UNIVERSITY OF TARTU
Faculty of Mathematics and Computer Science
Institute of Computer Science

Jaak Randmets

ABSTRACT MACHINE FOR A COMONADIC DATAFLOW LANGUAGE

Master's Thesis (30 ECTS)

Supervisor: *prof.* Varmo Vene

Author: . . . . "…" May 2012
Supervisor: . . . . "…" May 2012

Allowed for defence
Professor: . . . . "…" May 2012

Tartu 2012

Contents

Chapter 1

Introduction

The conventional approach to programming languages is to manipulate individual primitive values one at a time. For example, a programming language might have concept of integers, objects, or functions, but they are almost never manipulated in bulk by the language operations. Collections of primitive values and operations on them are mostly implemented using the language facilities themselves. A class of exceptions are array processing languages, and data-parallel languages that have arrays as a primitive data and many of the language primitives operate point-wise on them. Other class of exceptions are dataflow languages that manipulate primitive concepts of the language as infinite streams of values. As such, the types in a dataflow language denote streams of types, literals of the language are constant streams of values, and primitive operations manipulate input streams point-wise. Dataflow languages can be used to model electronic circuits and real-time systems for hardware control. This work focuses on dataflow languages.

Higher-order functions are a powerful method for abstraction in functional languages. In dataflow languages they appear to cause some issues. Either the language only implements first order functions or for example, in Lucid Synchrone the programmer has to explicitly state if the functional parameter is dataflow dependant. The approach taken by Lucid Synchrone also implies orthogonality of data types and dataflow types. Reactive programming languages could potentially overcome those issues but, to our knowledge, so far only higher-order reactive programming languages are implemented as embedded domain-specific languages in Haskell, and thus require the dataflow types to be specified explicitly.

Semantics for higher-order dataflow languages, and reactive programming languages exist, but to our knowledge practical non-embedded implementations do not. Earliest instances of semantics to higher-order dataflow languages were based on the concept of arrows [Hug00]. Uustalu and Vene proposed in [UV05] that arrows are overly general for this purpose, and demonstrated that semantics to higher-order dataflow languages can be based on a simpler concept of comonads. The semantics to functional reactive programming languages has still been evolving around arrows, especially as the category-theoretic understanding of them has continuously been improving. As of late it has been shown that linear temporal logic types functional reactive programs [Jef12]. Curry-Howard correspondence between linear temporal logic and functional reactive programming has been established.

Both the functional reactive languages embedded in Haskell and comonadic dataflow language implementations have yet to enjoy particularly efficient implementations. Both suffer from issues with large and opaque memory consumption, and the comonadic semantics additionally has problems with time efficiency due to unnecessary repeated computations.

In order to move closer to an efficient implementation of the comonadic semantics we will develop semantically equivalent abstract machine [HM92]. The abstract machine is closer to hardware, and its behaviour is operationally much simpler than the behaviour of the denotational semantics. The abstract machine is intuitively a simple term rewriting system.

The reader is expected to be familiar with basics of the non-strict functional programming language Haskell [M^{+}10], and the basics of the theory of programming language semantics. Language evaluators, various program transformations and examples are presented in Haskell. While strictly not required it helps if reader is familiar with the concept of monads, arrows, or comonads. Familiarity with λ-calculus, continuation-passing style transformation, thunk-based simulation, and defunctionalization could also be of help but is not required.

1.1 Outline

In Chapter 2 we give an overview of comonadic dataflow languages. We start by presenting a short introduction to programming in a dataflow language Lucid Synchrone to provide some context. Next, we will introduce the notion of comonads, and present a comonadic evaluator for a higher-order non-strict dataflow language.

The Chapter 3 presents the methodology developed by Ager, Biernacki, Danvy, and Midtgaard for deriving abstract machines from denotational evaluators in [ABDM03]. We start by defining the notion of abstract machines based on [HM92] and present two abstract machines for evaluating λ-calculus terms as examples. The first is the CEK abstract machine for call-by-value evaluation, and the other is the Krivine's abstract machine for call-by-name evaluation. The Danvy's methodology uses two major code transformations. First is the continuation-passing style transformation, and the second is the defunctionalization. We shall introduce these, and the thunk-based simulation in Section 3.2. Finally, we will derive an abstract machine for an extremely simple expression-language evaluator to demonstrate the methodology.

The Chapter 4 focuses entirely on deriving an abstract machine for the comonadic evaluator defined in the second chapter. The derivation goes through the following steps: *a*) higher-order functions in value domain are eliminated, and become closures (function objects) via defunctionalization; *b*) the evaluator is simplified and the comonad specific constructs are inlined; *c*) call-by-name behaviour is simulated via thunks; *d*) call-by-value continuation-passing style transformation is applied to reach a tail-recursive evaluator; and *e*) higher-order functions are eliminated by defunctionalization. The call-by-value continuation-passing style transformation can be applied because thunk-based simulation results in evaluation order indifferent program. Finally, the abstract machine, which we show is equivalent to the final evaluator, is presented. Each evaluator in the derivation sequence is equivalent to the previous one, and thus the final abstract machine is equivalent to the initial denotational evaluator. The equivalence of each evaluator to the previous one is due to the correctness of the used program transformations.

1.2 Contributions

As a major result we present an abstract machine for a higher-order non-strict dataflow language. The abstract machine can be a basis for future efficient implementations, and gives insight to the operational behaviour of the language programs. The second contribution is that we demonstrate that the Danvy's methodology [ABDM03] can be applied to evaluators implemented in a non-strict host language. While Danvy has shown in [Dan04] that the methodology results in both the CEK, and the Krivine's machine if applied to call-by-name evaluator implemented in a strict host language. We will propose that the same result is reached from evaluator implemented in a non-strict language. This result is significant because the implementation of a call-by-name evaluator is much simpler and corresponds more directly to the denotational semantics in a call-by-name host language.

The resulting abstract machine for the non-strict dataflow language is similar to the Krivine's abstract machine. Compared to the Krivine's machine our machine has two extra rules for evaluating the additional dataflow language construct, and uses overloaded notion of environment updating and environment lookup. The two extra rules are simple and have constant-time behaviour. While this result is not surprising it does give additional guarantee of correctness. We do not present operational semantics to the dataflow language, but the similarity of the abstact machine to the Krivine's machine suggests that the operational semantics are too similar to the standard call-by-name λ-calculus operational semantics.

Chapter 2

Comonadic dataflow languages

In this chapter we give an overview of the semantics of a comonadic dataflow language. We start with giving an overview of dataflow languages, comonads, and then we present a denotational comonadic semantics for a simple causal higher-order dataflow language. The semantics are based on the works by Uustalu and Vene in [UV05].

2.1 Dataflow languages

Where conventional programming languages manipulate individual values, dataflow languages on the other hand are designed to manipulate infinite streams of data, which can be viewed as a discrete-time signals. In this section we give short overview of dataflow languages based on some examples of Lucid Synchrone [Pou06].

Lucid Synchrone is functional, strict, and strongly typed Objective Caml like programming language that manipulates infinite sequences as primitive values. The language is extended with many dataflow primitives, and the type system is extended to support the added primitives. For example, at the type system level the notion of *clocks* supplies a way to specify different execution rates in a program. In this section we explore some simpler concepts of the language in order to gain some intuition in dataflow programming.

Because this section acts as a basic introduction to dataflow languages we will completely ignore more advanced features of the language. For example, we will gloss over the type system entirely by only looking at primitive types, and focus on some of the simpler dataflow features of the language.

2.1.1 Point-wise operations

Every type and every scalar values in Lucid Synchrone is implicitly lifted to streams. For instance:

- `int` stands for the type of integer streams;
- language literal expressions stand for constant streams of values;
- pairs of values denote streams of pairs; and
- arithmetic expressions, such as `x + y`, operate point-wise on input streams.

Program execution can be represented as *chronograms*, showing the sequence of values taken by streams during the execution. The chronogram below shows five streams. The first column denotes the expression, and rest of the columns denote the values that the expression takes. Below we have defined stream of booleans c, two integer streams x and y, and an if-expression.

Expression	Value					
c	**true**	**false**	**true**	**true**	**false**	...
x	x_0	x_1	x_2	x_3	x_4	...
y	y_0	y_1	y_2	y_3	y_4	...
x + y	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	...
if c **then** x **else** y	x_0	y_1	x_2	x_3	y_4	...

The let-expression is used to define new streams and functions. For example, to define two constant streams, and a function to average two input streams point-wise we could write:

Listing 2.1: Simple streams

```
let dt = 0.001
let g = 9.81
let average (x,y) = (x + y) / 2
```

Function application is denoted with space, and to compute the point-wise average of streams dt, and g we can write average (dt,g). Blocks of mutually recursive functions can be defined using let-expression with the **rec**, and **and** keywords:

Listing 2.2: Mutual recursion

```
let rec odd  n = if n == 0 then false else even (n - 1)
    and even n = if n == 0 then true  else odd  (n - 1)
```

So far we have only seen how to define new constant streams, and how to manipulate streams point-wise. In order to escape the world of constant streams we will introduce some dataflow language specific constructs.

2.1.2 Delays

Lucid Synchrone is causal dataflow language: values that a function takes may only depend on either the current, or previous values of input streams. The language supports many operators to introduce stream delays. Most important for us is the *followed-by* operator. The expression x **fby** y reads as "x followed by y", and takes first value of the stream x and after that a previous value of stream y. In another words, it delays y by a single instant, and is initialized by the first value of x. The following chronogram demonstrates the behaviour of the followed-by construct:

x	x_0	x_1	x_2	x_3	x_4	...
y	y_0	y_1	y_2	y_3	y_4	...
x **fby** y	x_0	y_0	y_1	y_2	y_3	...

Lucid Synchrone also has a conceptually simpler delay operation **pre** that delays an input stream by a single instant. The expression **pre** x, reads as "previous x", is undefined at the first position and at every other position has the value of stream x at the previous position. The following chronogram shows the effect of the previous-operator:

x	x_0	x_1	x_2	x_3	x_4	$\ldots$
pre x	$\perp$	x_0	x_1	x_2	x_3	$\ldots$

A static analysis is used to guarantee that undefined values introduced by **pre** are never used directly.

To eliminate the undefined behaviour introduced by previous-operator an operator is supplied to replace the first element of an input stream with a well defined value:

x	x_0	x_1	x_2	x_3	x_4	$\ldots$
y	y_0	y_1	y_2	y_3	y_4	$\ldots$
x -> y	x_0	y_1	y_2	y_3	y_4	$\ldots$

The followed-by expression `x fby y` can now be redefined as `x -> pre y`.

We call an expressions *sequential* if it may produce time evolving value when the free variables are kept constant. Otherwise, the expression is call *combinatorial*. While there are other methods to construct sequential expressions for our needs the expression is combinatorial if it does not contain any of the delay operators. In Lucid Synchrone it is required to annotate let-expressions that introduce sequential functions into global scope with **node** keyword. Sequential non-functional values may not be bound with a let-expression in the global scope.

2.1.3 Examples

At this point we are equipped with powerful enough tools to define more interesting streams. One of the simplest examples is the stream of natural numbers. While we can not directly declare a stream of integers, we overcome this by defining the stream as a sequential function from the unit type to integers.

To define stream of integers we need to exploit recursion. Stream of natural numbers starts with value 0, and is followed by stream of natural numbers with 1 larger value. This gives us rather straightforward implementation:

Listing 2.3: Stream of natural numbers

```
let node nat () = nat' where
    rec nat' = 0 fby (nat' + 1)
```

Note the use of the **where** keyword to define local variables.

Another standard example is stream of factorials of natural numbers, which can be defined as 1 followed by factorials multiplied by the stream of natural numbers that starts with 1:

Listing 2.4: Factorials

```
let node fact () = fact' where
    rec fact' = 1 fby (fact' * (nat () + 1))
```

As for slightly more complicated example we will define a stream of Fibonacci numbers. The stream starts with 1, and the value of every other position is the sum of previous two. To compute the sum of two previous positions we add the stream to 0 followed by the original stream. The constant 0 acts as an element before the initial value 1.

Listing 2.5: Fibonacci numbers

```
let node fib () = fib' where
    rec fib' = 1 fby (fib' + (0 fby fib'))
```

To demonstrate how `fib` is computed we show the intermediate values that its subexpressions take in the following table:

`fib'`	1	1	2	3	5	8	...
`0 fby fib'`	0	1	1	2	3	5	...
`fib' + (0 fby fib')`	1	2	3	5	8	13	...
`1 fby (fib' + (0 fby fib'))`	1	1	2	3	5	8	...

2.2 Comonads

Monad is a well known abstraction in functional programming community, and as shown by Moggi [Mog90, Mog89], and Wadler [Wad92] can be used to provide additional (effectful) structure for denotational semantics. However, its category theoretic dual, comonad, has not enjoyed nearly as much use. Monads are widely used to represent effectful computations and input/output in Haskell, but there is no well known practical use for its dual. While various context dependent computations are known to have a comonadic structure the general abstraction is almost never needed in practise.

