Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures

Nils Anders Danielsson
Chalmers University of Technology
nad@cs.chalmers.se

Abstract

Okasaki and others have demonstrated how purely functional data structures that are efficient even in the presence of persistence can be constructed. To achieve good time bounds essential use is often made of laziness. The associated complexity analysis is frequently subtle, requiring careful attention to detail, and hence formalising it is valuable.

This paper describes a simple library which can be used to make the analysis of a class of purely functional data structures and algorithms almost fully formal. The basic idea is to use the type system to annotate every function with the time required to compute its result. An annotated monad is used to combine time complexity annotations.

The library has been used to analyse some existing data structures, for instance the deque operations of Hinze and Paterson's finger trees.

Categories and Subject Descriptors F.2.m [*Analysis of Algorithms and Problem Complexity*]: Miscellaneous; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; E.1 [*Data Structures*]

General Terms Languages, performance, theory, verification

1. Introduction

Data structures implemented in a purely functional language automatically become persistent; even if a data structure is updated, the previous version can still be used. This property means that, from a correctness perspective, users of the data structure have less to worry about, since there are no problems with aliasing. From an efficiency perspective the picture is less nice, though: different usage patterns can lead to different time complexities. For instance, a common implementation of FIFO queues has the property that every operation takes constant amortised time if the queues are used single-threadedly (i.e. if the output of one operation is always the input to the next), whereas for some usage patterns the complexity of the tail function becomes linear (Okasaki 1998).

Despite this a number of purely functional data structures exhibiting good performance no matter how they are used have been developed (see for instance Okasaki 1998; Kaplan and Tarjan 1999; Kaplan et al. 2000; Hinze and Paterson 2006). Many of these data structures make essential use of laziness (non-strictness with memoisation, also known as call-by-need) in order to ensure good performance; see Section 8.1 for a detailed example. However, the resulting complexity analysis is often subtle, with many details to keep track of.

To address this problem the paper describes a simple library, THUNK, for semiformal verification of the time complexity of purely functional data structures. The basic idea is to annotate the code (the actual code later to be executed, not a copy used for verification) with ticks, representing computation steps:

$$\checkmark : \mathit{Thunk}\ n\ a \rightarrow \mathit{Thunk}\ (1+n)\ a$$

Time complexity is then tracked using the type system. Basically, if a value has type *Thunk n a*, then a weak head normal form (WHNF) of type *a* can be obtained in *n* steps amortised time, no matter how the value is used. *Thunk* is a monad, and the monadic combinators are used to combine time complexities of subexpressions.

Note that the *Thunk* type constructor takes a value (n) as argument; it is a *dependent* type. The THUNK library is implemented in the dependently typed functional language Agda (Norell 2007; The Agda Team 2007), which is described in Section 2. The approach described in the paper is not limited to Agda—it does not even need to be implemented in the form of a library—but for concreteness Agda is used when presenting the approach.

In order to analyse essential uses of laziness THUNK makes use of a simplified version of Okasaki's banker's method (1998). This version is arguably easier to explain (see Section 8), but it is less general, so fewer programs can be properly analysed. A generalisation of the method, also implemented, is discussed in Section 11, and remaining limitations are discussed in Section 12.

Despite any limitations the methods are still useful in practice. The following algorithms and data structures have been analysed:

- Linear-time minimum using insertion sort, the standard example for time complexity analysis of call-by-name programs (see Section 7).
- Implicit queues (Okasaki 1998), which make essential use of laziness (see Section 8).
- The deque operations of Hinze and Paterson's finger trees (2006).[1]
- Banker's queues (Okasaki 1998), by using the generalised method described in Section 11.[1]

The time bounds obtained using the library are verified with respect to an operational semantics for a small, lazy language; see Section 9. To increase trust in the verification it has been checked mechanically (also using Agda, which doubles as a proof assistant).

[1] Using an earlier, but very similar, version of the library.

The source code for the library, the examples mentioned above, and the mechanisation of the correctness proof are available from the author's web page (currently `http://www.cs.chalmers.se/~nad/`). A technical report also describes the mechanisation in more detail (Danielsson 2007).

To summarise, the contributions of this work are as follows:

- A simple, lightweight library for semiformal verification of the time complexity of a useful class of purely functional data structures.
- The library has been applied to real-world examples.
- The library has a well-defined semantics, and the stated time bounds have been verified with respect to this semantics.
- The correctness proofs have been checked using a proof assistant.

The rest of the paper is structured as follows: Section 2 describes Agda and Section 3 describes the basics of THUNK. The implementation of the library is discussed in Section 4, some rules for how the library must be used are laid down in Section 5, and Sections 6–8 contain further examples on the use of THUNK. The correctness proof is outlined in Sections 9–10, Section 11 motivates and discusses a generalisation of the library, and Section 12 describes some limitations. Finally related work is discussed in Section 13 and Section 14 concludes.

