Abstract

The Theory and Practice of Causal Commutative Arrows

Hai Liu

2011

Arrows are a popular form of abstract computation. Being more general than monads, they are more broadly applicable, and in particular are a good abstraction for signal processing and dataflow computations. Most notably, arrows form the basis for *Yampa*, a functional reactive programming (FRP) language embedded in Haskell. Our primary interest is in better understanding the class of abstract computations captured by Yampa. Unfortunately, arrows are not concrete enough to do this with precision for the lack of a domain specific knowledge.

In this thesis, we present a more constrained class of arrows called *causal commutative arrows* (CCA) that introduces an *init* operator to capture the causal nature of arrow effects, as well as two additional laws. Our key contribution is the identification of a normal form for CCA, and by defining a normalization procedure we have developed an optimization strategy that yields dramatic improvements in performance over conventional implementations of arrows.

To study this abstract class of computation more concretely, we explore three different and yet related applications of CCA, namely, synchronous dataflow, ordinary differential equation, and functional reactive programming. For each application, we develop an arrow based DSL that is an instance of CCA, and we show their significant advantages at improving program's run-time behavior, such as eliminating hideous space leaks, and boosting performances by orders of magnitude.

The Theory and Practice of Causal Commutative Arrows

A Dissertation

Presented to the Faculty of the Graduate School

of

Yale University

in Candidacy for the Degree of

Doctor of Philosophy

by

Hai Liu

Dissertation Director: Professor Paul Hudak

Department of Computer Science

May 2011

Acknowledgements

I owe my deepest gratitude to my advisor, Paul Hudak, for giving me the opportunity and freedom to pursue this research, for his encouragement and guidance througout my graduate study, and for his enthusiasm and persistence that have influnced me in ways beyond just my work.

I would like to thank my committee, Paul Hudak, Zhong Shao, Drew McDermott, and Henrik Nilsson, for providing guidance and valuable feedbacks to my talks and papers.

Thanks to Henrik Nilsson and Antony Courtney for their work on Yampa that has been the inspiration for my work. Thanks to Eric Cheng, Andreas Voellmy, and people at the Yale Haskell Group for many insightful discussions and feedbacks to my talks. It is also an honor for me to collaborate with Richard Yang, Hao Wang, and George Mou for works outside of my thesis.

Finally, I want to thank my families and especially my wife, Lihua He, for their understanding and sacrifice, and for having faith in me even in the most difficult times.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Domain Specific Languages (DSLs) are a novel approach to tackle domain specific problems, with a focus on domain abstractions that bring expressiveness and productivity. Their restricted semantics lend themselves to advanced optimization techniques that are often not readily available in general purpose languages. The ability to abstract over domain knowledges is a key criteria in designing DSLs, and very often such designs are based upon formal models for their properties are well studied, and programs can also benefit from techniques such as type analysis and formal reasoning.

Arrows introduced by Hughes [2000] are a popular form of such formalisms, and they have enjoyed a wide range of applications, often as an embedded DSL, including signal processing [Nilsson et al., 2002], graphical user interface [Courtney and Elliott, 2001], parsers and printers [Jansson and Jeuring, 1999], and so on. One notable example of such applications is in the area of functional reactive programming, or

FRP, which models hybrid reactive system in a high-level and declarative way. In particular, arrows form the basis for a DSL called *Yampa*, which has been used in a variety of applications, including robotics [Hudak et al., 2003, Peterson et al., 1999b,a], sound synthesis [Giorgidze and Nilsson, 2008, Cheng and Hudak, 2008], animations [Hudak et al., 2003], video games [Courtney et al., 2003, Cheong, 2005], bio-chemical processes [Hudak et al., 2008], control systems [Oertel, 2006], and graphical user interfaces [Courtney and Elliott, 2001, Courtney, 2004].

One of the key abstractions in conventional FRP languages is to use signals to represent time-changing values:

$$Signal\ a \approx Time \rightarrow a$$

A signal can be viewed as a function from time to a value, and this simple but concise abstraction gives rise to a rich semantics for programming reactive systems with a functional language. Despite its conceptual elegance, signals are not without problems. Semantics wise, there is an ambiguity in defining higher-order signals: whether they shall keep changing along with time, or be frozen until the moment of resumption. Implementation wise, there was a space leak problem, associated with recursive signal definitions troubling early FRP implementations.

These problems motivated Yampa, also known as arrow-ized FRP or AFRP, where instead of representing signals directly, we make a further abstraction with signal functions:

$$SF\ a\ b \approx Signal\ a \rightarrow Signal\ b$$

Signal functions can be viewed as a basic building block mapping one signal into another, and we write programs by composing simpler signal functions together to

form more complex ones. Arrows become an ideal abstract model for signal functions by supplying a minimal set of combinators to support both sequential and parallel compositions, as well as looping structures.

Programming at the signal function level is more restricted since we lose the ability to manipulate signals directly. As a consequence of this kind of imposed disciplines, we are no longer troubled by higher-order signals since they are not even representable in Yampa. Instead, we now have higher-order signal functions, which play a key role in modeling structural reactivity. Moreover, the space leak problem plaguing conventional FRP implementations has simply ceased to exist in Yampa programs. A common folklore is that since a representation of signals has to lug around their entire history of values, they are prone to space leak problems. While there is a certain degree of truth in this, the real problem is more subtle and requires careful analysis.

A good DSL shall capture the essence of the given domain and provide the right level of abstraction. Being a generic computation model, however, arrows must be extended by domain specific operators when applied to a particular problem domain. In order to better understand the class of abstract computation represented by Yampa, we develop a more constrained class of arrows called causal commutative arrows, or CCAs, and study its theory and application in depth.

In particular, we notice that there are two essential properties of FRP not captured by conventional arrows:

1. Parallel composition of signal functions are *commutative* since all side effects are entirely local and do not interfere with one another, and yet the paral-

lel composition of generic arrows presumes that the compositional order is of importance.

2. FRP computations must be *causal*, i.e., computations shall only depend on the past and current inputs, but not future ones. Usually we cannot avoid mentioning time when we talk about causality, but time itself is hidden in the abstraction of signal functions.

To generalize arrows that are commutative, we introduce a commutativity law. To generalize causality, we introduce a new arrow combinator called *init* together with a product law that describes its property. Arrows that are both commutative and causal are called causal commutative arrows (CCA).

1.2 Dissertation Overview

In order to formalize CCA and its properties, we introduce a language for CCA, which is basically a typed lambda calculus with arrow combinators added. A notable simplification in the CCA language setting it apart from the lambda calculus is that there is no lambda or recursion at the arrow level, which effectively means all arrow programs in this language are only compositions of a finite number of arrow combinators. We then prove an important property of CCA, i.e., all CCA programs can be transformed into a canonical representation that we call *Causal Commutative Normal Form*, or CCNF. We prove that the normalization procedure is sound, based on equational reasoning using only the CCA laws.

The discovery of a normal form for CCA is of importance both in theory and in

practice. It has become common in the Haskell research community to characterize a particular model of computation via a type class, its operators, and their associated laws. But the soundness of the laws is not typically established with respect to a formal semantics, and the usual notions of confluence, termination, normal form, etc. are absent. We attempt here to address these concerns while still keeping the computational model relatively abstract, thus giving a stronger characterization of the model itself.

Our formalization of the CCA language is fairly abstract since it makes no mention of either a denotational or an operational semantics. This is indeed desirable because CCA by itself is only governed by the set of CCA laws, or in other words, an axiomatic semantics. It also means CCA could be more broadly applicable than Yampa itself. We further explore the applications of CCA in three main areas:

Synchronous dataflow Most FRP implementations, including Yampa, belong to a larger family of synchronous dataflow languages. We look at simple languages for discrete dataflow and compare the usual stream based DSL to an arrow based one. We argue that the CCA normalization is an effective compile-time transformation for the latter that improves its run-time performances. Further more, we relate the CCNF of arrow programs to an operational semantics for dataflow languages based on Mealy machines, a form of automata.

Ordinary Differential Equation We explore the DSL design space for ordinary differential equations, ranging from a shallow embedding to a deep embedding, and middle-grounds in between. A naive implementation based on the *lazy tower of derivatives* [Karczmarczuk, 1998] is straightforward but has serious

time and space leaks due to the loss of sharing when traversing cyclic and infinite data structures. We argue that an arrow based solution can help to capture both sharings and recursions and at the same time avoid space leaks. We further identify this is an application of CCA besides dataflow, and improve the run-time ODE performance by normalizing arrows at compile-time.

Functional Reactive Programming Finally, we come back to our motivating area of FRP, and explore how arrows help to solve a similar and yet different space leak problem than the ODE one. We also present a unified approach for CCA to capture both the discrete and continuous time-changing aspects in Yampa, and explore the limitation of CCA normalization in supporting dynamic structures.

Besides the main theme on CCAs, through out the chapters we also evangelize arrows as a suitable computation model in the design and implementation of embedded DSLs. The compositional structure of arrow programs maintains sharing of computation by nature, and entails easy traversals and transformations.

1.3 Contributions

We summarize our contributions as follows:

1. We define a notion of *commutative arrow* by extending the conventional set of arrow laws to include a commutativity law.

2. We define an *ArrowInit* type class with an *init* operator that captures a generalized notion of computational causality and satisfies a *product law*.

3. We define a restricted language for *CCA*, in which the above ideas are manifest. For such arrows we establish:

   (a) a *normal form*, and

   (b) a *normalization procedure*.

   We achieve this result with only the CCA laws, without referring to any concrete semantics or implementation.

4. We define an *optimization technique* for CCA that yields substantial improvements in performance over previous attempts to optimize arrow combinators and arrow syntax.

5. We implement CCA normalization and optimization techniques in Haskell utilizing Template Haskell as a means to seamlessly integrate with existing arrow programs.

6. We demonstrate how arrows and CCAs can be successfully applied in three applications including synchronous dataflow, ordinary differential equation, and functional reactive programming. For each of the three applications, we show that:

   (a) an arrow based DSL effectively models computation in the target domain;

   (b) the arrow is a CCA instance because it satisfies the commutativity law, and implements a domain specific *init* operator satisfying the product law;

   (c) there are practical benefits of using arrows and CCAs including the elimination of certain space leaks, and improved run-time performances through

CCA normalization.

7. We identify a space leak problem in both the DSLs for ODE and early FRP implementations, trace its origin to the sharing of beta reductions in the standard call-by-need evaluation, and show that arrow based DSLs can avoid the leak by employing a finite data structure and a custom fixed-point operator.

1.4 Advice to the Reader

The remaining chapters are organized as follows. Chapter 2 gives a brief overview of arrows and Yampa. The knowledgeable reader may prefer skipping directly to Chapter 3 where we give a formal treatment of the CCA language and its properties including normalization, and a discussion of possible extensions. We then take an in-depth look at three applications of CCA: synchronous dataflow in Chapter 4, ordinary differential equation in Chapter 5, and functional reactive programming in Chapter 6. Finally we conclude and discuss future work in Chapter 7.

We assume the reader has a basic knowledge of functional programming and the Haskell programming language, since most programs in this dissertation are written in Haskell, and compiled with the Glasgow Haskell Compiler (GHC). Chapter 3 also contains proofs of theorems, although the bulk of them are relegated to the Appendix.

Chapter 2

Arrows and FRP

2.1 Arrows

Arrows [Hughes, 2000] are a generalization of monads that relax the stringent linearity imposed by monads, while retaining a disciplined style of composition. Arrows have enjoyed a wide range of applications, often as a domain-specific embedded language (DSEL [Hudak, 1996, 1998]), including the many Yampa applications cited earlier, as well as parsers and printers [Jansson and Jeuring, 1999], parallel computing [Huang et al., 2007], and so on. Arrows also have a theoretical foundation in category theory, where they are strongly related to (but not precisely the same as) *Freyd categories* [Atkey, 2008, Power and Thielecke, 1999].

2.1.1 Conventional Arrows

Like monads, arrows capture a certain class of abstract computations, and offer a way to structure programs. In Haskell this is achieved through the *Arrow* type class:

$$\textbf{class } \mathit{Arrow}\ a \textbf{ where}$$

$$\mathit{arr} \quad :: (b \to c) \to a\ b\ c$$

$$(\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d$$

$$\mathit{first} \ :: a\ b\ c \to a\ (b, d)\ (c, d)$$

The combinator *arr* lifts a function from b to c to a "pure" arrow computation from b to c, namely $a\ b\ c$ where a is the arrow type. The output of a pure arrow entirely depends on the input (it is analogous to *return* in the *Monad* class). $\ggg$ composes two arrow computations by connecting the output of the first to the input of the second (and is analogous to bind $\gg=$ 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:

$$\mathit{second} \quad :: (\mathit{Arrow}\ a) \Rightarrow a\ b\ c \to a\ (d, b)\ (d, c)$$

$$\mathit{second}\ f = \mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ f \ggg \mathit{arr}\ \mathit{swap}$$

$$\textbf{where } \mathit{swap}\ (a, b) = (b, a)$$

Parallel composition can be defined as a sequence of first and second:

left identity	$arr\ id \ggg f = f$
right identity	$f \ggg arr\ id = f$
associativity	$(f \ggg g) \ggg h = f \ggg (g \ggg h)$
composition	$arr\ (g\ .\ f) = arr\ f \ggg arr\ g$
extension	$first\ (arr\ f) = arr\ (f \times id)$
functor	$first\ (f \ggg g) = first\ f \ggg first\ g$
exchange	$first\ f \ggg arr\ (id \times g) = arr\ (id \times g) \ggg first\ f$
unit	$first\ f \ggg arr\ fst = arr\ fst \ggg f$
association	$first\ (first\ f) \ggg arr\ assoc = arr\ assoc \ggg first\ f$
	where $assoc\ ((a, b), c) = (a, (b, c))$

Figure 2.1: Conventional Arrow Laws

$$(\star\star\star) \quad :: (Arrow\ a) \Rightarrow a\ b\ c \to a\ b'\ c' \to a\ (b, b')\ (c, c')$$

$$f \star\star\star g = first\ f \ggg second\ g$$

A mere implementation of the arrow combinators, of course, does not make it an arrow – the implementation must additionally satisfy a set of *arrow laws*, which are shown in Figure 2.1.

2.1.2 Looping Arrows

To model recursion, we can introduce a *loop* combinator [Paterson, 2001]. For example, many applications in signal processing would require feedback loops from the output to the input. In Haskell this combinator is captured in the *ArrowLoop* class:

left tightening $loop\ (first\ h \ggg f) = h \ggg loop\ f$

right tightening $loop\ (f \ggg first\ h) = loop\ f \ggg h$

sliding $loop\ (f \ggg arr\ (id \times k)) = loop\ (arr\ (id \times k) \ggg f)$

vanishing $loop\ (loop\ f) = loop\ (arr\ assoc^{-1} \ggg f \ggg arr\ assoc)$

superposing $second\ (loop\ f) = loop\ (arr\ assoc \ggg second\ f \ggg arr\ assoc^{-1})$

extension $loop\ (arr\ f) = arr\ (trace\ f)$

$\textbf{where}\ trace\ f\ b = \textbf{let}\ (c, d) = f\ (b, d)\ \textbf{in}\ c$

Figure 2.2: Arrow Loop Laws

$\textbf{class}\ Arrow\ a \Rightarrow ArrowLoop\ a\ \textbf{where}$

$loop :: a\ (b, d)\ (c, d) \to a\ b\ c$

Intuitively, the second output of the arrow inside *loop* is immediately fed back to its second input, and thus becomes a form of recursion. A valid instance of this class should satisfy the additional laws shown in Figure 2.2. This class and its associated laws are related to the trace operator in [Street et al., 1996, Hasegawa, 1997], which was generalized to arrows in [Paterson, 2001].

We find that arrows are best viewed pictorially, especially for applications such as signal processing, where domain experts commonly draw signal flow diagrams. Figure 2.3 shows some of the basic combinators in this manner, including *loop*.

2.1.3 Arrow Notation

Arrow expressions we have seen so far maintain a point-free style that requires explicit "plumbing" using arrow combinators, and may be obscure and inconvenient in some

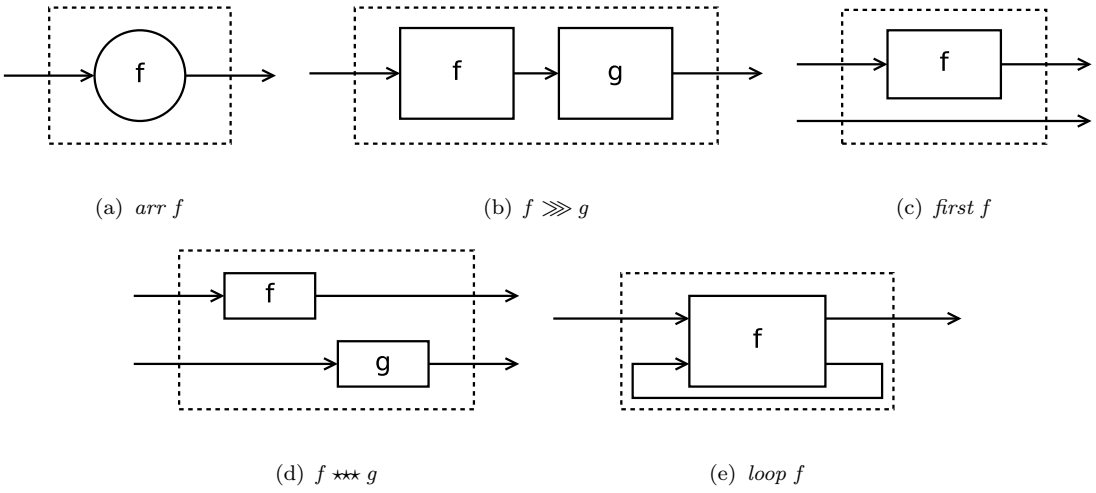
$$arr \;\; :: Arrow\ a \Rightarrow (b \to c) \to a\ b\ c$$

$$(\ggg) :: Arrow\ a \Rightarrow a\ b\ c \to a\ c\ d \to a\ b\ d$$

$$first \;\; :: Arrow\ a \Rightarrow a\ b\ c \to a\ (b, d)\ (c, d)$$

$$(\star\star\star) :: Arrow\ a \Rightarrow a\ b\ c \to a\ b'\ c' \to a\ (b, b')\ (c, c')$$

$$loop \;\; :: Arrow\ a \Rightarrow a\ (b, d)\ (c, d) \to a\ b\ c$$



(a) *arr f* (b) $f \ggg g$ (c) *first f*

(d) $f \star\star\star g$ (e) *loop f*

Figure 2.3: Commonly Used Arrow Combinators

$$
\begin{aligned}
exp &= ... \\
&\mid \textbf{proc}\ pat \rightarrow cmd \\
cmd &= exp \rightarrowtail exp \\
&\mid \textbf{do}\ \{\, stmt; ...; stmt; cmd \,\} \\
stmt &= pat \leftarrow cmd \\
&\mid cmd \\
&\mid \textbf{rec}\ \{\, stmt; ...; stmt \,\}
\end{aligned}
$$

Figure 2.4: Grammar for Arrow Notations

cases. Paterson [2001] devises a set of *arrow notation* (also called *arrow syntax*) that help users to express arrows in a "point-ful" style with improved presentation, with the grammar defined in Figure 2.4. Programs written in such special syntax can be automatically translated by a pre-processor back to the combinator form. GHC in fact has built-in support for arrow notation.

We omit the detail translation rules of arrow notation, and instead briefly explain through the example of the parallel composition ($\star\!\!\star\!\!\star$) that is re-written in arrow notation as follows:

$$
\begin{aligned}
&(\star\!\!\star\!\!\star) \quad :: Arrow\ a \Rightarrow a\ b\ c \rightarrow a\ b'\ c' \rightarrow a\ (b, b')\ (c, c') \\
&f \star\!\!\star\!\!\star g \quad = \textbf{proc}\ (x, y) \rightarrow \textbf{do} \\
&\quad x' \leftarrow f \rightarrowtail x \\
&\quad y' \leftarrow g \rightarrowtail y \\
&\quad returnA \rightarrowtail (x', y') \\
&returnA :: Arrow\ a \Rightarrow a\ b\ b
\end{aligned}
$$

$returnA = arr\ (\lambda x \rightarrow x)$

The **proc** keyword starts an arrow expression whose input is a pair (x, y), and whose output is the output of the last command in the **do**-block. The **do**-block allows one to use variable bindings as "points" to interconnect arrows, e.g., $x' \leftarrow f \prec x$ passes a value x through an arrow f and names the result x'. So the **proc** expression is really just another way to express arrow compositions by naming the "points", in contrast to the point-free style.

2.2 Yampa: Arrow-ized FRP

As mentioned in Chapter 1, *Yampa* [Hudak et al., 2003] is a flavor of FRP that instead of having time varying values (also called *signals*) as first-class objects, represents *signal functions*, or signal transformers, as first-class. The compositional model of Yampa is based on arrows, and hence we sometimes refer to Yampa or its variants as *Arrow-ized FRP*.

2.2.1 Yampa Primitives

Yampa models both the continuous and discrete aspects of a hybrid reactive system. The primary arrow type in Yampa is called *SF*, and it is declared as instances of the *Arrow* and *ArrowLoop* classes (implementation details are omitted below):

data *SF a b* = ...

instance *Arrow SF* **where** ...

instance *ArrowLoop SF* **where** ...

As an example of continuous modeling, consider the program for a robot simulator given by Hudak et al. [2003]:

$$
\begin{aligned}
&xSF :: SF\ SimbotInput\ Distance \\
&xSF = \mathbf{proc}\ inp \rightarrow \mathbf{do} \\
&\quad vr \leftarrow vrSF \quad \prec inp \\
&\quad vl \leftarrow vlSF \quad \prec inp \\
&\quad \theta \leftarrow thetaSF \prec inp \\
&\quad i \leftarrow integral \prec (vr + vl) * cos\ \theta \\
&\quad returnA \prec i\ /\ 2
\end{aligned}
$$

The above *xSF* models the computation of the x position of a differential drive robot governed by the following mathematical equation:

$$x(t) = \frac{1}{2}\int_0^t (v_r(t) + v_l(t))cos(\theta(t))dt$$

Even for those not familiar with arrow syntax or Haskell, the close correspondence between the mathematics and the Yampa program should be clear. As in most high-level language designs, this is the primary motivation for developing a language such as Yampa: reducing the gap between program and specification.

More importantly, we make two observations in the above Yampa definition for *xSF*:

1. There is an explicit input *inp* to drive both the left and right velocities. This is because the velocities are obtained from the input data gathered by sensors (represented here as the data type *SimbotInput*), which is considered as an external input to our program. Such kind of information or specification is

entirely hidden in the math equation. The nature of arrows require us to state the explicit mapping from inputs to outputs, and hence is a more disciplined way of writing programs.

2. Time t itself is not mentioned anywhere in the program. In a similar manner to conventional FRP programs, by not having to deal with time directly, we keep signal functions abstract and hence allow more flexible implementations. The primary effect in the above program is the *integral* arrow with the following type (here slightly simplified):

   $$integral :: Fractional\ a \Rightarrow a \rightarrow SF\ a\ a$$

   What *integral* does is to numerically integrate the incoming signal over time, and the only argument it takes is an initial value. It basically transforms a continuous-time signal to its numerical integral that is also continuous-time.

Besides transforming continuous signals, Yampa also supports discrete events. Events differ from continuous values in their nature of occurrence: they are instantaneous and have no duration. Events are represented by an *Event* type in Yampa, which is isomorphic to the commonly used *Maybe* type in Haskell:

$$\begin{aligned} &\textbf{data}\ Event\ a = Event\ a \\ &\qquad\qquad\quad \mid NoEvent \end{aligned}$$

Individual events may carry a value, and a very common event operation is to tag the event with a different value:

$$tag :: Event\ a \rightarrow b \rightarrow Event\ b$$

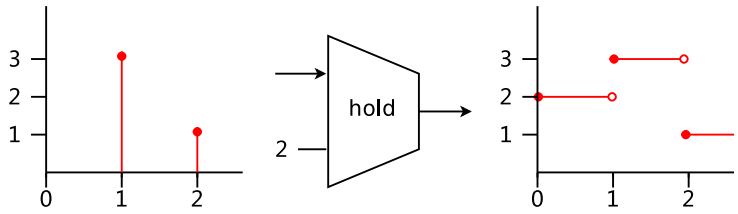


Figure 2.5: Semantics of *hold*

$$Event \_ \text{`}tag\text{`} \ v = Event \ v$$

$$NoEvent \ \text{`}tag\text{`} \ \_ = NoEvent$$

We can coerce from a discrete-time event stream to continuous-time signal by "holding" a previous event value over the period of *NoEvent* until a new event occurs. We illustrate the semantics of a *hold* function in Figure 2.5, and give its type below:

$$hold :: a \rightarrow SF \ (Event \ a) \ a$$

Although Yampa supplies *hold* as a pre-defined signal function, actually it can be expressed in terms of a more primitive signal function called *iPre* in Yampa, which represents an infinitesimal delay of a signal:

$$iPre :: a \rightarrow SF \ a \ a$$

We briefly explain the semantics of *iPre* as follows. The output signal of *iPre* i will take the value of i initially, and then subsequently at time t yield the input signal at time $t - \epsilon$, where ϵ is an infinitesimal value close to but never equal to 0.

With *iPre*, we can define *hold* as a looping arrow as follows:

$$hold \ i = \textbf{proc} \ e \rightarrow \textbf{do}$$

$$\quad \textbf{rec} \ y \leftarrow iPre \ i \rightarrowtail z$$

$$\begin{aligned}
&\textbf{let } z = \textbf{case } e \textbf{ of } \mathit{Event}\ x \ \rightarrow x \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{NoEvent} \rightarrow y \\
&\mathit{returnA} \prec z
\end{aligned}$$

In the above definition, the output z takes the event value of x if there is actually an event coming from its input, otherwise it maintains its old value passed through an *iPre* arrow.