In Haskell comonads can be represented as the following type class:

class *Comonad w* **where**
  $counit :: w\ a \to a$
  $cobind :: (w\ a \to b) \to w\ a \to w\ b$

The class declaration states that each instance of comonad has to define function *counit*, which is used to retrieve a value from comonadic structure, and *cobind*, that is used to lift comonadic functions. Functions of type $w\ a \to b$, where w is a comonad, are known as co-Kleisli arrows from type a to type b.

The type class requires programmer to supply properly typed functions for every comonad instance, but does not express the laws the functions have to abide to. It's programmers responsibility that for every defined comonad w, and for every function $f :: w\ b \to c$, and $g :: w\ a \to b$ the following three laws hold:

$$\begin{aligned} &cobind\ counit &&\equiv id \\ &counit \circ cobind\ f &&\equiv f \\ &cobind\ f \circ cobind\ g &&\equiv cobind\ (f \circ cobind\ g) \end{aligned}$$

Concrete examples of comonads include: identity comonad, product comonad, costate comonad, and non-empty list comonad. Surprisingly cellular automaton can be viewed as a comonad [Pip06, CU10], and non-empty arrays with one element focused can also be viewed as such [Pip08]. In fact, in Haskell, every functor gives rise to a comonad via the cofree comonad construction.

2.2.1 Product comonad

The product data type:

$$\textbf{data}\ \textit{Prod e a} = \textit{Prod e a}$$

is a classic example that give rise to a family of comonads that can be easily interpreted as a value in a context. The implementation is symmetrical, but the first component can be viewed as the context, and second component as the value.

For every type *e* the partial application *Prod e* defines a comonad:

```
instance Comonad (Prod e) where
  counit (Prod _ x) = x
  cobind d (Prod e x) = Prod e (d (Prod e x))
```

The unit operation extract the value from context, and the binding operation applies the function to the input product, and inserts the result back into the context supplied by the input.

The proofs for the comonad laws are easy for the first two, and simply account to expanding the definitions of *counit* and *cobind*. For first law we substitute the right-hand side of *counit* and then *cobind* as follows:

$$\begin{aligned}
&\textit{cobind counit}\ (\textit{Prod e x}) \\
\equiv\ &\textit{Prod e}\ (\textit{counit}\ (\textit{Prod e x})) \\
\equiv\ &\textit{Prod e x}
\end{aligned}$$

For the second rule we do the same in reverse order:

$$\begin{aligned}
&\textit{counit}\ (\textit{cobind f}\ (\textit{Prod e x})) \\
\equiv\ &\textit{counit}\ (\textit{Prod e}\ (f\ (\textit{Prod e x}))) \\
\equiv\ &f\ (\textit{Prod e x})
\end{aligned}$$

The correctness of the last rule is somewhat more complicated to show. First, we will expand both *cobind* operations, and reintroduce the innermost *cobind*. The second step is to introduce function composition of *f*, and *cobind g* by rewriting the right-hand side of function composition operator with the left side. Finally, the outermost *cobind* is reintroduced by replacing the right-hand side of the definition with the left giving exactly what is required.

$$\begin{aligned}
&\textit{cobind f}\ (\textit{cobind g}\ (\textit{Prod e x})) \\
\equiv\ &\textit{cobind f}\ (\textit{Prod e}\ (g\ (\textit{Prod e x}))) \\
\equiv\ &\textit{Prod e}\ (f\ (\textit{Prod e}\ (g\ (\textit{Prod e x})))) \\
\equiv\ &\textit{Prod e}\ (f\ (\textit{cobind g}\ (\textit{Prod e x}))) \\
\equiv\ &\textit{Prod e}\ ((f \circ \textit{cobind g})\ (\textit{Prod e x})) \\
\equiv\ &\textit{cobind}\ (f \circ \textit{cobind g})\ (\textit{Prod e x})
\end{aligned}$$

Note that we can't skip expanding the innermost *cobind g*, as to expand the outer *cobind f* we need to know that it was passed value constructed with the *Prod* data constructor.

It's important to note that this proof is not quite sufficient as we have not accounted for undefined values. Luckily, the proof is straightforward to extend, and we shall not go through this.

While strictly not always required due to [DHJG06] we will expect that the data structures are defined, and the comonad instance implemented, in such manner that the laws hold for undefined values too. An example of an incorrect definition of the lifting operator for the product comonad would be:

```
cobind d p = Prod e (d p)
  where Prod e x = p
```

because *cobind counit* $\perp$ would evaluate to *Prod* $\perp$ $\perp$ which, in Haskell, is distinguishable from undefined values.

Every comonad has to be equipped with operations that distinguish it from others. For product comonad we can ask for the context, and perform local computations on it:

```
ask :: Prod e a → e
ask (Prod e _) = e
local :: (e → e′) → Prod e a → Prod e′ a
local f (Prod e a) = Prod (f e) a
```

2.2.2 Non-empty list comonad

Non-empty lists are a more complex example of comonads. They can be constructed by either taking a single value to construct a singleton list, or appending a value to already existing non-empty list giving list of at least two elements long. This can be represented by the following data structure:

```
data LV a = One a | Cons a (LV a)
```

The data structure either constructs a singleton list tagged with *One*, or longer list from already constructed non-empty list with data constructor *Cons*. This is very close to the definition of regular lists with the difference that instead of constructing an empty list we construct a singleton list.

The unit of comonad can be defined because it's always possible to extract one element from a non-empty list. The unit extracts the first value:

```
instance Comonad LV where
  counit (One x)       = x
  counit (Cons x xs)   = x
```

The definition for lifting is more complicated, but it simply constructs a new non-empty list by applying the argument function to all the tails of the input list. Resulting list is of same length as the input list.

```
  cobind d (One x)     = One  (d (One x))
  cobind d (Cons x xs) = Cons (d (Cons x xs)) (cobind d xs)
```

Proofs for the laws are by induction on the lists length, and take some simple equational reasoning. We only prove the first law, but the rest follow in quite similar vein.

Proposition 1. *For non-empty lists cobind counit ≡ id.*

Proof. For bottom-valued lists the equation holds due to strictness of cobind. We show that the claim holds for singleton lists. Let x be an arbitrary value of some type a.

$$\begin{aligned}
& \mathit{cobind}\ \mathit{counit}\ (\mathit{One}\ x) \\
\equiv\ & \quad \{ \text{ definition of cobind } \} \\
& \mathit{One}\ (\mathit{counit}\ (\mathit{One}\ x)) \\
\equiv\ & \quad \{ \text{ definiton of counit } \} \\
& \mathit{One}\ x
\end{aligned}$$

Assume that the proposition holds for a non-empty list $xs :: LV\ a$. Let x be a value of type a. We will show that the proposition holds for list $\mathit{Cons}\ x\ xs$ by the following reasoning:

$$\begin{aligned}
& \mathit{cobind}\ \mathit{counit}\ (\mathit{Cons}\ x\ xs) \\
\equiv\ & \quad \{ \text{ definition of cobind } \} \\
& \mathit{Cons}\ (\mathit{counit}\ (\mathit{Cons}\ x\ xs))\ (\mathit{cobind}\ \mathit{counit}\ xs) \\
\equiv\ & \quad \{ \text{ definition of counit } \} \\
& \mathit{Cons}\ x\ (\mathit{cobind}\ \mathit{counit}\ xs) \\
\equiv\ & \quad \{ \text{ induction assumption } \} \\
& \mathit{Cons}\ x\ xs
\end{aligned}$$

□

Every comonad has to be associated with comonad specific operations. For non-empty lists we define so called *followed-by* construct:

```
fby :: a → LV a → a
fby x (One y)       = x
fby x (Cons y ys) = counit ys
```

The meaning of the construct might be difficult to extract, but it will become clear if we look at an example of its use. The expression *cobind* (*fby x*), for example, shifts the argument list to left, and replaces the last element with x. Let us view a sequence of values $x_1\ x_2\ \dots\ x_n$ in reverse as $\mathit{Cons}\ x_n\ (\ldots(\mathit{Cons}\ x_2\ (\mathit{One}\ x_1)))$. Using such view we can represent the behaviour of the example as follows:

y	y_0	y_1	y_2	y_3	y_4
cobind (*fby x*) *y*	x	y_0	y_1	y_2	y_3

We define two additional functions. First is *cmap* for applying a function f over every element of the input list. It is defined as *cobind* ($f \circ \mathit{counit}$), but we give it more explicit but equivalent definition:

```
cmap :: (a → b) → LV a → LV b
cmap f (One x)       = One  (f x)
cmap f (Cons x xs) = Cons (f x) (cmap f xs)
```

The second function is for zipping two lists together, and is defined as follows:

```
czip :: LV a → LV b → LV (a, b)
czip (Cons x xs) (Cons y ys) = Cons (x, y) (czip xs ys)
czip xs          ys          = One (counit xs, counit ys)
```

As the zipping function defined in the Haskell prelude the *czip* for non-empty lists discards elements from the longer one.

2.3 Comonadic denotational semantics for a dataflow language

In this section we present an evaluator for a causal higher-order dataflow language. The evaluator is equivalent to that presented in the works by Uustalu and Vene [UV05], but is slightly simplified. The first simplification is that of the data structure, and the second is elimination of a type class as we are only dealing with a single evaluator instead of multiple.

The language is based on lambda calculus and has variables, integer literals, lambda-expressions, function application, and additionally *followed-by* construction to introduce stream delays. The language is not given concrete syntax, and is simply represented in Haskell by the following abstract syntax tree:

```
data Term
  = Lit  Int
  | Var  String
  | Lam String Term
  | App  Term  Term
  | Fby  Term  Term
```

Everything but the *followed-by* construction is a standard for a lambda-calculus. The language does not include fixed point combinators as those can be implemented in a non-typed setting easily.

The value domain of the language is composed of integers, and functions mapping values to values. Because the semantics of the terms are given in a co-Kleisli category the functions need to be co-Kleisli arrows of type *LV Value* → *Value*. The data constructor *LV* is one for non-empty lists defined previously in Subsection 2.2.2. The data type to represent the value domain of the language is defined in Haskell as follows:

```
data Value
  = I  Int
  | F (LV Value → Value)
```

The environment maps variables to values, and is implemented as a list of pairs composed of a variable, and a value:

```
type Env = [(String, Value)]
```

We define generic functions for looking up variables, and for inserting values into the environment. To map a variable to a value the variable-value pair is simply inserted into beginning of the list. We also provide uncurried version of *insert* and name it *repair*:

$$
\begin{array}{l}
\mathit{insert} :: k \to v \to [(k,v)] \to [(k,v)] \\
\mathit{insert}\ x\ v\ e = (x,v) : e \\
\mathit{repair} :: k \to (v, [(k,v)]) \to [(k,v)] \\
\mathit{repair}\ x = \mathit{uncurry}\ (\mathit{insert}\ x)
\end{array}
$$

The operator ! is used to look up inserted values from the environment. The function compares the variable to the first components of the pairs stored in the environment until a match is found. The second component of the first match is returned:

$$
\begin{array}{lll}
(!) :: \mathit{Eq}\ k \Rightarrow [(k,v)] \to k \to v & & \\
[\,] \qquad\qquad\ !\, x & & = \mathit{error}\ \texttt{"Type error!"} \\
((k,v) : \mathit{kvs})\ !\, x & \mid k \equiv x & = v \\
 & \mid \mathit{otherwise} & = \mathit{kvs}\ !\, x
\end{array}
$$

If the variable is not found in the environment a type error is raised. Those run-time errors can be eliminated by performing a simple static check to guarantee that all used variables are within scope. The variable lookup and insertion are implemented such that the same variable can be inserted multiple times into the same environment, and the latest inserted one will be returned by the lookup function. This gives us variable shadowing.

The denotation of terms are given by co-Kleisli arrow from environment to value. The evaluator follows from [UV05], and directly corresponds to the denotational semantics:

$$
\begin{array}{ll}
\mathit{eval} :: \mathit{Term} \to \mathit{LV}\ \mathit{Env} \to \mathit{Value} & \\
\mathit{eval}\ (\mathit{Lit}\ n) & \mathit{de} = I\ n \\
\mathit{eval}\ (\mathit{Var}\ x) & \mathit{de} = \mathit{counit}\ \mathit{de}\ !\, x \\
\mathit{eval}\ (\mathit{Lam}\ x\ t) & \mathit{de} = F\ (\lambda \mathit{dv} \to \mathit{eval}\ t\ (\mathit{repair}\ x\ \text{`}\mathit{cmap}\text{`}\ \mathit{czip}\ \mathit{dv}\ \mathit{de})) \\
\mathit{eval}\ (\mathit{App}\ t_0\ t_1) & \mathit{de} = \mathit{unF}\ (\mathit{eval}\ t_0\ \mathit{de})\ (\mathit{cobind}\ (\mathit{eval}\ t_1)\ \mathit{de}) \\
\mathit{eval}\ (\mathit{Fby}\ t_0\ t_1) & \mathit{de} = \mathit{fby}\ (\mathit{eval}\ t_0\ \mathit{de})\ (\mathit{cobind}\ (\mathit{eval}\ t_1)\ \mathit{de})
\end{array}
$$

The comonadic operations *counit*, *cobind*, *cmap*, *czip* and *fby* have been defined previously in Subsection 2.2.2.

