[5] Adam Bakewell and Colin Runciman. A model for comparing the space usage of lazy evaluators. In *Principles and Practice of Declarative Programming*, pages 151–162, 2000.

[6] William D. Clinger. Proper tail recursion and space efficiency. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 174–185, New York, NY, USA, 1998. ACM Press.

[7] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proc. of the 2001 Haskell Workshop*, September 2001.

[8] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[9] Conal Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.

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

[11] Jorgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In *International Conference on Functional Programming*, pages 265–276, 2001.

[12] L. Huang, P. Hudak, and J. Peterson. Hporter: Using arrows to compose parallel processes. In *Proc. Practical Aspects of Declarative Languages*, pages 275–289. Springer Verlag LNCS 4354, January 2007.

[13] Liwen Huang and Paul Hudak. Dance: A declarative language for the control of humanoid robots. Technical Report YALEU/DCS/RR-1253, Yale University, August 2003.

[14] Paul Hudak. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge University Press, New York, NY, USA, 2000.

[15] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[16] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[17] John Lamping. An algorithm for optimal lambda calculus reduction. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–30, New York, NY, USA, 1990. ACM Press.

[18] John Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.

[19] Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université Paris VII, 1978.

[20] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.

[21] Simon Marlow. *Deforestation for Higher Order Functional Programs*. PhD thesis, University of Glasgow, 1996.

[22] Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, continued. In *ACM SIGPLAN 2002 Haskell Workshop*, October 2002.

[23] Ross Paterson. A new notation for arrows. In *ICFP'01: International Conference on Functional Programming*, pages 229–240, Firenze, Italy, 2001.

[24] John Peterson, Gregory Hager, and Paul Hudak. A language for declarative robotic programming. In *International Conference on Robotics and Automation*, 1999.

[25] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *First International Workshop on Practical Aspects of Declarative Languages*. SIGPLAN, Jan 1999.

[26] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[27] Vincent van Oostrom, Kees-Jan van de Looij, and Marijn Zwitserlood. Lambdascope another optimal implementation of the lambda-calculus. In *Workshop on Algebra and Logic on Programming Systems (ALPS)*, 2004.

[28] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88. European Symposium on Programming, Nancy, France, 1988 (Lecture Notes in Computer Science, vol. 300)*, pages 344–358. Berlin: Springer-Verlag, 1988.

[29] P. L. Wadler. Strictness analysis on non-flat domains by abstract interpretation over finite domains. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood, Chichester, UK, 1987.

[30] Philip L. Wadler. Fixing some space leaks with a garbage collector. *Software Practice and Experience*, 17(9):595–609, 1987.

[31] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *Proceedings of Sixth ACM SIGPLAN International Conference on Functional Programming*, Florence, Italy, September 2001. ACM.

Appendix

Because `SF` is isomorphic to a function type, we'll abbreviate the type constructors in the following reduction steps to make things easier to follow.

The direct definition of `eSF` without using the arrow syntax is:

```
eSF = loop (second (integralSF 1) >>> arr dup2)
  where
    second f (z, x) = ((z, y), second ∘ f')
      where (y, f') = f x
    dup2 (x, y) = (y, y)
```

The reduction of `eSF` is as follows:

```
   eSF
↪ loop (second (integralSF 1) >>> arr dup2)
↪ let f = second (integralSF 1) >>> arr dup2
   in loop f                                          -- introduce f
```

Note that in one step it reaches a form that corresponds to the diagram in Figure 7(a). There is no point to further reduce `eSF` because reducing `loop f` one more step will result in a weak-head normal form.