Likewise, we can implement an edge condition checker that generates an unit event when its input signal changes from *False* to *True*:

$$\begin{aligned}
&\mathit{edge} :: \mathit{SF}\ \mathit{Bool}\ (\mathit{Event}\ ()) \\
&\mathit{edge} = \textbf{proc } x \rightarrow \textbf{do} \\
&\quad x' \leftarrow \mathit{iPre}\ \mathit{True} \prec x \\
&\quad \mathit{returnA} \prec \textbf{if } \neg\, x' \wedge x \textbf{ then } \mathit{Event}\ () \textbf{ else } \mathit{NoEvent}
\end{aligned}$$

The *edge* arrow compares the current value of its input to its older value, and outputs an event when the edge condition is satisfied. Note the use of *True* as the initial value to *iPre*, which is to prevent triggering a false event if the input signal begins with a *True* value at the start of time.

Thus far we have given a few examples illustrating some of the key features of Yampa, and it suffices to say both *integral* and *iPre* are important primitives for the Yampa arrow. Also we cannot help but noticing a strong similarity between their types: both take an initial value of type a and return an arrow of type $\mathit{SF}\ a\ a$, except that *integral* requires its input to be "integratable" over time. One of the primary motivations for CCA is to capture such domain specific operators more abstractly, and we will keep revisiting them in the remaining chapters of this thesis.

2.2.2 Switching

Another important feature of Yampa is that programs can respond to events by *switching modes*. The simplest form of *switch* has the following signature:

$$switch :: SF\ a\ (b, Event\ c) \rightarrow (c \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$

The intuition is that when a switching event occurs, the signal function changes from one into another. The first argument to switch is an arrow that models the original behavior of the result signal function as well as the switching event, and the second argument is an event handler that yields the signal function to switch into when supplied an event. Note that this kind of mode switching is *permanent*, i.e., the event checking is only done before the switch happens, after which no more event is produced or checked, and thus the mode has changed permanently. On the other hand, if we want to model recurrent event handling, we can either adopt a looping arrow with conditionals, or use recursion as demonstrated in the example below.

To illustrate how *switch* can be used, we model in the program below the physics of a bouncing ball with perfect elastic collision when hitting ground:

$$\begin{aligned}
&\textbf{type}\ Height\ \ = Double\\
&\textbf{type}\ Velocity = Double\\
&\textbf{type}\ Ball\ \ \ \ = (Height, Velocity)\\
&ball :: Ball \rightarrow SF\ ()\ (Height, Event\ Ball)\\
&ball\ (h, v) = \textbf{proc}\ () \rightarrow \textbf{do}\\
&\quad v \leftarrow integral\ v0 \prec -9.8\\
&\quad h \leftarrow integral\ h0 \prec v
\end{aligned}$$

$$e \leftarrow edge \qquad \prec h \leqslant 0$$

$$returnA \prec (h, e \text{ `}tag\text{` } (h, v))$$

$$bouncing :: Ball \rightarrow SF\ ()\ Height$$

$$bouncing\ (h0, v0) = ball\ (h0, v0)\ \text{`}switch\text{`}\ (\lambda(h, v) \rightarrow bouncing\ (h, -v))$$

The *ball* arrow models the physics of a free falling ball, as well as the event detection when its height becomes 0. Due to imperfect numerical precision of computer simulations, the edge condition we use here is $h \leqslant 0$ instead of $h = 0$. The event is then tagged with the current value of the ball's height h, and velocity v. The *bouncing* arrow models a bouncing ball by inverting its velocity when it hits the ground. Note that *bouncing* is recursively defined because we want the ball to continue bouncing, and the arrow to switch into would just be itself, but with an inverted velocity.

Yampa supports a number of different switch primitives including parallel ones that handle a collection of arrows. We omit such discussions in this short introduction since they are not the focus of this thesis, and instead refer our readers to Courtney [2004].

2.2.3 Animating Signal Functions

So far the examples we have covered are just specifications written as signal functions, and what is missing is a way to run the actual simulation modelled by the Yampa arrow. To do this we need to connect our program to the external world including the I/O system. Yampa provides a *reactimate* function (here slightly simplified) for this purpose:

$$reactimate :: IO\ (DTime, a) \rightarrow (b \rightarrow IO\ ()) \rightarrow SF\ a\ b \rightarrow IO\ ()$$

$$reactimate\ sense\ actuate\ sf = ...$$

The first argument *sense* is an IO action that gets the next input sample along with the amount of time elapsed since the previous sample. The second argument *actuate* takes the current output and produces an IO action. The third argument *sf* is the actual signal function to execute. Although denotationally a Yampa arrow can model both continuous and discrete behaviors, the *reactimate* function can only approximate the continuous behavior through discrete sampling. Therefore the actual signal function is run in iterations, and *reactimiate* will invoke *sense* at the start and *actuate* at the end of each iteration.

The design for *reactimate* is to deliberately ensure a Yampa program stays true to its denotational semantics, and is only discretized and interacts with the I/O system when run with *reactimate*.

Chapter 3

Causal Commutative Arrows

3.1 Definition

First, as mentioned in the introduction, the set of arrow and arrow loop laws is not strong enough to capture interesting computations modelled by Yampa. In particular, the *commutativity law* shown in Figure 3.1 establishes a non-interference property for concurrent computations – effects are still allowed, but this law guarantees that they cannot interfere with each other. We say that an arrow is *commutative* if it satisfies the conventional laws as well as this critical additional law. Yampa is in fact based on commutative arrows.

commutativity $\mathit{first}\ f \ggg \mathit{second}\ g = \mathit{second}\ g \ggg \mathit{first}\ f$

product $\mathit{init}\ i \star\!\!\star\!\!\star \mathit{init}\ j = \mathit{init}\ (i, j)$

Figure 3.1: Causal Commutative Arrow Laws

Second, we note that Yampa has a primitive operator called *iPre* that is used to inject an infinitesimal delay into a computation; indeed it is one of the primary effects imposed by the Yampa arrow. Similar operators, often called *delay*, also appear in dataflow programming [Wadge and Ashcroft, 1985], stream processing [Stephens, 1997, Thies et al., 2002], and synchronous languages [Caspi et al., 1987, Colaço et al., 2004]. In all cases, the operator introduces stateful computations into an otherwise stateless setting.

In an effort to make this operation more abstract, we rename it *init* and capture it in the following type class:

class *ArrowLoop* $a \Rightarrow$ *ArrowInit* a **where**

$init :: b \rightarrow a\ b\ b$

Our intention is for the *init* operator to represent a generalized concept of causality, namely the computational dependency between the arrow's input and output. The type of *init* makes sure such a computation is polymorphic over the type of its argument that we call *initial value*, which is the same type as the input and output of the returned arrow. When applied to the application domain of dataflow computations, this dependency refers to the usual concept of temporal causality, i.e., the current output depends only on the current as well as previous inputs. But in the general case, we make no other assumptions about the nature of these values besides computational causality. The very existence of an *init* operator is an indication as well as an evidence of causal computations.

Additionally, a valid instance of the *ArrowInit* class must satisfy the *product* law shown in Figure 3.1. This law states that two *init*s paired together are equivalent to

one *init* of a pair. Here we use the $\star\star\star$ operator instead of its expanded definition *first*... $\ggg$ *second*... to imply that the product law assumes commutativity.

We will see in later sections that *init* and its product law are critical to our normalization and optimization strategies. But *init* is also important in allowing us to define operators that were previously taken as domain-specific primitives, for instance, the *hold* and *edge* arrows in Yampa.

3.2 A Language for CCA

To study the properties of CCA more rigorously, we first introduce a language for CCA programs in Figure 3.2, which is an extension of a typed lambda calculus with the unit and product types, a trace operator, and arrows. We elaborate our design decisions below:

- In Haskell we use type *Arrow* $a \Rightarrow a\ b\ c$ to represent an arrow type a mapping from type b to type c. However, CCA does not have type classes, and thus we write $A \rightsquigarrow B$ instead.

- The CCA language syntax separates expressions from arrow programs and hence all arrow programs are only defined using the arrow combinators, and of a static structure. This is both a simplification and a desirable property for normalization as we shall see in the next section.

- For the same reason as above, we provide all five arrow combinators: *arr*, $\ggg$, *first*, *loop* and *init*, as primitive operators rather than pre-defined constants of

Syntax

$$
\begin{array}{lrcl}
\text{Variables} & V & ::= & x \mid y \mid z \mid ... \\
\text{Types} & A, B, C & ::= & 1 \mid M \times N \mid A \to B \mid A \rightsquigarrow B \\
\text{Expressions} & M, N & ::= & () \mid V \mid (M, N) \mid \mathit{fst}\ M \mid \mathit{snd}\ M \mid \lambda V.M \mid M\ N \mid \mathit{trace}\ M \\
\text{Programs} & P, Q & ::= & \mathit{arr}\ M \mid P \ggg Q \mid \mathit{first}\ P \mid \mathit{loop}\ P \mid \mathit{init}\ M \\
\text{Environment} & \Gamma & ::= & x_0 : A_0, ..., x_n : A_n
\end{array}
$$

Typing Rules

$$\text{(UNIT)}\quad \Gamma \vdash () : 1 \qquad \text{(VAR)}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{(TRACE)}\ \frac{\Gamma \vdash M : A \times C \to B \times C}{\Gamma \vdash \mathit{trace}\ M : A \to B}$$

$$\text{(ABS)}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \text{(APP)}\ \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\text{(PAIR)}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad \text{(FST)}\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathit{fst}\ M : A} \qquad \text{(SND)}\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathit{snd}\ M : B}$$

$$\text{(ARR)}\ \frac{\vdash M : A \to B}{\vdash \mathit{arr}\ M : A \rightsquigarrow B} \qquad \text{(SEQ)}\ \frac{\vdash P : A \rightsquigarrow B \quad \vdash Q : B \rightsquigarrow C}{\vdash P \ggg Q : A \rightsquigarrow C}$$

$$\text{(FIRST)}\ \frac{\vdash P : A \rightsquigarrow B}{\vdash \mathit{first}\ P : A \times C \rightsquigarrow B \times C} \qquad \text{(LOOP)}\ \frac{\vdash P : A \times C \rightsquigarrow B \times C}{\vdash \mathit{loop}\ P : A \rightsquigarrow B}$$

$$\text{(INIT)}\ \frac{\vdash M : A}{\vdash \mathit{init}\ M : A \rightsquigarrow A}$$

Figure 3.2: *CCA*: a Language for Causal Commutative Arrows

more liberal types. This is in contrast to the arrow calculus given by Lindley et al. [2010].

- All arrow programs have an empty type environment in the typing rules, i.e., they are closed terms with no free variables.

- The *trace* operator here is similar to the *trace* function we have previously seen in Figure 2.2. It provides a form of general recursion for lambda expressions that is akin to the *fix* operator. In fact, we can define *fix* in terms of *trace*:

$$fix = \lambda f \ . \ snd \ (trace \ (\lambda x \rightarrow ((), f \ (snd \ x))))$$

  And *trace* can also be defined from *fix*, of course.

Besides satisfying the usual beta law for lambda expressions, arrows in CCA must also satisfy the nine conventional arrow laws (Figure 2.1), the six arrow loop laws (Figure 2.2), and the two CCA laws (Figure 3.1).

3.3 Normalization

In most implementations, arrow programs carry a run-time overhead, primarily due to the use of a data structure for arrow instances, as well as the extra tupling forced onto function's arguments and return values. There have been several attempts [Hughes, 2004, Nilsson, 2005] to optimize arrow-based programs using arrow laws, but the result has not been entirely satisfactory. Although conventional arrow and arrow loop laws offer ways to combine pure arrows and collapse nested loops, they are not specific enough to target *effectful* arrows, such as the *init* combinator. Certain

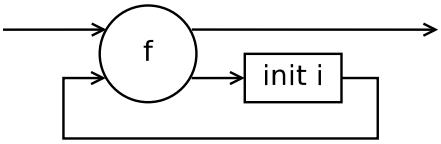


Figure 3.3: Diagram for *loopD*

effectful arrows are dynamically optimized in Nilsson [2005], but they are based on somewhat ad-hoc laws, and there are no normal forms.

Our new strategy is based on the following rather striking observation: *any CCA program can be transformed into a single loop containing at most one pure arrow and one initial state*. More precisely, any CCA program can be normalized into either the form *arr f* or:

$$loop\ (arr\ f \ggg second\ (init\ i))$$

where f is a pure function and i is an initial state. Note that all the *essential* arrow combinators, namely *arr*, $\ggg$, *second*, *loop* and *init*, are used *exactly once*, and therefore all of the interpretive overheads associated with multiple occurrences and compositions of these combinators are eliminated. Not surprisingly, the resulting improvement in performance can be rather dramatic, as we will see later chapters when we discuss the applications of CCA.

We define a combinator called *loopD* that can be viewed as syntactic sugar for the above form:

$$loopD\ i\ f = loop\ (f \ggg second\ (init\ i))$$

A pictorial view of *loopD* is given in Figure 3.3. The second argument to *loopD* is a pure function mapping a tuple of type (b, d) to (c, d), where the value of type d

is initialized before looping back, and is often regarded as an internal state. Such an initialized loopback is a form of *decoupled cycles* [Sculthorpe and Nilsson, 2009], and any immediate loop without going through *init* will be collapsed using the *trace* operator and become part of the pure function f.

3.3.1 Normalization Strategy

Our basic idea behind the normalization of CCA is to extend arrow loop laws to *loopD*, so that we only get the *loopD* form as a result. Formally we define a single step reduction $\mapsto$ for CCA as a set of rules shown in Figure 3.4, and a normalization procedure in Figure 3.5. The normalization relation $\Downarrow$ can be seen as a big step reduction following an innermost strategy, and is indeed a function.

Note that some of the reduction rules resemble the arrow laws of the same name. However, there are some subtle but important differences. First, unlike the laws, reductions are directed. Second, the reduction rules are extended to handle *loopD* instead of *loop*.

To see how this works, it is helpful to visualize a few examples of the reduction rules in Figure 3.4, as shown in Figure 3.6. We omit simple rules that follow directly from the laws, and only show those that involve *loopD*. The diagrams in Figure 3.6 can be explained as follows:

(a) **left tightening**. Figure 3.6(a) shows that we can move a pure arrow from a left composition inside a *loopD* arrow. This follows directly from the left tightening law for *loop*.

(b) **right tightening**. Figure 3.6(b) shows that we can move a pure arrow from a

composition	$arr\ f \ggg arr\ g \mapsto arr\ (g\ .\ f)$
left tightening	$arr\ f \ggg loopD\ i\ g \mapsto loopD\ i\ (g\ .\ (f \times id))$
right tightening	$loopD\ i\ f \ggg arr\ g \mapsto loopD\ i\ ((g \times id)\ .\ f)$
sequencing	$loopD\ i\ f \ggg loopD\ j\ g \mapsto loopD\ (i,j)\ (assoc'\ (juggle'\ (g \times id)\ .\ (f \times id)))$
extension	$first\ (arr\ f) \mapsto arr\ (f \times id)$
superposing	$first\ (loopD\ i\ f) \mapsto loopD\ i\ (juggle'\ (f \times id))$
loop-extension	$loop\ (arr\ f) \mapsto arr\ (trace\ f)$
vanishing	$loop\ (loopD\ i\ f) \mapsto loopD\ i\ (trace\ (juggle'\ f))$

$f \times g\ (x,y) = (f\ x, g\ y)$ $\qquad$ $swap\ (x,y) = (y,x)$

$assoc \quad ((x,y),z) = (x,(y,z))$ $\qquad$ $juggle\ ((x,y),z) = ((x,z),y)$

$assoc^{-1}\ (x,(y,z)) = ((x,y),z)$ $\qquad$ $juggle'\ f = juggle\ .\ f\ .\ juggle$

$assoc'\ f = assoc\ .\ f\ .\ assoc^{-1}$

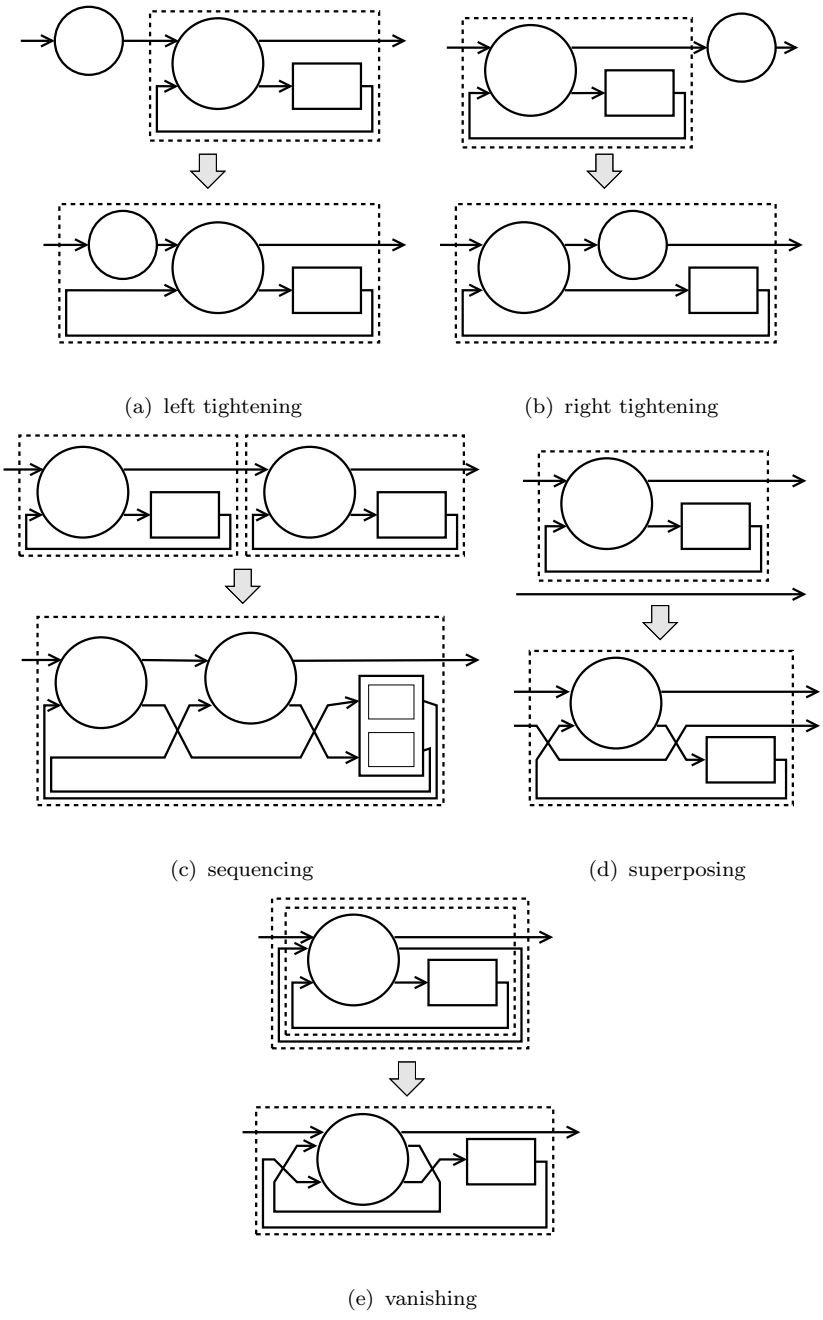
Figure 3.4: Single Step Reduction for CCA

$$\text{(NORM1)}\ \frac{}{arr\ f \Downarrow arr\ f} \qquad \text{(NORM2)}\ \frac{}{loopD\ i\ f \Downarrow loopD\ i\ f}$$

$$\text{(INIT)}\ \frac{}{init\ i \Downarrow loopD\ i\ swap} \qquad \text{(SEQ)}\ \frac{p_1 \Downarrow p_1' \quad p_2 \Downarrow p_2' \quad p_1' \ggg p_2' \mapsto p}{p_1 \ggg p_2 \Downarrow p}$$

$$\text{(FIRST)}\ \frac{f \Downarrow f' \quad first\ f' \mapsto p}{first\ f \Downarrow p} \qquad \text{(LOOP)}\ \frac{f \Downarrow f' \quad loop\ f' \mapsto p}{loop\ f \Downarrow p}$$

Figure 3.5: Normalization of CCA



(a) left tightening

(b) right tightening

(c) sequencing

(d) superposing

(e) vanishing

Figure 3.6: Illustrations of Reduction Rules

right composition inside a *loopD* arrow so that it fuses with the pure function inside the *loopD*. This follows directly from the left tightening law for *loop* and the commutativity law.

(c) **sequencing**. Figure 3.6(c) shows that we can combine two *loopD* arrows into one. In doing so, we have to re-route part of the computation, and make use of the product law to fuse two *init* arrows into one. Notice that there is also a re-ordering of a pure arrow and an *init* arrow, which is due to the commutativity law.

(d) **superposing**. Figure 3.6(d) shows a variant of the superposing law for *loopD* using *first* instead of *second*. Instead of an outer parallel composition, the second line simply passes through the *loopD* unchanged with some re-routing.

(f) **vanishing**. Figure 3.6(e) shows an extension of the vanishing law for *loop* to handle *loopD*. Since the outer loop only acts on the pure function, it can be moved inside and composed with the *trace* operator due to the extension law for *loop*.

3.3.2 Causal Commutative Normal Form

In order to prove that the normalization strategy for the CCA language always produces a normalized arrow term, we first show that the reduction rules are sound with respect to the arrow laws, and the normalization procedure always terminates for all CCA programs.

Lemma 3.3.1 (Soundness) *The reduction rules given in Figure 3.4 are both type*

and semantics preserving, i.e., if $p \mapsto p'$ then $p = p'$ is algebraically derivable from the set of CCA laws.

Proof: By equational reasoning using arrow laws. The **composition**, **extension** and **loop-extension** reduction rules are directly based on the arrow laws with the same name; **left** and **right tightening**, **superposing** and **vanishing** reduction rules follow the definition of *loopD*, the commutativity law, the product law, and the arrow loop laws with the same name. The proof of the **sequencing** rule is more involved, and is given in Appendix B.1. □

Note that the set of reduction rules shown in Figure 3.4 is sound but not complete, because the loop combinator can introduce general recursion at the value level, and hence no equivalence relation of pure functions, and consequently of arrow terms, can be complete.

Lemma 3.3.2 (Termination) *The normalization procedure for CCA given in Figure 3.5 terminates for all CCA programs.*

Proof: By structural induction over all possible definitions of a CCA program. The rules NORM1, NORM2, and INIT are the base case, and SEQ, FIRST, and LOOP cover the inductive case, where the sub arrows (such as p_1 and p_2 in $p_1 \ggg p_2$, and f in *first* f and *loop* f) are normalized inductively. It also explains why there are exactly 8 reduction rules for $\mapsto$, because the premises of SEQ, FIRST and LOOP require 4, 2 and 2 reduction rules respectively. □

Theorem 3.3.1 (CCNF) *For all well typed CCA program $p : A \rightsquigarrow B$, there exists a normal form p_{norm}, called the Causal Commutative Normal Form, which is either*

of the form arr f, or loopD i f for some i and f, such that $p_{norm} : A \rightsquigarrow B$*, and* $p \Downarrow p_{norm}$*. In unsugared form, the second form is equivalent to loop* (*arr f* $\ggg$ *second* (*init i*))*.*

Proof: Follows directly from Lemmas 3.3.1 and 3.3.2. □

3.4 Implementation

So far we have restricted all discussions to the setting of the CCA language (Figure 3.2), but for real implementations, we choose to realize CCA as an embedded language in Haskell for the following reasons:

- By making it an embedded language, we can re-use Haskell's syntax and compiler, and save the effort of re-inventing yet another functional language from scratch.
- We want to extend the normalization process to generic Haskell arrows of the *ArrowInit* class.
- We want the implementation to be *non-intrusive* by making the normalization step optional, so that a valid arrow program is still valid even when the normalizer is turned off.

A major property of the CCA language is that all arrow programs are composed only from the set of CCA combinators, while the same is not true for arrow programs written in Haskell. There are in fact a number of restrictions and challenges of what we can achieve at embedding CCA in Haskell:

- Not all normalization of valid *ArrowInit* instances will terminate, for instance, recursively defined arrows.

- For the benefit of optimal performance and the ease of implementation, we choose to only implement the normalizer statically, i.e., at compile time. This further restricts the set of Haskell arrows that we can normalize to the ones of static structures.

- Even when the final program is an arrow of static structure, programmers would often use auxiliary functions and modular definitions. In order to avoid the complexity of separate compilation, we choose to inline all definitions so as to obtain a single piece of the final arrow program.

3.4.1 Compile-time Translation using Template Haskell

We implement CCA normalizations in Haskell with the help from Template Haskell [Sheard and Peyton Jones, 2002], an extension to Haskell that allows type-safe compile-time meta-programming. Our compilation process consists of three steps:

1. The source arrow program is translated to an abstract syntax tree (AST).

2. The AST is then normalized to CCNF.

3. The result is spliced back into the original program before the Haskell compiler finishes the rest of compilation.

The first step requires inlining of all arrow terms in preparation for the second step that does the actual normalization. So merely grabbing the AST of a single CCA

definition is not enough since it may contain references to other definitions. Our solution here is to allow a generic *ArrowInit* instance to be instantiated as an AST directly from within Haskell, so that when we evaluate such a term, we get a full AST. This is performed by the Haskell compiler at the meta level and can help achieving similar effects of inlining or substitutions. The following code snippet demonstrates this approach:

```
data AExp = Arr ExpQ
          | First AExp
          | AExp :>>> AExp
          | Loop AExp
          | Init ExpQ
          | LoopD ExpQ ExpQ
newtype ASyn b c = AExp AExp
```

The *AExp* data type represents an AST for CCA, and *ASyn b c* employs phantom types so that we can declare *ASyn* to be an instance of the *Arrow*, *ArrowLoop* and *ArrowInit* type classes. The *ExpQ* type used here is the internal syntactic representation of a Haskell expression provided by Template Haskell.