The helper function *unF* is used to unwrap function from value data type. The operation may fail if a non-function valued expression is used in application, but all such cases can be caught by a type checker.

$$
\begin{array}{l}
\mathit{unF} :: \mathit{Value} \to (\mathit{LV}\ \mathit{Value} \to \mathit{Value}) \\
\mathit{unF}\ (I\ n) = \mathit{error}\ \texttt{"Type error!"} \\
\mathit{unF}\ (F\ f) = f
\end{array}
$$

To compute a value of a term e at some stream position i we apply the evaluator to the term e and a length i non-empty constant list of initial environments. The symbol ε is used to denote the initial environment.

$$
\begin{array}{l}
\mathit{runEval} :: \mathit{Term} \to \mathit{Int} \to \mathit{Value} \\
\mathit{runEval}\ t\ i = \mathit{eval}\ t\ (\varepsilon \uparrow i) \\
(\uparrow) :: a \to \mathit{Int} \to \mathit{LV}\ a \\
x \uparrow 1 = \mathit{One}\ \ x \\
x \uparrow n = \mathit{Cons}\ x\ (x \uparrow (n-1))
\end{array}
$$

2.3.1 Code examples

In order to demonstrate the evaluator we are going to define the initial environment with some primitive operations predefined. The operations are represented as functional values.

```
ε :: Env
ε = [("add", binOp (+)), ("mul", binOp (*))]
```

The helper function *binOp* is used to lift arbitrary binary integer functions into value domain.

```
binOp :: (Int → Int → Int) → Value
binOp op = F (λdv₀ → F (λdv₁ → I (unI (counit dv₀) ‘op‘ unI (counit dv₁))))
  where
    unI (I x)  = x
    unI (F f) = error "Type error!"
```

While it’s possible to construct some simple programs with given terms, and primitive operations; we need a fixed-point combinator to define recursive programs. Because the language is not typed it’s possible to define the fixed-point operator using existing terms. The fixed point term corresponds the lambda calculus **Y** combinator $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, and is defined by the following term:

```
Y :: Term
Y = Lam "f" (App
  (Lam "x" (App (Var "f") (App (Var "x") (Var "x"))))
  (Lam "x" (App (Var "f") (App (Var "x") (Var "x")))))
```

A simple example of dataflow program that can not be defined without recursion is one to compute the stream of natural numbers. The term to do just that can be defined as follows:

```
nat :: Term
nat = App Y (Lam "nat" (Fby (Lit 0)
  (App (App (Var "add") (Lit 1)) (Var "nat"))))
```

To run the program we evaluate expression *map* (*runEval nat*) [1 . .], which evaluates to infinite list of values [*I* 1, *I* 2, . .].

The terms presented directly in Haskell using the abstract syntax tree are quite difficult to read. In the rest of this section we are going to use some syntactic sugar. The previously defined terms in the new form look as follows:

```
Y   = λf. (λx. f (x x)) (λx. f (x x))
nat = Y (λnat. Fby 0 (add 1 nat))
```

The lambda function constructors have been presented in a nicer form. Application constructors, variable constructors, and literal constructors have been removed. It’s always clear from the context which constructor has been applied. Parentheses have been used to resolve the cases where function application constructors could be applied ambiguously. The function application is left-associative.

Factorial function can be defined by reusing the stream of natural numbers as follows:

$$\begin{aligned}&\mathit{fact} :: \mathit{Term}\\&\mathit{fact} = \mathbf{Y}\ (\lambda\,\texttt{fact}\,.\ \mathit{Fby}\ 1\ (\texttt{mul fact}\ (\texttt{add}\ \mathit{nat}\ 1)))\end{aligned}$$

Finally, to complete the examples presented in the section about data-flow languages here follows the term to compute stream of Fibonacci numbers:

$$\begin{aligned}&\mathit{fib} :: \mathit{Term}\\&\mathit{fib} = \mathbf{Y}\ (\lambda\,\texttt{fib}\,.\ \mathit{Fby}\ 1\ (\texttt{add fib}\ (\mathit{Fby}\ 0\ \texttt{fib})))\end{aligned}$$

Note that the examples presented here correspond exactly to the examples presented in the Section 2.1 about dataflow languages.

Chapter 3

From evaluator to abstract machine

3.1 Abstract machines

There are many definitions of abstract machines, and countless different abstract machines following any of these definitions. Well known examples of abstract machines are the CEK machine [FF86], Krivine's machine [Kri85], CLS machine [HM92], SECD machine [Lan64], and STG machine [PJS89]. We follow the definition by Hannan and Miller [HM92] which states that an abstract machine is a term rewriting system with a strict initialization condition and halting conditions, and rewrite rules that: *a*) are deterministic; *b*) are always applied to a term at its root; and *c*) must not contain repeated variables in the left.

We also distinguish abstract machines from virtual machines. The latter operate on an instruction set while the former directly on a source language terms. Virtual machines have an additional compilation step before evaluation.

Following our definition a set of mutually tail recursive functions, operating on language terms, under strict evaluation, is an abstract machine. If the set of mutually recursive functions is evaluator for some language, and we can prove that initial evaluator is equivalent to said set of recursive functions then we can say that the evaluator is equivalent to the corresponding abstract machine.

3.1.1 The CEK abstract machine

First example is the CEK abstract machine which evaluates lambda calculus terms with call-by-value strategy. The language terms are denotes with t, variables with x, expressible values with v, evaluation contexts with k, and environments with *env*. The expressible values are closures consisting of a variable, a term, and an environment. The evaluation context is either: *a*) a stop marker; *b*) a function marker storing a closure value, and an evaluation context; or *c*) an argument marker consisting of a term, an environment, and an evaluation context. Environment e maps variables to expressible values. Terms, values, and contexts are described by the following abstract syntax:

$$\begin{array}{rcl} t & ::= & x \mid \lambda x.t \mid t_0\, t_1 \\ v & ::= & [x, t, e] \\ k & ::= & \texttt{stop} \mid \texttt{fun}(v, k) \mid \texttt{arg}(t, e, k) \end{array}$$

The CEK machine has initial transition, final transition, three evaluation rules, and two continuation rules for reducing the evaluation context:

t	$\Rightarrow_{eval}$	$\langle t, nil, \texttt{stop}\rangle$
$\langle x, e, k\rangle$	$\Rightarrow_{eval}$	$\langle k, e(x)\rangle$
$\langle \lambda x.t, e, k\rangle$	$\Rightarrow_{eval}$	$\langle k, [x, t, e]\rangle$
$\langle t_0\ t_1, e, k\rangle$	$\Rightarrow_{eval}$	$\langle t_0, e, \texttt{arg}(t_1, e, k)\rangle$
$\langle \texttt{arg}(t_1, e, k), v\rangle$	$\Rightarrow_{cont}$	$\langle t_1, e, \texttt{fun}(v, k)\rangle$
$\langle \texttt{fun}([x, t, e], k), v\rangle$	$\Rightarrow_{cont}$	$\langle t, e[x \mapsto v], k\rangle$
$\langle \texttt{stop}, v\rangle$	$\Rightarrow_{cont}$	v

The three evaluation rules operate on triples composed of a term, an environment, and an evaluation context. The continuation rules operate on pairs composed of an evaluation context, and a value.

The evaluation rule pattern matches on the term structure, and if the term is a function application then the evaluation context is updated to evaluate the function parameter. Otherwise the evaluation rule transitions into reducing the evaluation context with either the closure value in case of lambda abstraction, or the value of the variable in case of the variable expression.

The continuation rules, however, pattern match on the evaluation context. If the evaluation context is a function argument, then the expressible value v is known to be a functional value and the rule transitions into evaluating the argument with the given functional value on the stack. If the context is a function object, then the value is known to be argument to the function and the rule transitions into evaluating the body of the function with environment updated to map the lambda variable to the value.

3.1.2 Krivine's abstract machine

The second example is the Krivine's abstract machine for evaluating lambda terms under call-by-name (CBN) evaluation strategy. Compared to the CEK machine the Krivine's machine operates on lambda terms that represent variables with De Bruijn indices. Variables are expressed with natural numbers denoting the number of in scope lambda binders until one corresponding to the variable. Lambda terms no longer introduce variables by name. Expressible values v are pairs composed of a term, and an environment. Environment e is a sequence of values. Stack of values is denoted with s.

$$\begin{array}{rcl} t & ::= & n \mid \lambda t \mid t_0\ t_1 \\ v & ::= & [t, e] \\ s & ::= & \bullet \mid v :: s \\ e & ::= & nil \mid v :: e \end{array}$$

The abstract machine has transition for initialization, transition for halting, and three transition rules for evaluating the expression.

t	$\Rightarrow$	$\langle t, nil, \bullet\rangle$
$\langle n, e, s\rangle$	$\Rightarrow$	$\langle t, e', s\rangle$ where $[t, e'] = e_n$
$\langle \lambda t, e, [t', e'] :: s\rangle$	$\Rightarrow$	$\langle t, [t', e'] :: e, s\rangle$
$\langle t_0\ t_1, e, s\rangle$	$\Rightarrow$	$\langle t_0, e, [t_1, e] :: s\rangle$
$\langle \lambda t, e, \bullet\rangle$	$\Rightarrow$	$[t, e]$

In case of variable n a pair consisting of a term and an environment is looked up from the currently active environment, and the evaluation is continued with the resulting term and environment. In case of lambda abstraction the parameter is assumed to be stored on the stack, and the environment is simply updated to map to the parameter. In case of function application the parameter is pushed onto the stack with currently active environment, and the evaluation is continued with the function in the unmodified environment.

3.2 Code transformations

The derivation from denotational evaluator to abstract machine uses two non-trivial code transformations. The first transformation is to convert the program into a continuation-passing style resulting in an equivalent tail-recursive program. The second transformation is called defunctionalization and is used to eliminate higher-order functions from the input program. In this section we present both of the transformations in mostly informal manner, and demonstrate the use of them based on simple examples.

3.2.1 CPS transformation

The continuation-passing style (CPS) transformation is a code transformation that converts functions into a form in which the control is passed explicitly in the form of a continuation. The transformation is extremely useful as it converts the function into a tail-recursive, but higher-order, form. This section uses the results by Plotkin from [Plo75], but is mostly based on the works by Hatcliff and Danvy in [HD97].

The CPS transformed program has a fixed control flow, and is indifferent to the evaluation strategy. Informally this means that it does not matter if the transformed program is evaluated under call-by-name, or call-by-value. Depending on which behaviour of the underlying program needs to be preserved either call-by-name, or call-by-value CPS transformation can be applied.

To formally define both of the CPS transformations we consider Λ, the untyped λ-calculus. The λ-calculus term t is either a literal constant c, a variable x, a lambda-abstraction, or a function application.

$$\begin{array}{rcl} t & \in & \Lambda \\ t & ::= & c \mid x \mid \lambda x.t \mid t_0\, t_1 \end{array}$$

The set of values is defined depending on which evaluation strategy the terms are under. Under call-by-name the value v is either a constant, or a lambda abstraction; and under call-by-value the value v is either a constant, a variable, or a lambda abstraction.

$$\begin{array}{rcl} v & \in & \mathit{Values}_n[\Lambda] \\ v & ::= & c \mid \lambda x.t \end{array} \qquad \begin{array}{rcl} v & \in & \mathit{Values}_v[\Lambda] \\ v & ::= & c \mid x \mid \lambda x.t \end{array}$$

The term evaluation functions $eval_n$, and $eval_v$ are defined in terms of reflexive, transitive closure of small-step operational semantic rules defined in Figure 3.1, and Figure 3.2 respectively.

$$\begin{array}{lcl} eval_n(t) = v & \text{iff} & t \longmapsto_n^* v \\ eval_v(t) = v & \text{iff} & t \longmapsto_v^* v \end{array}$$

$$(\lambda x.t_0)\ t_1 \longmapsto_n t_0[x := t_1]$$

$$\frac{t_0 \longmapsto_n t_0'}{t_0\ t_1 \longmapsto_n t_0'\ t_1}$$

Figure 3.1: Call-by-name small-step operational semantics

$$(\lambda x.t_0)\ v \longmapsto_v t_0[x := v]$$

$$\frac{t_0 \longmapsto_v t_0'}{t_0\ t_1 \longmapsto_v t_0'\ t_1} \qquad \frac{t_1 \longmapsto_v t_1'}{(\lambda x.t_0)\ t_1 \longmapsto_v (\lambda x.t_0)\ t_1'}$$

Figure 3.2: Call-by-value small-step operational semantics

Note that both of the small-step transition rules, and both of the evaluation functions operate on closed lambda terms (terms with no free variables).

We say that two lambda terms are equivalent $t_0 \simeq t_1$ if either both t_0, and t_1 are undefined, or both are defined and are equal up renaming of the variables. We write $t_0 \simeq_{\beta_i} t_1$ if either both terms are undefined, or are defined and denote β_i-convertible terms. By β_i-convertibility we don't strictly have β-reduction in mind, but arbitrary small-step operational semantic rules. For example, a pair of terms might be n-convertible to state that they are convertible under call-by-name small-step rules. Informally two terms are convertible under a reduction strategy if they will eventually coverge into same terms.