2. Host language

This section discusses some aspects of Agda (Norell 2007; The Agda Team 2007), the language used for the examples in the paper, in order to make it easier to follow the text. Agda is a dependently typed functional language, and for the purposes of this paper it may be useful to think of it as a total variant of Haskell (Peyton Jones 2003) with dependent types and generalised algebraic data types, but no infinite values or coinduction.

THUNK is not tied to Agda, but can be implemented in any language which supports the type system and evaluation orders used, see Sections 4 and 9.

Hidden arguments Agda lacks (implicit) polymorphism, but has *hidden arguments*, which in combination with dependent types compensate for this loss. For instance, the ordinary list function *map* could be given the following type signature:

$$map : \{a, b : \star\} \to (a \to b) \to List\ a \to List\ b$$

Here $\star$ is the type of (small) types. Arguments within $\{\ldots\}$ are hidden, and need not be given explicitly, if the type checker can infer their values from the context in some way. If the hidden arguments cannot be inferred, then they can be given explicitly by enclosing them within $\{\ldots\}$:

$$map\ \{Int\}\ \{Bool\} : (Int \to Bool) \to List\ Int \to List\ Bool$$

The same syntax can be used to pattern match on hidden arguments:

$$map\ \{a\}\ \{b\}\ f\ (x :: xs) = \ldots$$

Inductive families Agda has inductive families (Dybjer 1994), also known as generalised algebraic data types or GADTs. Data types are introduced by listing the constructors and giving their types. Natural numbers, for instance, can be defined as follows:

$$\begin{array}{l} \textbf{data}\ \mathbb{N} : \star\ \textbf{where} \\ \quad \mathsf{zero} : \mathbb{N} \\ \quad \mathsf{suc}\ \ : \mathbb{N} \to \mathbb{N} \end{array}$$

As an example of a *family* of types consider the type *Seq a n* of sequences (sometimes called vectors) of length *n* containing elements of type *a*:

$$\begin{array}{l} \textbf{data}\ Seq\ (a : \star) : \mathbb{N} \to \star\ \textbf{where} \\ \quad \mathsf{nil}\ \ : Seq\ a\ \mathsf{zero} \\ \quad (::) : \{n : \mathbb{N}\} \to a \to Seq\ a\ n \to Seq\ a\ (\mathsf{suc}\ n) \end{array}$$

Note how the *index* (the natural number introduced after the last : in the first line) is allowed to vary between the constructors. *Seq a* is a family of types, with one type for every index *n*.

To illustrate the kind of pattern matching Agda allows for an inductive family, let us define the tail function:

$$\begin{array}{l} tail : \{a : \star\} \to \{n : \mathbb{N}\} \to Seq\ a\ (\mathsf{suc}\ n) \to Seq\ a\ n \\ tail\ (x :: xs) = xs \end{array}$$

We can and need only pattern match on (::), since the type of nil does not match the type *Seq a* (suc *n*) given in the type signature for *tail*. As another example, consider the definition of the append function:

$$\begin{array}{l} (+\!\!+) : Seq\ a\ n_1 \to Seq\ a\ n_2 \to Seq\ a\ (n_1 + n_2) \\ \mathsf{nil} \qquad +\!\!+\ ys = ys \\ (x :: xs) +\!\!+\ ys = x :: (xs +\!\!+ ys) \end{array}$$

In the nil case the variable n_1 in the type signature is unified with zero, transforming the result type into *Seq a* n_2, allowing us to give *ys* as the right-hand side. (This assumes that zero $+ n_2$ evaluates to n_2.) The (::) case can be explained in a similar way.

Note that the hidden arguments of ($+\!\!+$) were not declared in its type signature. This is not allowed by Agda, but often done in the paper to reduce notational noise. Some other minor syntactic changes have also been made in order to aid readability.

Run-time and compile-time code Agda evaluates code during type checking; two types match if they reduce to the same normal form. Hence it is useful to distinguish between compile-time code (code which is only evaluated at type-checking time) and run-time code (code which is executed at run-time). The principal purpose of the THUNK library is to annotate run-time code; the compile-time code will not be executed at run-time anyway, so there is not much point in annotating it.

Unfortunately Agda has no facilities for identifying compile-time or run-time code. As a crude first approximation types are not run-time, though.

3. Library basics

An example will introduce the core concepts of THUNK. By using the library combinators the append function can be proved to be linear in the length of the first sequence:

$$\begin{array}{l} (+\!\!+)\ \ :\ \ Seq\ a\ m \to Seq\ a\ n \\ \qquad\quad \to Thunk\ (1 + 2 * m)\ (Seq\ a\ (m + n)) \\ \mathsf{nil} \qquad +\!\!+\ ys = {}^{\checkmark}\ return\ ys \\ (x :: xs) +\!\!+\ ys = {}^{\checkmark} \\ \quad xs +\!\!+ ys \gg\!= \lambda xsys \to {}^{\checkmark} \\ \quad return\ (x :: xsys) \end{array}$$

The rest of this section explains this example and the library in more detail.