For example, the *Arrow* instance of *Asyn* can be declared as follows:

```
instance Arrow ASyn where
  arr f = error "use arr' instead"
  AExp f >>> AExp g = AExp (f :>>> g)
  first (AExp f) = AExp (First f)
```

As we can see here, the usual arrow combinators are just syntactic operations over the *AExp* data type. The problem, however, is in defining the *arr* combinator. For instance, consider the following program:

$$f :: Arrow\ a \Rightarrow a\ b\ c$$

$$g :: Arrow\ a \Rightarrow a\ b'\ c'$$

$$h :: Arrow\ a \Rightarrow a\ (b, b')\ (c, c')$$

$$h = first\ f \ggg arr\ swap \ggg first\ g \ggg arr\ swap$$

We can obtain the AST for h by instantiating the generic arrow type a to $Asyn$. This step is automatic because any function over type $ASyn\ u\ v$ can be applied to any generic arrow of type $Arrow\ a \Rightarrow a\ u\ v$. Simply evaluating $h :: ASyn\ (b, b')\ (c, c')$ in a Haskell interpreter such as GHCi shall return its AST as something like below:

$$AExp\ (((First\ f' :\ggg Arr\ swap) :\ggg First\ g') :\ggg Arr\ swap)$$

where f' and g' stand for the AST for f and g respectively. The instantiation of f and g is automatic because the concrete arrow types for f and g are inferred to be $ASyn$ too. Another way to look at this is that $AExp\ f' :: ASyn\ b\ c$ and $AExp\ g' :: ASyn\ b'\ c'$ are instances of the generic arrow f and g.

The real issue here, however, is that we cannot automatically reify a Haskell function such as *swap* to the meta level, so the above AST for h has a type error, because *Arr swap* would not type check.

Template Haskell can reify certain expressions to the meta level, and this step is called *quotation*. But it cannot do so for all values, and requires explicit quotation using a special syntax [|...|] for certain things, such as references to global definitions.

To work around this problem, we ask the programmer to always state the quotation explicitly, and disallow direct usage of *arr* as indicated in the arrow instance delcaration for *ASyn*. Instead of *arr*, we provide an arr' function that additionally takes a quoted expression, for example:

$$arr'\ [|\ \lambda x \rightarrow (x, x)\ |]\ (\lambda x \rightarrow (x, x))$$

The above would give us the needed AST for the pure function $\lambda x \rightarrow (x, x)$ in addition to the function itself, where the [|...|] operation is a special Template Haskell syntax for quoting ("quotable") Haskell expression into an *ExpQ* representation. We provide arr' (as well as $init'$, since *init* faces a similar problem) in the *ArrowInit* class defined below:

$$\begin{aligned}
&\textbf{class}\ (Arrow\ a, ArrowLoop\ a) \Rightarrow ArrowInit\ a\ \textbf{where}\\
&\quad init :: b \rightarrow a\ b\ b\\
&\quad arr' :: ExpQ \rightarrow (b \rightarrow c) \rightarrow a\ b\ c\\
&\quad init' :: ExpQ \rightarrow b \rightarrow a\ b\ b\\
&\quad loopD :: e \rightarrow ((b, e) \rightarrow (c, e)) \rightarrow a\ b\ c
\end{aligned}$$

Since asking the programmer to write in arr' gets tedious over time, we provide a modified arrow syntax translator that directly outputs combinator programs written in arr' and $init'$ instead of *arr* and *init*.

The actual normalization of an *AExp* can be straightforwardly implemented as a traversal using the algorithms given in Figures 3.4 and 3.5. We omit the details here by just saying the CCNF of an *AExp* is eventually represented by either the *Arr* or the *LoopD* constructors.

We provide the normalization function as a Template Haskell splice operation:

$$norm :: ArrowInit\ a \Rightarrow a\ b\ c \rightarrow ExpQ$$

So if we evaluate $\$(norm\ e)$ for any generic arrow $e :: ArrowInit\ a \Rightarrow a\ b\ c$, what happens in the background is that it will first instantiate e to an *ASyn* arrow which is internally represented by the *AExp* data type, then normalized to a *LoopD* (or *Arr*) form, and finally spliced back as an Haskell expression $loopD\ i\ f$ (or $arr\ f$) for some i and f. The \$(...) operation is a special Template Haskell syntax for splicing. Since GHC has native support for Template Haskell, the entire process happens without any user intervention during either the compilation of a Haskell source program using GHC, or an intepretive session using the interactive Haskell evaluator *GHCi*.

3.4.2 Technical Limiations

The use of Template Haskell allows a very simple implementation of the meta operations required for CCA normalization, and its integration with GHC gives a seamless user experience. The only caveat to this approach, however, has to do with a limitation of Template Haskell. For example, if we were to define a constant arrow like this:

$$constant :: ArrowInit\ a \Rightarrow c \rightarrow a\ b\ c$$
$$constant\ x = \mathbf{proc}\ \_ \rightarrow returnA \prec x$$

and when we try to instantiate the above arrow type a to *ASyn*, the compiler will complain that the type for *constant* is wrong, and insist that type c must be a member of the *Lift* class, which is how Template Haskell reifies a Haskell value to the meta

level. To see exactly where is the problem, we translate the above from arrow syntax to combinators using arr':

$$constant\ x = arr'\ [|\ \lambda\_ \rightarrow x\ |]\ (\lambda\_ \rightarrow x)$$

The quotation of the lambda expression $\lambda\_ \rightarrow x$ above has a reference to x, a parameter of the function *constant*, and Template Haskell cannot quote it unless it knows how to reify the value of x to the meta level. and hence requires that the type for x is an instance of the *Lift* class.

On the other hand, our goal in the first step of inlining CCA defitions is not to lift arbitrary Haskell values, but to generate top-level code that could somehow still relate the occurrence of variable x in a quotation to the actual parameter of *constant* symbolically, so that we know how to handle substitution without evaluating the actual value of x. In order to do so in Template Haskell would require quoting the entire definition of *constant* and restricting variable x to be of the *ExpQ* type, the type for quoted expressions. This will prevent us from re-using the same unmodified code without CCA normalization, and hence compromise one of our initial goals of making the implementation non-intrusive.

A more effective solution is to perform inlinings and substitutions at the meta level, e.g., using a heavily customized arrow syntax preprocessor, which will no longer rely on the evaluation of Haskell terms, and hence bypass the whole reification issue. Unfortunately this is beyond what Template Haskell does, and we leave it to future work.

3.4.3 Direct Interpretation

The CCA language presented in Section 3.2 has a number restrictions, of which the most significant one is that lambdas are not allowed at the program level, which implies two things:

1. All arrow programs must be first-order.

2. Arrow programs can not mix with lambda expressions even when they only contain variables at value level.

The second restriction is far stronger than the first one. In our Template Haskell based implementation we actually relax on these two restrictions by allowing helper functions to be used in defining the main arrow program, so long as we can extract its full AST representation at compile-time.

We must stress that this restriction on the CCA language is only a simplification for the sake of clarity in discussing the normalization procedure, and entirely artificial. In other words, not all CCA can be expressed in our CCA language. It is then worth asking the question whether the normalization procedure applies to all CCAs.

In order to answer this question, and as an alternative to compile-time transformations, we can express the CCA normalization procedure as an interpretation of arrow programs in Haskell. This is actually a more direct approach that puts no restriction on the source arrow program.

Figure 3.7 shows the implementation of such an interpretation. We first introduce a generalized algebraic data type (GADT) called *CCNF*, whose two constructors, *Arr* and *LoopD* represent the two cases for CCNF. We use GADT here because the type e in the *LoopD* case is hidden in the signature of the $CCNF\ b\ c$ type, and

```
data CCNF b c where
  Arr     :: (b → c) → CCNF b c
  LoopD :: e → ((b, e) → (c, e)) → CCNF b c

instance Arrow CCNF where
  arr = Arr
  (Arr f)       ⋙ (Arr g)       = Arr (g . f)
  (Arr f)       ⋙ (LoopD i g) = LoopD i (g . (f × id))
  (LoopD i f) ⋙ (Arr g)       = LoopD i ((g × id) . f)
  (LoopD i f) ⋙ (LoopD j g) = LoopD (i, j) (assoc' (juggle' (g × id) . (f × id)))
  first (Arr f)       = Arr (f × id)
  first (LoopD i f) = LoopD i (juggle' (f × id))

instance ArrowLoop CCNF where
  loop (Arr f)       = Arr (trace f)
  loop (LoopD i f) = LoopD i (trace (juggle' f))

instance ArrowInit CCNF where
  init i = LoopD i swap
```

Figure 3.7: *CCNF* as an Instance of *ArrowInit*

an alternative choice here would be to use existential types. Then we declare that *CCNF* is an instance of *Arrow*, *ArrowLoop*, and *ArrowInit*. The instance functions are almost line by line translation of the reduction and normalization rules shown in Figure 3.4 and Figure 3.5.

Our program indeed demonstrates that all generic *ArrowInit* arrows can be instantiated to the *CCNF* data type, which captures the CCA normal form by construction. A programmer is free to use any Haskell functions, including lambdas and general recursion the arrow level, to construct a generic *ArrowInit* arrow and hopefully obtain its normal form this way, that is, subject to the termination of evaluating a term of *CCNF* type. In other words, Lemma 3.3.2 no longer holds, and the normal form could be bottom. So long as we accept this relaxation on termination, we can apply CCA normalization to all generic CCAs that are intances of *ArrowInit*.

An important difference between this approach and the Template Haskell based implementation is that the actual construction of CCNF is now at run-time rather than compile-time. Therefore, we cannot rely on GHC to take the pure function and state captured in a CCNF and produce optimized code at compile-time. Thus the overhead of interpreting arrow combinators still remains, and the main benefit of CCA normalization, namely performance improvements over conventional arrow implementations, is much weakened.

Our point here, however, is that our design of an abstract language for CCA as shown in Figure 3.2 is purposely simplified, and shall not diminish the interest in CCA and its normalization. Our staged compilation for CCA based on Template Haskell is merely a first step towards making good use of CCA normalization techniques despite its limitations. Developing a comprehensive staged and modular compiler for CCA is

certainly our long term goal, and we leave it to furture work.

3.5 Extensions

Normalization is a strong property, and the CCA language is of limited scope. As discussed in last section, only a subset of Haskell arrows of the *ArrowInit* class can be normalized at compile time. But on the other hand, we can still extend the CCA language while retaining the normalization property, and admit more programs to this family.

3.5.1 ArrowChoice

Many dataflow and stream programming languages provide conditionals, such as **if-then-else**, as part of the language [Wadge and Ashcroft, 1985, Caspi et al., 1987]. In Haskell, conditionals at the arrow level are captured by the *ArrowChoice* class together with a set of the arrow choice laws shown in Figure 3.8. In the case of the CCA language, we need an extension of the sum type in order support the *left* combinator for *ArrowChoice*. Modifications to the CCA language syntax and types are given in Figure 3.9.

We extend the types to include a sum type $A + B$, and the base expressions with the left (ι_l) and right (ι_r) injections for sum, as well as a $\oplus$ operator that consumes a sum type with either a function applying to the left branch, or another to the right. The $\oplus$ operator is an equivalent definition of the commonly used case operator for sum types. At the program level, we introduce a new *left* combinator. The typing rules for the CCA language are also suitably extended as shown in Figure 3.9.

class $Arrow\ a \Rightarrow ArrowChoice\ a$ **where**

$left :: a\ b\ c \rightarrow a\ (Either\ b\ d)\ (Either\ c\ d)$

extension	$left\ (arr\ f) = arr\ (f \oplus id)$
functor	$left\ (f \ggg g) = left\ f \ggg left\ g$
exchange	$left\ f \ggg arr\ (id \oplus g) = arr\ (id \oplus g) \ggg left\ f$
unit	$arr\ left \ggg left\ f = f \ggg arr\ left$
association	$left\ (left\ f) \ggg arr\ assocsum = arr\ assocsum \ggg left\ f$

$$f\ (\oplus)\ g\ (Left\ x) = Left\ (f\ x) \qquad assocsum\ (Left\ (Left\ x)) = Left\ x$$
$$f\ (\oplus)\ g\ (Right\ y) = Right\ (g\ y) \qquad assocsum\ (Left\ (Right\ x)) = Right\ (Left\ x)$$
$$assocsum\ (Right\ x) = Right\ (Right\ x)$$

Figure 3.8: *ArrowChoice* Class and Its Laws

Syntax Addition

$$\begin{array}{lrcl} \text{Types} & A, B, C & ::= & \ldots \mid A + B \\ \text{Expressions} & M, N & ::= & \ldots \mid \iota_l\ M \mid \iota_r\ M \mid M \oplus N \\ \text{Programs} & P, Q & ::= & \ldots \mid \mathit{left}\ P \end{array}$$

Typing Rules Addition

$$\text{(CASE)}\ \frac{\Gamma \vdash M : A \to C \quad \Gamma \vdash N : B \to C}{\Gamma \vdash M \oplus N : A + B \to C}$$

$$\text{(LEFT)}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \iota_l\ M : A + B} \qquad \text{(RIGHT)}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash \iota_r\ M : A + B}$$

$$\text{(LEFTARROW)}\ \frac{\vdash P : A \rightsquigarrow B}{\vdash \mathit{left}\ P : A + C \rightsquigarrow B + C}$$

Figure 3.9: CCA Language Extension for *ArrowChoice*

To complete the normalization, we further extend the CCA reduction rules in Figure 3.4 to handle *ArrowChoice* as follows:

extension $\quad left\ (arr\ f) \mapsto arr\ (f \oplus id)$

superposition $\quad left\ (loopD\ i\ f) \mapsto loopD\ i\ (tag^{-1}\ .\ (f \oplus id)\ .\ tag)$

where the function tag and tag^{-1} are defined below:

$$\begin{aligned} fmap\ f &= (\iota_l\ .\ f) \oplus (\iota_r\ .\ f) \\ tag\ (z, y) &= fmap\ (\lambda x \rightarrow (x, y))\ z \\ tag^{-1}\ x &= (fmap\ fst\ x, (snd \oplus snd)\ x) \end{aligned}$$

The soundness of the above extension to the reduction rules can be easily proved with respect to the arrow choice laws shown in Figure 3.8, and we omit the details here.

We also need a new inference rule for the normalization procedure shown in Figure 3.5, which is given below:

$$\text{(LEFT)} \quad \frac{f \Downarrow f' \quad left\ f' \mapsto p}{left\ f \Downarrow p}$$

Similarly, termination can be proved for the above extension. It can also be easily shown that the above extensions requires no modification to CCNF, and Theorem 3.3.1 still holds. In other words, CCA extended with *ArrowChoice* can still be normalized to the same normal form as in the original CCA.

3.5.2 Multi-sort *init*s

As shown in earlier sections, the *init* arrow and its product law are essential to CCA and its normalization. However, it is sometimes useful to consider more than one kind of init, and each of them would obey a product law of their own.

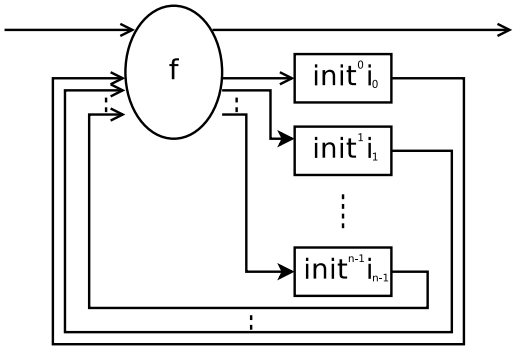


Figure 3.10: CCNF for $\text{CCA}_n^\dagger$, an Extension of CCA with Multi-sort *init*s

We define an extension to CCA that we call $\text{CCA}_n^\dagger$, such that instead of just one *init* combinator, we have n combinators $init^0, init^1, \ldots, init^{n-1}$, and instead of a single product law, we have $init^k\ i \star\!\star\!\star\ init^k\ j = init^k\ (i, j)$ holds true for all integers $k \in [0, n)$. Our intuition tells us that the normal form of $\text{CCA}_n^\dagger$ would become something like the one illustrated in Figure 3.10.

Before we sketch out the complete proof, we first need some custom notations. Let us make a shorthand by denoting a nested tuple $(i_0, (i_1, (\ldots, i_{n-1})\ldots))$ that has at least one element as a *sequence*: $I = i_0, i_1, \ldots, i_{n-1}$. Then we define a *bracketed sequence*: $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ where each $x_k \in X$ is either *missing* (denoted by $x_k = \diamond$), or *present* with a value (denoted by $x_k = \underline{i_k}$ for some i_k). The injection function ι_n^k injects into a bracketed sequence X of size n a single value i at position k ($x_k = \underline{i}$, and $x_j = \diamond$ for all $0 \leq j < n$ and $j \neq k$). We also define a *point-wise product* function $\star$ for bracketed sequences as follows:

$$\langle\ x_0, x_1, \ldots, x_{n-1}\ \rangle\ \star\ \langle\ y_0, y_1, \ldots, y_{n-1}\ \rangle = \langle\ x_0\ *\ y_0, x_1\ *\ y_1, \ldots, x_{n-1}\ *\ y_{n-1}\ \rangle$$

where $\diamond\ *\ \diamond = \diamond$

$$\underline{i}\ \ *\ \diamond = \underline{i}$$

$$\diamond \ * \ \underline{j} = \underline{j}$$

$$\underline{i} \ * \ \underline{j} = \underline{(i,j)}$$

Further more, we denote the *packing* of all present values in a bracketed sequence X as an sequence $\lfloor X \rfloor = \lfloor x_{k_0}, x_{k_1}, \ldots, x_{k_{m-1}} \rfloor = i_{k_0}, i_{k_1}, \ldots, i_{k_{m-1}}$ for all $x_{k_j} = \underline{i_{k_j}}$, where $k_p < k_q$ for all $0 \leq p < q < m$, and m is the number of present values in X. Conversely, we denote the *unpacking* of a sequence $I = i_0, i_1, \ldots, i_{m-1}$ into an X-shaped bracketed sequence $Y = \lceil I \rceil^X$, where $y_k = \diamond$ iff $x_k = \diamond$, and $\lfloor Y \rfloor = I$. Finally we define a syntactic sugar $init^\dagger$ for multi-sort inits as follows (assuming $\star\!\!\star\!\!\star$ is left associative):

$$init^\dagger X = init^{k_0} \ i_0 \ \star\!\!\star\!\!\star \ init^{k_1} \ i_1 \ \star\!\!\star\!\!\star \ \ldots \ \star\!\!\star\!\!\star \ init^{k_{m-1}} \ i_{m-1}$$

where $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle$, and $\lfloor X \rfloor = \lfloor x_{k_0}, x_{k_1}, \ldots, x_{k_{m-1}} \rfloor = i_0, i_1, \ldots, i_{m-1}$. It is easy to see that $init^k \ i = init^\dagger(\iota_n^k \ i)$.

Now with the help from the above notations, we construct a *generalized product law* as a theorem for $\mathrm{CCA}_n^\dagger$, and we prove that it is sound.

Theorem 3.5.1 (Generalized Product) *The following law holds for all $CCA_n^\dagger$, with I and J being some bracked sequences of size n:*

$$init^\dagger I \ \star\!\!\star\!\!\star \ init^\dagger J = arr \ p \ggg init^\dagger \ (I \star J) \ggg arr \ p^{-1}$$

$$where \ p \ (x, y) = \lfloor \ \lceil x \rceil^I \star \lceil y \rceil^J \rfloor$$

Proof: The above can be proved by induction over the size of I and J. The base case degenerates to the ordinary product law when both I and J are single valued sequences and p becomes an identity function. The inductive case has two sub-cases which are symmetric, because now either I or J can have one extra value. Without

loss of generality, suppose I has an extra value to the front, and this value can be either missing or present. In both cases the proof is straight forward, and follows directly from the expansion of definitions. □

With the generalized product law, and by converting all *init*s to $init^\dagger$, we can normalize any arrow in $\mathrm{CCA}_n^\dagger$ following a similar strategy to the original CCA. We define a combinator called $loopD^\dagger$ that can be seen as a syntactic sugar:

$$loopD^\dagger\ I\ f = loop\ (arr\ f \ggg second\ (init^\dagger\ I))$$

where I is a bracketed sequence of size n.

The single step reduction $\mapsto^\dagger$ for $\mathrm{CCA}_n^\dagger$ is basically the same as $\mapsto$ for CCA (Figure 3.4) except that *loopD* is now replaced by $loopD^\dagger$, and the **sequencing** rule becomes:

$$loopD\ I\ f \ggg loopD\ J\ g \mapsto^\dagger loopD\ (I \star J)\ (route\ (juggle'\ (g \times id)\ .\ (f \times id)))$$

$$\textbf{where}\ route\ f\ = id \times p^{-1}\ .\ assoc'\ f\ .\ id \times p$$

$$p\ (x, y) = \lfloor\ \lceil x \rceil^I \star \lceil y \rceil^J \rfloor$$

The soundness of $\mapsto^\dagger$ can be easily proved following the proof for $\mapsto$, and we omit them here.

The normalization procedure $\Downarrow^\dagger$ for $\mathrm{CCA}_n^\dagger$ is also similar to that of $\Downarrow$ for CCA (Figure 3.5) with the INIT rule replaced by a new rule for $init^\dagger$:

$$(\mathrm{INIT}_\mathrm{n}^\dagger)\quad \frac{}{init^k\ i \Downarrow^\dagger loopD^\dagger\ (\iota_n^k\ i)\ swap}$$

Similarly we can prove that $\Downarrow^\dagger$ terminates for all arrows in $\mathrm{CCA}_n^\dagger$. Details are omitted since it is similar to that of $\Downarrow$.

Theorem 3.5.2 (CCNF of $\mathbf{CCA}_n^\dagger$) *For all well typed $CCA_n^\dagger$ program $p :: A \rightsquigarrow B$, there exists a normal form p_{norm}, which is either of the form arr f, or loop$D^\dagger$ I f for some bracketed sequence I of size n and a pure function f, such that $p_{norm} :: A \rightsquigarrow^\dagger B$, and $p \Downarrow^\dagger p_{norm}$. In unsugared form, the second form is equivalent to loop $(arr\ f \ggg second\ (init^\dagger\ I))$.*

Proof: follows directly from the soundness of $\mapsto^\dagger$ and the termination of $\Downarrow^\dagger$.

3.6 Discussion

Apart from arrows, other formalisms such as monads, comonads and applicative functors have been used to model computations over data streams [Bjesse et al., 1998, Uustalu and Vene, 2005, McBride and Paterson, 2008]. Central to many of these approaches are the representation of streams and computations about them. However, notably missing are the connections between stream computations and their related laws. For example, Uustalu and Vene [2005] concluded that comonad is a suitable model for dataflow computation, but did not make the connection to comonadic laws.

In contrast, it is the very idea of making sense out of arrow and arrow loop laws that motivated our work. We argue that arrows are a suitable abstract model for stream computation not only because we can implement stream functions as arrows, but also because abstract properties like the arrow laws help bring more insights to our target application domain.

Besides having to satisfy respective laws for these formalisms, each abstraction has to introduce domain specific operators, otherwise it would be too general to be useful. With respect to causal streams, many have introduced *init* (also known as

delay) as a primitive to enable stateful computation, but few seem to have made the connection of its properties to program optimization.

Recently, Lindley et al. [2010] give a more explicit explanation of the arrow laws by constructing an arrow calculus and turning the nine arrow laws into five laws for the calculus, and discover a redundancy in the original nine arrow laws. Unfortunately, arrow loop is not included in their formulation.

The *loop* combinator and its arrow loop laws play a key role in CCA normalization because multiple loops can be fused together, and nested loops can be collapsed into just one. This is actually very close to yet another instantiation of the Folk Theorem [Harel, 1980] that all computer programs can be simulated by a single while-loop, if not for the fact that arrows and arrow loop only model a specific subset of but not all computations. We are interested in both the generality and the discipline brought forward by the laws.

CCA originates from an attempt to capture the essence of computations over causal stream. However, by making it more abstract, and specifically, by not giving the CCA language a definitive denotational semantics, we have deliberately made a generalization and perhaps broadened its scope.

Also, when we consider possible laws for the *init* combinator, the following equation makes a lot of sense in the context of dataflow if we assign the meaning of a unit delay to *init*:

$$init\ i \ggg arr\ f = arr\ f \ggg init\ (f\ i)$$

As we shall later see in Chapter 5, when we apply CCA to areas outside of dataflow, the above no longer holds true, but the product law still does. Therefore, careful

generalization of selected properties is a key criteria in determining the usefulness and applicability of CCA.

But first and foremost, we have established CCA as an abstract model for causal stream computations. In this area, the co-algebraic property of streams is well known, and most relevant to our work is a functional representation of stream and stream functions by Caspi and Pouzet [1998]. They also use a primitive similar to the trace operator (and hence the arrow loop combinator) to model recursion. Their compilation technique, however, lacks a systematic approach to optimize nested recursions. We consider our technique more effective and more abstract.

Also relevant is the work of Rutten [2006] on higher-order functional stream derivatives. We believe that arrows are a more general abstraction than functional stream derivatives, because the latter still exposes the structure of a stream. Moreover, arrows give rise to a high-level language with richer algebraic properties than the 2-adic calculus considered in Rutten [2006].

Chapter 4

Application: Synchronous Dataflow

4.1 Dataflow Languages

One of the notable applications of arrows is in the area of dataflow languages, and in particular, synchronous dataflow languages that include Yampa. In fact, CCA was initially designed to model applications in this domain, where *init* would correspond to the domain specific *delay* primitive. In this chapter we first give an overview of dataflow languages, and then we look at a popular design of an arrow based DSL for synchronous dataflow and its embedded implementation in Haskell. Finally we illustrate that the normalization of CCA serves as a practical and effective normalization technique that brings drastic performance improvements.