Both of the CPS transformations are defined as functions taking lambda terms to lambda terms. The call-by-value transformation C_v, and call-by-name transformation C_n are defined in Figure 3.3.

Plotkin showed in [Plo75] that both call-by-name, and call-by-value CPS transformed closed terms are indifferent to the evaluation strategy. Given identity function I this is stated formally as follows:

$$eval_v(C_n⟪t⟫\ I) \simeq eval_n(C_n⟪t⟫\ I)$$
$$eval_v(C_v⟪t⟫\ I) \simeq eval_n(C_v⟪t⟫\ I)$$

A second property that Plotkin [Plo75] showed was that call-by-name behaviour is simulatable in call-by-value setting by performing call-by-name CPS transformation. The other way around call-by-value can be simulated in call-by-name setting.

$$\begin{array}{rcl} C_n\langle eval_n(t)\rangle & \simeq_{\beta_i} & eval_v(C_n⟪t⟫\ I) \\ C_v\langle eval_v(t)\rangle & \simeq & eval_n(C_v⟪t⟫\ I) \end{array}$$

Example

Informally, the call-by-value CPS transformation consists of naming all significant intermediate results of terms, sequentializing their computation, and introducing continuations. For example, let us consider non-tail-recursive factorial function defined in Haskell:

```
fact :: Int → Int
fact n = if n ≡ 1 then 1
                   else  n ∗ fact (n − 1)
```

$$\begin{array}{rcl}
\mathcal{C}_n\llbracket\cdot\rrbracket & : & \Lambda \to \Lambda \\
\mathcal{C}_n\llbracket v\rrbracket & = & \lambda k.k\ \mathcal{C}_n\langle v\rangle \\
\mathcal{C}_n\llbracket x\rrbracket & = & \lambda k.x\ k \\
\mathcal{C}_n\llbracket t_0\ t_1\rrbracket & = & \lambda k.\mathcal{C}_n\llbracket t_0\rrbracket(\lambda y_0.y_0\ \mathcal{C}_n\llbracket t_1\rrbracket\ k) \\
\\
\mathcal{C}_n\langle\cdot\rangle & : & Values_n[\Lambda] \to \Lambda \\
\mathcal{C}_n\langle c\rangle & = & c \\
\mathcal{C}_n\langle \lambda x.t\rangle & = & \lambda x.\mathcal{C}_n\llbracket t\rrbracket \\
\\
\mathcal{C}_v\llbracket\cdot\rrbracket & : & \Lambda \to \Lambda \\
\mathcal{C}_v\llbracket v\rrbracket & = & \lambda k.k\ \mathcal{C}_v\langle v\rangle \\
\mathcal{C}_v\llbracket t_0\ t_1\rrbracket & = & \lambda k.\mathcal{C}_v\llbracket t_0\rrbracket(\lambda y_0.\mathcal{C}_v\llbracket t_1\rrbracket(\lambda y_1.y_0\ y_1\ k)) \\
\\
\mathcal{C}_v\langle\cdot\rangle & : & Values_v[\Lambda] \to \Lambda \\
\mathcal{C}_v\langle c\rangle & = & c \\
\mathcal{C}_v\langle x\rangle & = & x \\
\mathcal{C}_v\langle \lambda x.t\rangle & = & \lambda x.\mathcal{C}_v\llbracket t\rrbracket
\end{array}$$

Figure 3.3: Call-by-name and call-by-value CPS transformations

The first step in transforming the function to continuation-passing style is to name and sequentialize all the intermediate computations. We do not transform constant literals or other primitives such as arithmetic operators. The only significant intermediate result in factorial function is the recursive call to the function itself. We denote the result of the recursive call with a fresh variable *v*:

```
fact n = if n ≡ 1
  then 1
  else let v = fact (n − 1) in n ∗ v
```

The next step is to introduce the continuation *k*, and redefine the original function as CPS-transformed function applied to the identity continuation. The transformed program, after moving the lambda-abstraction parameter *k* to left-hand side of the helper function definition, looks as follows:

```
fact_CPS :: Int → (Int → a) → a
fact_CPS n k = if n ≡ 1
  then k 1
  else fact_CPS (n − 1) (λv → k (n ∗ v))
fact :: Int → Int
fact n = fact_CPS n (λv → v)
```

The resulting factorial function is tail-recursive. However, while the original program built a high call stack, the transformed program builds a large closure.

In the derivation we have cheated a bit. Namely the host language is evaluated under call-by-name strategy, but we have applied call-by-value CPS transformation. While this does not not preserve the semantics of the original program in general case in the current case it's correct as the factorial function is strict.

3.2.2 Defunctionalization

Defunctionalization is a global code transformation that turns programs with higher-order functions into first-order ones. We are using the defunctionalization as described by Reynolds in [Rey98]. The method represents functions with data types, and replaces function abstractions with data constructors.

For every function space the transformation denotes the type of the function with a new data type, and every lambda abstraction of that type becomes a data constructor in the new data type. Free variables of lambda abstractions become parameters to the corresponding data constructors. Function applications are represented with first-order functions, that take the representation of the function and the function parameters as input, and perform the application as dictated by the original lambda abstraction body. Correctness of defunctionalization has previously been established by Nielsen in [Nie00].

Note that we do not consider the case of transforming polymorphic programs. We assume that before defunctionalization is applied the program is converted into monomorphic form via code duplication.

As an example we will apply defunctionalization to the higher-order tail-recursive factorial function derived in the previous subsection. The example has only single function space of type $Int \to Int$ that we will denote with a new data type *Fun*. Inhabitants of the only function space are given rise to by two lambda abstractions: *a*) $\lambda v \to v$ with no free variables; and *b*) $\lambda v \to k\ (n * v)$ with $n :: Int$, and $k :: Int \to Int$ that will be represented with *Fun* data type, as free variables. Thus we create a data type with two constructors, and an appropriately typed application function:

```
data Fun = L₀
         | L₁ Fun Int
app₀ :: Fun → Int → Int
app₀ L₀        v = v
app₀ (L₁ k n) v = app₀ k (n ∗ v))
```

In the original program we have replaced the abstractions with data constructors and applications of higher-order functions with calls to the new first-order function. This gives us the following defunctionalized program:

```
fact_DEF :: Int → Fun → Int
fact_DEF n k = if n ≡ 1
  then app₀ k 1
  else fact_DEF (n − 1) (L₁ k n)
fact :: Int → Int
fact n = fact_DEF n L₀
```

Note that the data structure has a list structure, and thus the new program can be made more readable and intuitively understandable by using already existing list type:

```
app₁ :: [Int] → Int → Int
app₁ (n : k) v = app₁ k (n ∗ v)
app₁ [ ]     v = v
```

```
$fact_{LIST} :: Int \to [Int] \to Int$
$fact_{LIST}\ n\ k = $ **if** $n \equiv 1$
    **then** $app_1\ k\ 1$
    **else** $fact_{LIST}\ (n-1)\ (n : k)$
$fact :: Int \to Int$
$fact\ n = fact_{LIST}\ n\ [\,]$
```

By combining the continuation-passing style with defunctionalization we have reduced the higher-order non-tail-recursive factorial function into two first-order tail-recursive functions. One of which constructs a stack of values, and other multiplies the elements of the stack together. With further non-automatic optimizations it's possible to reach a tail-recursive factorial function which operates on integer accumulator, but those transformations are beyond the scope of this work.

3.2.3 Thunk-based simulation

In Section 3.2.1 we saw that call-by-name evaluation strategy can be simulated in call-by-value setting by performing a call-by-name CPS transformation. Turns out that there is a simpler way to simulate CBN programs by transforming the program to explicitly suspend the computation of function arguments. We call a delayed or suspended computation (or parameterless procedure) a *thunk*.

Formally the thunk-based simulation takes place in λ-calculus extended with constructs to explicitly force or delay computations. The syntax for the extended lambda calculus is as follows:

$$\begin{array}{rcl} t & \in & \Lambda_\tau \\ t & ::= & c \mid x \mid \lambda x.t \mid t_0\, t_1 \mid \mathit{delay}\ t \mid \mathit{force}\ t \end{array}$$

The evaluation functions for extended language are obtained by adding following transition rules to both the call-by-name, and the call-by-value operational semantics defined in Figure 3.1 and 3.2.

$$\frac{t \longmapsto t'}{\mathit{force}\ t \longmapsto \mathit{force}\ t'} \qquad \mathit{force}\ (\mathit{delay}\ t) \longmapsto t$$

The transformation to thunk-based form from regular λ-calculus is achieved by annotating function parameters with explicit delay, and forcing the evaluation of variables.

$$\begin{array}{rcl} \mathcal{T}\langle\!\langle \cdot \rangle\!\rangle & : & \Lambda \to \Lambda_\tau \\ \mathcal{T}\langle\!\langle c \rangle\!\rangle & = & c \\ \mathcal{T}\langle\!\langle x \rangle\!\rangle & = & \mathit{force}\ x \\ \mathcal{T}\langle\!\langle \lambda x.t \rangle\!\rangle & = & \lambda x.\mathcal{T}\langle\!\langle t \rangle\!\rangle \\ \mathcal{T}\langle\!\langle t_0\ t_1 \rangle\!\rangle & = & \mathcal{T}\langle\!\langle t_0 \rangle\!\rangle(\mathit{delay}\ \mathcal{T}\langle\!\langle t_1 \rangle\!\rangle) \end{array}$$

An optimization to the transformation has been supplied by Danvy and Hatcliff in [DH92]. The improved transformation is performed on λ-terms with strictness annotations. The parameters to strict functions are not suspended, and thus the variables that have been introduced by strict functions are not explicitly forced. This result allows for the thunk-based simulation to be guided by the results of a strictness analysis.

Thunk-transformed closed terms are indifferent to evaluation strategy, and simulate call-by-name evaluation in call-by-value setting:

$$\begin{array}{rcl} eval_v(\mathcal{T}\langle\!\langle t\rangle\!\rangle) & \simeq & eval_n(\mathcal{T}\langle\!\langle t\rangle\!\rangle) \\ \mathcal{T}\langle\!\langle eval_n(t)\rangle\!\rangle & \simeq_\tau & eval_v(\mathcal{T}\langle\!\langle t\rangle\!\rangle) \end{array}$$

Here the equivalence relation $t_1 \simeq_\tau t_2$ means that the terms are both n-convertible, and v-convertible given that the conversion rules are extended to account for the force and delay primitives. The evaluation rules too need to account for the new primitives too.

Let us finally note that the call-by-name CPS transformation factors into composition of thunk-based simulation followed by call-by-value CPS transformation. Informally: $C_v \circ \mathcal{T} \equiv C_n$, and the result was shown by Hatcliff and Danvy in [HD97].

3.3 The Hutton's razor

Hutton's razor [HW04] can be considered the simplest non-trivial expression language. The language term is either integer literal, or addition of two terms. Abstract syntax of the language is represented with Haskell data type, and the semantics of the language is defined by a denotational evaluator. The evaluator directly corresponds to denotational semantics.

```
data Term
  = Lit   Int
  | Add Term Term

eval  :: Term → Int
eval (Lit x)       = x
eval (Add t0 t1) = eval t0 + eval t1
```

In order to demonstrate the methodology developed by Ager *et al.* in [ABDM03] we will, as an introduction to it, derive an abstract machine from the given evaluator. First we note that the *eval* function itself is not suitable as an abstract machine because it is not tail-recursive, and is not presented as a term rewriting system.

3.3.1 CPS transformation

The first step is to perform continuation-passing style transformation to convert the evaluator to tail-recursive form. We choose to evaluate the left-hand side of the addition operator first as the semantics of the language has left the evaluation order of its parameters open.

```
eval0  :: Term → (Int → a) → a
eval0 (Lit x)       k = k x
eval0 (Add t0 t1) k = eval0 t0 (λv0 → eval0 t1 (λv1 → k (v0 + v1)))

runEval0 :: Term → Int
runEval0 t = eval0 t (λv → v)
```

The equivalence of *eval* to *runEval*$_0$ is due to the correctness of the CPS transformation. While the evaluator is now linear, it's no longer first-order. We have applied the call-by-value CPS transformation, as the original program was strict. Defunctionalization is used next to eliminate the use of higher-order functions.

3.3.2 Defunctionalization

The CPS transformed program has a single function space of type $Int \to Int$. While the type of evaluation function is more generic, it is specialized to integer at the call site in the $runEval_0$ function. If the evaluation function was called on multiple different concrete types (for instance, additionally $eval_0\ t\ print$) we would specialize the function for all types, and perform the defunctionalization on all of the specializations individually.

The only function space shall be denoted with *Fun* data type. Within the single function space we have three function abstractions:

1. $\lambda v \to v$ with no free variables;
2. $\lambda v_1 \to k\ (v_0 + v_1)$ with two free variables $v_0 :: Int$, and $k :: Int \to Int$ that will be represented with *Fun*; and
3. $\lambda v_0 \to eval_0\ t_1\ (\lambda v_1 \to k\ (v_0 + v_1))$ with two free variables $t_1 :: Term$, and $k :: Int \to Int$ that will be represented with *Fun*.