Ticks As mentioned above the user has to insert ticks manually:

$${}^{\checkmark} : Thunk\ n\ a \to Thunk\ (1 + n)\ a$$

The basic unit of cost is the rewriting of the left-hand side of a definition to the right-hand side. Hence, for every function clause, lambda abstraction etc. the user has to insert a tick. (The ${}^{\checkmark}$ function is a prefix operator of low precedence, reducing the need for parentheses.)

By design the library is lightweight: no special language support for reasoning about time complexity is needed. It would be easy to turn the library from being semiformal into being formal by modifying the type-checker of an existing language to ensure that ticks were always inserted where necessary (and a few other requirements listed in Section 5). However, the primary intended use of THUNK is the analysis of complicated data structures; it should not interfere with "ordinary" code. Furthermore the freedom to choose where to insert ticks gives the user the ability to experiment with different cost models.

Thunk monad The type *Thunk* is an "annotated" monad, with the following types for the unit (*return*) and the bind operator ($\gg=$):

$$\begin{aligned}&\mathit{return} : a \to \mathit{Thunk}\ 0\ a\\&(\gg=)\ : \mathit{Thunk}\ m\ a \to (a \to \mathit{Thunk}\ n\ b) \to \mathit{Thunk}\ (m+n)\ b\end{aligned}$$

The monad combinators are used to combine the time complexities of subexpressions. It makes sense to call this a monad since the monoid laws for 0 and $+$ make sure that the monad laws are still "type correct".

Time bounds Let us now discuss the time complexity guarantees established by the library. Assume that t has type

$$a \equiv \mathit{Thunk}\ n_1\ (\mathit{Thunk}\ n_2 \ldots (\mathit{Thunk}\ n_k\ b)\ldots),$$

where b is not itself equal to *Thunk* something. The library then guarantees that, if t evaluates to WHNF, then it does so in at most $n \equiv n_1 + n_2 + \ldots + n_k$ steps. Denote the number n by *time a*.

The precondition that t must evaluate to WHNF is not a problem in Agda, since Agda is a total language. In partial languages one has to be more careful, though. Consider the following code, written in some partial language:

$$\begin{array}{ll}\omega : \mathbb{N} & \mathit{ticks} : \mathit{Thunk}\ \omega\ a\\ \omega = 1 + \omega & \mathit{ticks} = {}^{\checkmark}\,\mathit{ticks}\end{array}$$

The value *ticks* does not have a WHNF. Since Agda is total the precondition above will implicitly be assumed to be satisfied when the examples in the rest of the paper are discussed.

One can often extract more information than is at first obvious from the given time bounds. For instance, take two sequences $xs : \mathit{Seq}\ a\ 7$ and $ys : \mathit{Seq}\ a\ 3$ (for some a). When evaluating $xs +\!\!+\ ys$ a WHNF will be obtained in $\mathit{time}\ (\mathit{Thunk}\ 15\ (\mathit{Seq}\ a\ 10)) = 15$ steps. This WHNF has to be $z :: zs$ for some $z : a$, $zs : \mathit{Seq}\ a\ 9$. Since $\mathit{time}\ (\mathit{Seq}\ a\ 9) = 0$ this means that zs evaluates to WHNF in zero steps. Continuing like this we see that $xs +\!\!+\ ys$ really evaluates to spine-normal form in 15 steps; even normal form if a does not contain embedded *Thunk*s. This example shows that types without embedded *Thunk*s are treated as if they were strict. Section 7 shows how non-strict types can be handled.

Run function There is a need to interface annotated code with "ordinary" code, which does not run in the *Thunk* monad. This is done by the *force* function:

$$\mathit{force} : \mathit{Thunk}\ n\ a \to a$$

This function must of course not be used in code which is analysed.

Equality proofs The Agda type checker does not automatically prove arithmetical equalities. As a result, the definition of $(+\!\!+)$ above does not type check: Agda cannot see that the tick count of the right-hand side of the last equation, $1+((1+2*m)+(1+0))$ (for some variable $m : \mathbb{N}$), is the same as $1 + 2 * (1 + m)$. This problem can be solved by inserting a proof demonstrating the equality of these expressions into the code. The problem is an artifact of Agda, though; simple arithmetical equalities such as the one above could easily be proved automatically, and to aid readability no such proofs are written out in the paper, with the exception of a discussion of equality proofs in Section 10.

Summary The basic version of the library consists of just the *Thunk* monad, $^{\checkmark}$, *force*, and the function *pay*, which is introduced in Section 8; *pay* is the key to taking advantage of lazy evaluation. The following list summarises the primitives introduced so far:

$$\begin{array}{ll}\mathit{Thunk} & : \mathbb{N} \to \star \to \star\\ {}^{\checkmark} & : \mathit{Thunk}\ n\ a \to \mathit{Thunk}\ (1+n)\ a\\ \mathit{return} & : a \to \mathit{Thunk}\ 0\ a\\ (\gg=) & : \mathit{Thunk}\ m\ a \to (a \to \mathit{Thunk}\ n\ b) \to \mathit{Thunk}\ (m+n)\ b\\ \mathit{force} & : \mathit{Thunk}\ n\ a \to a\end{array}$$

4. Implementation

In the implementation of THUNK the type *Thunk n a* is just a synonym for the type a; n is a "phantom type variable" (Leijen and Meijer 1999). However, this equality must not be exposed to the library user. Hence the type is made *abstract*:

abstract
  $\mathit{Thunk} : \mathbb{N} \to \star \to \star$
  $\mathit{Thunk}\ n\ a = a$

Making a type or function abstract means that its defining equations are only visible to other abstract definitions in the same module. Hence, when type checking, if $x : \mathit{Thunk}\ n\ a$, then this reduces to $x : a$ in the right-hand sides of the library primitives below, but in other modules the two types a and *Thunk n a* are different.