4.1.1 Overview

The dataflow programming model represents programs as directed graphs where the nodes represent instructions, and the data flows between the nodes along the directed

arcs [Arvind and Culler, 1986, Davis and Keller, 1982]. The original motivation for dataflow was the exploitation for parallelism because instructions can be executed as soon as their operands becomes available. Many developments have taken place in designing both the hardware architecture and software systems for dataflow. Active research has also gone into designing programming languages for dataflow, such as Signal [Gautier et al., 1987], Lustre [Caspi et al., 1987, Halbwachs et al., 1991a], Esterel [Berry and Cosserat, 1985], Lucid [Wadge and Ashcroft, 1985], and so on.

Very often, dataflow programming languages are thought of as a specific type of functional languages because in a pure dataflow model, the computation at each node is entirely local and there is no global data store. Indeed, many of the FRP languages, including Yampa, can be seen as variants of dataflow languages.

4.1.2 Synchronous Dataflow

One particular model that became widely used for programming reactive systems is the synchronous paradigm [Halbwachs, 1992, Lee and Messerschmitt, 1987], where the behavior of a program is a sequence of reactions, and each reaction is considered as an *atomic* cycle of reading current inputs, producing current outputs, and updating internal states [Halbwachs et al., 1991a]. These atomic cycles are perceived as *taking no time* (logically), and hence all events occurring during a single cycle are considered *simultaneous*.

The synchronous paradigm is of course just a hypothetical ideal, and in practice, the assumption is that the program is able to react to an external event before any further event occurs, and hence the ideal behavior becomes a sensible abstraction.

4.2 Stream

Synchronous dataflow languages such as Esterel, Signal, and Synchronous Lucid, provide high-level and modular ways to program reactive systems following both the synchronous paradigm and the dataflow model, in which expressions denote infinite streams (also known as sequences or flows) of values, and common operations on basic types are lifted to the stream level point-wise.

The individual values in a stream are usually indexed by a clock, which is basically a sequence of natural numbers if we only consider discrete streams. A common practice is to index the n-th value in a stream of the *base clock* at the n-th cycle during the program execution. In synchronous languages, the notion of clock represents a *logical time* that is decoupled from the physical time it actually takes to perform the computation steps.

Conceptually if we represent a stream of values as an abstract data type $S\ a$, it can be viewed as a function over the clock:

$$S\ a \approx Clock \rightarrow a$$

Many dataflow language also consider something called *multi-clockrate*, where different streams are associated with different clocks. Streams of different clocks can be simulated by *partial streams*, which is a stream that may have empty positions to indicate the pace of its clock with respect to the base clock. For simplicity, if we assume that all streams are discrete and only of the base clock, we can give a concrete representation of our stream type $S\ a$ as follows:

$$\textbf{newtype}\ S\ a = S\ \{\, hd :: a, tl :: S\ a \,\}$$

In other words, a stream of type a is represented by its head (a value of type a), and its tail that is also a stream. This is a very common representation of streams in Haskell.

As an example, we show in Figure 4.1 a Haskell implementation of a DSL for synchronous dataflow. It makes use of various Haskell type classes to define a number of primitives, and we elaborate on the details below:

- We make use of the *Functor* and *Applicative* classes to implement the lifting of values (as well as functions) onto the stream level, where
  - *pure* x creates a stream of constant value x;
  - f $<\star>$ x applies a stream of functions (represented by f) to a stream of values (represented by x) pointwise;
  - *fmap* f lifts a pure function to the stream level so that it can be applied to each element of an input stream. Also, the *applicative law* for the *Applicative* class states that *fmap* f x = *pure* f $<\star>$ x.
  - In addition, we define a set of lifting functions, *lift*, *lift2* and *lift3*, and they lift functions that take one, two, or three arguments to the stream level.
- We make use of the various type classes for basic Haskell types to overload their operators to the stream level. For example, with the *Eq* instance for the data type S, we can directly write $x \equiv y$ to compare the equality of two streams x and y. Similarly, common arithmetic operators can directly operate on streams too.

```
instance Functor S where
  fmap f (S x xs) = S (f x) (fmap f xs)
instance Applicative S where
  pure x = S x (pure x)
  (S f fs) <*> (S x xs) = S (f x) (fs <*> xs)
i `fby` x       = S i x
lift  f x       = pure f <*> x                  -- or lift = fmap
lift2 f x y     = pure f <*> x <*> y            -- or lift2 f x y = lift f x <*> y
lift3 f x y z = pure f <*> x <*> y <*> z        -- or lift3 f x y z = lift2 f x y <*> z
instance Eq a => Eq (S a) where
  (==) = lift2 (==)
instance Ord a => Ord (S a) where
  (<=) = lift2 (<=)
instance Num a => Num (S a) where
  (+) = lift2 (+)
  (*) = lift2 (*)
  negate = lift negate
  abs    = lift abs
  signum = lift signum
instance Fractional a => Fractional (S a) where
  (/) = lift2 (/)
  fromRational = lift romRational
```

Figure 4.1: A Stream Based Dataflow DSL

- Following the convention of Lucid, we define a '*fby*' (followed-by) infix operator that delays an input stream by one clock cycle and sets its initial value. It is the primary operator for stateful computation in dataflow languages. A different but equivalent design choice is to separate the delay and initialization into two operators, e.g., Lustre uses *pre* for delay, and $\rightarrow$ for initialization.

With the simple DSL primitives defined in Figure 4.1, we can already make interesting dataflow programs, and we give some examples below.

$$
\begin{aligned}
&\mathit{ones} = \mathit{constant}\ 1 \\
&\mathit{sum}\ x = x + 0\ `\mathit{fby}`\ \mathit{sum}\ x \\
&\mathit{nats} = \mathit{sum}\ \mathit{ones} \\
&\mathit{fibs} = \mathbf{let}\ f = 0\ `\mathit{fby}`\ g \\
&\qquad\qquad\quad g = 1\ `\mathit{fby}`\ (f + g) \\
&\qquad\ \mathbf{in}\ f
\end{aligned}
$$

The stream *ones* is a constant stream of 1s, and *sum* is a function that maps from a stream of numbers to a stream of its pointwise sumation, and when applied to *ones*, we obtain a stream of natural numbers *nats*. The stream *fibs* represents the Fibonacci sequence inductively defined by the set of mathematical equations $f_0 = 0, f_1 = 1, f_{n+2} = f_{n+1} + f_n$.

Notice that in this DSL, all primitives are also Haskell functions, and we can directly write dataflow programs using many other Haskell features such as lambda abstraction, let expression, overloaded operators, etc. It not only saves us effort of having to build and parse a new language syntax, but also enables a much richer language design by re-using Haskell native functions. This kind of DSL embedding is

also known as *shallow embedding*.

A dataflow program written in our DSL is just a definition of a stream. One of the many ways to evaluate such a program is to look up its stream value at n-th position:

$$run_s :: Int \to S\ a \to a$$
$$run_s\ n\ (S\ v\ x) = \textbf{if}\ n \equiv 0\ \textbf{then}\ v\ \textbf{else}\ run_s\ (n-1)\ x$$

The handful primitives provided in Figure 4.1 can already make up a fairly rich set of dataflow programs, but there are still a few things we cannot express in this DSL:

- The current value of a stream can only depend on past values (by using '*fby*'), but not the future. This is a property of causality we can ensure by limiting the choice of primitives.

- Moreover, the current value of a stream can not depend on stream values in an arbitrary past since '*fby*' only provides an unit delay, not a delay of arbitrary length. This is usually one of the properties that can help ensure that the space requirement for each computation cycle are bounded so that the overall synchrony hypothesis can be met for real-time reactive systems.

One caveat of shallow embedding in Haskell, however, is that not all properties of the DSL can be safe-guarded in the implementation. For example, even though 'fby' of arbitrary length is not provided, we can still accumulate all historical values in an unbounded Haskell list. So unless one writes programs only using primitive types (whose size are bounded) and with no recursion, we cannot fulfill the bounded space requirement given above.

4.3 Stream Transformer

As we might have noticed, all the primitive operators for our stream based dataflow DSL (Figure 4.1) map an input stream to an output stream. In other words, they are *stream transformers* (or *stream functions*). Just like we can capture streams with an abstract data type $S\ a$, we can also capture stream transformers as follows:

$$SF\ a\ b\ \approx S\ a \rightarrow S\ b$$

Not incidentally, stream transformers are arrows, or to be more specific, CCAs. Instead of manipulating first-class stream objects, we now compose stream transformers using arrow combinators:

- *arr* f produces an output stream by applying the pure function f to each element in the input stream.
- $f \ggg g$ composes two stream functions together, passing the input stream through f and then through g.
- *first* f accepts a stream of tuples (that is isomorphic to a tuple of streams), and passes the first stream through f without changing the second one.
- *loop* f creates a form of loopback around the stream function f by connecting the second part of f's output stream back to the second part of f's input, and thus enables recursive dataflow.
- *init* i delays its input stream by one cycle, and sets i as the initial value of its output stream. Just like '*fby*', *init* is the primary operator that introduces causal and stateful computation to an arrow based dataflow DSL.

The same kind of program that previously was written using the dataflow DSL can now be written in an arrow form. The following implements the same examples we saw, but now written as arrows in the arrow syntax:

$$
\begin{aligned}
&\mathit{constant}\ x = \mathit{arr}\ (\lambda\_ \rightarrow x)\\
&\mathit{ones} = \mathit{constant}\ 1\\
&\mathit{sum} = \mathbf{proc}\ x \rightarrow \mathbf{do}\\
&\quad \mathbf{rec}\ s \leftarrow \mathit{init}\ 0 \prec s + x\\
&\quad \mathit{returnA} \prec s\\
&\mathit{nats} = \mathit{ones} \ggg \mathit{sum}\\
&\mathit{fibs} = \mathbf{proc}\ \_ \rightarrow \mathbf{do}\\
&\quad \mathbf{rec}\ f \leftarrow \mathit{init}\ 0 \prec g\\
&\qquad\quad g \leftarrow \mathit{init}\ 1 \prec (f + g)\\
&\quad \mathit{returnA} \prec f
\end{aligned}
$$

As can be seen above, the use of arrow syntax actually helps to visualize the dataflow through our arrows. There are a few important differences between a stream based program and an arrow based one:

- There is no standalone streams in an arrow based program, and even *constant* must be represented as a stream function mapping from *any* stream to a constant stream.

- Since there is no stream, we no longer have to overload arithmetic operators such as those provided by the *Num* and *Fractional* classes. For example, in the definition of *sum*, we simply write $s + x$ where the $+$ is a plain math operator.

- We no longer need recursive definition for streams or stream functions in the arrow program, and instead we use the *loop* operator to represent recursions in a dataflow at the value level (rather than the arrow level). Notice that the use of **rec** keyword in the arrow syntax would translate to *loop* after desugaring. This is actually a very important feature because we now have the freedom to traverse and transform a composite arrow without worrying about running into recursive definitions. Later in Chapter 5 and Chapter 6 we will see the real advantage of eliminating space leaks using the arrow loop form in stead of the general recursion.

We give a sample implementation of a dataflow DSL that captures a causal stream transformer using arrows, as shown in Figure 4.2. A few details are explained as follows:

- $SF\ a\ b$ is an arrow representing transformers from streams of type a to streams of type b. It is essentially a recursively defined data type consisting of a function with its continuation, a concept closely related to a form of finite state automaton called a *Mealy Machine* [Mealy, 1955]. The same data type was called *Auto* in [Paterson, 2001].

- SF is declared an instance of type classes *Arrow*, *ArrowLoop* and *ArrowInit*. These instances obey all of the arrow laws, including the two additional laws for CCA.

- $run_{sf} :: SF\ a\ b \rightarrow S\ a \rightarrow S\ a$ converts an SF arrow into a plain function that maps an input stream to an output stream.

$$\textbf{newtype}\ SF\ a\ b = SF\ (a \to (b, SF\ a\ b))$$

$$\textbf{instance}\ Arrow\ SF\ \textbf{where}$$

$$\begin{array}{lll}
arr\ f = SF\ h & \textbf{where}\ h\ x & = (f\ x, SF\ h) \\
first\ (SF\ f) = SF\ (h\ f) & \textbf{where}\ h\ f\ (x, z) & = \textbf{let}\ (y, SF\ f') = f\ x \\
 & & \quad \textbf{in}\ ((y, z), SF\ (h\ f')) \\
SF\ f \ggg SF\ g = SF\ (h\ f\ g) & \textbf{where}\ h\ f\ g\ x & = \textbf{let}\ (y, SF\ f') = f\ x \\
 & & \quad\ \ (z, SF\ g') = g\ y \\
 & & \quad \textbf{in}\ (z, SF\ (h\ f'\ g'))
\end{array}$$

$$\textbf{instance}\ ArrowLoop\ SF\ \textbf{where}$$

$$\begin{array}{lll}
loop\ (SF\ f) = SF\ (h\ f) & \textbf{where}\ h\ f\ x & = \textbf{let}\ ((y, z), SF\ f') = f\ (x, z) \\
 & & \quad \textbf{in}\ (y, SF\ (h\ f'))
\end{array}$$

$$\textbf{instance}\ ArrowInit\ SF\ \textbf{where}$$

$$\begin{array}{lll}
init\ i = SF\ (h\ i) & \textbf{where}\ h\ i\ x & = (i, SF\ (h\ x))
\end{array}$$

$$run_{sf} :: SF\ a\ b \to S\ a \to S\ b$$

$$\begin{array}{lll}
run_{sf}\ (SF\ f) = g\ f & \textbf{where}\ g\ f\ (S\ x\ xs) & = \textbf{let}\ (y, SF\ f') = f\ x \\
 & & \quad \textbf{in}\ S\ y\ (g\ f'\ xs)
\end{array}$$

Figure 4.2: A Stream Transformer Based Dataflow DSL

We must stress that the *SF* type is but one example CCA instance for dataflow programming, and alternative implementations such as the synchronous circuit type *SeqMap* in [Paterson, 2001] and the stream function type (incidentally also called) *SF*) in [Hughes, 2004] also qualify as valid instances of CCA. The abstract properties of CCA such as normal forms are applicable to any of these instances, and thus illustrates that CCA is more broadly applicable than optimization techniques based on a specific semantic model, such as the one considered in [Caspi and Pouzet, 1998].

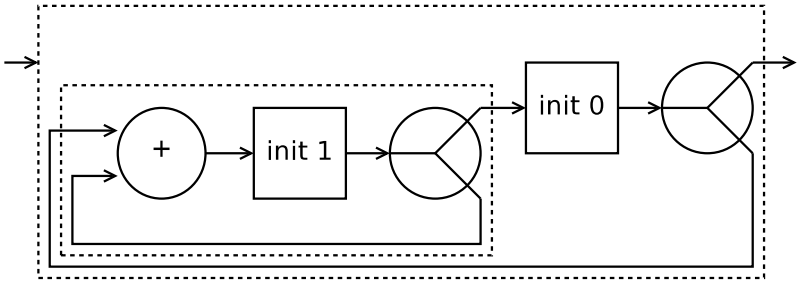
4.4 Optimization

As discussed in Chapter 3, the usual arrow instance declarations come with interpretive overheads associated with the arrow data type and combinators, and one of the key properties of defining something as a CCA is to have a normal form that is free of such overheads.

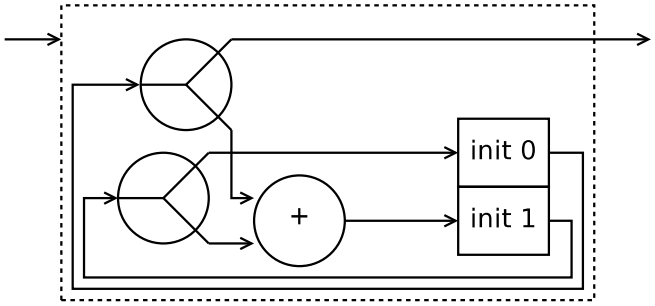
First of all, by Theorem 3.3.1, all *SF* arrow can be normalized to either a pure arrow or a *loopD* form. As an example, Figure 4.3 shows the arrow diagram of *fibs* before (a) and after normalization (b). In unsugared form, the *fibs* presented in Section 4.3 is equivalent to the following definition written in arrow combinators, whose diagram is shown in Figure 4.3(a):

$$\begin{aligned} \mathit{fibs} = \mathit{loop}\ (\mathit{arr}\ \mathit{snd} \ggg \mathit{loop}\ (\mathit{arr}\ (\mathit{uncurry}\ (+)) \ggg \mathit{init}\ 1 \ggg \mathit{arr}\ \mathit{dup}) \ggg \\ \mathit{init}\ 0 \ggg \mathit{arr}\ \mathit{dup}) \\ \textbf{where}\ \mathit{dup}\ x = (x, x) \end{aligned}$$

If we express the normal form of *fibs* in Haskell, it is equivalent to the following definition, whose diagram is shown in Figure 4.3(b):



(a) Original *fibs*



(b) Normalized *fibs*

Figure 4.3: Normalization of *fibs*

$$
\begin{aligned}
ccnf_{fibs} &= loopD\ (0,1)\ (\lambda(\_,(x,y)) \rightarrow (x,(y,x+y))) \\
&= loop\ (arr\ (\lambda(\_,(x,y)) \rightarrow (x,(y,x+y))) \ggg second\ (init\ (0,1)))
\end{aligned}
$$

Notice that in Figure 4.3(b), the pure function looks a bit more complex than what is presented in the code above, and $init\ (0,1)$ is actually drawn as $init\ 0 \star\star\star init\ 1$. This is intentional because we want to show that the two diagrams in Figure 4.3 are indeed isomorphic to each other: they have exactly the same set of nodes and edges, and the only difference is the layout.

In the remainder of this section we describe a simple sequence of optimizations that dramatically improve the run-time speed of our stream transformer arrows, and thus demonstrate the real power of CCA normalization when used as a program optimization technique for dataflow DSLs.

SF Arrow One observation is that instead of defining *loopD* as syntactic sugar, we can implement it directly using the *SF* data type:

$$
loopD\ i\ f = SF\ (g\ i)\ \textbf{where}\ g\ i\ x = \textbf{let}\ (y,i') = f\ (x,i)\ \textbf{in}\ (y, SF\ (g\ i'))
$$

As a special case of run_{sf} defined in Figure 4.2, usually we are only interested in computing the n^{th} element of the output stream when the input is a constant unit stream. This gives the following function that avoids constructing the output stream as an $S\ a$ object:

$$
\begin{aligned}
&nth_{sf} :: Int \rightarrow SF\ ()\ b \rightarrow b \\
&nth_{sf}\ n\ (SF\ f) = x\ \text{`}seq\text{`}\ \textbf{if}\ n \equiv 0\ \textbf{then}\ x\ \textbf{else}\ nth_{sf}\ (n-1)\ f'\ \textbf{where}\ (x,f') = f\ ()
\end{aligned}
$$

Notice that we use *seq* to force strict evaluation in each iteration.

CCNF Tuple Given the fact that a CCNF is no more than a pair of a state and a pure function, we can drop the *SF* data structure altogether by simply using the pair instead of an arrow written in *loopD* form. We call the pair (i, f) a *CCNF tuple* for a CCNF in the form $loopD\ i\ f$. Correspondingly we can define the stream transformer run_{ccnf} and the n^{th} element evaluator nth_{ccnf} as follows:

$$
\begin{aligned}
&run_{ccnf} :: (d, (b, d) \to (c, d)) \to S\ b \to S\ c \\
&run_{ccnf}\ (i, f) = g\ i \\
&\quad \textbf{where}\ g\ i\ (S\ x\ xs) = \textbf{let}\ (y, i') = f\ (x, i)\ \textbf{in}\ S\ y\ (g\ i'\ xs) \\
&nth_{ccnf} :: Int \to (d, ((), d) \to (c, d)) \to c \\
&nth_{ccnf}\ n\ (i, f) = aux\ n\ i \\
&\quad \textbf{where}\ aux\ n\ i = x\ `seq`\ \textbf{if}\ n \equiv 0\ \textbf{then}\ x\ \textbf{else}\ aux\ (n-1)\ i' \\
&\qquad\qquad \textbf{where}\ (x, i') = f\ ((), i)
\end{aligned}
$$

Instead of taking an arrow, the above two functions just take a CCNF tuple and use the pure function to update the state in a loop computation. In doing so, we have successfully transformed away all arrow instances, including the *SF* data structure used to implement them!

Inlining As we have mentioned in Section 3.4, all arrow definitions must be inlined in the final program at compile-time before the CCA normalization takes place. Besides this inlining step performed by Template Haskell, a Haskell compiler such as GHC may perform more inlining after we obtain a normal form, which will produce further optimized code.

For example, the actual code generated by our CCA normalizer $\$(norm\ fibs)$ is

as follows:

$$
\begin{aligned}
&\$\,(norm\ fibs) = loopD\ i_{fibs}\ f_{fibs}\\
&i_{fibs} = (0, 1)\\
&f_{fibs} = trace\ (juggle\ .\ assoc\ .\ (dup \times id\ .\ swap) \times id\ .\ juggle\ .\\
&\quad trace\ (juggle\ .\ dup \times id\ .\ swap\ .\ (uncurry\ (+)) \times id\ .\ juggle) \times id\ .\\
&\quad assoc^{-1}\ .\ snd \times swap\ .\ juggle)
\end{aligned}
$$

The CCNF tuple for the *fibs* arrow is just (i_{fibs}, f_{fibs}). Notice the sequence of function compositions in f_{fibs} is the result of our CCA normalization procedure, which can be further simplified through inlining. The built-in optimization techniques of GHC can already do this without any user intervention.

We demonstrate this step with the CCNF for *fibs*, though the technique is equally applicable to any CCNF. To compute the n^{th} element of the Fibonacci sequence, we can just apply nth_{ccnf} to the CCNF tuple (i_{fibs}, f_{fibs}) like this:

$$
\begin{aligned}
&nth_{fibs} :: Int \to Double\\
&nth_{fibs}\ n = nth_{ccnf}\ n\ (i_{fibs}, f_{fibs})
\end{aligned}
$$

When given proper optimization flags, GHC is able to aggressively optimize the above code and fully inline all functions in the definition of f_{fibs} and nth_{fibs}. The following is the equivalent intermediate representation extracted from GHC (also know as Core files) after optimization:

$$
\begin{aligned}
&nth_{fibs}\ n = \mathbf{case}\ n\ \mathbf{of}\ \{\, I\#\ m \to go\ 0\ 1\ m\,\}\\
&go :: Int\# \to Int\# \to Int\# \to Int\\
&go\ x\ y\ n = \mathbf{case}\ n\ \mathbf{of}
\end{aligned}
$$

$$\_\_DEFAULT \rightarrow go\ y\ (x+y)\ (n-1)$$

$$0 \rightarrow I\#\ x$$

As we can see, GHC has successfully inlined not only the function nth_{ccnf} but also i_{fibs} and f_{fibs}, and transformed everything into a tight loop using only strict and unboxed types (those marked by #). Notice that the tuple in i_{fibs} is actually turned into curried form and passed as arguments to the *go* function. This kind of aggressive optimization essentially results in compiling a CCA program directly to a tight loop that is free of any memory allocation of intermediate data structures.

4.5 Operational Semantics and CCA

The CCNF tuple of a CCA has a close relationship to a *Mealy machine* [Mealy, 1955], a form of finite state automaton. Formally, a Mealy machine with inputs in set A and outputs in set B is a pair (S, ϕ), where S is a set of states, and $\phi : S \rightarrow (B \times S)^A$ is a transition function. This function maps each $s_0 \in S$ to a function that produces for every input $a \in A$ an unique pair (b, s_1), consisting of the output b and the next state s_1. In categorical terms, it is a coalgebra of the functor $F : Set \rightarrow Set$ on the category of sets and functions, which is defined as $F(S) = (B \times S)^A$.

If we look at the function f in a CCNF tuple (i, f), it has the type $(b, d) \rightarrow (c, d)$, which is isomorphic to the type of ϕ in a Mealy machine, where d corresponds to the S set. Then the initial state i in a CCNF tuple would represent the starting state $s_0 \in S$.

Mealy machines are interesting because they implement causal functions over streams. A Mealy machine that takes an input stream $<a_0, a_1, \cdots, a_k, \cdots>$ and yields

an output stream $\lhd b_0, b_1, \cdots, b_k, \cdots \rhd$ can be described as a sequence of transitions:

$$s_0 \xrightarrow{a_0|b_0} s_1 \xrightarrow{a_1|b_1} \cdots \xrightarrow{a_k|b_k} s_k \xrightarrow{a_{k+1}|b_{k+1}} \cdots$$

where a single-step transition is defined as:

$$s_i \xrightarrow{a_i|b_i} s_{i+1} \quad == \quad (b_i, s_{i+1}) = \phi(s_i)\ a_i$$

If we translate the above to a Haskell function with a CCNF tuple (i, f) representing a Mealy machine, where $i = s_0$ and $f = \phi$, it would just be the run_{ccnf} function defined in Section 4.4.

Therefore, by normalizing a stream transformer program from its original arrow form to a CCNF tuple, we have essentially discovered a Mealy machine implementing an operational semantics for stream transformers. It is also interesting to note that the normalization step can be regarded as an application of the axiomatic semantics for stream transformers stated in the form of CCA laws.

We know that axiomatic semantics is more abstract, and hence a language characterized by its axiomatic semantics admits more programs than one only defined by an operational semantics. This is also true in the case of CCA. As we shall see later in Chapter 5, where we give a very different operational semantics to CCA when it is applied in an application domain that is not dataflow.