The enumeration of the lambda function constructors gives us the following data type to represent the function space:

```
data Fun
  = L_0
  | L_1 Int  Fun
  | L_2 Term Fun
```

Every lambda function constructor within the evaluation function is replaced by the corresponding data constructor. Free variables of the lambda functions become arguments to the data constructors.

```
eval_1 :: Term → Fun → Int
eval_1 (Lit x)        k = app k x
eval_1 (Add t_0 t_1) k = eval_1 t_0 (L_2 t_1 k)
runEval_1 :: Term → Int
runEval_1 t = eval_1 t L_0
```

A new function *app* is used to apply the representation of the function space to actual values. The body of the application function is derived from the definition of the lambda functions. Note that the transformation has been applied to the lambda function bodies themselves, and the inner lambda function has been eliminated this way:

```
app :: Fun → Int → Int
app L_0            v   = v
app (L_1 v_0 k) v_1 = app k (v_0 + v_1)
app (L_2 t_1 k) v_0 = eval_1 t_1 (L_1 v_0 k)
```

Due to the correctness of the defunctionalization we have that $runEval_0\ t \equiv runEval_1\ t$ for every expression t. Technically the presented evaluator is in curried form, and thus is still higher-order. However, because all of the function applications are total, we are not going to uncurry the functions explicitly.

3.3.3 The abstract machine

The final evaluator is ready to be presented in an abstract machine form. Syntactically the expression language terms are denoted with t, expressible values with v, and the evaluation contexts with k. The language literal values $\bar{v}$ are marked with an over-bar to distinguish them from expressible values. Expressible values v are integers. Evaluation context k has a stack structure, and is either empty stack *nil*, or has expressible value or expression as the topmost element.

$$\begin{array}{rcl} v & \in & \mathbb{Z} \\ t & ::= & \bar{v} \mid t_0 + t_1 \\ k & ::= & nil \mid v :: k \mid t :: k \end{array}$$

Note that the abstract machine compared to the Haskell evaluator does not consider undefined values $\perp$, and can have arbitrary integers as expressible values.

The transition system has an initialization rule, two transition rules, and a stopping rule. The two evaluation transitions directly correspond to the definition of *eval* function, and the two application transitions correspond to the *app* function. Notice that the evaluation of addition operator is very similar to evaluation of function application in the CEK abstract machine.

t	$\Rightarrow_{eval}$	$\langle t, nil\rangle$
$\langle t_0 + t_1, k\rangle$	$\Rightarrow_{eval}$	$\langle t_0, t_1 :: k\rangle$
$\langle \bar{v}, k\rangle$	$\Rightarrow_{eval}$	$\langle k, v\rangle$
$\langle t_1 :: k, v_0\rangle$	$\Rightarrow_{app}$	$\langle t_1, v_0 :: k\rangle$
$\langle v_0 :: k, v_1\rangle$	$\Rightarrow_{app}$	$\langle k, v_0 + v_1\rangle$
$\langle nil, v\rangle$	$\Rightarrow_{app}$	v

The first *app* transition corresponds to the case where left-hand side of the operator has been evaluated to v_0, and the right-hand side t_1 has not. In that case the system transition into evaluating the right-hand side, and stores the value of the left-hand side on the stack. The second rule corresponds to case where both sides of the operator have been evaluated, and in that case the rule performs the operation, and possibly continues reducing expressible values on the stack.

For example, to evaluate expression $(\bar{1} + \bar{2}) + (\bar{4} + \bar{5})$ we would follow the transition rules as follows:

$$\begin{array}{rl} (\bar{1} + \bar{2}) + (\bar{4} + \bar{5}) & \Rightarrow_{eval} \\ \langle (\bar{1} + \bar{2}) + (\bar{4} + \bar{5}), nil\rangle & \Rightarrow_{eval} \\ \langle (\bar{1} + \bar{2}), (\bar{4} + \bar{5}) :: nil\rangle & \Rightarrow_{eval} \\ \langle \bar{1}, \bar{2} :: (\bar{4} + \bar{5}) :: nil\rangle & \Rightarrow_{eval} \\ \langle \bar{2} :: (\bar{4} + \bar{5}) :: nil, 1\rangle & \Rightarrow_{app} \\ \langle \bar{2}, 1 :: (\bar{4} + \bar{5}) :: nil\rangle & \Rightarrow_{eval} \\ \langle 1 :: (\bar{4} + \bar{5}) :: nil, 2\rangle & \Rightarrow_{app} \\ \langle (\bar{4} + \bar{5}) :: nil, 3\rangle & \Rightarrow_{app} \\ \langle \bar{4} + \bar{5}, 3 :: nil\rangle & \Rightarrow_{eval} \\ \langle \bar{4}, \bar{5} :: 3 :: nil\rangle & \Rightarrow_{eval} \\ \langle \bar{5} :: 3 :: nil, 4\rangle & \Rightarrow_{app} \\ \langle \bar{5}, 4 :: 3 :: nil\rangle & \Rightarrow_{eval} \\ \langle 4 :: 3 :: nil, 5\rangle & \Rightarrow_{app} \\ \langle 3 :: nil, 9\rangle & \Rightarrow_{app} \\ \langle nil, 12\rangle & \Rightarrow_{app} \quad 12 \end{array}$$

Notice that the transitions respect left-to-right evaluation order, and the parameters to the operator are passed in correct order. This machine would correctly evaluate expressions even in the case of non-associative, and non-commutative underlying operator.

There is no formal proof of correctness for the presented abstract machine, but there is clear equivalence to the last evaluator in the derivation sequence. We can see that the abstract syntax corresponds directly to the *Term* data type, and the stack structure exactly to the *Fun* data type. The evaluation and application rules correspond to $eval_1$, and *app* functions. The initial transition corresponds to $runEval_1$ function. Due to the obvious correspondence to the last evaluator and its equivalence to the initial one we can conclude that the presented abstract machine is indeed correct in respect to the initial evaluator.

Chapter 4

Deriving abstract machine for a dataflow language

In Chapter 3 we gave overview of the methodology developed by Ager *et al.* in [ABDM03] for deriving abstract machines from denotational evaluators. The notion of abstract machines was defined in the Section 3.1. In this chapter we will apply the methodology to derive an abstract machine for the comonadic denotational dataflow language evaluator defined in Section 2.3.

In this chapter we will first recap the previously defined comonadic evaluator and apply a series of code transformations to it in order to reach an equivalent abstract machine. The first transformation is to remove the higher-order functions from the value domain of the language. We achieve this via defunctionalization. Next we will inline various components to simplify the evaluator. At this point we will diverge slightly from the methodology and convert the evaluator to use thunks in order to simulate the lazy host-language semantics. To reach a tail-recursive form we will apply call-by-value CPS transformation. Finally we will defunctionalize the introduced continuations in order to reach a first-order form.

We have diverged only slightly from the methodology. It would have been possible to avoid thunk-based simulation, and perform call-by-name CPS transformation directly, but we found the thunk-based simulation approach to be simpler to understand and apply.

4.1 Initial evaluator

Let us recap the evaluator presented in Section 2.3 that stands as stepping stone for the derivation process. The language term is either a variable, a literal integer, a lambda abstraction, a function application, or a followed-by construct. Only the followed-by construct is unique, and rest of the evaluator is standard for lambda calculus. In Haskell the abstract syntax tree is defined as follows:

```
data Term
  = Lit  Int
  | Var  String
  | Lam String Term
  | App  Term  Term
  | Fby  Term  Term
```

The expressible value is either co-Kleisli arrow from a value to value, or an integer. The data structure to represent values and the denotational evaluator itself are defined as follows:

```
data Value
  = I  Int
  | F (LV Value → Value)
eval :: Term → LV Env → Value
eval (Lit n)       de = I n
eval (Var x)       de = counit de ! x
eval (Lam x t)     de = F (λdv → eval t (repair x ‘cmap‘ czip dv de))
eval (App t₀ t₁)  de = unF (eval t₀ de) (cobind (eval t₁) de)
eval (Fby t₀ t₁)  de = fby (eval t₀ de) (cobind (eval t₁) de)
```

Throught the derivation the type of the value domain will change, and thus the type of the environment as well. In particular all of the transformations would need to be applied to the values stored in the environment too. For this reason we will assume that the initial environment is empty:

```
type Env = [(String, Value)]
ε :: Env
ε = [ ]
```

Using this assumption only the type of the initial environment will change throughout the derivation and the implementation remains as is. How to handle the case where the initial environment contains values has previously been covered by Ager, Danvy, and Midtgaard in [ADM05] for monadic evaluators.

We would like to point out various characteristics of the evaluator that need to be addressed in the derivation process before the abstract machine form is reached. In particular the evaluator: *a*) contains higher-order function in the value domain; *b*) passes higher-order function in the form *eval* t_1 to *cobind*; *c*) is evaluated lazily due to the host-language semantics; *d*) is not tail-recursive; and *e*) is not presented as a term rewriting system.

4.2 From higher-order functions to closures

As a first step we eliminate the use of higher-order functions from the expressible value domain. Instead the value domain will store closures in the form of triples consisting of a variable, a term, and an environment.

The transformation is performed by defunctionalizing the function domain *LV Value* → *Value* in the value data type. The only inhabitants of this function domain are constructed in the lambda abstraction case, and are consumed in the lambda application case. The lambda function constructor has three free variables *x* :: *String*, *t* :: *Term*, and *de* :: *LV Env*.

The defunctionalization is performed as usual, but the resulting data type with only one constructor is inlined into the value data type. This results in the following data structure:

```
data Value₀ = I₀  Int
            | C₀ String Term (LV Env₀)
type Env₀   = [(String, Value₀)]
```

Finally, we inline the constructed application function into the evaluator. As a result we have replaced the right-hand side of lambda expression where function objects are constructed, and modified the application case where the constructed functions are consumed. Integer values need to be constructed with the appropriate data constructor. This gives us following code for the evaluator with rest unmodified:

```
$eval_0 :: Term \to LV\ Env_0 \to Value_0$
$eval_0\ (Lit\ n)\quad\ de = I_0\ n$
$eval_0\ (Lam\ x\ t)\ \ de = C_0\ x\ t\ de$
$eval_0\ (App\ t_0\ t_1)\ de =$
   case $eval_0\ t_0\ de$ of
      $C_0\ x'\ t'\ de' \to$
         $eval_0\ t'\ (repair\ x'\ `cmap`\ czip\ (cobind\ (eval_0\ t_1)\ de)\ de')$
 ...
```

The *runEval* function needs to be updated to call the new evaluation function, and reflect the new value type:

```
$runEval_0 :: Term \to Int \to Value_0$
$runEval_0\ t\ i = eval_0\ t\ (\varepsilon \uparrow i)$
```

Definition 1. *We say that two non-empty lists are equivalent, if they are of the same length and are point-wise equivalent. Two functions are equivalent if they map equivalent inputs to equivalent outputs. The equivalence relation between values of type Value, and Value$_0$ is defined as follows:*

$$\frac{}{\bot \cong \bot} \qquad \frac{}{I\ n \cong I_0\ n}$$

$$\frac{f \cong \lambda dv' \to eval_0\ t\ (repair\ x\ `cmap`\ czip\ dv'\ de)}{F\ f \cong C_0\ x\ t\ de}\ .$$

Proposition 2 (Full correctness)**.** *For every language term $t :: Term$, and $i > 0$ if evaluating runEval t i yields a value then runEval$_0$ t i yields an equivalent value.*

Proof. Due to correctness of defunctionalization. □

4.3 Inlining the comonad

The second step of the transformation is to inline some of the comonad specific functions. We extract the environment updating from the application case of the evaluator, and introduce it as a helper function:

```
$update_0 :: String \to Term \to LV\ Env_0 \to LV\ Env_0 \to LV\ Env_0$
$update_0\ x'\ t_1\ de\ de' =$
   $repair\ x'\ `cmap`\ czip\ (cobind\ (eval_1\ t_1)\ de)\ de'$
```

First we will inline the definitions of *cobind*, *czip* and *repair* into the helper function, and after some simplification we are left with following code:

```
$update_1 :: String \to Term \to LV\ Env_0 \to LV\ Env_0 \to LV\ Env_0$
$update_1\ x'\ t_1\ de@(Cons\ e\ es)\ (Cons\ e'\ es') =$
   $Cons\ (insert\ x'\ (eval_1\ t_1\ de)\ e')\ (update_1\ x'\ t_1\ es\ es')$
$update_1\ x'\ t_1\ de\ de' = One\ (insert\ x'\ (eval_1\ t_1\ de)\ (counit\ de'))$
```

Proof of the equivalence of the two functions is presented in Appendix A.1.

As both update functions are equivalent we can replace the occurrences of right-hand side of *update*$_0$ with left-hand side of *update*$_1$:

```
$eval_1 :: Term \to LV\ Env_0 \to Value_0$
...
$eval_1\ (App\ t_0\ t_1)\ de =$
   **case** $eval_1\ t_0\ de$ **of**
      $C_0\ x'\ t'\ de' \to eval_1\ t'\ (update_1\ x'\ t_1\ de\ de')$
...
```

By introducing *update*$_1$ function we have gotten rid of all uses of functions *repair* and *czip*.