The primitive operations of the library are basically identity functions; *return* and ($\gg=$) form an annotated identity monad:

abstract
  ${}^{\checkmark} : \mathit{Thunk}\ n\ a \to \mathit{Thunk}\ (1+n)\ a$
  ${}^{\checkmark}x = x$

  $\mathit{return} : a \to \mathit{Thunk}\ 0\ a$
  $\mathit{return}\ x = x$

  $(\gg=) : \mathit{Thunk}\ m\ a \to (a \to \mathit{Thunk}\ n\ b) \to \mathit{Thunk}\ (m+n)\ b$
  $x \gg= f = f\ x$

  $\mathit{force} : \mathit{Thunk}\ n\ a \to a$
  $\mathit{force}\ x = x$

This ensures minimal run-time overhead, and also that the implementation of THUNK corresponds directly to the erasure function used to prove the library correct (see Section 9.1).

It may be possible to implement a variant of the library in a strict language with explicit support for laziness (with memoisation). The correctness statement and proof, and perhaps also some type signatures, would probably need to be modified, though.

5. Conventions

There are some conventions about how the library must be used which are not captured by the type system:

- Every run-time function clause (including those of anonymous lambdas) has to start with $^{\checkmark}$.
- The function *force* may not be used in run-time terms.
- Library functions may not be used partially applied.

The correctness of the library has only been properly verified for a simple language which enforces all these rules through syntactic restrictions (see Section 9.1); Agda does not, hence these conventions are necessary. Further differences between Agda and the simple language are discussed in Section 10.

The rest of this section discusses and motivates the conventions.

Run-time vs. compile-time It would be very awkward to have to deal with thunks in the *types* of functions, so the rules for $^{\checkmark}$

only apply to terms that will actually be executed at run-time. The function *force* may obviously not be used in run-time terms, since it can be used to discard annotations.

Ticks everywhere One might think that it is possible to omit $\checkmark$ in non-recursive definitions, and still obtain asymptotically correct results. This is not true in general, though. Consider the following function, noting that the last anonymous lambda is not ticked:

```
build : (n : ℕ) → Thunk (1 + 2 ∗ n) (ℕ → Thunk 1 ℕ)
build zero    = ✓ return (λn → ✓ return n)
build (suc n) = ✓
  build n ≫= λf → ✓
  return (λn → f (suc n))
```

The function *build* n, when forced, returns a function $f : \mathbb{N} \to$ *Thunk* $1\ \mathbb{N}$ which adds n to its input. However, f is not a constant-time function, so this is clearly wrong. The problem here is the lambda which we have not paid for.

Partial applications The guarantees given by THUNK are verified by defining a function $\ulcorner\cdot\urcorner$ which erases all the library primitives, and then showing that, for every term t whose *time* is n, the erased term $\ulcorner t\urcorner$ takes at most n steps amortised time to evaluate to WHNF (see Section 9).

Now, $\ulcorner \mathit{return}\ t\urcorner = \ulcorner t\urcorner$, so if partial applications of library functions were allowed we would have $\ulcorner \mathit{return}\urcorner = \lambda x \to x$. However, an application of the identity function takes one step to evaluate, whereas *return* has zero overhead. Hence partial applications of library functions are not allowed. (It may be useful to see them as annotations, as opposed to first-class entities.)

6. Some utility functions

Before moving on to some larger examples a couple of utility functions will be introduced.

When defining functions which have several cases the types of the different case branches have to match. For this purpose the following functions, which increase the tick counts of their arguments, are often useful:

```
wait : (n : ℕ) → Thunk m a → Thunk (1 + n + m) a
wait zero    x = ✓ x
wait (suc n) x = ✓ wait n x

return_w : (n : ℕ) → a → Thunk (1 + n) a
return_w zero    x = ✓ return x
return_w (suc n) x = ✓ return_w n x
```

Note that return_w cannot be defined in terms of *wait*; the extra tick would give rise to a different type:

```
return_w : (n : ℕ) → a → Thunk (2 + n) a
return_w n x = ✓ wait n (return x)
```

Note also that, to improve performance (as opposed to time bounds), it is a good idea to add these functions to the trusted code base:

```
abstract
  wait : (n : ℕ) → Thunk m a → Thunk (1 + n + m) a
  wait _ x = x

  return_w : (n : ℕ) → a → Thunk (1 + n) a
  return_w _ x = x
```

This does not increase the complexity of the main correctness proof, since we know that the functions *could* be implemented in the less efficient way above.