4.6 Benchmarks

We ran a set of benchmarks to measure the performance of several programs written in arrow syntax, but compiled and optimized in different ways. For each program, we:

1. Compiled with GHC, which has a built-in translator for arrow syntax, and ran nth_{sf} on the result arrow. (GHC)

2. Translated using Paterson's *arrowp* pre-processor to arrow combinators, compiled with GHC, and ran nth_{sf} on the result arrow. ($arrowp$)

3. Normalized to CCNF, compiled with GHC and ran nth_{sf} on the normalized arrow. ($CCNF_{sf}$)

4. Normalized to CCNF, compiled with GHC and ran nth_{ccnf} on the CCNF tuple. ($CCNF_{tuple}$)

The four benchmarks are: a sine wave with fixed frequency using Goertzel's method [Goertzel, 1958], the Fibonacci and factorial arrows given earlier, and a bounded counter taken from Courtney [2004]. The programs are compiled with GHC version 7.1 (development branch) and run on an Intel Atom N270 1.6GHz machine with a 32-bit Linux OS. We use the compilation flags `-O2 -fvia-C -fno-method-sharing -fexcess-precision` with GHC, and measure the CPU time used to run a program using the Criterion benchmark package [O'Sullivan]. The results are shown in Figure 4.1, where the numbers represent normalized speedup ratios, and we include the source program for all benchmarks in Appendix A.

The results show dramatic performance improvements using normalized arrows. We note that:

1. Based on the same arrow implementation, the performance gain of CCNF over the first two approaches is entirely due to program transformations at the source level. This means that the run-time overhead of arrows is significant, and cannot

Table 4.1: Dataflow Benchmark Speed Ratio (greater is better)

Name	*GHC*	*arrowp*	$CCNF_{sf}$	$CCNF_{tuple}$
sine	1.0	2.40	17.05	470.56
fibonacci	1.0	1.87	16.48	123.15
factorial	1.0	3.09	15.84	22.62
bounded counter	1.0	3.22	44.48	98.91

be neglected for real applications.

2. With the help from GHC's optimization technique, the CCNF tuple produces high-performance code that is free of dynamic memory allocation and intermediate data structures (with varying degrees [1]), and can be orders of magnitude faster than its arrow-based predecessors.

3. GHC's arrow syntax translator does not do as well as Paterson's original translator for the sample programs we chose, though both are significantly outperformed by our normalization techniques.

4. We notice that the speed-ups brought by the CCNF tuple in the factorial case is much lower than the others. This is because for this benchmark we use Haskell's unbounded *Integer* type, whereas in the Fibonacci case we only use the 32-bit *Int* type. A significant portion of the computation time is spent in doing big integer arithmetics, and hence the benefit of removing the arrow overhead seems

[1] This is subject to how much GHC can do in terms of unboxing types and currying function arguments, and in general not guaranteed as discussed in Section 4.7.

less compared to other benchmarks, but is still significant.

As mentioned in Section 4.4, our implementation of the CCA optimization relies on a Haskell compiler to carry out the final inlining of the pure function, and strict-ness analysis to remove unboxed types. One reason for taking this approach is that we shall leave it to the compiler to do what it is already capable of doing, and as seen in the micro benchmarks here, it is already very effective.

On the other hand, exactly what GHC does to further optimize the CCNF tuple remains obscure to the programmer. For example, when we examine the Core program generated by GHC, we find that it is very good at unboxing all intermediate values and currying the state tuple for the sine function, but only partially unboxed some values for the Fibonacci function. As for the bounded counter, GHC does not unbox any intermediate values or curry the state tuple. Apparently there is more room for improvements, and perhaps implementing a separate inlining and strictness analysis module just for CCNF tuples could be more beneficial to the end programmers. We leave it to the future work.

4.7 Discussion

Most synchronous languages, including the one introduced by Caspi and Pouzet [1998], are able to compile stream programs into a form called *single loop code* by performing a causality analysis to break the feedback loop of recursively defined values. Many efforts have been made to generate efficient single loop code [Halbwachs et al., 1991b, Amagbegnon et al., 1995], usually by a compilation from a high level dataflow source language to a target language that is usually imperative and low level,

but few express the transformation at the source level to reach a normal form with strong characterization. Our discovery of CCNF is original, and the optimization by normalization approach is targeting a lazy functional language, namely Haskell, and making use of an advanced Haskell compiler to further optimize and produce low level code.

Biernacki et al. [2008] propose a modular compilation technique for synchronous languages by introducing an intermediate language to represent transition functions. One of the major problems addressed by their work is that traditional modular compilation of synchronous languages imposes too strong a causality constraint that every feedback loop must cross an explicit delay. However, such a problem simply cease to exist when we adopt a lazy functional language as an intermediate or even the target language, for instance, as in our staged compilation for CCA. This is because the ability to represent immediate loopbacks, or recursions at value level, is a coherent feature of lazy languages. As discussed in Section 3.4.3, CCA by itself does not preclude modular compilations, even though our abstract CCA language and its current implementation through Template Haskell requires full inlining of arrow terms and hence are not modular.

The normalization procedure for CCA is applicable to causal stream functions in general. It is interesting to compare it to the stream fusion technique introduced by Coutts et al. [2007]. Stream fusion can help fuse zips, left folds, and nested lists into efficient loops. But on its own, it does not optimize recursively and lazily defined streams effectively.

Consider a stream generating the Fibonacci sequence. One way of writing it in Haskell is to exploit laziness and zip the stream with itself:

$$fibs :: [Int]$$

$$fibs = 0 : 1 : zipWith\ (+)\ fibs\ (tail\ fibs)$$

While the code is concise and elegant, such programming style relies too much on the definition of a co-inductively defined structure. The explicit sharing of the stream *fibs* in the definition is both a blessing and a curse. On one hand, it runs in linear time and constant space. On the other hand, the presence of the stream structure gets in the way of optimization. None of the current fusion or deforestation techniques are able to effectively eliminate cons cell allocations in this example. Real-world stream programs are usually much more complex and involve more feedbacks, and the time spent in allocating intermediate structure and by the garbage collector could degrade performance significantly.

We can certainly write a stream in stepper style that generates the Fibonacci sequence following the techniques by Coutts et al. [2007]:

$$\textbf{data}\ Stream\ a = forall\ s\ .\ Stream\ (s \to Step\ a\ s)\ s$$

$$\textbf{data}\ Step\ a\ s = Yield\ a\ s$$

$$fib_{Stream} :: Stream\ Int$$

$$fib_{Stream} = Stream\ next\ (0, 1)$$

$$\quad \textbf{where}\ next\ (a, b) = Yield\ r\ (b, r)\ \textbf{where}\ r = a + b$$

$$fibNth :: Int \to Int$$

$$fibNth\ n = nth_{Stream}\ n\ fib_{Stream}$$

Stream fusion will fuse nth_{Stream} and fib_{Stream} to produce an efficient loop for *fibNth*. For a comparison, with our technique the arrow version of the Fibonacci sequence

shown in Section 4.3 compiles to the same efficient loop, and yet retains the benefit of being abstract and concise.

We must stress that writing stepper functions is not always as easy as in trivial examples like *fibs* and *fact*. Most non-trivial stream programs that we are concerned with contain many recursive parts, and expressing them in terms of combinators in a non-recursive way can get unwieldy. Moreover, this kind of coding style exposes a lot of operational details which are arguably unnecessary for representing the underlying algorithm.

In contrast, arrow syntax relieves the burden of coding in combinator form and allows recursion via the **rec** keyword. It also completely hides the actual implementation of the underlying stream structure and is therefore more abstract.

On the topic of program optimization under an FRP or arrow setting, Burchett et al. [2007] introduce a concept called "lowering" that helps fuse pure functions in FrTime, a strict FRP language embedded in Scheme, but unfortunately it does not handle stateful computation such as the single unit delay; Nilsson [2005] makes use of several arrow laws and generalized algebraic types to optimize Yampa implementation, and in particular some stateful computations and event processing; Sculthorpe and Nilsson [2008] consider change propagation as a means to optimize Yampa programs with a dynamic structure.

Chapter 5

Application: Ordinary Differential Equations

In this chapter, we study a number of embedded DSLs for *autonomous ordinary differential equations* (autonomous ODEs) in Haskell. A naive implementation based on the *lazy tower of derivatives* is straightforward but has serious time and space leaks due to the loss of sharing when handling cyclic and infinite data structures. In seeking a solution to fix this problem, we explore a number of DSLs ranging from shallow to deep embeddings, and middle-grounds in between. We advocate a solution based on *arrows*, which happens to capture both sharing and recursion elegantly. We further relate our arrow-based DSL to CCA, whose normalization leads to a staged compilation technique improving ODE performance by orders of magnitude.

5.1 Introduction

Consider the following stream representation of the "lazy tower of derivatives" [Karczmarczuk, 1998] in Haskell:

$$\textbf{data}\ D\ a = D\ \{val_D :: a, der_D :: D\ a\}\ \textbf{deriving}\ (Eq, Show)$$

Mathematically it represents an infinite sequence of derivatives $f(t_0)$, $f'(t_0)$, $f''(t_0)$, ..., $f^{(n)}(t_0)$, ... for a function f that is continuously differentiable at some value t_0. This representation has been used frequently in a technique called *Functional Automatic Differentiation* [Karczmarczuk, 1998, Elliott, 2009]. The usual trick in Haskell is to make $D\ a$ an instance of the *Num* and *Fractional* type classes, and overload the mathematical operators to simultaneously work on all values in the tower of derivatives as follows:

$$\begin{aligned}
&\textbf{instance}\ Num\ a \Rightarrow Num\ (D\ a)\ \textbf{where}\\
&\quad D\ x\ x' + D\ y\ y' &&= D\ (x+y)\ (x'+y')\\
&\quad u@(D\ x\ x') * v@(D\ y\ y') &&= D\ (x*y)\ (x'*v + u*y')\\
&\quad negate\ (D\ x\ x') &&= D\ (-x)\ (-x')\\
&\quad fromInteger &&= const_D\ .\ fromInteger\\
&zero_D &&:: Num\ a \Rightarrow D\ a\\
&const_D, var_D &&:: Num\ a \Rightarrow a \rightarrow D\ a\\
&zero_D &&= D\ 0\ zero_D\\
&const_D\ c &&= D\ c\ zero_D\\
&var_D\ \ c &&= D\ c\ (const_D\ 1)
\end{aligned}$$

For example, if we want to calculate the derivative of a Haskell function $f\ x =$

$x^2 + 2 * x + 3$ at $x = 1$, instead of passing 1 to f, we evaluate $f\ (var_D\ 1)$, and get $D\ 6\ (D\ 4\ (D\ 2\ (D\ 0\ (...))))$, which corresponds to the tower of derivatives $f(1), f'(1), f''(1), \ldots$

5.1.1 Autonomous ODEs and the Tower of Derivatives

We first present a simple but novel use of the "lazy tower of derivatives" to implement a domain specific language (DSL) for *autonomous ordinary differential equations* (autonomous ODEs). Mathematically, an equation of the form:

$$f^{(n)} = F(t, f, f', \ldots, f^{(n-1)})$$

is called an ordinary differential equation of order n for an unknown function $f(t)$, with its n^{th} derivative described by $f^{(n)}$, where the types for f and t are $\mathbb{R} \to \mathbb{R}$ and $\mathbb{R}$ respectively. A differential equation not depending on t is called *autonomous*. An *initial value problem* of a first order autonomous ODE is of the form:

$$f' = F(f) \quad s.t. \quad f(t_0) = f_0$$

where the given pair $(t_0, f_0) \in \mathbb{R} \times \mathbb{R}$ is called the *initial condition*. The solution to a first-order ODE can be stated as:

$$f(t) = \int f'(t)dt + C$$

where C is the *constant of integration*, which is chosen to satisfy the initial condition $f(t_0) = f_0$.

In Haskell we represent the above integral operation as $init_D$ that takes an initial value f_0:

$$init_D :: a \to D\ a \to D\ a$$
$$init_D = D$$

As an example, consider the simple ODE $f' = f$, whose solution is the well known exponential function, and can be defined in terms of $init_D$:

$$e = init_D\ 1\ e$$

which is a valid Haskell definition that evaluates to a concrete value: the value of e at $t_0 = 0$, namely 1, along with a recursively defined tower of derivatives of e at t_0, each again equal to 1.

In general, by harnessing the expressive power of recursive data types and overloaded arithmetic operators, we can directly represent autonomous ODEs as a set of Haskell definitions. Such a representation precisely captures the mathematical relation among the derivatives of the unknown function at a given point, with proper initialization. We give a few more examples in Figure 5.1. Note that in the sine wave and damped oscillator examples, we translate higher-order ODEs into a system of first-order equations.

The solution to the initial value problem of an ODE can often be approximated by numerical integration. Here is a program that integrates a tower of derivatives at t_0 to its next step value at $t_0 + h$ using the Euler method, for an infinitesimal step size h:

$$euler_D \quad :: Num\ a \Rightarrow a \to D\ a \to D\ a$$
$$euler_D\ h\ f = D\ (val_D\ f + h * val_D\ (der_D\ f))\ (euler_D\ h\ (der_D\ f))$$

The function $euler_D$ lazily traverses and updates every value in the tower of deriva-

Sine wave	$y'' = -y$	$y = init_D\ y_0\ y'$ $y' = init_D\ y_1\ (-y)$
Damped oscillator	$y'' = -cy' - y$	$y = init_D\ y_0\ y'$ $y' = init_D\ y_1\ (-c * y' - y)$
Lorenz attractor	$x' = \sigma(y - x)$ $y' = x(\rho - z) - y$ $z' = xy - \beta z$	$x = init_D\ x_0\ (\sigma * (y - x))$ $y = init_D\ y_0\ (x * (\rho - z) - y)$ $z = init_D\ z_0\ (x * y - \beta * z)$

Figure 5.1: A Few ODE Examples

tives by their next step values. By repeatedly applying $euler_D$, we can sample the approximate solution to an ODE:

$$sample_D \quad :: Num\ a \Rightarrow a \to D\ a \to [a]$$
$$sample_D\ h = map\ val_D\ .\ iterate\ (euler_D\ h)$$

For instance, evaluating $sample_D\ 0.001\ e$ generates an infinite sequence of the exponential function $\exp(t)$ sampled at a 0.001 interval starting from $t = 0$:

$$[1.0, 1.001, 1.002001, 1.003003001, 1.004006004001, ...$$

5.1.2 Time and Space Leak

Thus far, we have designed a DSL embedded in Haskell for autonomous ODEs. However, our DSL, despite its elegant implementation, has but one problem: *the numerical solver has serious time and space leaks.* For instance, unfolding the sequence *sample* 0.001 e in GHCi exhibits a quadratic time behavior instead of linear. Evaluating more complex definitions than e can exhibit even worse leaks.

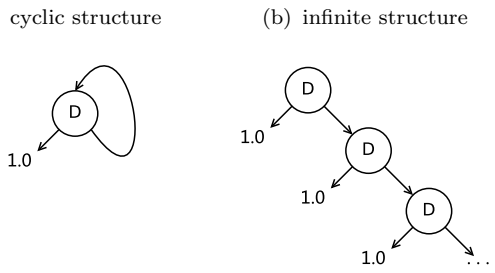


Figure 5.2: Two Structural Diagrams for e

The problem is that data sharing is lost when we update a recursive structure. In a lazy and pure functional setting, cyclic and infinite data structures are indistinguishable when they semantically denote the same value, as illustrated in Figure 5.2. Usually an implementation of a lazy language allows one to "tie the knot" using recursive definitions such as $e = init_D\ 1\ e$, which would create an internal data structure as pictured in Figure 5.2(a). This kind of knot tying, however, is very limited, and even the simplest traversal such as the identity traversal below loses sharing:

$$id_D\ (D\ v\ d) = D\ v\ (id_D\ d)$$

When evaluating $id_D\ e$, a lazy (call-by-need) strategy fails to recognize that in the unfolding of $id_D\ e = id_D\ (D\ 1\ e) = D\ 1\ (id_D\ e)$, the last and first occurrences of $id_D\ e$ could share the same value, and therefore produces something like in Figure 5.2(b). Repeatedly evaluating an update function such as $euler_D$ on a recursively defined value of type $D\ a$ will force unfolding the structure indefinitely, and hence create leaks both in space and time.

5.1.3 Approach

In the remainder of this chapter we embark on a journey seeking the best way to implement our DSL for ODEs with varying degrees of embedding. Specifically, we make the following contributions:

1. We study the problem of handling cyclic and infinite structures by analyzing different DSL representations and implementations, from shallow to deep embeddings, and mid-grounds in between.

2. We present an arrow-based DSL that captures sharing implicitly but without the usual deficiency of having to observe and compare equivalences using tags or references. Additionally the use of *arrow notation* [Paterson, 2001] enables succinct syntax for ODEs.

3. We illustrate that sharing and recursion in an object language can be better captured by arrows than higher-order abstract syntax (HOAS), even though both are mixing shallow and deep embeddings.

4. We make use of the arrow properties, and specifically the normal form of *causal commutative arrows* (CCA), to compile our DSL and eliminate all overhead introduced by the abstraction layer.

Finally if we compare the DSL for ODE to FRP languages that model continuous signals, they actually share some common characteristics including the way programs are written. There are a number of differences, however, and we elaborate below:

1. Semantically, continuous signals in FRP denote time-changing values, while the

variables in ODE denote functions that are differentiable, and not necessarily related to passage of time or a logical clock.

2. Operationally, the integral operator in FRP takes one time-changing signal and generates another time-changing signal, which implies tangible results are produced at every cycle of running a FRP program. The integral operator in modeling an ODE, however, only represents the relationship between a function and its derivative symbolically, and it is only when we want to solve an ODE, we consider the actual numerical method that samples the ODE function. There is nothing preventing us from using an algebraic method to solve ODEs symbolically, and yet the leak problems discussed here would still surface since they are related to the representation of an ODE but not necessarily the numerical methods.

3. The temporal causality of FRP signals demands a forward integration, but an ODE can be solved in either direction, forward or backward. In other words, the usual temporal view of causality, namely, current outputs depending on current and past inputs, falls apart when we consider ODEs. Instead, we shall resort to the generalized notion of computational causality that is still applicable in the case of the integral operator for ODEs.

5.2 Sharing of Computation

5.2.1 A Tagged Solution

To distinguish cyclic from infinite data structures, we can make the sharing of substructures explicit by labeling them with unique tags [O'Donnell, 1992]. The traversal of a tagged structure must keep track of all visited tags and skip those that are already traversed in order to avoid endless loops.

It must be noted, however, that not all infinite data structures can be made cyclic. This can be demonstrated by the multiplication of two towers of derivatives $x, x', \ldots, x^{(m-1)}, \ldots$ and $y, y', \ldots, y^{(n-1)}, \ldots$, which produces the following sequence:

$$
\begin{aligned}
&xy \\
&x'y + xy' \\
&x''y + x'y' + x'y' + xy'' \\
&\ldots
\end{aligned}
$$

Even if both sequences of x and y are cyclic, i.e., $x^{(i)} = x^{(i \bmod m)}$, $y^{(j)} = y^{(j \bmod n)}$, for some m and n, and any $i \geq m, j \geq n$, the resulting sequence does not necessarily have a repeating pattern that loops over from the beginning, or any part in the middle. Therefore merely adding tags to the tower of derivatives is not enough; we need to represent mathematical operations symbolically so that they become part of the data structure and hence subject to traversal as well. For instance:

```
data E a   = EI a   (T a)              -- init operator
           | E1 Op (T a)               -- unary arithmetic
           | E2 Op (T a) (T a)         -- binary arithmetic
```

$$\textbf{type } T\ a \quad = Tag\ (E\ a)$$

$$\textbf{data } Tag\ a = Tag\ Int\ a$$

$$\textbf{type } Op \quad = String$$

This is a simple DSL that supports initialization (*EI*) in addition to both unary (*E1*) and binary (*E2*) operations. Since every node in a ($T\ a$) structure is tagged, we can easily detect sharing or cycles by comparing tags. There are different ways to generate unique tags; we follow Bjesse et al. [1998] and use a state monad. We give the rest of the DSL implementation in Figure 5.3, where the *State* type and functions like *modify* and *get* are from the standard Haskell module *Control.Monad.State*, and *liftM* and *liftM2* are monadic lifting functions from the module *Control.Monad*.

To demonstrate the usage for this tag based DSL, we show below a program for the lorenz attractor, previously shown in Figure 5.1:

$$\begin{aligned}
&lorenz_M\ (x0, y0, z0) = mdo \\
&\quad x_ \leftarrow init_M\ x0\ (\sigma * (y - x)) \\
&\quad y_ \leftarrow init_M\ y0\ (x * (\rho - z) - y) \\
&\quad z_ \leftarrow init_M\ z0\ (x * y - \beta * z) \\
&\quad \textbf{let } x = return\ x_ \\
&\qquad\ \ y = return\ y_ \\
&\qquad\ \ z = return\ z_ \\
&\quad return\ z_
\end{aligned}$$

In the above definition, we use the *mdo* syntax for recursive Monads [Erkök and Launchbury, 2002, Erkök, 2002] to give recursive definitions within a Monadic setting. Note that the mathematical operators such as $*$, $+$, and $-$ are all lifted operations

```
type M a = State Int (T a)      -- monad that returns T a

instance Num a ⇒ Num (M a) where
  x + y     = liftM2 (E2 "+") x y >>= tag
  x * y     = liftM2 (E2 "*") x y >>= tag
  negate x  = liftM (E1 "-") x >>= tag
  fromInteger = const_M . fromInteger

instance Fractional a ⇒ Fractional (M a) where
  fromRational = const_M . fromRational
  x / y = liftM2 (E2 "/") x y >>= tag

newtag     :: State Int Int        -- to get fresh new tag
newtag     = modify (+1) >> get
tag        :: E a → M a            -- tag a node with new tag
tag x      = newtag >>= λi → return (Tag i x)
init_T     :: a → T a → M a        -- init with a new tag
init_T v d = tag (EI v d)
init_M     :: a → M a → M a
init_M v   = (>>=init_T v)
zero_M     :: Num a ⇒ M a
zero_M     = mfix (init_T 0)
const_M    :: Num a ⇒ a → M a
const_M c  = init_M c zero_M
```

Figure 5.3: A Tag Based DSL for Autonomous ODE

at the Monad level. The programas certainly looks a bit cumbersome compared to the one written using the "tower of derivative", and this is because we have to use a monad so that each time when $init_M$ is called, a new tag can be generated and tagged to the newly created EI node.

Since our DSL now represents all operations as part of its data structure, we no longer need the chain rule to evaluate multiplication, and instead we just represent it symbolically. Such a technique is often called *deep embedding* in contrast to our first DSL, which is a *shallow embedding* since all its operators are ordinary Haskell functions.

To evaluate our DSL, we need a val_T and a val_E function that yield the current value of a $T\ a$ and $E\ a$ structure respectively:

$$
\begin{aligned}
&val_T && :: Fractional\ a \Rightarrow T\ a \rightarrow a \\
&val_T\ (Tag\ \_\ x) && = val_E\ x \\
&val_E && :: Fractional\ a \Rightarrow E\ a \rightarrow a \\
&val_E\ (EI\ v\quad \_) && = v \\
&val_E\ (E1\ \texttt{"-"}\ x) && = -\ val_T\ x \\
&val_E\ (E2\ \texttt{"+"}\ x\ y) && = val_T\ x + val_T\ y \\
&val_E\ (E2\ \texttt{"-"}\ x\ y) && = val_T\ x - val_T\ y \\
&val_E\ (E2\ \texttt{"*"}\ x\ y) && = val_T\ x * val_T\ y \\
&val_E\ (E2\ \texttt{"/"}\ x\ y) && = val_T\ x\ /\ val_T\ y
\end{aligned}
$$

To solve the ODE represented by our DSL, we also need an $euler_T$ and a $sample_M$ function that approximate the solution:

$$
euler_T :: Fractional\ a \Rightarrow a \rightarrow T\ a \rightarrow T\ a
$$

$$
\begin{aligned}
&\mathit{euler}_T\ h\ x \quad = \ \mathit{evalState}\ (\mathit{aux}_T\ x)\ [\,]\\
&\quad \textbf{where}\\
&\qquad \mathit{aux}_T\ (\mathit{Tag}\ i\ x) \quad = \mathit{fmap}\ (\mathit{lookup}\ i)\ \mathit{get} \gg\!= \mathit{maybe}\ (\mathit{mfix}\ f)\ \mathit{return}\\
&\qquad\quad \textbf{where}\ f\ y \quad = \mathit{modify}\ ((i, y):) \gg \mathit{aux}_E\ x \gg\!= \mathit{return}\ .\ \mathit{Tag}\ i\\
&\qquad \mathit{aux}_E\ (\mathit{EI}\ v\ d) \quad = \mathit{liftM}\ (\mathit{EI}\ (v + \mathit{val}_T\ d * h))\ (\mathit{aux}_T\ d)\\
&\qquad \mathit{aux}_E\ (\mathit{E1}\ \mathit{op}\ x) \quad = \mathit{liftM}\ (\mathit{E1}\ \mathit{op})\ (\mathit{aux}_T\ x)\\
&\qquad \mathit{aux}_E\ (\mathit{E2}\ \mathit{op}\ x\ y) = \mathit{liftM2}\ (\mathit{E2}\ \mathit{op})\ (\mathit{aux}_T\ x)\ (\mathit{aux}_T\ y)\\
&\mathit{sample}_M \quad :: \ \mathit{Fractional}\ a \Rightarrow a \to M\ a \to [\,a\,]\\
&\mathit{sample}_M\ h\ x = \ \mathit{map}\ \mathit{val}_T\ \$\ \mathit{iterate}\ (\mathit{euler}_T\ h)\ \$\ \mathit{evalState}\ x\ 0
\end{aligned}
$$

The euler_T function traverses a tagged structure and updates the current values of every *EI* node to their next step values by numerical integration, with derivative values calculated by applying val_T to the derivative node. It also remembers all visited nodes in a state monad, and reuses them when repeating tags are encountered.