We observe that in case of followed-by term the evaluation function is strict on the environment parameter. This is due to the fact that both *fby*, and *cobind* are strict on the second parameter. The strictness allows us to pattern match on the second evaluator parameter, and simplify using simple case analysis. For singleton environments:

```
   $eval_1\ (Fby\ t_0\ t_1)\ (One\ e)$
$\equiv$   { definition of $eval_1$ }
   $fby\ (eval_1\ t_0\ (One\ e))\ (cobind\ (eval_1\ t_1)\ (One\ e))$
$\equiv$   { definition of $cobind$ }
   $fby\ (eval_1\ t_0\ (One\ e))\ (One\ (eval_1\ t_1\ (One\ e)))$
$\equiv$   { definition of $fby$ }
   $eval_1\ t_0\ (One\ e)$
```

and for larger environments:

```
   $eval_1\ (Fby\ t_0\ t_1)\ (Cons\ e\ de)$
$\equiv$   { definition of $eval_1$ }
   $fby\ (eval_1\ t_0\ (Cons\ e\ de))$
      $(cobind\ (eval_1\ t_1)\ (Cons\ e\ de)$
$\equiv$   { definition of $cobind$ }
   $fby\ (eval_1\ t_0\ (Cons\ e\ de))$
      $(Cons\ (eval_1\ t_1\ (Cons\ e\ de))\ (cobind\ (eval_1\ t_1)\ de))$
$\equiv$   { definition of $fby$ }
   $counit\ (cobind\ (eval_1\ t_1)\ de))$
$\equiv$   { comonad law }
   $eval_1\ t_1\ de$
```

The simplification gives us the following followed-by construct cases:

```
...
$eval_1\ (Fby\ t_0\ t_1)\ (One\ \ \ e)\ \ \ \ \ = eval_1\ t_0\ (One\ e)$
$eval_1\ (Fby\ t_0\ t_1)\ (Cons\ e\ de) = eval_1\ t_1\ de$
```

The updated *runEval* function simply calls the new evaluation function:

$runEval_1 :: Term \to Int \to Value_0$
$runEval_1\ t\ i = eval_1\ t\ (\varepsilon \uparrow i)$

Proposition 3 (Full correctness)**.** *For every language term $t :: Term$, and $i > 0$ if $runEval_0\ t\ i$ yields some value, then it's equal to $runEval_1\ t\ i$.*

Proof. Due to the correctness of applied simplifications. □

4.4 Introduction of thunks

The state of the evaluator we have after inlining has two issues. Firstly the newly constructed $update_1$ function is mutually recursive to $eval_1$. This can not be the case if we wish it to become primitive construct in the abstract machine.

Second issue is that the correct semantics of the evaluator depends on the non-strict semantics of Haskell. Namely, the results of $eval_1$ function applications performed in the *update* function are inserted into environment but are only forced when evaluating variables. We wish for the abstract machine to be indifferent to the evaluation order. One option is to apply call-by-name CPS transformation to gain correct evaluation order indifferent and tail-recursive evaluator, but we found thunk-based simulation to be a better option.

We will make the laziness explicit by introducing *force*, and *delay* constructs into the evaluator. The constructs are placed based on an ad-hoc strictness analysis. We are going to assume that terms, and environment lists passed to the evaluation function are always defined, and thus can be strictly evaluated. The *update*, and *counit* functions are strict.

Following the assumptions we notice that the only non-strict function in the evaluator is the environment insertion function. The insert is lazy in the value parameter, and the lookup returns potentially delayed value. The only values inserted into the environment are the results of calls to the evaluation function. Because of that we are not implementing generic suspension operation, but specialize it.

$force :: Value_1 \to Value_1$
$force\ (T_1\ t\ de) = eval_2\ t\ de$
$delay :: Term \to LV\ Env_1 \to Value_1$
$delay\ t\ de = T_1\ t\ de$

Notice that for every term *t*, and environment *de* if $eval_2\ t\ de$ yields some value, then it's equal to $force\ (delay\ t\ de)$.

The suspended computations will be stored directly in the value domain as thunks.

```
data Value₁
    = I₁ Int
    | C₁ String Term (LV Env₁)
    | T₁        Term (LV Env₁)
type Env₁ = [(String, Value₁)]
```

There might also arise question if we could modify the environment to include only unevaluated thunks instead of arbitrary expressible values. The answer is yes, but we choose not to as the *delay* operation can be slightly optimized. For example, there is no reason to delay literal values and closures.

We will note that delayed values of type $Value_0$ can also be expressed as functions from the unit type () to the domain itself $() \to Value_0$. If this approach was taken the environment has to contain functions of this form. The introduced higher-order functions have to be eliminated using defunctionalization which will eventually, after merging the value domain with created data type, result in exactly the same evaluator. The indirect approach through higher-order functions is more formal, but is much more verbose.

Mutual recursion and implicit non-strict behaviour is now eliminated by introducing explicit delay into the $update_1$ function giving us:

```
$update_2$ :: $String \to Term \to LV\ Env_1 \to LV\ Env_1 \to LV\ Env_1$
$update_2\ x'\ t_1\ de@(Cons\ e\ es)\ (Cons\ e'\ es') =$
    $Cons\ (insert\ x'\ (delay\ t_1\ de)\ e')\ (update_2\ x'\ t_1\ es\ es')$
$update_2\ x'\ t_1\ de\ de' =$
    $One\ (insert\ x'\ (delay\ t_1\ de)\ (counit\ de'))$
```

Values are forced to evaluate only when looked up from the environment, and function application invokes the new $update_2$ function:

```
$eval_2$ :: $Term \to LV\ Env_1 \to Value_1$
...
$eval_2\ (Var\ x)$      $de = force\ (counit\ de\ !\ x)$
$eval_2\ (App\ t_0\ t_1)\ de =$
    case $eval_2\ t_0\ de$ of
        $C_1\ x'\ t'\ de' \to eval_2\ t'\ (update_2\ x'\ t_1\ de\ de')$
...
```

Notice that evaluation function can never give rise to thunks. The rest of the evaluator remains unmodified apart from updated types and data constructors.

The only place where values from the environment are read is the variable lookup. Previously the values inserted into the environment were evaluated when the result of the variable lookup was required. This behaviour was due to the call-by-name reduction strategy of the host language. Now we delay the evaluation explicitly, and force the evaluation of values when they are actually looked up from the environment. Under non-strict semantics that's correct.

We have gotten rid of mutual recursion of $eval_1$ and $update_1$, but introduced it to $eval_2$ and *force*. However, this is not an issue as *force* is conceptually much simpler function.

To update the *runEval* function the data type is updated, and the new evaluation function is called:

```
$runEval_2$ :: $Term \to Int \to Value_1$
$runEval_2\ t\ i = eval_2\ t\ (\varepsilon \uparrow i)$
```

Definition 2. *We say that two environments are equivalent if they are of equal length, and pointwise map the same variable to equivalent values. The equivalence relation between values of* *type $Value_0$, and $Value_1$ is defined as follows:*

$$\frac{}{\bot \cong \bot} \qquad \frac{}{I_1\ n \cong I_1\ n}$$

$$\frac{de \cong de'}{C_0\ x\ t\ de \cong C_1\ x\ t\ de'}$$

$$\frac{v_1 \cong eval_2\ t\ de}{v_1 \cong T_1\ t\ de}$$

Proposition 4 (Full correctness)**.** *For every language term $t :: Term$, and $i > 0$ if $runEval_1\ t\ i$ yields some value, then it's equivalent to $runEval_2\ t\ i$.*

Proof. Due to correctness of thunk-based simulation. □

Proposition 5 (Indifference)**.** *The evaluation function $eval_2$ is indifferent to the host language evaluation strategy.*

Proof. Due to indifference property of the thunk-based simulation. □

4.5 CPS transformation

In this section we will apply call-by-value CPS transformation to the call-by-name thunk-transformed evaluator. The previous and current transformation combined are equivalent to call-by-name CPS transformation.

We have decided to convert $Value_1$ to closure $(Value_1 \to a) \to a$. The transformation is straightforward and result in the following higher-order tail-recursive evaluator:

```
$eval_3 :: Term \to LV\ Env_1 \to (Value_1 \to a) \to a$
$eval_3\ (Lit\ n)$       $de$            $k = k\ (I_1\ n)$
$eval_3\ (Var\ x)$       $de$            $k = force_{CPS}\ (counit\ de\ !\ x)\ k$
$eval_3\ (Lam\ x\ e)$    $de$            $k = k\ (C_1\ x\ e\ de)$
$eval_3\ (App\ t_0\ t_1)$ $de$           $k = eval_3\ t_0\ de\ (\lambda y_0 \to$
    case $y_0$ of $C_1\ x'\ e'\ de' \to$
        $eval_3\ e'\ (update\ x'\ t_1\ de\ de')\ k)$
$eval_3\ (Fby\ t_0\ t_1)\ (One\ \ e)$     $k = eval_3\ t_0\ (One\ e)\ k$
$eval_3\ (Fby\ t_0\ t_1)\ (Cons\ e\ de)$ $k = eval_3\ t_1\ de\ k$
```

Due to mutual recursion the CPS conversion is also applied to the *force* function:

```
$force_{CPS} :: Value_1 \to (Value_1 \to a) \to a$
$force_{CPS}\ (T_1\ t\ de)\ k = eval_3\ t\ de\ k$
```

To execute the new evaluator we apply it to identity function:

```
$runEval_3 :: Term \to Int \to Value_1$
$runEval_3\ t\ i = eval_3\ t\ (\varepsilon \uparrow i)\ (\lambda v \to v)$
```

Proposition 6 (Full correctness)**.** *For every language term $t :: Term$, and $i > 0$ if $runEval_2\ t\ i$ yields some value then $runEval_3\ t\ i$ yields an equal value.*

Proof. Due to the indifference property of $eval_2$, and the correctness of call-by-value CPS transformation. □

4.6 Defunctionalization

We have performed multiple steps of transformations all of which are covered in Subsection 3.2.2.

To remove the higher-order functions introduced by the CPS transformation we need to defunctionalize the created closures. Turns out that the defunctionalization step is quite simple, as the evaluator only has two lambda abstractions of the same type. One of them is the identity function with no free variables, and the other has a term, an environment, and the closure itself as free variables. The enumeration of abstractions gives us a data type with a list structure. The elements of the closure representing list are pairs composed of a term, and an environment. We are somewhat more general, and represent the closure with a list of values, while in reality the only values will be thunks. The previous reasoning gives up the following type for the new evaluation function:

$$eval_4 :: Term \to LV\ Env_1 \to [Value_1] \to Value_1$$

Environment, and the value domain remain as is.

In the evaluator the continuation is applied in two positions. In the variable literal, and in the lambda abstraction case. The evaluation function with transformed higher-order function applications looks as follows:

```
$eval_4\ (Lit\ n)$        $de$           $k = app\ k\ (I_1\ n)$
$eval_4\ (Var\ x)$        $de$           $k = force_{DEF}\ (counit\ de\ !\ x)\ k$
$eval_4\ (Lam\ x\ e)$     $de$           $k = app\ k\ (C_1\ x\ e\ de)$
$eval_4\ (App\ t_0\ t_1)$ $de$           $k = eval_4\ t_0\ de\ (T_1\ t_1\ de : k)$
$eval_4\ (Fby\ t_0\ t_1)$ $(One\ \ e)$     $k = eval_4\ t_0\ (One\ e)\ k$
$eval_4\ (Fby\ t_0\ t_1)$ $(Cons\ e\ de)$ $k = eval_4\ t_1\ de\ k$
```

Defunctionalization of the $force_{CPS}$ function simply transforms the type, and the code remains as is:

```
$force_{DEF}\ :: Value_1 \to [Value_1] \to Value_1$
$force_{DEF}\ (T\ t\ de)\ k = eval_4\ t\ de\ k$
```

The application function for the only function space is constructed from lambda abstraction bodies, and is defined as follows:

```
$app\ :: [Value_1] \to Value_1 \to Value_1$
$app\ (T\ t_1\ de : k)\ (C\ x'\ t'\ de') = eval_4\ t'\ (update\ x'\ t_1\ de\ de')\ k$
$app\ []$              $v$              $= v$
```

Finally, the identity function in $runEval_3$ needs to be transformed to empty list giving us:

```
$runEval_4 :: Term \to Int \to Value_1$
$runEval_4\ t\ i = eval_4\ t\ (\varepsilon \uparrow i)\ []$
```

Proposition 7 (Full correctness)**.** *For every language term $t :: Term$ and $i > 0$ if the expression $runEval_3\ t\ i$ yields some value v then it's equal to $runEval_4\ t\ i$.*

Proof. Due to correctness of defunctionalization. □

4.7 The final abstract machine

In this section we present the abstract machine in formal manner, and show that it's equivalent to the evaluator presented in previous section. The host language is λ-calculus extended with the followed-by construct. A language term t is either variable x, literal n, lambda abstraction, function application, or a followed-by construct.

$$t \ ::= \ x \mid n \mid \lambda x.t \mid t_0\, t_1 \mid t_0 \leftarrow t_1$$

The abstract machine operates on a non-empty stack of environments de, and an environment e maps variables to thunks. Thunks are pairs composed of a term, and an environment stack.

$$\begin{array}{rcl} de & ::= & e \mid e \lhd de \\ e & ::= & \varepsilon \mid e[x \mapsto \mathbf{T}\ t\ de] \end{array}$$

Note that a while environment itself may be empty, a sequence of environments is defined in such manner that it's not possible to construct an empty sequence.