The function $(=\!\!\ll)$, bind with the arguments flipped, is also included in the trusted core:

```
(=≪) : (a → Thunk m b) → Thunk n a → Thunk (n + m) b
f =≪ c = c ≫= f
```

This function does not add any overhead to the correctness proof since it is identical to bind (except for a different argument order). Furthermore it is useful; it is used several times in the next section.

The following thunkified variant of if-then-else will also be used:

```
if_then_else_ : Bool → a → a → Thunk 1 a
if true  then x else y = ✓ return x
if false then x else y = ✓ return y
```

7. Non-strict data types

Data types defined in an ordinary way are treated as strict. In order to get non-strict behaviour *Thunk* has to be used in the definition of the data type. To illustrate this a linear-time function which calculates the minimum element in a non-empty list will be defined by using insertion sort.

First lazy sequences are defined:

```
data Seq_L (a : ⋆) (c : ℕ) : ℕ → ⋆ where
  nil_L  : Seq_L a c 0
  (::_L) : a → Thunk c (Seq_L a c n) → Seq_L a c (1 + n)
```

$\mathit{Seq}_L\ a\ c\ n$ stands for a lazy sequence of length n, containing elements of type a, where every tail takes c steps to force; note the use of *Thunk* in the definition of $(::_L)$. A variant where different tails take different numbers of steps to force is also possible (see Section 11), but not needed here.

The function *insert* inserts an element into a lazy sequence in such a way that if the input is sorted the output will also be sorted. To compare elements *insert* uses the function $(\leq) : a \to a \to$ *Thunk* 1 *Bool*; for simplicity it is assumed that comparisons take exactly one step.[2]

```
insert  :  {c : ℕ} → a → Seq_L a c n
             → Thunk 4 (Seq_L a (4 + c) (1 + n))
insert {c} x nil_L       = ✓
  return_w 2 (x ::_L return_w (3 + c) nil_L)
insert {c} x (y ::_L ys) = ✓
  x ≤ y ≫= λb → ✓
  if b then x ::_L wait (2 + c) (wait_L 2 (y ::_L ys))
       else  y ::_L (insert x =≪ ys)
```

When $x \leq y$ the function wait_L is used to ensure that the resulting sequence has the right type:

```
wait_L  :  (c : ℕ) → Seq_L a c′ n
             → Thunk 1 (Seq_L a (2 + c + c′) n)
wait_L c nil_L        = ✓ return nil_L
wait_L c (x ::_L xs)  = ✓ return (x ::_L wait c (wait_L c =≪ xs))
```

By using wait_L all elements in the tail get assigned higher tick counts than necessary. It would be possible to give *insert* a more precise type which did not overestimate any tick counts, but this type would be rather complicated. The type used here is a compromise which is simple to use and still precise enough.

Note that the library does not give any help with solving recurrence equations; it just checks the solution encoded by the user through type signatures and library primitives. (The arguments to functions like *wait* can often be inferred automatically in Agda, obviating the need for the user to write them. For clarity they are included here, though.)

[2] Agda has parameterised modules, so $(\leq)$ does not need to be an explicit argument to *insert*.

Insertion sort, which takes an ordinary sequence as input but gives a lazy sequence as output, can now be defined as follows:

```
sort : Seq a n → Thunk (1 + 5 * n) (Seq_L a (4 * n) n)
sort nil       =✓ return nil_L
sort (x :: xs) =✓ insert x =≪ sort xs
```

Note that the time needed to access the first element of the result is linear in the length of the input, whereas the time needed to force the entire result is quadratic. Using *sort* and *head* the *minimum* function can easily be defined for non-empty sequences:

```
head : Seq_L a c (1 + n) → Thunk 1 a
head (x ::_L xs) =✓ return x

minimum : Seq a (1 + n) → Thunk (8 + 5 * n) a
minimum xs =✓ head =≪ sort xs
```

As a comparison it can be instructive to see that implementing *maximum* using insertion sort and *last* can lead to quadratic behaviour:

```
last : Seq_L a c (1 + n) → Thunk (1 + (1 + n) * (1 + c)) a
last (x ::_L xs) =✓ last′ x =≪ xs
  where
  last′ : a → Seq_L a c n → Thunk (1 + n * (1 + c)) a
  last′ x nil_L        =✓ return x
  last′ x (y ::_L ys) =✓ last′ y =≪ ys

maximum : Seq a (1 + n) → Thunk (13 + 14 * n + 4 * n^2) a
maximum xs =✓ last =≪ sort xs
```

Fortunately there are better ways to implement this function.

8. Essential laziness

The time bound of the *minimum* function only requires non-strictness, not memoisation. To make use of laziness to obtain better time bounds *pay* can be used:

```
abstract
  pay : (m : ℕ) → Thunk n a → Thunk m (Thunk (n − m) a)
  pay _ x = x
```

(Here $n - m = 0$ whenever $n < m$.)