With the same exponential example now defined as $e = \mathit{mfix}\ (\mathit{init}_T\ 1)$, [1] repeatedly sampling its value in GHCi now exhibits a linear time behavior, and runs in constant space as one would have expected.

After moving from shallow to deep embedding, and with the help of tags, we are now able to recover sharing in the interpretation of our tagged DSL because:

- Deep embedding enables program transformations beyond what is possible with shallow embedding.

- Sharing or cycles can be better retained in a DSL program than in the result it

[1] Function *mfix* computes the fixed point of a monad, and is of type $\mathit{MonadFix}\ m \Rightarrow (a \to m\ a) \to m\ a$

computes.

By abstracting the computation about the tower of derivatives, we alleviate the burden of maintaining proper sharing in computing an infinite data structure to the representation of a DSL of which cyclic structures are made explicit. Although the tagged solution successfully avoids space leaks, it is cumbersome due to the extra baggage of generating unique tags.

For example, if we are to provide a monadic $euler_M :: a \to M\ a \to M\ a$, not only must all existing tags in the argument remain unique, a new set of tags will have to be created to accommodate the returned result. All this tag handling gets in the way of expressing our intended algorithm.

5.2.2 Higher Order Abstract Syntax

Although the tagged solution successfully avoids space leaks, it is cumbersome due to the overhead of generating and maintaining unique tags. One way to avoid dealing with tags is to mimic *Let*-expressions for sharing, and *Letrec* for recursion. However, *Let*-expressions in the object language require variable bindings and their interpretations. Indeed, variables are just lexically scoped tags, and they are remembered in an environment instead of a state monad.

An alternative solution that avoids variable bindings in the object language is to use higher-order abstract syntax (HOAS). For example, we may modify our DSL to include both *Let* and *Letrec* as follows:

```
data H a = HI    a     (H a)        -- init operator
         | H1    Op    (H a)        -- unary operator
```

$$
\begin{array}{lllll}
\mid & \mathit{H2} & \mathit{Op}\ \ (H\ a) & (H\ a) & \text{-- binary operator} \\
\mid & \mathit{Let} & (H\ a \to H\ a)\ (H\ a) & & \\
\mid & \mathit{LetRec}\ (H\ a \to H\ a) & & & \\
\mid & \mathit{Var} & \mathit{Int} & & \text{-- for internal use only}
\end{array}
$$

Where $\mathit{Let}\ f\ x$ introduces the sharing of x in the result of $f\ x$, and $\mathit{LetRec}\ f$ introduces an explicit cycle in computing the fixed point of f. When traversing *Let* and *LetRec*, however, we have to remember shared values for later lookups in an environment. For this reason we need to use $\mathit{Var}\ i$ to represents an index i in such an environment. We give the complete implementation of this DSL in Figure 5.4. Like in the tag based DSL, we make a complete syntax tree of the DSL with recursive definitions handled by *LetRec*, and all mathematical operators, including multiplication, are represented symbolically.

To evaluate the HOAS based DSL, we define a val_H function below that returns the current value of an $H\ a$ structure. In order to handle *Let* and *LetRec*, we need to remember variable bindings in an environment that maps variable names (represented as an integer) to a pair consisting of the actual parameter, and its current value. The former is needed to create the closure used in *Let* or *LetRec* during an update traversal, and further details will be explained shortly.

$$
\begin{array}{l}
\textbf{type}\ \mathit{Env}\ a = [(\mathit{Int}, (H\ a, a))] \\
\mathit{val}_H :: \mathit{Fractional}\ a \Rightarrow \mathit{Env}\ a \to H\ a \to a \\
\mathit{val}_H\ \mathit{env}\ (\mathit{HI}\ x\ \_) \quad\ = x \\
\mathit{val}_H\ \mathit{env}\ (\mathit{H1}\ \texttt{"-"}\ x) \quad = -\mathit{val}_H\ \mathit{env}\ x \\
\mathit{val}_H\ \mathit{env}\ (\mathit{H2}\ \texttt{"+"}\ x\ y) = \mathit{val}_H\ \mathit{env}\ x + \mathit{val}_H\ \mathit{env}\ y
\end{array}
$$

```
instance Num a ⇒ Num (H a) where
  x + y      = (H2 "+") x y
  x * y      = (H2 "*") x y
  negate x   = (H1 "-") x
  fromInteger = const_H . fromInteger
instance Fractional a ⇒ Fractional (H a) where
  fromRational = const_H . fromRational
  x / y = (H2 "/") x y
init_H :: a → H a → H a
init_H = HI
zero_H :: Num a ⇒ H a
zero_H = LetRec (init_H 0)
const_H :: Num a ⇒ a → H a
const_H v = init_H v zero_H
```

Figure 5.4: A HOAS Based DSL for Autonomous ODE

$$
\begin{aligned}
&val_H\ env\ (H2\ \texttt{"-"}\ x\ y) = val_H\ env\ x - val_H\ env\ y \\
&val_H\ env\ (H2\ \texttt{"*"}\ x\ y) = val_H\ env\ x * val_H\ env\ y \\
&val_H\ env\ (H2\ \texttt{"/"}\ x\ y) = val_H\ env\ x\ /\ val_H\ env\ y \\
&val_H\ env\ (Let\ f\ x) \quad = val_H\ env\ (f\ x) \\
&val_H\ env\ (LetRec\ f) \quad = val_H\ env\ (fix\ f) \\
&val_H\ env\ (Var\ i) \quad = \textbf{case}\ lookup\ i\ env\ \textbf{of} \\
&\quad Just\ (\_, x) \to x \\
&\quad Nothing \quad \to error\ (\texttt{"variable "} +\!\!+ show\ i +\!\!+ \texttt{" not found"})
\end{aligned}
$$

To solve the ODE represented by our DSL, we also need an $euler_H$ and a $sample_H$ function that approximate the solution:

$$
\begin{aligned}
&euler_H \quad :: Fractional\ a \Rightarrow a \to H\ a \to H\ a \\
&euler_H\ h = aux\ [\,] \\
&\quad \textbf{where} \\
&\quad\quad aux\ env\ (HI\ v\ x) \quad = \\
&\quad\quad\quad \textbf{let}\ v' = v + h * val_H\ env\ x \\
&\quad\quad\quad \textbf{in}\ HI\ v'\ (aux\ env\ x) \\
&\quad\quad aux\ env\ (H1\ op\ x) \quad = H1\ op\ (aux\ env\ x) \\
&\quad\quad aux\ env\ (H2\ op\ x\ y) \quad = H2\ op\ (aux\ env\ x)\ (aux\ env\ y) \\
&\quad\quad aux\ env\ (Let\ f\ x) \quad = \\
&\quad\quad\quad \textbf{let}\ x' \quad = aux\ env\ x \\
&\quad\quad\quad\quad\quad i \quad = length\ env \\
&\quad\quad\quad\quad\quad f'\ y \ = aux\ ((i, (y, val_H\ env\ x)) : env)\ (f\ (Var\ i)) \\
&\quad\quad\quad \textbf{in}\ \ Let\ f'\ x'
\end{aligned}
$$

```
    aux env x@(LetRec f) =
      let i      = length env
          f' x' = aux ((i, (x', val_H env x)) : env) (f (Var i))
      in LetRec f'
    aux env (Var i)        =
      case lookup i env of
        Just (x, _) → x
        Nothing     → error ("variable " ++ show i ++ " not found!")
sample_H  :: Fractional a ⇒ a → H a → [a]
sample_H h = map (val_H []) . iterate (euler_H h)
```

The $euler_H$ function will traverse the syntax tree, and update all *HI* nodes to the next time step. The only tricky parts are the *Let* and *LetRec* nodes that must be "opened up" and "duplicated" during the traversal. As an example, let us take the actual code snippet that traverses the *Let* node and look at the details:

```
aux env (Let f x) =
  let x'  = aux env x
      i   = length env
      f' y = aux ((i, (y, val_H env x)) : env) (f (Var i))
  in  Let f' x'
```

The function *aux* remembers shared values in an environment variable *env* during a traversal. To update a node of *Let* f x is to create a new function f' out of f in some way, and return *Let* f' x'. In computing f' it must reference the environment to get the shared value of x using val_H *env* x. Therefore f' is really a new closure.

Also notice that the way we traverse the body of function f is to supply a dummy variable $Var\ i$ as its argument, and later tie the reference to $Var\ i$ back to the parameter y of function f. This is why we need to remember y as part of the mapping of an environment variable, in addition to just the variable's value $val_H\ env\ x$.

Since our host language Haskell is not able to introspect or evaluate under lambdas, repeatedly updating HOAS structures in this way will result in building larger and larger closures, and hence creating a new kind of space leak.

Therefore, even though we can now define the same exponential ODE as $e = LetRec\ (init_H\ 1)$, evaluating $sample_H\ e$ in GHCi will still result in a space leak.

A possible remedy to this situation is *memoization* [Michie, 1968]. For example, we can have a pair of conversion functions between the HOAS language and the tagged language:

$$\begin{aligned} to_T &:: H\ a \to T\ a \\ from_T &:: T\ a \to H\ a \end{aligned}$$

Computation over $H\ a$ can then be expressed in terms of computations over $T\ a$. As a result of to_T, the intermediate tagged structure is of fixed size (relative to the input), and hence $from_T$ will create a HOAS structure also of fixed size.

Unfortunately, this approach introduces considerably more runtime overhead and begins to feel just as cumbersome as tagging. Therefore we consider HOAS inadequate as a technique for object languages that require careful sharing.

5.2.3 Summary

As discussed in above sections, by abstracting the computation about the tower of derivatives, we alleviate the burden of maintaining proper sharing in computing an infinite data structure to the representation of a DSL of which cyclic structures are made explicit.

The initial DSL around type $D\ a$ is *shallow embedding*, and such DSL programs do not easily lend to further analysis or transformation because they are indistinguishable from programs written in the host language. The tagged solution, on the other hand, is *deep embedding*, and despite its obvious interpretive overhead, it is able to recover proper sharing through the manipulation of the DSL program itself.

HOAS creates a middle ground – while the usual operators remain symbolic, the use of DSL level *Let* and *LetRec* directly employs functions from the host language – even though a correct implementation must sacrifice efficiency in order to fix a new leak problem. The question is, does there exist a solution that's both succinct at presentation and efficient at run-time?

The answer is yes. By leveraging on arrows, not only are we able to provide a clean representation of ODE, but also to discover better implementations.

5.3 ODE and Arrows

We begin with an abstract view of an ODE program without committing to a particular implementation. Here is the exponential example written in arrow syntax:

Sine wave	$y'' = -y$	**proc** () → **do** **rec** $y \leftarrow init\ y_0 \prec y'$ $y' \leftarrow init\ y_1 \prec -y$ $returnA \prec y$
Damped oscillator	$y'' = -cy' - y$	**proc** () → **do** **rec** $y \leftarrow init\ y_0 \prec y'$ $y' \leftarrow init\ y_1 \prec -c * y' - y$ $returnA \prec y$
Lorenz attractor	$x' = \sigma(y - x)$ $y' = x(\rho - z) - y$ $z' = xy - \beta z$	**proc** () → **do** **rec** $x \leftarrow init\ x_0 \prec \sigma * (y - x)$ $y \leftarrow init\ y_0 \prec x * (\rho - z) - y$ $z \leftarrow init\ z_0 \prec x * y - \beta * z$ $returnA \prec (x, y, z)$

Figure 5.5: ODE Examples in Arrow Notation

$e =$ **proc** () → **do**
  rec $e \leftarrow init\ 1 \prec e$
  $returnA \prec e$

We give more examples in Figure 5.5 by re-writing in arrow syntax the same ODEs given in Figure 5.1. We can easily tell that the arrow programs are almost line by line translation of the ODE equations.

To implement such an ODE arrow, we simply lift all arithmetic operations to pure arrows, and the only domain specific operator we need is *init*. Following our previous

two DSL designs, we have to traverse the internal structure of our DSL and update all initial values. Hence a natural choice is to implement our arrow to reflect this kind of traversal. We present the complete implementation in Figure 5.6, including the evaluation functions *euler* and *sample*.

The *ODE* type is parameterized by the type of initial value s, and implemented as a function that takes an *Updater* and an input value of type a, and returns a pair: output value of type b, and an updated ODE. The only place we actually apply the *Updater* is in the *init* combinator, where both the initial value and the current input are given to the *Updater* to produce an updated initial value:

All other arrow combinators simply pass the *Updater* around to complete a full traversal. The numerical integration is done by passing the *euler* function as the *Updater*.

This approach is not only elegant, it is also efficient – there are no space leaks. For example, unfolding *sample* 0.001 e in GHCi executes correctly and exhibits a linear time behavior. This is because

1. The representation of an ODE is composed from a fixed number of arrows with no cycles, and thus the traversal will always terminate.

2. Although the arrow itself is implemented as a higher-order function, unlike the HOAS implementation, it makes no environment lookup, and does not build new closure upon existing ones.

3. The traversal of all arrows returns new arrows of the same size, which can be proved by a structural induction as follows:

   (a) The traversal of a pure arrow always returns a pure arrow of the same size.

```
newtype ODE s a b = ODE (Updater s → a → (b, ODE s a b))

type    Updater s  = s → s → s

instance Arrow (ODE s) where

  arr f                 = ODE h where h u x      = (f x, arr f)

  ODE f ⋙ ODE g = ODE h where h u x      = let (y, f') = f u x
                                              (z, g') = g u y
                                          in  (z, f' ⋙ g')

  first (ODE f)         = ODE h where h u (x, z) = let (y, f') = f u x
                                          in  ((y, z), first f')

instance ArrowLoop (ODE s) where

  loop (ODE f)          = ODE h where h u x      = let ((y, z), f') = f u (x, z)
                                          in  (y, loop f')

init  :: s → ODE s s s

init i = ODE h

  where h f x = (i, init (f i x))

euler       :: Num s ⇒ s → Updater s

euler h i x = i + h * x

sample      :: Num s ⇒ s → ODE s () c → [c]

sample h (ODE f) = y : sample h f'

  where (y, f') = f (euler h) ()
```

Figure 5.6: An Arrow Based DSL for Autonomous ODE

(b) The traversal of all arrow compositions ($\ggg$, *first*, and *loop*) always returns a composition of the same structure, and of the same size.

(c) The update of initial values is only within the *init* arrow, which also returns a new arrow of the same size.

Of course the above is only an informal proof; a formal proof would depend on a more precise definition of size, and the lazy (call-by-need) semantics of the host language. It must be noted, however, that much of the above reasoning has little to do with the actual implementation of the arrow and its combinators. In other words, *arrows capture sharing by design*.

This intuition becomes more evident when we look at arrow programs written using combinators. As a slightly more complex example, we translate the program for a damped oscillator given in Figure 5.5 to combinators below:

$$loop\ (arr\ snd \ggg loop\ (arr\ f \ggg init\ y1 \ggg arr\ dup) \ggg init\ y_0 \ggg arr\ dup)$$

where $dup\ x = (x, x)$

$$f\ (y, y') = -c * y' - y$$

It is obvious that the above program consists of a fixed number of arrows that are easy to traverse or manipulate. The same program is presented pictorially in Figure 5.7 where the loops represent the values of y (outer) and y' (inner) being fed back to the inputs. Their values are shared at all the "points". For instance, the function *dup* only evaluates its argument once.

Both HOAS and arrow-based DSLs can be viewed as middle grounds between shallow and deep embeddings. We advocate the use of arrows because, Unlike HOAS, lambdas in the object language are represented as compositions of arrow combinators,

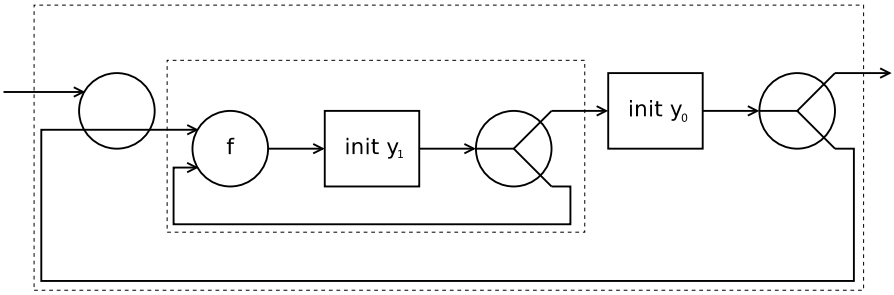


Figure 5.7: Arrow Diagram of Damped Oscillator

which lends to easy program manipulation. Also, We no longer have to deal with variable bindings, environments or open terms since all arrows translate to combinators that are always closed, and do not require memoization.

5.4 ODE and CCA

The *init* arrow for ODE introduces an internal state that is subject to both intentional computation (for being an arrow) and extensional examination (for being part of a traversal). More importantly, it is commutative and also satisfies the product law for CCA if we extend the mathematical integral from scalar values to vectors. Therefore, we can apply CCA normalization to ODE arrows. For example, the arrow program for damped oscillator is normalized to a CCNF below:

$$
\begin{aligned}
&\mathit{loop}\ (\mathit{arr}\ f \ggg \mathit{second}\ (\mathit{init}\ i)) \\
&\quad \textbf{where}\ i = (y_0, y_1) \\
&\qquad f\ (\_, (y, y')) = \textbf{let}\ y'' = -c * y' - y \\
&\qquad\qquad\qquad\qquad \textbf{in}\ \ (y, (y', y''))
\end{aligned}
$$

Just like what we saw in Chapter 4, we can utilize the CCA normalization as a staged compilation technique that effectively turns an arrow program into a CCNF tuple (i, f), where

- The state i is a nested tuple that can be viewed as a vector since all states in our ODEs are of the same numerical types.
- The pure function f computes the derivative of a state vector.

With this result in mind, we can implement a new sampling function that approximates the solution to ODE arrows with just the CCNF tuple, which is shown in Figure 5.8. The *VectorSpace* class captures state vectors with a scalar multiplication operator $*\hat{}$, and also regains the homogeneous type required by *euler*. Such tuples are made instances of the *Num* class, where arithmetic operators are overloaded point-wise. The *sample* function then takes the CCNF tuple (i, f) we obtain from the normalization of an arrow program, uses function f to calculate the derivative of i, and computes its next step value using *euler*.

Now it becomes even clearer that there is no space leak because only the state vector is updated during the repeated sampling, while the pure function remains unchanged.

5.5 Benchmark

We compare the DSL performance of the tagged solution, the ODE arrow and CCA-based staged compilation by running ODE examples listed in Figure 5.1 and Figure 5.5. We do not consider the very first DSL and the HOAS version because they

```
class VectorSpace v a where
  (*^) :: v → a → a

instance Num a ⇒ VectorSpace a a where
  x *^ y = x * y

instance (VectorSpace v a, VectorSpace v b) ⇒ VectorSpace v (a, b) where
  k *^ (x, y) = (k *^ x, k *^ y)

instance (Num a, Num b) ⇒ Num (a, b) where
  negate (x, y) = (negate x, negate y)
  (x, y) + (u, v) = (x + u, y + v)
  (x, y) - (u, v) = (x - u, y - v)
  (x, y) * (u, v) = (x * u, y * v)

euler :: (VectorSpace v a, Num a) ⇒ v → a → a → a
euler h i i' = i + h *^ i'

sample :: (VectorSpace v a, Num a) ⇒ v → (a, ((), a) → (b, a)) → [b]
sample h (i, f) = aux i
    where aux i = x : aux j
              where (x, i') = f ((), i)
                    j = euler h i i'
```

Figure 5.8: Approximate Solutions to ODE with CCNF Tuples

Table 5.1: ODE Benchmark Speed Ratio (greater is better)

Name	Tagged	Arrow	CCA
Exponential	1	0.17	83.72
Sine wave	1	0.35	27.52
Damped oscillator	1	1.13	82.34
Lorenz attractor	1	3.55	159.54

both have space leaks, and neither do we include results from the memoized HOAS version since it is always slower than the tagged DSL. The benchmarks were compiled with GHC version 7.1 (development branch) and run on an Intel Atom N270 1.6GHz machine with a 32-bit Linux OS. All programs are compiled to compute 10^4 samples using GHC 6.10.4 with compilation flag `-O2 -fvia-C -fno-method-sharing -fexcess-precision`, and the speed is measured using the Criterion benchmark package [O'Sullivan]. The results are given in Table 5.1, where all numbers are speed-up ratios measured in CPU time normalized to the speed of the first column. We make the following observations:

1. As the ODE gets more complex (from exponential, to sine, to oscillator, and to Lorenz), the tagged version becomes slower compared to the arrow version since it incurs more overhead interpreting the DSL, as well as remembering and comparing visited tags.

2. The arrow version is slower than the tagged version for simpler ODEs, which is attributed to the overhead of interpreting the arrow combinators.

3. For the micro benchmarks presented here, the CCA version is orders of magnitude faster since it is free of all arrow and arrow syntax overhead. Just like what we have shown in Section 4.4, the intermediate *Core* program generated by GHC also confirms that the CCA optimization leads to very efficient target code in a tight loop.

5.6 Discussion

Before discussing the sharing problem in general, one may ask why we take the long road implementing a DSL for ODEs, when they can be directly represented in Haskell as a function that computes derivatives. For example, the damped oscillator ODE in Figure 5.1 can be described as follows:

$$f\ (y, y') = \textbf{let}\ y'' = -c * y' - y$$
$$\textbf{in}\ \ (y', y'')$$

Coupled with a set of initial values (y_0, y_1), they can be used to compute numerical solutions to the ODE. A major drawback, however, is that such a pair is at too low level because it is unable to:

1. express the function represented by an ODE as a single value;
2. express compositions such as $y * y$ where y is defined by the above ODE;
3. make room for new extensions.

The lack of abstraction renders such a direct representation a poor choice for a DSL. Moreover, the purpose of this chapter is not to solve differential equations, but to

illustrate an use case for CCA, and explore the design space of embedded DSLs that preserves sharing of computation. It is also worth noting that our staged compilation through CCA yields a similar pair of function and state. In other words, we compile from a more expressive DSL at a higher level of abstraction to lower level programs.

Memoization caches previous computation results and later re-uses instead of re-computes them. A generic *memo* function builds an internal lookup table that may interfere with garbage collection, and the prompt release of cached data is critical to the success of this technique. A *memo* function can be implemented purely but must sacrifice laziness of the function being memoized, while a lazy *memo* cannot be pure since it requires pointer comparisons [Hughes, 1985].

The sharing problem discussed in this paper is of course not new. A majority of efforts have focused on detecting cycles and properly representing them. O'Donnell [1992] uses integer tags for explicit labelling, while Claessen and Sands [1999] suggest a non-conservative extension using references. Gill [2009] introduces type-safe observable sharing using stable names within the IO monad. These techniques usually translate a lazy cyclic structure into an equivalent graph representation, but are inefficient at handing updates.

Introducing variable bindings to denote sharing or recursion in an algebraic data type is not new either. Fegaras and Sheard [1996] adopt HOAS, while Ghani et al. [2006] employ de Bruijn indices in a nested data type [Bird and Paterson, 1999]. There are also efforts to give an alternative semantics to the fixed point operator so that cycles can be successfully recovered [Weddig, 2005].

For DSLs both shallow and deep embeddings have their respective pros and cons, and therefore it is worth exploring middle grounds. In particular, HOAS is known for

its less interpretive overhead, but at the same time it does not allow easy manipulations. Atkey et al. [2009] recently suggest a reification of HOAS into deep embedding for further processing. On the other hand, Carette et al. [2007] propose a finally tagless encoding that enables very shallow embedding, but unfortunately it does not handle cycles.

Historically the normal order reduction of a combinator program is known to preserve sharing in a similar way to lazy (call-by-need) evaluation [Turner, 1979], but such a style has rarely been used to represent sharing or cycles in algebraic data types despite having less overhead than both variable bindings and de Bruijn indices. The arrow abstraction gives rise to a rich algebra in a combinator style, which makes it a suitable candidate for traversals and updates, as well as transformations using the set of arrow laws. Further more, CCA normalization can be applied as a staged compilation technique that eliminates all interpretive overhead.

Chapter 6

Application: Functional Reactive Programming

Functional Reactive Programming (FRP) is a general framework for programming hybrid systems that supports both continuous-time and discrete-time signals. The conceptually continuous signals in FRP is what distinguishes it from the synchronous dataflow model we considered in Chapter 4 that is only discrete. In this chapter we first take a look at different implementations of continuous signals in FRP and how arrows help eliminating a space leak problem plaguing earlier FRP implementations. Then we further abstract over the hybrid model and generalize it to CCA with multi-sort inits, which gives rise to a new compilation technique for arrow based FRP languages such as Yampa. Finally we explore the limitations of CCA being an abstract programming model for hybrid systems, and give a general discussion on the general space leak problem and its relationship to evaluation strategies of functional languages.

6.1 Conventional FRP

As mentioned in Chapter 1, continuous values in FRP, which are called *signals* (or behaviors), are time-varying values that can be conceptually thought of as functions of time:

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

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 the point-wise sum of the functions representing *s1* and *s2*. More importantly, stateful computations such as integration and differentiation may be applied to signals. For example, the integral of signal s is another signal that can be just written as *integral* i s, where i represents the initial value of the integral at the starting time. Therefore it is easy to write integral or differential equations commonly used to describe dynamic systems, in similar ways to the DSLs for ODEs considered in Chapter 5.

In retrospect, even though FRP is more general than ODEs, and their respective implementations are vastly different[1], at a high level and especially when we only consider continuous signals, they share many common characteristics including the way programs are written. We will come back to explore the connections between them in Section 6.2.2.

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

[1]With the exception of arrow based implementations, which are actually similar because both employ data structures based on continuation.

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 problem that we were once puzzled upon.

6.1.1 Stream Based FRP

In the book *The Haskell School of Expression* [Hudak, 2000] 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 6.1 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, $integral_B$, which takes an initial value and returns a signal that is the numerical integration of the input signal using the Euler integration method.