The environment updating notion is extended to environment sequences. If the sequence is a singleton then the only environment is updated as is. To update a non-singleton sequence of environments with a thunk we need to pattern match on the environment sequence of the thunk. If the thunk is stored with a history, then we update the first environment to map to given thunk, but rest of the sequence is updated to map the variable to a same thunk but with shorter history. If we map a variable to a thunk with singleton environment in a non-singleton environment sequence, then the rest of the sequence is simply discarded resulting in singleton environment.

$$\begin{array}{lcl} (e' \lhd de')[x \mapsto \mathbf{T}\ t\ (e \lhd de)] & = & e'[x \mapsto \mathbf{T}\ t\ (e \lhd de)] \lhd de'[x \mapsto \mathbf{T}\ t\ de] \\ (e' \lhd de')[x \mapsto \mathbf{T}\ t\ e] & = & e'[x \mapsto \mathbf{T}\ t\ e] \end{array}$$

The $\uparrow$ operator is used to map an environment to a specified sized stack of environments, and *counit* is used to extract the first environment from a sequence.

$$\begin{array}{lcll} e \uparrow i & \equiv & \underbrace{e \lhd \ldots \lhd e}_{i}, & \text{if } i > 0 \\ counit\ e & \equiv & e & \\ counit\ (e \lhd de) & \equiv & e & \end{array}$$

Variable lookup on non-singleton environment looks the variable up from the first environment:

$$de\ !\ x \equiv counit\ de\ !\ x \ \ .$$

Expressible value v is either an integer, a thunk, or a closure. Continuation stack k is composed of values.

$$\begin{array}{rcl} v & ::= & n \mid \mathbf{T}\ t\ de \mid \mathbf{C}\ x\ t\ de \\ k & ::= & nil \mid v :: k \end{array}$$

Finally, the abstract machine has initialization rule for every positive integer i, final transition rule, and eight reduction rules. Evaluation rules operate on triples composed of a term, an environment sequence, and a continuation stack. Application rules operate on pairs composed of a continuation stack, and an expressible value.

t	$\Rightarrow^{i}_{eval}$	$\langle t,\ \varepsilon \uparrow i, nil\rangle$ if $i > 0$
$\langle x,\ de,\ k\rangle$	$\Rightarrow_{eval}$	$\langle t', de', k\rangle$ where $\mathbf{T}\ t'\ de' = de\ !\ x$
$\langle n,\ de,\ k\rangle$	$\Rightarrow_{eval}$	$\langle k, n\rangle$
$\langle \lambda x.t,\ de,\ k\rangle$	$\Rightarrow_{eval}$	$\langle k,\ \mathbf{C}\ x\ t\ de\rangle$
$\langle t_0\ t_1,\ de,\ k\rangle$	$\Rightarrow_{eval}$	$\langle t_0,\ de,\ \mathbf{T}\ t_1\ de\ ::\ k\rangle$
$\langle t_0 \leftarrow t_1,\ e,\ k\rangle$	$\Rightarrow_{eval}$	$\langle t_0,\ e,\ k\rangle$
$\langle t_0 \leftarrow t_1,\ e \lhd de,\ k\rangle$	$\Rightarrow_{eval}$	$\langle t_1,\ de,\ k\rangle$
$\langle \mathbf{T}\ t_1\ de\ ::\ k,\ \mathbf{C}\ x'\ t'\ de'\rangle$	$\Rightarrow_{app}$	$\langle t',\ de'[x' \mapsto \mathbf{T}\ t_1\ de],\ k\rangle$
$\langle nil,\ v\rangle$	$\Rightarrow_{app}$	v

The abstract machine presented here corresponds directly to the evaluator defined in the previous section. We note that: *a*) the abstract syntax corresponds to the *Term* data type; *b*) expressible values corresponds to $Value_1$ data type; *c*) the environment lookup corresponds to *counit de* ! *x* expression; *d*) environment updating corresponds to the *update* function; and *e*) the initialization transitions corresponds to the $runEval_4$ function.

The abstract machine is very similar to Kirvne's abstract machine apart from few differences. A minor difference is that our language contains regular variables, while the Kirvne's machine operated on terms with De Bruijn indices. Because of difference representation of variables our abstract machines expressible value domain contains closures. The second difference is the two additional rules to handle the followed-by construct. Finally, the language operates on non-empty sequence of environments, and uses overloaded notions of variable lookup, and variable updating.

Due to the direct correspondence and Proposition 7 we can conclude that the machine presented in this section is an abstract machine for a higher-order call-by-name dataflow language.

Chapter 5

Conclusions

In this thesis we explore the behaviour of programming languages that manipulates primitive values as infinite time-evolving streams of data. These programming languages are known as dataflow languages, and are used to model electronic circuits and to control hardware devices in real-time.

In particular we look at the operational behaviour of a simple higher-order call-by-name dataflow language. The language is syntactically λ-calculus that is extended with a causal dataflow language construct. The dynamic semantics of the language is given in comonadic denotational form which, however, is unfit to convey the operational behaviour. One way to impart the operational behaviour is either small-step or big-step operational semantics. As an alternative to operational semantics we have chosen to present the language evaluator in a form of an abstract machine.

An abstract machine is a term rewriting system that has few properties that make it easy for a machine to evaluate. Rather informally a term rewriting system is an abstract machine if it has clear initialization, and halting conditions; does not behave undeterministically; and the rules are always applied to terms at their roots. In addition to that it must be efficient to select which of the transition rules to apply. Abstract machines operate directly on the source language terms. Our goal was to derive an abstract machine for a hihger-order comonadic dataflow language.

Ager *et al.* developed a methodology in [ABDM03] for deriving abstract machines from program language evaluators implemented in a denotational style. The methodology applies a series of well understood code transformations to the evaluator until the sought form is attained. Major steps of the transformation are the continuation-passing style (CPS) transformation to convert the evaluator to a tail-recursive form, and the defunctionalization for eliminating the use of higher-order functions. The defunctionalization transformation is applied twice: first to convert the value domain of the language to contain closures instead of functions, and the second time to eliminate the higher-order functions introduced by the continuation-passing style transformation.

We apply the methodology to a comonadic language evaluator implemented in a non-strict language Haskell. The denotational evaluator is by Uustalu and Vene from [UV05] and corresponds directly to the formal denotational semantics of the language. Because we are working in a non-strict host language we have to apply the call-by-name CPS transformation in the derivation. However, we found that it's simpler to split the call-by-name CPS transformation into composition of thunk-based simulation and regular call-by-value CPS transformation. We guide the thunk-based simulation using ad-hock strictness analysis to avoid inserting unnecessary evaluation delays.

The derivation presented in this work results in an abstract machine which is nearly equivalent to the Krivine's abstract machine for call-by-name λ-calculus. Only significant differences are that the machine for the dataflow language has additional two rules for evaluating the causal dataflow operator, and uses overloaded notion of environment lookup and updating. A minor difference is that our language denotes variables by name while Krivine's machines uses De Bruijn indices. This tells us that operationally dataflow language functions exactly like call-by-name lambda calculus, but represents the environment in more complicated manner to track history of variables. We have also demonstrated that the methodology of deriving abstract machines can be directly applied to an evaluator implemented in a non-strict language.

5.1 Future work

While the abstract machine does give insight into the operational behaviour of the language it does not directly lead to an efficient implementation. Similar situation held with call-by-name language evaluators as the Kirivne's machine was well known before any efficient implementations emerged. Additionally, the fact that implementing functional reactive programming in Haskell has proved to be very tricky suggests that the same holds for implementing call-by-name functional dataflow languages. Recent results in developing efficient implementation of functional reactive programming in Haskell [Jel11] suggests that these approaches can be adopted.

The most significant time inefficiency of the dataflow language evaluator is due to the fact that all of the values of the resulting stream are computed independent to the previous ones. Intermediate results from the past computations could be reused. One, but not the most efficient, approach would be memoisation.

The evaluator also suffers from memory inefficiency due to having to keep all previous values of variables in memory. Either a type system, or program analysis could be used to figure out how much of the history the evaluator needs to keep around. This would be major efficiency gain as most dataflow programs only depend on finite number of previous values. However, it's very probable that this kind of analysis does not work well in the presence of higher-order functions.

Abstraktne masin komonaadilisele andmevookeelele

Magistritöö (30 EAP)

Jaak Randmets

Resümee

Klassikaline lähenemine programmeerimiskeeltele on töötada baastüüpidega (täisarvud, tõeväärtused) ühe väärtuse kaupa. Keel võib kohelda baasväärtustena arve, objekte või funktsioone, aga keele operatsioonid käsitlevad neid harva hulgi. Väärtuste kollektsioonid ja neid töötlevad operatsioonid on enamasti realiseeritud keele eraldi vahenditega, mis võimaldavad konstrueerida baastüüpidest keerulisemaid tüüpe. Üheks eraldi klassiks on massiivitöötluse ja andmeparalleelsed keeled, mille baasväärtusteks võivad olla jadad, maatriksid või isegi suvalised n-mõõtmelised massiivid. Paljud nende keelte baasoperatsioonid töötlevad massiive punktiviisiliselt. Teiseks eri klassiks on andmevookeeled, kus käsitletakse kõiki keele baasobjekte kui lõputuid andmevooge. Seega on keele tüübid lõputud vood tüüpidest, literaalid tähistavad konstantseid vooge väärtustest ja näiteks arimeetilised operatsioonid töötavad sisendvoogudel punktiviisiliselt. Andmevookeeli kasutatakse mikroskeemide modelleerimiseks ja reaalajasüsteemide juhtimiseks. Käesolevas töös uurimegi selliseid andmevookeeli.

Me ütleme, et keel on kõrgemat järku, kui ta käsitleb funktsioone baasväärtustena mis tähendab, et lubab kasutaja defineeritud funktsioonidel parameetriks võtta ja tagastada teisi funktsioone. Funktsioon on kõrgemat järku, kui ta saab kas sisendiks või tagastab väärtusena funktsiooni. Andmevookeeltes on paraku kõrgemat järku funktsioonide esitamisega probleeme ja paljud andmevookeeled kas ei käsitle neid esimest järku objektidena või nõuavad, et tüüpides oleks ilmutatult öeldud kas parameetrina edastatud funktsioon käitub puhtalt punktiviisiliselt või mitte.

Kõrgemat järku andmevookeeletes kirjutatud programmidele saab formaalse tähenduse anda kasutades kategooriateooriast tuntud mõisteid. Üks võimalus on seda teha kasutades nooli, aga leidub ka konkreetsem viis komonaadide näol. Mõlemad semantikad on antud denotatsioonilisel kujul ja edastavad seega hästi programmide tähendust, jättes aga kõrvale programmmide arvutuskäigu.

Programmide operatsioonilise käitumise mõistmiseks on paar võimalust. Esimene neist on anda keelele selle denotatsioonilise semantikaga samaväärne operatsiooniline loomulik või struktuurne semantika. Teine võimalus on konstrueerida korrektne keele väärtustaja abstrakt-

se masina kujul. Käesoleva töö eesmärgiks on tuletada kõrgemat järku andmevookeelele selle komonaadilise semantikaga samaväärne abstraktne masin.

Abstraktne masin on termide asendussüsteem, millel on mõned omadused, mis muudavad selle väärtustamise arvutile lihtsamaks. Nimelt, termide asendussüsteemi nimetame abstraktseks masinaks kui: sellel on selge alustamise reegel ning seiskamise reeglid, see käitub deterministlikult, selle reeglite vasakud pooled ei sisalda korduvaid muutujaid ja reegleid rakendatakse alati termide juurtel.

Abstraktse masina tuletamiseks kasutame metoodikat, mis seisneb algsele denotatsioonilisel kujul olevale väärtustajale järjest hästi tundutud kooditeisenduste rakendamises, kus abstraktse masina samaväärsus esialgse väärtustajaga tuletatakse rakendatud kooditeisenduste korrektsusest. Suuremad sammud programmiteisenduste ahelas on: jätkuedastusstiili teisendus, et jõuda sabarekursiivse väärtustajani, ning defunktsionaliseerimine, et teisendada programm esimest järku kujule. Defunktsionaliseerimist rakendatakse sealjuures kaks korda: esimene kord, et asendada interpretaatori väärtuste domeenis kõrgemat järku funktsioonid esimest järku sulundite esitusega ning teine kord, et eemaldada jätkuedastusstiili teisenduse poolt tekitatud kõrgemat järku funktsioonid.

Me rakendame kirjeldatud metoodikat komonaadilise semantikaga samaväärsele Haskellis realiseeritud väärtustajale. Keele interpretaator on esitatud denotatsioonilises stiilis ning vastab täpselt selle keele denotatsioonilisele semantikale. Lisaks peame arvestama ka sellega, et meie interpretaator on realiseeritud mitteagaras keeles. Ainus kooditeisendus kus see oluliseks osutub on jätkuedastusstiili teisendus, mis määrab kindlaks teisendatava programmi kontrollvoo. Jätkuedastusstiili teisendusest on kaks versiooni: üks, mis säilitab nimekutse (*call-by-name*) semantika ning teine, mis säilitab väärtuskutse (*call-by-value*) semantika.