The correctness of *pay* is obvious, since

$$time\ (Thunk\ n\ a) \leq time\ (Thunk\ m\ (Thunk\ (n-m)\ a)).$$

However, more intuition may be provided by the following interpretations of *pay*:

1. When *pay m t* is executed (as part of a sequence of binds) the thunk *t* is executed for *m* steps and then suspended again.
2. When *pay m t* is executed the thunk *t* is returned immediately, but with a new type. If *t* is never forced, then we have paid *m* steps too much. If *t* is forced exactly once, then we have paid the right amount. And finally, if *t* is forced several times, then it is memoised the first time and later the memoised value is used, so the amount paid is still a correct upper bound.

The first way of thinking about *pay* may be more intuitive. Furthermore, if it could be implemented, it would lead to worst-case, instead of amortised, time bounds (assuming a suitably strict semantics). However, the extra bookkeeping needed by the first approach seems to make it hard to implement without non-constant overheads; consider nested occurrences of *pay*.

8.1 Implicit queues

The interpretations above do not explain why *pay* is useful. To do this I will implement implicit queues (Okasaki 1998), FIFO queues with constant-time head, snoc and tail.[3] In this example using *pay* corresponds to paying off so-called *debits* in Okasaki's banker's method (1998), hence the name.

When using the THUNK library debits are represented explicitly using thunked arguments in data type definitions. Implicit queues are represented by the following nested data type:

```
data Q (a : ⋆) : ⋆ where
  empty   : Q a
  single  : a → Q a
  twoZero : a → a → Thunk 5 (Q (a × a))       → Q a
  twoOne  : a → a → Thunk 3 (Q (a × a)) → a → Q a
  oneZero : a →     Thunk 2 (Q (a × a))       → Q a
  oneOne  : a →             Q (a × a)   → a → Q a
```

The recursive constructors take queues of pairs of elements, placed after the first one or two elements, and before the last zero or one elements. Okasaki's analysis puts a certain number of debits on the various subqueues. These invariants are reflected in the thunks above (modulo some details in the analysis).

The *snoc* function adds one element to the end of a queue. Okasaki's analysis tells us that this function performs $\mathcal{O}(1)$ unshared work and discharges a certain number of debits. We do not need to keep these two concepts separate (even though we could), hence the following type for *snoc*:

```
snoc : Q a → a → Thunk 5 (Q a)
snoc empty                x₁ =✓ return_w 3 (single x₁)
snoc (single x₁)          x₂ =✓
  return_w 3 (twoZero x₁ x₂ (return_w 4 empty))
snoc (twoZero x₁ x₂ xs₃)    x₄ =✓
  pay 2 xs₃ ≫= λxs′₃ → ✓
  return_w 0 (twoOne x₁ x₂ xs′₃ x₄)
snoc (twoOne x₁ x₂ xs₃ x₄) x₅ =✓
  xs₃ ≫= λxs′₃ → ✓
  return (twoZero x₁ x₂ (snoc xs′₃ (x₄, x₅)))
snoc (oneZero x₁ xs₂)       x₃ =✓
  xs₂ ≫= λxs′₂ → ✓
  return_w 0 (oneOne x₁ xs′₂ x₃)
snoc (oneOne x₁ xs₂ x₃)    x₄ =✓
  pay 3 (snoc xs₂ (x₃, x₄)) ≫= λxs₂₃₄ → ✓
  return (oneZero x₁ xs₂₃₄)
```

Note how the invariants encoded in the data structure, together with the use of *pay*, ensure that we can show that the function takes constant amortised time even though it is recursive.

Note also that using call-by-value or call-by-name to evaluate *snoc* leads to worse time bounds. Consider a "saturated" queue *q*, built up by repeated application of *snoc* to empty:

$$q = \mathsf{twoOne}\ x\ x\ (\mathsf{twoOne}\ (x,x)\ (x,x)\ (\ldots \mathsf{empty} \ldots)\ (x,x))\ x$$

In a strict setting it takes $\mathcal{O}(d)$ steps to evaluate *snoc q x*, where *d* is the depth of the queue. If *snoc q x* is evaluated *k* times, this will take $\mathcal{O}(kd)$ steps, and by choosing *k* high enough it can be ensured that the average number of steps needed by *snoc* is not constant. If call-by-name is used instead, then the lack of memoisation means that $q = snoc\ (snoc\ (\ldots \mathsf{empty} \ldots)\ x)\ x$ will be evaluated to WHNF each time *snoc q x* is forced, leading to a similar situation.