The reduction of `eSF1` under call-by-need is as follows (to conserve space, multiple steps are merged into one when there is no ambiguity):

```
   eSF1
↪ snd (eSF ()) dt
```

```
↪ let f = second (integralSF 1) >>> arr dup2
   in snd (loop f ()) dt                              -- unfold eSF, introduce f
↪ let f = second (integralSF 1) >>> arr dup2
       f1 dt = loop (f2 dt)
       ((z2, z), f2) = f ((), z)
   in f1 dt
-- unfold loop, reduce snd, introduce f1
↪ let f = second (integralSF 1) >>> arr dup2
       g' = loop (f2 dt)
       ((z2, z), f2) = f ((), z)
   in g'                                              -- reduce (f1 dt), introduce g'
↪ let g1 = second (integralSF 1)
       h1 = arr dup2
       f x1 = (z1, λdt → g1' dt >>> h1' dt)
         where (y1, g1') = g1 x1
               (z1, h1') = h1 y1
       g' = loop (f2 dt)
       ((z2, z), f2) = f ((), z)
   in g'                                              -- introduce g1 h1, unfold >>>
↪ let g1 = second (integralSF 1)
       h1 = arr dup2
       (y1, g1') = g1 ((), z)
       (z1, h1') = h1 y1
       g' = loop (f2 dt)
       ((z2, z), f2) = (z1, λdt → g1' dt >>> h1' dt)
   in g'                                               -- reduce (f ((), z))
↪ let g1 = second (integralSF 1)
       h1 = arr dup2
       (y1, g1') = g1 ((), z)
       (z1, h1') = h1 y1
       f' = g1' dt >>> h1' dt
       (z2, z) = z1
   in loop f'                                          -- projection, reduce (f2 dt),
                                                          introduce f'
```

Next, in order to show that `eSF1` is indeed the same as pictured in Figure 7(b), we need to prove that `f' = second (integralSF i') >>> arr dup2` by further reducing `g1'` and `h1'`.

```
   f'
↪ let g1 = second (integralSF 1)
       h1 = arr dup2
       (y1, g1') = g1 ((), z)
       (z1, h1') = h1 y1
       (z2, z) = z1
   in g1' dt >>> h1' dt
↪ let k = integralSF 1
       g1 = second k
       h1 = arr dup2
       (y1, g1') = g1 ((), z)
       (z1, h1') = h1 y1
       (z2, z) = z1
   in g1' dt >>> h1' dt                          -- introduce k
↪ let k = integralSF 1
       h1 = arr dup2
       (y1, g1') = (((), x), second ∘ k')
       (x, k') = k z
       (z1, h1') = h1 y1
       (z2, z) = z1
   in g1' dt >>> h1' dt                          -- unfold second, reduce (g1 ((), z))
↪ let h1 = arr dup2
       (y1, g1') = (((), x), second ∘ k')
       (x, k') = (1, λdt → integralSF (1 + dt * z))
```

```
        (z1, h1') = h1 y1
        (z2, z) = z1
    in g1' dt >>> h1' dt                    -- unfold integralSF, reduce (k z)
↪ let h1 = arr dup2
        (y1, g1') = ((), 1), second ∘ k')
        k' dt = integralSF (1 + dt * z)
        (z1, h1') = h1 y1
        (z2, z) = z1
    in g1' dt >>> h1' dt                    -- projection, eliminate x
↪ let h1 = arr dup2
        i' = 1 + dt * z
        y1 = ((), 1)
        (z1, h1') = h1 y1
        (z2, z) = z1
    in second (integralSF i') >>> h1' dt    -- projection, reduce (g1' dt),
                                               introduce i'
↪ let i' = 1 + dt * z
        y1 = ((), 1)
        (z1, h1') = (dup2 y1, λdt → arr dup2)
        (z2, z) = z1
    in second (integralSF i') >>> h1' dt    -- unfold arr, reduce (h1 y1)
↪ let i' = 1 + dt * z
        z1 = (1, 1)
        h1' dt = arr dup2
        (z2, z) = z1
    in second (integralSF i') >>> h1' dt     -- projection, reduce (dup2 y1)
↪ let i' = 1 + dt * 1
    in second (integralSF i') >>> arr dup    -- unfold (h1' dt), eliminate z
```