Kuna meie väärtustaja on implementeeritud mitteagaras keeles, siis tuleb alloleva keele nimekutse semantika säilitamiseks rakendada nimekutse jätkuedastusstiili teisendust. Lihtsam lahendus on simuleerida nimekutse käitumist, teisendades programmi kujule, kus arvutusi viivitatakse ning sunnitakse ilmutatult. Kuna programmid, kus kõik viivitused on esitatud ilmutatud kujul, on sõltumatud väärtustamisjärjekorra suhtes, siis rakendame interpretaatorile väärtuskutse jätkuedastusstiili teisendust. Need kaks teisendussammu järjestikku rakendatuna on samaväärsed nimekutse jätkuedastusstiili teisendusega.

Töös esitatud kooditeisenduste järjend annab meile esimest järku sabarekursiivse interpretaatori mitteagarale kõrgemat järku andmevookeelele. Esitame selle interpretaatori samaväärse termide asendussüsteemina, millel on kõik abstraktse masina omadused. Tänu rakendatud kooditeisenduste korrektsusele ning viimase väärtustaja üksühesele vastavusele selle abstraktse masinaga saame öelda, et toodud abstraktne masin on korrektne keele denotatsioonilise semantika suhtes.

Esitatud abstraktne masin on väga sarnane hästi tuntud Krivine'i abstraktsele masinale. Ainsateks oluliseks erinevusteks on kaks uut reeglit väärtustamaks andmevookeele spetsiifilist baasoperatsiooni ning muutujate keskkonna keerukam esitus. Kuna keskkonna esitus ei ole sama, mis Krivine'i masinal, on seega ka keskkonnast otsimine ning selle uuendamine ülelaaditud tähendusega. Tuletatud masina sarnasus hästi tuntud abstraktsele masinale annab tugeva korrektsuse garantii ning ütleb, et mitteagarate andmevookeelte operatsiooniline käitumine on sarnane tavaliste mitteagarate funktsionaalsete keelte käitumisele.

Bibliography

[ABDM03] M.S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 8–19. ACM, 2003.

[ADM05] M.S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Scienve*, 342:149–172, 2005.

[CU10] S. Capobianco and T. Uustalu. A categorical outlook on cellular automata. *Arxiv preprint arXiv:1012.1220*, 2010.

[Dan04] O. Danvy. On evaluation contexts, continuations, and the rest of the computation. In *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College*, pages 13–23, 2004.

[DH92] O. Danvy and J. Hatcliff. Cps-transformation after strictness analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(3):195–212, 1992.

[DHJG06] N.A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *ACM SIGPLAN Notices*, volume 41, pages 206–217. ACM, 2006.

[FF86] M. Felleisen and D.P. Friedman. Control operators, the secd machine, and the λ-calculus, m. wirsing, editor. *Formal Description of Programming Concepts III*, pages 193–217, June 1986.

[HD97] J. Hatcliff and O. Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3):303–319, 1997.

[HM92] J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(04):415–459, 1992.

[Hug00] J. Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1):67–111, 2000.

[HW04] G. Hutton and J. Wright. Compiling exceptions correctly. In *Mathematics of Program Construction*, pages 211–227. Springer, 2004.

[Jef12] A. Jeffrey. Ltl types frp: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming languages meets program verification*, pages 49–60. ACM, 2012.

[Jel11] W. Jeltsch. *Strongly typed and efficient functional reactive programming*. PhD thesis, Universitätsbibliothek, 2011.

[Kri85] J.L. Krivine. Un interprète du λ-calcul. *Brouillon. Available online* `http://www.pps.jussieu.fr/~krivine/`, 1985.

[Lan64] P.J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[M$^+$10] S. Marlow et al. Haskell 2010 language report. *Available online* `http://www.haskell.org/onlinereport/haskell2010/`, 2010.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 14–23. IEEE, 1989.

[Mog90] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 1990.

[Nie00] L.R. Nielsen. A denotational investigation of defunctionalization. Research Series RS-00-47, BRICS, Department of Computer Science, University of Aarhus, 2000.

[Pip06] D. Piponi. Evaluating cellular automata is comonadic, December 2006. Blog post available at: `http://blog.sigfpe.com/2006/12/evaluating-cellular-automata-is.html` (May 2012).

[Pip08] D. Piponi. Comonadic arrays, March 2008. Blog post available at: `http://blog.sigfpe.com/2008/03/comonadic-arrays.html` (May 2012).

[PJS89] S.L. Peyton Jones and J. Salkild. The spineless tagless g-machine. In *Proceedings of the fourth international conference on FPCA*, pages 184–201. ACM, 1989.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

[Pou06] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI, April 2006. Distribution available at: `www.di.ens.fr/~pouzet/lucid-synchrone` (May 2012).

[Rey98] J.C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation*, 11(4):363–397, 1998.

[UV05] T. Uustalu and V. Vene. The essence of dataflow programming. *Programming Languages and Systems*, pages 2–18, 2005.

[Wad92] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992.

Appendix A

Extra proofs

A.1 Equivalence of update functions

Proposition 8. *The following function:*

$$\begin{array}{l} \mathit{update}_1\ x'\ t_1\ \mathit{de}@(\mathit{Cons}\ e\ \mathit{es})\ (\mathit{Cons}\ e'\ \mathit{es}') = \\ \quad \mathit{Cons}\ (\mathit{insert}\ x'\ (\mathit{eval}_1\ t_1\ \mathit{de})\ e')\ (\mathit{update}_1\ x'\ t_1\ \mathit{es}\ \mathit{es}') \\ \mathit{update}_1\ x'\ t_1\ \mathit{de}\ \mathit{de}' = \mathit{One}\ (\mathit{insert}\ x'\ (\mathit{eval}_1\ t_1\ \mathit{de})\ (\mathit{counit}\ \mathit{de}')) \end{array}$$

is equivalent to:

$$\begin{array}{l} \mathit{update}_0\ x'\ t_1\ \mathit{de}\ \mathit{de}' = \\ \quad \mathit{repair}\ x'\ `\mathit{cmap}`\ \mathit{czip}\ (\mathit{cobind}\ (\mathit{eval}_1\ t_1)\ \mathit{de})\ \mathit{de}' \end{array}$$

Proof. The proof is by induction over the length of argument lists. First we show that the statement holds if the first argument is a singleton list and the second list of arbitrary length. Let x', t_1, e, and de' be some values of types directed by the type of the update_0 function. Then the following equalities hold:

$$\begin{array}{ll} & \mathit{update}_0\ x'\ t_1\ (\mathit{One}\ e)\ \mathit{de}' \\ \equiv & \{\ \text{definition of } \mathit{update}_0\ \} \\ & \mathit{repair}\ x'\ `\mathit{cmap}`\ \mathit{czip}\ (\mathit{cobind}\ (\mathit{eval}_1\ t_1)\ (\mathit{One}\ e))\ \mathit{de}' \\ \equiv & \{\ \text{definition of } \mathit{cobind}\ \} \\ & \mathit{repair}\ x'\ `\mathit{cmap}`\ \mathit{czip}\ (\mathit{One}\ (\mathit{eval}_1\ t_1\ (\mathit{One}\ e)))\ \mathit{de}' \\ \equiv & \{\ \text{definition of } \mathit{czip}\ \} \\ & \mathit{repair}\ x'\ `\mathit{cmap}`\ \mathit{One}\ (\mathit{eval}_1\ t_1\ (\mathit{One}\ e), \mathit{counit}\ \mathit{de}') \\ \equiv & \{\ \text{definition of } \mathit{cmap} \text{ and } \mathit{repair}\ \} \\ & \mathit{One}\ (\mathit{insert}\ x'\ (\mathit{eval}_1\ t_1\ (\mathit{One}\ e))\ (\mathit{counit}\ \mathit{de}')) \\ \equiv & \{\ \text{definiton of } \mathit{update}_1\ \} \\ & \mathit{update}_1\ x'\ t_1\ (\mathit{One}\ e)\ \mathit{de}' \end{array}$$

As for second case we show that the proposition holds if the first argument is of any length and the second argument is a singleton list. Again, let the free variables be properly typed arbitrary values:

$$\begin{aligned}
& \mathit{update}_0\ x'\ t_1\ \mathit{de}\ (\mathit{One}\ e') \\
\equiv\quad & \{\ \text{definiton of } \mathit{update}_0\ \} \\
& \mathit{repair}\ x'\ `\mathit{cmap}`\ \mathit{czip}\ (\mathit{cobind}\ (\mathit{eval}_1\ t_1)\ \mathit{de})\ (\mathit{One}\ e') \\
\equiv\quad & \{\ \text{definiton of } \mathit{czip}\ \} \\
& \mathit{repair}\ x'\ `\mathit{cmap}`\ \mathit{One}\ (\mathit{counit}\ (\mathit{cobind}\ (\mathit{eval}_1\ t_1)\ \mathit{de}), e') \\
\equiv\quad & \{\ \text{second comonad law}\ \} \\
& \mathit{repair}\ x'\ `\mathit{cmap}`\ \mathit{One}\ (\mathit{eval}_1\ t_1\ \mathit{de}, e') \\
\equiv\quad & \{\ \text{definition of } \mathit{repair} \text{ and } \mathit{cmap}\ \} \\
& \mathit{One}\ (\mathit{insert}\ x'\ (\mathit{eval}_1\ t_1\ \mathit{de})\ e') \\
\equiv\quad & \{\ \text{definition of } \mathit{counit}\ \} \\
& \mathit{One}\ (\mathit{insert}\ x'\ (\mathit{eval}_1\ t_1\ \mathit{de})\ (\mathit{counit}\ (\mathit{One}\ e'))) \\
\equiv\quad & \{\ \text{definiton of } \mathit{update}_1\ \} \\
& \mathit{update}_1\ x'\ \mathit{de}\ (\mathit{One}\ e')
\end{aligned}$$

Let us assume that the statement holds for non-empty lists *es* and *es*′, and show that it also holds for *Cons e es*, and *Cons e*′ *es*′ for any environemnt *e*, and *e*′. Let *x*′, and t_1 be chosen arbitrarily. The following reasoning holds:

$$\begin{aligned}
& \mathit{update}_0\ x'\ t_1\ (\mathit{Cons}\ e\ \mathit{es})\ (\mathit{Cons}\ e'\ \mathit{es}') \\
\equiv\quad & \{\ \text{definition of } \mathit{update}_0\ \} \\
& \mathit{repair}\ x'\ `\mathit{cmap}`\ \mathit{czip}\ \ (\mathit{cobind}\ (\mathit{eval}_1\ t_1)\ (\mathit{Cons}\ e\ \mathit{es})) \\
& \quad (\mathit{Cons}\ e'\ \mathit{es}') \\
\equiv\quad & \{\ \text{definition of } \mathit{cobind}\ \} \\
& \mathit{repair}\ x'\ `\mathit{cmap}`\ \mathit{czip}\ \ (\mathit{Cons}\ (\mathit{eval}_1\ t_1\ (\mathit{Cons}\ e\ \mathit{es}))\ (\mathit{cobind}\ (\mathit{eval}_1\ t_1)\ \mathit{es})) \\
& \quad (\mathit{Cons}\ e'\ \mathit{es}') \\
\equiv\quad & \{\ \text{definition of } \mathit{czip}\ \} \\
& \mathit{repair}\ x'\ `\mathit{cmap}`\ \mathit{Cons}\ (\mathit{eval}_1\ t_1\ (\mathit{Cons}\ e\ \mathit{es}), e') \\
& \quad (\mathit{czip}\ (\mathit{cobind}\ (\mathit{eval}_1\ t_1)\ \mathit{es})\ \mathit{es}') \\
\equiv\quad & \{\ \text{definition of } \mathit{cmap}\ \} \\
& \mathit{Cons}\ (\mathit{insert}\ x'\ (\mathit{eval}_1\ t_1\ (\mathit{Cons}\ e\ \mathit{es}))\ e') \\
& \qquad (\mathit{repair}\ x'\ `\mathit{cmap}`\ \mathit{czip}\ (\mathit{cobind}\ (\mathit{eval}_1\ t_1)\ \mathit{es})\ \mathit{es}') \\
\equiv\quad & \{\ \text{induction assumption}\ \} \\
& \mathit{Cons}\ (\mathit{insert}\ x'\ (\mathit{eval}_1\ t_1\ (\mathit{Cons}\ e\ \mathit{es}))\ e')\ (\mathit{update}_1\ \mathit{es}\ \mathit{es}') \\
\equiv\quad & \{\ \text{definiton of } \mathit{update}_1\ \} \\
& \mathit{update}_1\ x'\ t_1\ (\mathit{Cons}\ e\ \mathit{es})\ (\mathit{Cons}\ e'\ \mathit{es}')
\end{aligned}$$

It's easy to see that the statement holds for undefined lists due to strictness of *czip* and *cmap*. □