It remains to define the $view_{\prec}$ function (view left), which gives the first element and the rest of the queue. Forcing the tail takes longer than just viewing the head, so the following data type is defined to wrap up the result of $view_{\prec}$:

[3] The presentation used here is due to Ross Paterson (personal communication), with minor changes.

```
data View≺ (a : ⋆) : ⋆ where
  nil≺  : View≺ a
  cons≺ : a → Thunk 4 (Q a) → View≺ a
```

The function itself is defined as follows:

```
view≺ : {a : ⋆} → Q a → Thunk 1 (View≺ a)
view≺ empty        = ✓ return nil≺
view≺ (single x1) = ✓
  return (cons≺ x1 (return_w 3 empty))
view≺ (twoZero x1 x2 xs3) = ✓ return (cons≺ x1
  (pay 3 xs3 ≫= λxs'3 → ✓
  return (oneZero x2 xs'3)))
view≺ (twoOne x1 x2 xs3 x4) = ✓ return (cons≺ x1
  (xs3 ≫= λxs'3 → ✓
  return (oneOne x2 xs'3 x4)))
view≺ {a} (oneZero x1 xs2) = ✓
  return (cons≺ x1 (expand =≪ view≺ =≪ xs2))
  where
  expand : View≺ (a × a) → Thunk 1 (Q a)
  expand nil≺                     = ✓ return empty
  expand (cons≺ (y1, y2) ys3) = ✓
    return (twoZero y1 y2 (wait 0 ys3))
view≺ {a} (oneOne x1 xs2 x3) = ✓
  return (cons≺ x1 (expand =≪ view≺ xs2))
  where
  expand : View≺ (a × a) → Thunk 3 (Q a)
  expand nil≺                     = ✓ return_w 1 (single x3)
  expand (cons≺ (y1, y2) ys3) = ✓
    pay 1 ys3 ≫= λys'3 → ✓
    return (twoOne y1 y2 ys'3 x3)
```

8.2 Calculating invariants

It should be noted that the library does not help much with the *design* of efficient data structures, except perhaps by providing a clear model of certain aspects of lazy evaluation. It may still be instructive to see how the invariants used above can be obtained. Assuming that the general structure of the code has been decided, that the code is expected to be constant-time, and that the number of debits on all the subqueues is also expected to be constant, this is how it can be done:

1. Make all the subqueues thunked, also the one in the oneOne constructor.
2. Denote the time bounds and the number of debits on the various subqueues by variables. For instance:

   ```
   oneZero : a → Thunk d10 (Q (a × a))      → Q a
   oneOne  : a → Thunk d11 (Q (a × a)) → a → Q a
   ```

3. Assume a worst case scenario for how many *pay* annotations etc. are necessary. Calculate the amounts to pay using the variables introduced in the previous step. For instance, the oneOne case of *snoc* takes the following form, if s is the time bound for *snoc*:

   ```
   snoc (oneOne x1 xs2 x3) x4 = ✓
     xs2                                   ≫= λxs'2   → ✓
     pay (s − d10) (snoc xs'2 (x3, x4)) ≫= λxs234 → ✓
     return? (oneZero x1 xs234)
   ```

   (The function *return*? could be either *return* or $return_w$ n, for some n, depending on the outcome of the analysis.)

4. The basic structure of the code now gives rise to a number of inequalities which have to be satisfied in order for the code to be well-typed. For instance, the oneOne case of *snoc* gives rise to the inequality $3 + d_{11} + (s - d_{10}) \leq s$. Solve these inequalities. If no solution can be found, then one of the assumptions above was incorrect.
5. Optionally: If, for instance, a *pay* annotation was unnecessary, then it may be possible to tighten the time bounds a little, since a tick can be removed.

9. Correctness

The correctness of the library is established as follows:

1. Two small languages are defined: the *simple* one without the *Thunk* type, and the *thunked* one with the library functions as primitives. An erasure function $\ulcorner\cdot\urcorner$ converts thunked terms to simple ones.
2. A lazy operational semantics is defined for the simple language, and another operational semantics is defined for the thunked language. It is shown that, under erasure, the thunked semantics is equivalent to the simple one.
3. The THUNK library guarantees (see Section 3) are established for the thunked semantics. Since the two semantics are equivalent this implies that the guarantees hold for erased terms evaluated using the simple semantics.

As shown in Section 4 the library is implemented by ignoring all annotations. Hence what is actually run corresponds directly to the erased terms mentioned above, so the correctness guarantees extend also to the actual library (assuming that Agda has an operational semantics corresponding to the one defined for the simple language; currently Agda does not have an operational semantics). There are two caveats to this statement. One is the time needed to evaluate the library functions. However, they all evaluate in constant time, and I find it reasonable to ignore these times. The other is the difference between the small languages defined here and a full-scale language like Agda. These differences are discussed further in Section 10.

All nontrivial results discussed in this section have been proved formally using Agda.[4] There are some differences between the formalisation presented here and the mechanised one, most notably that de Bruijn indices are used to represent variables in the mechanisation. The verification of some of the extensions discussed in Sections 10–11 have also been mechanised. For more details, see Danielsson (2007).

9.1 Languages

Both of the two small languages are *simply* typed lambda calculi with natural numbers and products. Using dependently typed languages for the correctness proof would be possible, but this aspect of the type systems appears to be largely orthogonal to the correctness result. Furthermore it would have been considerably harder to mechanise the proofs. Hence I chose to use simply typed languages.