The $eval_B$ function in Figure 6.1 evaluates a signal at a given point of time, and turns a signal into a function of time, which in reality is only approximated by discrete samples at a fixed global time step *dt* using the $sample_B$ function. Notice that we are also using a strict version of the list index (!!) operator so as to force the evaluation of each sample, and hence remove any time and space leak caused by the standard lazy (!!) operator.

In similar ways to what we did for the dataflow DSL in Chapter 4, common arithmetic operations can be lifted to the signal level, and we omit them here.

```
type Time     = Double
type DTime    = Double
newtype B a = B ([DTime] -> [a])
integral_B :: Double -> B Double -> B Double
integral_B i (B f) = B (λdts -> scanl (+) i (zipWith (*) dts (f dts)))
eval_B :: B Double -> Time -> Double
eval_B (B f) t = sample_B !! (truncate (t / dt))
sample_B :: B Double -> [Double]
sample_B (B f) = f (repeat dt)
(x : xs) !! n = if n ≡ 0 then x else x `seq` (xs !! (n − 1))
```

Figure 6.1: Stream-Based Signal in FRP

```
newtype C a = C (a, DTime -> C a)
integral_C :: Double -> C Double -> C Double
integral_C i (C p) = C (i, λdt -> integral_C (i + fst p ∗ dt) (snd p dt))
eval_C :: C Double -> Time -> Double
eval_C x t = sample_C x !! (truncate t / dt)
sample_C :: C Double -> [Double]
sample_C (C p) = fst p : sample_C (snd p dt)
```

Figure 6.2: Continuation-Based Signal in FRP

6.1.2 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 giving rise to its future values. The full definition is given in Figure 6.2, where $integral_C$, $eval_C$ and $sample_C$ are the corresponding functions to $integral_B$, $eval_B$ and $sample_B$ shown in Figure 6.1.

6.1.3 Time and 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 [Clinger, 1998, Appel, 1992, Wadler, 1987a,b, 1988, Marlow, 1996]. Many of these techniques are now standard fare in modern day compilers such as the GHC. Not all optimization techniques are effective at all time, however, and in certain cases may result in worse behavior rather than better [Gustavsson and Sands, 2001]. There has also been work on *relative leakiness* [Bakewell and Runciman, 2000, Gustavsson and Sands, 2001, Bakewell, 2001], 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

$$
\begin{aligned}
e &= \mathit{integral}_C\ 1\ e \\
&= \mathbf{let}\ p = (1, \lambda dt \rightarrow \mathit{integral}_C\ (1 + \mathit{fst}\ p * dt)\ (\mathit{snd}\ p\ dt)) \\
&\quad \mathbf{in}\ C\ p \\
&= \mathbf{let}\ p = (1, f) \\
&\qquad f = \lambda dt \rightarrow \mathit{integral}_C\ (1 + 1 * dt)\ (f\ dt) \\
&\quad \mathbf{in}\ C\ p
\end{aligned}
$$

Figure 6.3: Unfolding e

mathematical formulation $e(t) = \int_0^t e(t)dt$:

$$e = \mathit{integral}_C\ 1\ e$$

Evaluating successive values of e using sample_C 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 $\mathit{integral}_B$ instead of $\mathit{integral}_C$.) Our intuition tells us that unfolding the time series of a signal such as e should be constant in space and linear in time. Yet in reality, the time complexity of evaluating the n^{th} value of e is $O(n^2)$ and the space complexity is $O(n)$.

To see where the leak occurs, let's unfold the definition of e using call-by-need evaluation. We adopt a familiar style of using let-expressions to denote sharing of terms [Launchbury, 1993, Ariola et al., 1995, Maraist et al., 1998]. The unfolding of e, where $e = \mathit{integral}_C\ 1\ e$, is shown in Figure 6.3.

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

$$f = \lambda dt \rightarrow integral_C\ (1 + dt)\ (f\ dt)$$

is indeed equivalent to:

$$f = \lambda dt \rightarrow \textbf{let}\ x = integral_C\ (1 + dt)\ x\ \textbf{in}\ x$$

The former definition causes work of computing $f\ dt$ to be repeated in evaluating the recursive body of f, whereas in the latter case the computation is shared.

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:

$$repeat = \lambda x \rightarrow x : repeat\ x$$

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

$$\begin{aligned} repeat = \lambda x \rightarrow\ & \textbf{let}\ xs = x : xs \\ & \textbf{in}\ xs \end{aligned}$$

This kind of optimization of space behavior is one of the memoization techniques in a lazy functional language, which caches the result of function application (the term $repeat\ x$) using a data structure (the list xs). Such a transformation is also called "knot tying" since it regains the sharing of a recursive function call site.

One may ask why we cannot do the same thing to optimize $integral_C\ e$? It is a non-trivial task because:

- As shown in Figure 6.3, we have to unfold both definitions of $integral_C$ and e, and fuse them together by performing beta reductions and substitutions, before we actually spot the "knot" that should be tied, namely, $f\ dt$.

- The standard call-by-need evaluation implemented by most interpreters and compilers for a lazy language does not do this.

- It is also difficult for a compile-time analysis to automate the unfolding steps like those in Figure 6.3 and then "tie-the-knot" because there is no knowledge of when to stop unfolding recursive definitions before a "loose knot" can be discovered, and the time complexity to do this is quadratic ($O(n^2)$) with respect to the number of beta redices n.

Since a similar space leak (or loss of sharing) problem of ODE implementations has been discussed in Chapter 5, it is interesting to compare the leak problem of conventional FRP implementations to it:

- The $D\ a$ type used to model ODE is a stream of derivatives, while the type $B\ a$ (or the continuation based $C\ a$) models a stream of time-based samples.

- The *init* operator for ODE represents an integral relationship, but it does not implement the actual integration method like the operator $integral_C$ (or $integral_B$) for FRP.

- The leak in ODE is exposed by traversals of recursive data structures, and can be avoided by turning the recusive data representation into a non-recursive one with arrow combinators; whereas the leak in FRP is caused by subsequent sampling of an infinite stream that is recursively defined.

In a lazy language like Haskell, however, infinite streams are always recursively defined, and sampling them do not always result in unexpected space behavor. A key observation is that in the definition of e, there are actually two levels of recursion, one in $integral_C$, another in e itself. More importantly in the definition of $integral_C$, the recursion takes place under a lambda.

To further illustrate this last point, consider an alternative implementation below, where we remove the abstraction of time as a parameter and use a global fixed dt instead:

$$\textbf{newtype}\ S\ a = S\ a\ (S\ a)$$

$$integralS :: Double \to S\ Double \to S\ Double$$

$$integralS\ i\ (S\ x\ y) = S\ i\ (integralS\ (i + x * dt)\ y)$$

Note that this $S\ a$ becomes the same stream datatype we considered in the dataflow DSL (Figure 4.1). It is easy to verify that with the above definition, sampling $e = integralS\ e$ will run in constant space and linear time.

In a real implementation of FRP, however, we cannot assume a fixed delta time, even though that is the case in our much simplified implementations of $eval_B$ and $eval_C$. Hence we cannot get rid of the time abstraction. A pragmatic solution to such kind of space leak problem is to memoize the function in question by caching its evaluation results. This is the workaround taken by Hudak [2000], but has always been considered as a hack since it requires impure operations.

More fundemantally, we ask the question whether we can avoid the "hazardous knot", i.e., a recursive reference to a function application? The answer is *yes*, as illustrated by an arrow based implementation that is at the core of Yampa.

6.2 Yampa and CCA

6.2.1 Signal Function

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

$$SF\ a\ b\ \approx\ Signal\ a \rightarrow Signal\ b$$

This is a very similar abstraction to the stream function (also called *SF*) discussed in Section 4.3. Programming at the level of signal functions instead of signals offers similar advantages to that of the stream functions. But in addition, as we shall see, it results in generally fewer space leaks.

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

$$\textbf{newtype}\ SF\ a\ b = SF\ (a \rightarrow (b, DTime \rightarrow SF\ a\ b))$$

The full definition of *SF* as an arrow, a definition of an integral function, and the definition of an $eval_{SF}$ function, is given in Figure 6.4.

Our running example of an exponential signal can be defined as a signal function using the arrow syntax as follows:

$$\begin{aligned}
&e_{SF} :: SF\ ()\ Double \\
&e_{SF} = \textbf{proc}\ () \rightarrow \textbf{do} \\
&\quad \textbf{rec}\ e \leftarrow integral_{SF}\ 1 \prec e \\
&\quad returnA \prec e
\end{aligned}$$

$$\textbf{newtype}\ SF\ a\ b = SF\ (a \to (b, DTime \to SF\ a\ b))$$

$$\textbf{instance}\ Arrow\ SF\ \textbf{where}$$

$$\begin{aligned}
arr\ f \qquad\qquad &= SF\ h\ \textbf{where}\ h\ x \qquad &&= (f\ x, \lambda dt \to arr\ f) \\
first\ (SF\ f) \quad &= SF\ h\ \textbf{where}\ h\ (x, z) &&= \textbf{let}\ (y, f') = f\ x \\
&&&\quad \textbf{in}\ ((y, z), first\ .\ f') \\
SF\ f \ggg SF\ g &= SF\ h\ \textbf{where}\ h\ x \qquad &&= \textbf{let}\ (y, f') = f\ x \\
&&&\qquad\ (z, g') = g\ y \\
&&&\quad \textbf{in}\ (z, \lambda dt \to f'\ dt \ggg g'\ dt)
\end{aligned}$$

$$\textbf{instance}\ ArrowLoop\ SF\ \textbf{where}$$

$$\begin{aligned}
loop\ (SF\ f) \quad &= SF\ h\ \textbf{where}\ h\ x \qquad &&= \textbf{let}\ ((y, z), f') = f\ (x, z) \\
&&&\quad \textbf{in}\ (y, loop\ .\ f')
\end{aligned}$$

$$integral_{SF} :: Double \to SF\ Double\ Double$$

$$integral_{SF}\ i = SF\ h\ \textbf{where}\ h\ x = (i, \lambda dt \to integral_{SF}\ (i + dt * x))$$

$$eval_{SF} :: SF\ ()\ Double \to t \to Double$$

$$eval_{SF}\ sf\ t = sample_{SF}\ sf\ !!\ (truncate\ (t\ /\ dt))$$

$$sample_{SF} :: SF\ ()\ Double \to [Double]$$

$$sample_{SF}\ (SF\ f) = \textbf{let}\ (v, c) = f\ ()\ \textbf{in}\ v : run\ (c\ dt)$$

Figure 6.4: Arrow-Based FRP

Note that the input (on the right) and output (on the left) to the signal function $integral_{SF}\ 1$ is the same (namely e), and thus this is a circular signal. This program expands into a combinator form as follows:

$$e_{SF} = loop\ (second\ (integral_{SF}\ 1) \ggg arr\ (\lambda(x, i) \rightarrow (i, i)))$$

Unfolding $sample_{SF}\ e_{SF}$ in GHC gives the expected time and space behavior without any leaks. The key different here is that e_{SF} is no longer recursively defined, i.e., we no longer have any recursive reference, or a "knot", to tie when we unfold the body of $integral_{SF}$, even though there is still a lambda taking a parameter of dt in its definition.

The arrow *loop* combinator provides a recusion in e_{SF} is only at the value level, but not at the arrow level. If we do not use *loop*, we can still define e_{SF} like this:

$$\hat{e}_{SF} = \hat{e}_{SF} \ggg integral_{SF}\ 1$$

It is perhaps surprising that the above definition works at all since it represents an infinite network of arrows. The magic of lazy evaluation does the trick, and $sample_{SF}\ \hat{e}_{SF}$ actually produces the correct sequence. But unfortunately, this definition suffers from the same time and space leak problem that we saw previously. This is not a coincident, but due to the fact that $\hat{e}_{SF}$ is using Haskell's default fixed point operator. Therefore, the ability to introduce a *loop* combinator as an alternative to do recursion for arrows at the value level allows more precise control over what to recurse and what not do, and hence avoids space leaks.

6.2.2 Unifying Discrete and Continuous Time

We notice that $integral_{SF}$ in Figure 6.4 indeed shares a very similar type signature to the *init* operator in CCA, and if we try to define what it means denotationally, it is indeed the same as the *init* operator for ODE arrow. The problem, however, is that in a dataflow language such as the arrow based DSL shown in Figure 4.2, *init* is already given the meaning of a unit delay. If we want to capture both the unit delay (discrete-time) and integral (continuous-time) aspects of FRP into a single CCA framework, we have to make use of the multi-sort init extension.

A DSL for the core FRP language consisting of *init*, *integral*, and the rest of CCA combinators corresponds to $\text{CCA}_2^\dagger$, where $init^0 = init$, and $init^1 = integral$.

By Theorem 3.5.2, the normal form for $\text{CCA}_2^\dagger$ is either $arr\ f$, or $loop\ (arr\ f \ggg second\ (init\ i \twoheadrightarrow integral\ j))$ for some pure function f, and initial states i and j, where i represents the state for discrete signals, and j represents the state for continuous ones. We denote the CCNF tuple $((i, j), f)$ for $\text{CCA}_2^\dagger$ as $CCNF_2$, and capture it as a generalized algebraic datatype below:

$$\begin{aligned}&\textbf{data}\ CCNF_2\ a\ b\ \textbf{where}\\ &\quad CCNF_2 :: (VectorSpace\ DTime\ d, Num\ d) \Rightarrow\\ &\qquad\qquad ((c,d),(a,(c,d)) \to (b,(c,d))) \to CCNF_2\ a\ b\end{aligned}$$

Here we use the *VectorSpace* class from Chapter 5 to indicate that the state for a continuous signal is indeed a vector that can be integrated over *Time*.

As illustrated in Chapter 4 and Chapter 5, the transformation of CCA programs is at the source level, and it is up to the programmer to define the semantical function for a $CCNF_2$ value. To do this for Yampa, we incorporate both the single unit delay

and the euler integration rule in our evaluation function as follows, assuming we still use a global time step dt:

$$
\begin{aligned}
&eval_{CCNF_2} :: CCNF_2\ ()\ a \rightarrow Time \rightarrow a \\
&eval_{CCNF_2}\ (CCNF_2\ ((i,j),f))\ t = run\ i\ j\ !!\ (truncate\ (t\ /\ dt)) \\
&\quad \textbf{where}\ run\ i\ j = \textbf{let}\ (y,(i1,j')) = f\ ((),(i,j)) \\
&\qquad\qquad\qquad\qquad\qquad j1 = euler\ dt\ j\ j' \\
&\qquad\qquad\qquad\quad \textbf{in}\ y : run\ i1\ j1
\end{aligned}
$$

In addition to $eval_{CCNF_2}$, we can define a *reactimate* function for $CCNF_2$ mimicing the *reactimate* of Yampa that we saw in Chapter 2 as follows:

$$
\begin{aligned}
&reactimate_{CCNF_2} :: IO\ (DTime, a) \rightarrow (b \rightarrow IO\ ()) \rightarrow CCNF_2\ a\ b \rightarrow IO\ () \\
&reactimate_{CCNF_2}\ sense\ actuate\ (CCNF_2\ ((i,j),f)) = run\ i\ j \\
&\quad \textbf{where}\ run\ i\ j = \textbf{do} \\
&\qquad\qquad (dt,x) \leftarrow sense \\
&\qquad\qquad \textbf{let}\ (y,(i1,j')) = f\ (x,(i,j)) \\
&\qquad\qquad\quad j1 = euler\ dt\ j\ j' \\
&\qquad\qquad actuate\ y \\
&\qquad\qquad run\ i1\ j1
\end{aligned}
$$

In a similar way to implementing an operational semantics based on CCNF tuples for the dataflow DSL in Chapter 4, we have now implemented an operational semantics for the Yampa language based on $CCNF_2$.

6.3 Discussion

6.3.1 Dynamic Structure and Switch

As mentioned in Chapter 2, Yampa programs can dynamically react to external events by using switches. Since our implementation of CCA normalization is at compile-time only, it is not possible to support the kind of complete dynamism enabled by the switches. One indication of this limitation is in the CCNF itself: the internal state i is given a static type, which effective means a CCNF cannot modify its own structure at run-time, while the general Yampa arrow such as the *SF* data type can.

A limited form of dynamic structure can be supported, however, if we implement the switch function as part of the dynamic semantics:

$$switch_{CCNF_2} :: CCNF_2\ a\ (b, Event\ c) \rightarrow (c \rightarrow CCNF_2\ a\ b) \rightarrow S\ a \rightarrow S\ b$$

We leave out the actual implementation, but would like to remark that $switch_{CCNF_2}$ is not composable and only at the top level, while the original Yampa switch is perfectly composable and can be used at any level during a composition since it returns an *SF* arrow as result.

Another possible work-around is not to use CCNF tuples such as the $CCNF_2$ data type but just the normalized CCNF in its arrow form, which can then be used afterwards just like any other arrow. Early benchmarks in Section 4.6 and Section 5.5 suggest that just going CCNF is worthwhile, although we would lose some of advanced optimizations specific to CCNF tuples. On the other hand we can still retain the flexibility of a full Yampa arrow.

A fully dynamic *switch* has the ability to compose the arrow to be switched into

at runtime, and so far our CCA implementation is not able to handle such a case. On the other hand, not all arrows to be switched into are dynamically composed, and often the handler function simply takes a parameter and produces an arrow still of a static structure, such as the bouncing ball example considered in Section 2.2.2, and hence there is no problem of making compile-time CCA normalization work with switches in such cases.

6.3.2 Space Leak and Evaluation Strategy

With our discussion on the space leak problem both in this chapter and in Chapter 5, it appears that the loss of sharing of function applications during either the traversal or unfolding of a recursive data structure is the root cause. In both cases we propose restructuring the DSL under the arrow framework could resolve the leak, which may not seem like a very general solution to the problem. After all, not all DSLs can be framed as arrows.

The inability to maintain sharing of function application is a fundamental issue inherent to the algorithm of call-by-need. By improving call-by-need, lambda evaluation strategies such as complete or optimal laziness can help recover lost sharing, but they are (so far) costly to implement, and generally not very well understood in terms of space and time complexity [Lamping, 1990, Asperti and Guerrini, 1998, Thyer, 1999, Sinot, 2008].

Lévy [1978] introduced the notion of *optimal reduction*, and Lamping [1990] is the first to invent an optimal reducer for the lambda calculus. Asperti and Guerrini [1998] summarize optimal reduction as:

Table 6.1: Reduction Steps of Unfolding $e_n = sample_C\ e\ !!\ n$

Exp	Call-by-need		Optimal	
	beta	arithmetic	beta	arithmetic
e_1	13	2	11	2
e_2	28	4	16	4
e_3	50	12	21	6
e_4	79	20	26	8
e_5	115	30	31	10
e_6	158	42	36	12
e_7	208	56	41	14

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. Verified by our own implementation [Liu, 2009] of both Lambdascope [van Oostrom et al., 2004], 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 $e_n = (sample_C\ e)\ !!\ n$ in Table 6.1.

The data confirms that the time complexity of unfolding e_n under call-by-need is

[2]The only arithmetic operation that we count is the calculation of $dt * i + j$, and we count it as 2 operations.

quadratic, because it redundantly re-evaluates e_{n-1}. We have:

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

where steps_{cbn} and steps_{opt} respectively stand for the number of reductions for call-by-need and for optimal to unfold e_n, and $\text{steps}_{opt}(n)$ is apparently linear in n as indicated by Table 6.1.

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.

Chapter 7

Conclusion and Future Work

We have formulated causal commutative arrows (CCA), an extension to the arrows framework, and its associated laws, and designed a language for CCA that is strongly normalizing. We give the normalization procedure and prove that it is sound and always terminating. The normalization property of CCA leads to an effective compile-time optimization technique that can help improving the run-time performance of such arrows by orders of magnitude. CCA captures the essence of causal stream computation and synchronous dataflow, and its normal form has a close relation to the operational semantics of dataflow languages defined by Mealy machines. Arrows and CCA are also used to model a DSL for ordinary differential equations (ODE) that not only improves performance but also resolves a space leak problem. Finally, CCA with a multi-sort init extension is shown to be a good abstraction for the core of Yampa, a functional reactive programming (FRP) language that models hybrid systems with both discrete and continuous components, even though CCA does not model the dynamic mode switching of Yampa programs.

In both Chapter 4 and Chapter 5, we make use of a number of micro-benchmarks to demonstrate the effectiveness of CCA optimization. It is our future work to explore the application of CCA in larger applications and under more complicated but real-world settings. In fact we are already doing it in the on-going development of *Euterpea* [Hudak et al., 2010]. Euterpea is a Haskell software library suitable for high-level music representation, algorithmic composition and analysis, mid-level concepts such as MIDI, and low-level audio processing, sound synthesis, and instrument design. We use an arrow based DSL for sound synthesis and signal processing that can be considered an instance of CCA. Preliminary results are already showing that we can achieve real-time synthesis in a Haskell program with a performance comparable to C-based implementations. More work still has to be done in order to fully evaluate the use of CCA normalization as an optimization technique in Euterpea.

Our implementation of CCA normalization is done at compile-time with the help from Template Haskell, which brings both conveniences as well as restrictions that put burdens on programmer. Not all the power of Template haskell is needed, however, and therefore it may be worth exploring alternative meta-programming approaches that provide the kind of program transformation needed by CCA, namely, the inlining of function definitions. It may also be worth exploring run-time alternatives such as a just-in-time (JIT) compiler with the ability to dynamically generate optimized run-time binary code for CCA.

With regard to optimization through CCA normalization, we also rely on advanced compilation offered by modern Haskell compilers such as GHC to further optimize the normal form using techniques like strictness analysis, unboxing data types, currying arguments, and function inlining. As mentioned in Section 4.7, GHC's behavior in

this regard is not completely tunable by programmers, and if possible we would like to have a separate module that output highly efficient source-level programs and give programmers more control over the process.

The only side-effect we have considered in CCA is brought by the *init* operator, and because of the commutativity law, such effects are non-interfering and isolated. Besides the operation of unit delay and numerical integration, we could also consider concurrent I/O operations that communicate with the environment but are oblivious to each other. For example, reading keyboard inputs is entirely independent to sending packets to the network interface. Currently the preferred way of a Yampa program to interact with the environment is through the *reactimate* function, but such a concurrent I/O extension would offer a much richer set of operators to work with, and perhaps even a fresh new way of I/O programming in FRP.

Bibliography

Pascalin Amagbegnon, Loc Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language signal. In *Conference on Programming Language Design and Implementation*, pages 163–173. ACM Press, 1995.

Andrew W. Appel. *Compiling with Continuation*. Cambridge University Press, New York, NY, USA, 1992. ISBN 9780521033114.

Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. In *POPL '95: the 22th symposium on Principles of Programming Languages*, pages 233–246, 1995.

Arvind and David E. Culler. *Dataflow architectures*, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986. ISBN 0-8243-3201-6.

Andrea Asperti and Stefano Guerrini. *The optimal implementation of functional programming languages*. Cambridge University Press, 1998. ISBN 0-521-62112-7.

Robert Atkey. What is a categorical model of arrows? In *Mathematically Structured Functional Programming*, 2008.

Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Proc. of the 2009 ACM SIGPLAN Haskell Symposium*. ACM, 2009.

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

Adam Bakewell and Colin Runciman. A model for comparing the space usage of lazy evaluators. In *PPDP'00: the 2nd ACM SIGPLAN International conference on Principles and Practice of Declarative Programming*, pages 151–162, 2000.

Gérard Berry and Laurent Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, 1985. Springer-Verlag. ISBN 3-540-15670-4.

Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008. ACM.

Richard S. Bird and Ross Paterson. de Bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, 1999. ISSN 0956-7968.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *ICFP '98: International Conference on Functional Programming*, pages 174–184. ACM, 1998.

Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: A

static optimization technique for transparent functional reactivity. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 71–80. ACM Press, 2007.

Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 222–238. Springer, November 2007. ISBN 978-3-540-76636-0.

P. Caspi, N. Halbwachs, D. Pilaud, and J.A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, January 1987.

Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998.

Eric Cheng and Paul Hudak. Look ma, no arrows – a functional reactive real-time sound synthesis framework. Technical Report YALEU/DCS/RR-1405, Yale University, May 2008.

Mun Hon Cheong. Functional programming and 3D games, November 2005. http://www.haskell.org/haskellwiki/Frag.

Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, pages 62–73. Springer Verlag, 1999.

William D. Clinger. Proper tail recursion and space efficiency. In *PLDI '98: Proc. of*

the ACM SIGPLAN conference on Programming language design and implementation, pages 174–185, New York, NY, USA, 1998. ACM Press. ISBN 0-89791-987-4.

Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *EMSOFT '04: Proc. of the 4th ACM international conference on Embedded software*, pages 230–239, New York, NY, USA, 2004. ACM. ISBN 1-58113-860-1.

Antony Courtney. *Modelling User Interfaces in a Functional Language*. PhD thesis, Department of Computer Science, Yale University, 2004.

Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell '01: Proc. of the 2001 ACM SIGPLAN Haskell Workshop*. ACM, 2001.

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

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *ICFP '07: Proc. of the ACM SIGPLAN International Conference on Functional Programming*, April 2007.

A. L. Davis and R. M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, 1982. ISSN 0018-9162.

Conal Elliott. Beautiful differentiation. In *ICFP '09: Proc. of International Conference on Functional Programming*, pages 191–202. ACM, 2009.

Levent Erkök. *Value recursion in monadic computations*. PhD thesis, OGI School of Science and Engineering, OHSU, Portland, Oregon, 2002.

Levent Erkök and John Launchbury. A recursive do for Haskell. In *Haskell '02: Proc. of the 2002 ACM SIGPLAN Haskell Workshop*, pages 29–37, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *POPL '96: the 23th symposium on Principles of Programming Languages*, pages 284–294. ACM, 1996.

Thierry Gautier, Paul Le Guernic, and Löic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 257–277, London, UK, 1987. Springer-Verlag. ISBN 0-387-18317-5.

N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *TFP '06: Proc. of the 7th Symposium on Trends in Functional Programming*, pages 173–188, 2006.

Andy Gill. Type-safe observable sharing in Haskell. In *Haskell '09: Proc. of the 2009 ACM SIGPLAN Haskell Symposium*. ACM, 2009.

George Giorgidze and Henrik Nilsson. Switched-on yampa. In Paul Hudak and David Scott Warren, editors, *PADL '08: Proc. of the 10th International Symposium on the Practical Aspects of Declarative Language*, volume 4902 of *Lecture Notes in Computer Science*, pages 282–298, San Francisco, CA, USA, January 2008. Springer. ISBN 978-3-540-77441-9.

Gerald Goertzel. An algorithm for the evaluation of finite trigonometric series. *American Mathematical Monthly*, 65:34–35, January 1958.

Jorgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In *ICFP '08: Proc. of International Conference on Functional Programming*, pages 265–276, 2001.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proc. of the IEEE*, pages 1305–1320, 1991a.

N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In J. Maluszyński and M. Wirsing, editors, *Proc. of the Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13207–218. Springer Verlag, 1991b.

Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1992. ISBN 0792393112.

David Harel. On folk theorems, 1980.

Masahito Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *TLCA '97: Proc. of the Third International Conference on Typed Lambda Calculi and Applications*, pages 196–213, London, UK, 1997. Springer-Verlag. ISBN 3-540-62688-3.

Liwen Huang, Paul Hudak, and John Peterson. Hporter: Using arrows to compose parallel processes. In *PADL '07: Proc. of the 9th International Symposium on*

Practical Aspects of Declarative Languages, pages 275–289. Springer Verlag LNCS 4354, 2007.

Paul Hudak. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-64408-9.

Paul Hudak. Modular domain specific languages and tools. In *Proc. of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, 1998.

Paul Hudak. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), 1996.

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.

Paul Hudak, Paul Liu, Michael Stern, and Ashish Agarwal. Yampa meets the worm. Technical Report YALEU/DCS/RR-1408, Yale University, July 2008.

Paul Hudak, Hai Liu, and Eric Cheng. The Euterpea project. `http://www.cs.yale.edu/c2/index.php/research/by-topic/topic/the-euterpea-project`, 2010.

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

John Hughes. Programming with arrows. In *Advanced Functional Programming*, pages 73–129, 2004.

John Hughes. Lazy memo-functions. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 129–146. Springer-Verlag New York, Inc., 1985. ISBN 3-387-15975-4.

Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In *ESOP '99: European Symposium on Programming*, pages 273–287, 1999.

Jerzy Karczmarczuk. Functional differentiation of computer programs. In *ICFP '98: Proc. of International Conference on Functional Programming*, pages 195–203, 1998.

John Lamping. An algorithm for optimal lambda calculus reduction. In *POPL '90: the 17th symposium on Principles of Programming Languages*, pages 16–30. ACM, 1990. ISBN 0-89791-343-4.

John Launchbury. A natural semantics for lazy evaluation. In *POPL '93: Proc. of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.

Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1): 24–35, 1987. ISSN 0018-9340.

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

Sam Lindley, Philip Wadler, and Jeremy Yallop. The arrow calculus. *Journal of Functional Programming*, 20(1):51–69, 2010.

Hai Liu. LambdaINet: Graphical interaction net evaluator for optimal evaluation. `http://hackage.haskell.org/package/LambdaINet`, 2009.

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

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

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008. ISSN 0956-7968.

G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

Donald Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *ICFP '05: Proc. of International Conference on Functional Programming*, pages 54–65, 2005.

Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, continued. In *Haskell '02: Proc. of the 2002 ACM SIGPLAN Haskell Workshop*. ACM, 2002.

John T. O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming Workshops in Computing*, pages 178–194. Springer-Verlag, 1992.

Clemens Oertel. *RatTracker: A Functional-Reactive Approach to Flexible Control of Behavioural Conditioning Experiments*. PhD thesis, Wilhelm-Schickard-Institute for Computer Science at the University of Tübingen, May 2006.

Bryan O'Sullivan. Criterion: robust, reliable performance measurement and analysis. `http://bitbucket.org/bos/criterion`.

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

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

John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *PADL '99: Proc. of the First International Workshop on Practical Aspects of Declarative Languages*. SIGPLAN, Jan 1999b.

John Power and Hayo Thielecke. Closed freyd- and kappa-categories. In *ICALP*, pages 625–634, 1999.

Jan J. M. M. Rutten. Algebraic specification and coalgebraic synthesis of mealy automata. *Electronic Notes in Theoretical Computer Science*, 160:305–319, 2006.

Neil Sculthorpe and Henrik Nilsson. Optimisation of dynamic, hybrid signal func-

tion networks. In *TFP '08: Proc. of the 9th Symposium on Trends in Functional Programming*, pages 97–112. Intellect, 2008.

Neil Sculthorpe and Henrik Nilsson. Safe functional reactive programming through dependent types. In *ICFP '09: Proc. of the 14th International Conference on Functional Programming*, pages 23–34. ACM, 2009.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *Haskell '02: Proc. of the 2002 ACM SIGPLAN Haskell Workshop*, pages 1–16. ACM Press, October 2002.

François-Régis Sinot. Complete laziness: a natural semantics. *Electron. Notes Theor. Comput. Sci.*, 204:129–145, 2008. ISSN 1571-0661.

Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

Ross Howard Street, A. Joyal, and D. Verity. Traced monoidal categories. *Mathematical Proc. of the Cambridge Philosophical Society*, 119(3):425–446, 1996.

William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr 2002.

Michael Jonathan Thyer. *Lazy Specialization*. PhD thesis, York University, 1999.

D.A. Turner. A new implementation technique for applicative languages. *Software-Practice and Experience*, 9:31–49, 1979.

Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. In Zoltán Horváth, editor, *CEFP*, volume 4164 of *Lecture Notes in Computer Science*, pages 135–167. Springer, 2005. ISBN 3-540-46843-9.

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.

William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985. ISBN 0-12-729650-6.

Philip 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, 1987a.

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

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

Hampus Weddig. Fixing the semantics. In *TFP '05: Proc. of the 6th Symposium on Trends in Functional Programming*, pages 236–251, 2005.

Appendix A

Benchmark Programs

The benchmark programs used in Table 4.1 are given below.

```
sr = 44100 :: Int

sine :: ArrowInit a ⇒ Double → a () Double

sine freq = proc _ → do

  rec x ← init i⤙ r

    y ← init 0⤙ x

    let r = c ∗ x − y

  returnA⤙ r

  where

    omh = 2 ∗ pi / (fromIntegral sr) ∗ freq

    i = sin omh

    c = 2 ∗ cos omh

fibs :: ArrowInit a ⇒ a () Int

fibs = proc _ → do
```

```
  rec       f ← init 0 ≺ g
            g ← init 1 ≺ (f + g)
  returnA ≺ f

ones :: ArrowInit a ⇒ a () Int
ones = arr (λ_ → 1)

sum = proc x → do
  rec s ← init 0 ≺ s + x
  returnA ≺ s

nats = proc x → do
  y ← ones ≺ x
  z ← sum ≺ y
  returnA ≺ z

fact :: ArrowInit a ⇒ a () Integer
fact = proc x → do
  n ← nats ≺ x
  rec f ← init 1 ≺ f * fromIntegral (n + 1)
  returnA ≺ f

type Event = Maybe

hold i = proc e → do
  rec y ← init i ≺ z
    let z = maybe y id e
  returnA ≺ z
```

```
dHold i = proc e → do
  y ← hold i-≺ e
  z ← init i-≺ y
  returnA-≺ z

tag :: Event a → b → Event b
tag e v = fmap (λ_ → v) e

accum i = proc f → do
  rec s ← init i-≺ t
    let t = (maybe id id f) s
  returnA-≺ f `tag` t

dAccumHold i = proc f → do
  x ← accum i-≺ f
  y ← dHold i-≺ x
  returnA-≺ y

gate :: Event a → Bool → Event a
_ `gate` False = Nothing
e `gate` True = e

boundedCounter max = proc incReq → do
  rec n ← dAccumHold 0-≺ incCount
    let incCount = (incReq `gate` (n < max)) `tag` (+1)
  returnA-≺ n

bcinput :: ArrowInit a ⇒ a () (Event ())
```

$$bcinput = \textbf{proc}\ () \rightarrow \textbf{do}$$

$$\quad \textbf{rec}\ t \leftarrow init\ False \prec \neg\ t$$

$$\quad returnA \prec Just\ ()\ `gate`\ t$$

$$boundedCounterTest :: ArrowInit\ a \Rightarrow a\ ()\ Int$$

$$boundedCounterTest = altTrue \ggg boundedCounter\ 100$$

When conducting the benchmarks using the Criterion benchmark package, we unfold *sine* 2 to its 10000^{th} sample, *fibs* to 37^{th} (before hitting the max 32-bit signed integer), *fact* to 100^{th}, and *boundedCounterTest* to 100^{th}.

Appendix B

Proofs

B.1 The *sequencing* rule of *loopD*

The sequencing rule from Figure 3.4 is directly derivable from the following equation:

$$loopD\ i\ f \ggg loopD\ j\ g = loopD\ (i,j)\ (assoc'\ (juggle'\ (g \times id)\ .\ (f \times id)))$$

In order to show the above is true, we first prove three lemmas.

Lemma B.1.1 *Given a function definition revjuggle* $(a,(b,c)) = (b,(a,c))$*, we show that for all* f*:*

$$second\ (second\ f) = arr\ revjuggle \ggg second\ (second\ f) \ggg arr\ revjuggle$$

Proof

We start from the left-hand side by equational reasoning.

$$\begin{aligned} & lhs \\ = {} & second\ (second\ f) \end{aligned}$$

definition of *second*

$= arr\ swap \ggg first\ (arr\ swap \ggg first\ f \ggg arr\ swap) \ggg arr\ swap$

functor of *first*, extension of *first*

$= arr\ swap \ggg arr\ (swap \times id) \ggg first\ (first\ f) \ggg arr\ (swap \times id) \ggg arr\ swap$

identity of $arr, id = assoc^{-1}\ .\ assoc$, and composition of *arr*

$= arr\ swap \ggg arr\ (swap \times id) \ggg first\ (first\ f) \ggg arr\ assoc \ggg arr\ assoc^{-1}$

$\ggg arr\ (swap \times id) \ggg arr\ swap$

association of *first*

$= arr\ swap \ggg arr\ (swap \times id) \ggg arr\ assoc \ggg first\ f \ggg arr\ assoc^{-1}$

$\ggg arr\ (swap \times id) \ggg arr\ swap$

composition of *arr*

$= arr\ (assoc\ .\ (swap \times id)\ .\ swap) \ggg first\ f \ggg arr\ (swap\ .\ swap \times id\ .\ assoc^{-1})$

unfold function definition and beta reduce

$= arr\ (\lambda(a,(b,c)) \to (c,(a,b))) \ggg first\ f \ggg arr\ (\lambda(c,(a,b)) \to (a,(b,c)))$

Then from the right-hand side:

rhs

$= arr\ revjuggle \ggg second\ (second\ f) \ggg arr\ revjuggle$

definition of *second*

$= arr\ revjuggle \ggg arr\ swap \ggg first\ (arr\ swap \ggg first\ f \ggg arr\ swap)$

$\ggg arr\ swap \ggg arr\ revjuggle$

functor of *first*, extension of *first*

$= arr\ revjuggle \ggg arr\ swap \ggg arr\ (swap \times id) \ggg first\ (first\ f)$

$\ggg arr\ (swap \times id) \ggg arr\ swap \ggg arr\ revjuggle$

identity of $arr, id = assoc^{-1} \;.\; assoc$, and composition of arr

$= arr\ revjuggle \ggg arr\ swap \ggg arr\ (swap \times id) \ggg first\ (first\ f)$

$\ggg arr\ assoc \ggg arr\ assoc^{-1} \ggg arr\ (swap \times id) \ggg arr\ swap$

$\ggg arr\ revjuggle$

association of *first*

$= arr\ revjuggle \ggg arr\ swap \ggg arr\ (swap \times id) \ggg arr\ assoc \ggg first\ f$

$\ggg arr\ assoc^{-1} \ggg arr\ (swap \times id) \ggg arr\ swap \ggg arr\ revjuggle$

composition of arr

$= arr\ (assoc\ .\ swap \times id\ .\ swap\ .\ revjuggle) \ggg first\ f$

$\ggg arr\ (revjuggle\ .\ swap\ .\ swap \times id\ .\ assoc^{-1})$

unfold function definition and beta reduce

$= arr\ (\lambda(a,(b,c)) \to (c,(a,b))) \ggg first\ f \ggg arr\ (\lambda(c,(a,b)) \to (a,(b,c)))$

Therefore, lhs = rhs. □

Lemma B.1.2 *We show that for all f and g:*

$$second\ (first\ f) \ggg arr\ (assoc\ .\ swap) = arr\ (assoc\ .\ swap) \ggg first\ f$$

Proof

We start from the left-hand side by equational reasoning:

lhs

$= second\ (first\ f) \ggg arr\ (assoc\ .\ swap)$

definition of *second*

$= arr\ swap \ggg first\ (first\ f) \ggg arr\ swap \ggg arr\ (assoc\ .\ swap)$

identity of $arr, id = assoc^{-1} \;.\; assoc$, and composition of arr

$= arr\ swap \ggg first\ (first\ f) \ggg arr\ assoc \ggg arr\ assoc^{-1} \ggg arr\ swap$

$\ggg arr\ (assoc\ .\ swap)$

association of *first*

$= arr\ swap \ggg arr\ assoc \ggg first\ f \ggg arr\ assoc^{-1} \ggg arr\ swap$

$\ggg arr\ (assoc\ .\ swap)$

composition of *arr*

$= arr\ (assoc\ .\ swap) \ggg first\ f \ggg arr\ (assoc\ .\ swap\ .\ swap\ .\ assoc^{-1})$

identity of $arr, id = assoc\ .\ swap\ .\ swap\ .\ assoc^{-1}$

$= arr\ (assoc\ .\ swap) \ggg first\ f$ □

Lemma B.1.3 *We show that for all f:*

$$first\ f = arr\ revjuggle \ggg second\ (first\ f) \ggg arr\ revjuggle$$

Proof

We start from the right-hand side by equational reasoning:

rhs

$= arr\ revjuggle \ggg second\ (first\ f) \ggg arr\ revjuggle$

definition of *second*

$= arr\ revjuggle \ggg arr\ swap \gg first\ (first\ f) \ggg arr\ swap \ggg arr\ revjuggle$

identity of $arr, id = assoc^{-1}\ .\ assoc$, and composition of *arr*

$= arr\ revjuggle \ggg arr\ swap \gg first\ (first\ f) \ggg arr\ assoc \ggg arr\ assoc^{-1}$

$\ggg arr\ swap \ggg arr\ revjuggle$

association of *first*

$= arr\ revjuggle \ggg arr\ swap \gg arr\ assoc \ggg first\ f \ggg arr\ assoc^{-1}$

$\ggg arr\ swap \ggg arr\ revjuggle$

identity of $arr, id = id \times swap\ .\ id \times swap$, and composition of arr

$= arr\ revjuggle \ggg arr\ swap \gg arr\ assoc \ggg arr\ (id \times swap)$

$\ggg arr\ (id \times swap) \ggg first\ f \ggg arr\ assoc^{-1}$

exchange of *first*

$= arr\ revjuggle \ggg arr\ swap \gg arr\ assoc \ggg arr\ (id \times swap) \ggg first\ f$

$\ggg arr\ (id \times swap) \ggg arr\ assoc^{-1}$

composition of arr

$= arr\ (id \times swap\ .\ assoc\ .\ swap\ .\ revjuggle) \ggg first\ f$

$\ggg arr\ (revjuggle\ .\ swap\ .\ assoc^{-1}\ .\ id \times swap)$

identity of $arr, id = id \times swap\ .\ assoc\ .\ swap\ .\ revjuggl$

and $id = revjuggle\ .\ swap\ .\ assoc^{-1}\ .\ id \times swap$

$= first\ f$ □

We then show that

$$loopD\ i\ f \ggg loopD\ j\ g = loopD\ (i, j)\ (assoc'\ (juggle'\ (g \times id)\ .\ (f \times id)))$$

holds by equational reasoning, starting from the left-hand side:

$loopD\ i\ f \ggg loopD\ j\ g$

definition of *loopD*

$= loop\ (arr\ f \ggg second\ (init\ i)) \ggg loop\ (arr\ g \ggg second\ (init\ j))$

left tightening of *loop*

$= loop\ (first\ (loop\ (arr\ f \ggg second\ (init\ i))) \ggg (arr\ g \ggg second\ (init\ j)))$

definition of *second*

$= loop\ (arr\ swap \ggg second\ (loop\ (arr\ f \ggg second\ (init\ i))) \ggg arr\ swap$

$\ggg (arr\ g \ggg second\ (init\ j)))$

superposing of *loop*

$= loop\ (arr\ swap \ggg loop\ (arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i))$

$\ggg arr\ assoc^{-1}) \ggg arr\ swap \ggg (arr\ g \ggg second\ (init\ j)))$

left tightening of *loop*

$= loop\ (loop\ (first\ (arr\ swap) \ggg arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i))$

$\ggg arr\ assoc^{-1}) \ggg arr\ swap \ggg (arr\ g \ggg second\ (init\ j)))$

extension of *first*

$= loop\ (loop\ (arr\ (swap \times id) \ggg arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i))$

$\ggg arr\ assoc^{-1}) \ggg arr\ swap \ggg (arr\ g \ggg second\ (init\ j)))$

associativity of $\ggg$

$= loop\ (loop\ (arr\ (swap \times id) \ggg arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i))$

$\ggg arr\ assoc^{-1}) \ggg (arr\ swap \ggg arr\ g \ggg second\ (init\ j)))$

right tightening of *loop*

$= loop\ (loop\ (arr\ (swap \times id) \ggg arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i))$

$\ggg arr\ assoc^{-1} \ggg first\ (arr\ swap \ggg arr\ g \ggg second\ (init\ j))))$

vanishing of *loop*

$= loop\ (arr\ assoc^{-1} \ggg arr\ (swap \times id) \ggg arr\ assoc \ggg second\ (arr\ f$

$\ggg second\ (init\ i)) \ggg arr\ assoc^{-1} \ggg first\ (arr\ swap \ggg arr\ g$

$\ggg second\ (init\ j)) \ggg arr\ assoc)$

identity of $arr, id = id \times swap\ .\ id \times swap$, composition of *arr*

$= loop\ (arr\ (id \times swap) \ggg arr\ (id \times swap) \ggg arr\ assoc^{-1} \ggg arr\ (swap \times id)$

$$\ggg arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i)) \ggg arr\ assoc^{-1}$$

$$\ggg first\ (arr\ swap \ggg arr\ g \ggg second\ (init\ j)) \ggg arr\ assoc)$$

sliding of *loop*

$$= loop\ (arr\ (id \times swap) \ggg arr\ assoc^{-1} \ggg arr\ (swap \times id) \ggg arr\ assoc$$

$$\ggg second\ (arr\ f \ggg second\ (init\ i)) \ggg arr\ assoc^{-1} \ggg first\ (arr\ swap$$

$$\ggg arr\ g \ggg second\ (init\ j)) \ggg arr\ assoc \ggg arr\ (id \times swap))$$

composition of $arr, swap\ .\ assoc^{-1} = assoc\ .\ (swap \times id)\ .\ assoc^{-1}\ .\ (id \times swap)$

$$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ f \ggg second\ (init\ i)) \ggg arr\ assoc^{-1}$$

$$\ggg first\ (arr\ swap \ggg arr\ g \ggg second\ (init\ j)) \ggg arr\ assoc$$

$$\ggg arr\ (id \times swap))$$

definition of *second*

$$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ f \ggg arr\ swap \ggg first\ (init\ i)$$

$$\ggg arr\ swap) \ggg arr\ assoc^{-1} \ggg first\ (arr\ swap \ggg arr\ g$$

$$\ggg second\ (init\ j)) \ggg arr\ assoc \ggg arr\ (id \times swap))$$

definition of *first*

$$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ f \ggg arr\ swap \ggg first\ (init\ i)$$

$$\ggg arr\ swap) \ggg arr\ assoc^{-1} \ggg arr\ swap \ggg second\ (arr\ swap \ggg arr\ g$$

$$\ggg second\ (init\ j)) \ggg arr\ swap \ggg arr\ assoc \ggg arr\ (id \times swap))$$

functor and extension of *second*

$$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg second\ (first\ (init\ i))$$

$$\ggg arr\ (id \times swap) \ggg arr\ assoc^{-1} \ggg arr\ swap \ggg arr\ (id \times swap)$$

$$\ggg second\ (arr\ g) \ggg second\ (second\ (init\ j)) \ggg arr\ swap \ggg arr\ assoc$$

$$\ggg arr\ (id \times swap))$$

Lemma B.1.1

$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg second\ (first\ (init\ i))$

$\ggg arr\ (id \times swap) \ggg arr\ assoc^{-1} \ggg arr\ swap \ggg arr\ (id \times swap)$

$\ggg second\ (arr\ g) \ggg arr\ revjuggle \ggg second\ (second\ (init\ j))$

$\ggg arr\ revjuggle \ggg arr\ swap \ggg arr\ assoc \ggg arr\ (id \times swap))$

composition of $arr, id = (id \times swap)\ .\ assoc\ .\ swap\ .\ revjuggle$, identity of arr

$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg second\ (first\ (init\ i))$

$\ggg arr\ (id \times swap) \ggg arr\ assoc^{-1} \ggg arr\ swap \ggg arr\ (id \times swap)$

$\ggg second\ (arr\ g) \ggg arr\ revjuggle \ggg second\ (second\ (init\ j)))$

composition of arr, $assoc\ .\ swap = (id \times swap)\ .\ swap\ .\ assoc^{-1}\ .\ (id \times swap)$

$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg second\ (first\ (init\ i))$

$\ggg arr\ (assoc\ .\ swap) \ggg second\ (arr\ g) \ggg arr\ revjuggle$

$\ggg second\ (second\ (init\ j)))$

Lemma B.1.2

$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg arr\ (assoc\ .\ swap)$

$\ggg first\ (init\ i) \ggg second\ (arr\ g) \ggg arr\ revjuggle$

$\ggg second\ (second\ (init\ j)))$

commutativity

$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg arr\ (assoc\ .\ swap)$

$\ggg second\ (arr\ g) \ggg first\ (init\ i) \ggg arr\ revjuggle$

$\ggg second\ (second\ (init\ j)))$

Lemma B.1.3

$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg arr\ (assoc\ .\ swap)$

$$\ggg \mathit{second}\ (\mathit{arr}\ g) \ggg \mathit{arr}\ \mathit{revjuggle} \ggg \mathit{second}\ (\mathit{first}\ (\mathit{init}\ i))$$

$$\ggg \mathit{arr}\ \mathit{revjuggle} \ggg \mathit{arr}\ \mathit{revjuggle} \ggg \mathit{second}\ (\mathit{second}\ (\mathit{init}\ j)))$$

composition of $\mathit{arr}, \mathit{id} = \mathit{revjuggle}\ .\ \mathit{revjuggle}$, identity of arr

$$= \mathit{loop}\ (\mathit{arr}\ (\mathit{swap}\ .\ \mathit{assoc}^{-1}) \ggg \mathit{second}\ (\mathit{arr}\ (\mathit{swap}\ .\ f)) \ggg \mathit{arr}\ (\mathit{assoc}\ .\ \mathit{swap})$$

$$\ggg \mathit{second}\ (\mathit{arr}\ g) \ggg \mathit{arr}\ \mathit{revjuggle} \ggg \mathit{second}\ (\mathit{first}\ (\mathit{init}\ i))$$

$$\ggg \mathit{second}\ (\mathit{second}\ (\mathit{init}\ j)))$$

extension of second, composition of arr

$$= \mathit{loop}\ (\mathit{arr}\ (\mathit{revjuggle}\ .\ (\mathit{id} \times g)\ .\ \mathit{assoc}\ .\ \mathit{swap}\ .\ \mathit{id} \times (\mathit{swap}\ .\ f)\ .\ \mathit{swap}\ .\ \mathit{assoc}^{-1})$$

$$\ggg \mathit{second}\ (\mathit{first}\ (\mathit{init}\ i)) \ggg \mathit{second}\ (\mathit{second}\ (\mathit{init}\ j)))$$

$\mathit{assoc}'\ (\mathit{juggle}'\ (g \times \mathit{id})\ .\ (f \times \mathit{id})) = \mathit{revjuggle}\ .\ (\mathit{id} \times g)\ .$

$$\mathit{assoc}\ .\ \mathit{swap}\ .\ \mathit{id} \times (\mathit{swap}\ .\ f)\ .\ \mathit{swap}\ .\ \mathit{assoc}^{-1}$$

$$= \mathit{loop}\ (\mathit{arr}\ (\mathit{assoc}'\ (\mathit{juggle}'\ (g \times \mathit{id})\ .\ (f \times \mathit{id}))) \ggg \mathit{second}\ (\mathit{first}\ (\mathit{init}\ i))$$

$$\ggg \mathit{second}\ (\mathit{second}\ (\mathit{init}\ j)))$$

functor of second

$$= \mathit{loop}\ (\mathit{arr}\ (\mathit{assoc}'\ (\mathit{juggle}'\ (g \times \mathit{id})\ .\ (f \times \mathit{id}))) \ggg \mathit{second}\ (\mathit{first}\ (\mathit{init}\ i)$$

$$\ggg \mathit{second}\ (\mathit{init}\ j)))$$

product of init

$$= \mathit{loop}\ (\mathit{arr}\ (\mathit{assoc}'\ (\mathit{juggle}'\ (g \times \mathit{id})\ .\ (f \times \mathit{id}))) \ggg \mathit{second}\ (\mathit{init}\ (i, j)))$$

□