The syntax of contexts, types and terms for the *simple* language is defined as follows (with x, y variables):

$$
\begin{aligned}
\Gamma &::= \epsilon \mid \Gamma, x : \tau \\
\tau &::= \mathsf{Nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \\
t &::= x \mid \lambda x.t \mid t_1 \cdot t_2 \\
&\quad\mid (t_1, t_2) \mid \mathsf{uncurry}\ (\lambda xy.t) \\
&\quad\mid \mathsf{z} \mid \mathsf{s}\ t \mid \mathsf{natrec}\ t_1\ (\lambda xy.t_2)
\end{aligned}
$$

Here natrec is the primitive recursion combinator for natural numbers, and uncurry is the corresponding combinator for products.

[4] Agda currently does not check that all definitions by pattern matching are exhaustive. Hence care has been taken to check this manually.

Common typing rules

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 \cdot t_2 : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma, x : \tau_1, y : \tau_2 \vdash t : \tau}{\Gamma \vdash \mathsf{uncurry}\ (\lambda xy.t) : \tau_1 \times \tau_2 \to \tau} \qquad \frac{}{\Gamma \vdash \mathsf{z} : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{s}\ t : \mathsf{Nat}} \qquad \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma, x : \mathsf{Nat}, y : \tau \vdash t_2 : \tau}{\Gamma \vdash \mathsf{natrec}\ t_1\ (\lambda xy.t_2) : \mathsf{Nat} \to \tau}$$

Extra typing rules for the thunked language

$$\frac{\Gamma \vdash t : \mathsf{Thunk}\ n\ \tau}{\Gamma \vdash {}^{\checkmark}t : \mathsf{Thunk}\ (1 + n)\ \tau} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{return}\ t : \mathsf{Thunk}\ 0\ \tau}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Thunk}\ n_1\ \tau_1 \qquad \Gamma \vdash t_2 : \tau_1 \to \mathsf{Thunk}\ n_2\ \tau_2}{\Gamma \vdash t_1 \gg\!= t_2 : \mathsf{Thunk}\ (n_1 + n_2)\ \tau_2}$$

$$\frac{\Gamma \vdash t : \mathsf{Thunk}\ n\ \tau}{\Gamma \vdash \mathsf{force}\ t : \tau}$$

$$\frac{\Gamma \vdash t : \mathsf{Thunk}\ n\ \tau}{\Gamma \vdash \mathsf{pay}\ m\ t : \mathsf{Thunk}\ m\ (\mathsf{Thunk}\ (n - m)\ \tau)}$$

Figure 1. The type systems. All freshness side conditions have been omitted.

The *thunked* language extends the syntax with the library primitives as follows:

$$\begin{aligned} \tau &::= \ldots \mid \mathsf{Thunk}\ n\ \tau \\ t &::= \ldots \mid {}^{\checkmark}t \mid \mathsf{return}\ t \mid t_1 \gg\!= t_2 \mid \mathsf{force}\ t \mid \mathsf{pay}\ n\ t \end{aligned}$$

Here n stands for a natural number, *not* a term of type Nat.

The type systems for the two languages are given in Figure 1. In the remaining text only well-typed terms are considered. No type annotations are present in the syntax above, but this is just to simplify the presentation. The mechanised versions of the languages are fully annotated.

As noted above an erasure operation taking types and terms from the thunked language to the simple one is defined:

$$\begin{aligned} \ulcorner \mathsf{Nat} \urcorner &= \mathsf{Nat} \\ \ulcorner \tau_1 \times \tau_2 \urcorner &= \ulcorner \tau_1 \urcorner \times \ulcorner \tau_2 \urcorner \\ \ulcorner \tau_1 \to \tau_2 \urcorner &= \ulcorner \tau_1 \urcorner \to \ulcorner \tau_2 \urcorner \\ \ulcorner \mathsf{Thunk}\ n\ \tau \urcorner &= \ulcorner \tau \urcorner \end{aligned}$$

$$\begin{aligned} \ulcorner x \urcorner &= x \\ \ulcorner \lambda x.t \urcorner &= \lambda x.\ulcorner t \urcorner \\ \ulcorner t_1 \cdot t_2 \urcorner &= \ulcorner t_1 \urcorner \cdot \ulcorner t_2 \urcorner \\ \ulcorner (t_1, t_2) \urcorner &= (\ulcorner t_1 \urcorner, \ulcorner t_2 \urcorner) \\ \ulcorner \mathsf{uncurry}\ (\lambda xy.t) \urcorner &= \mathsf{uncurry}\ (\lambda xy.\ulcorner t \urcorner) \\ \ulcorner \mathsf{z} \urcorner &= \mathsf{z} \\ \ulcorner \mathsf{s}\ t \urcorner &= \mathsf{s}\ \ulcorner t \urcorner \\ \ulcorner \mathsf{natrec}\ t_1\ (\lambda xy.t_2) \urcorner &= \mathsf{natrec}\ \ulcorner t_1 \urcorner\ (\lambda xy.\ulcorner t_2 \urcorner) \\ \ulcorner {}^{\checkmark}t \urcorner &= \ulcorner t \urcorner \\ \ulcorner \mathsf{return}\ t \urcorner &= \ulcorner t \urcorner \\ \ulcorner t_1 \gg\!= t_2 \urcorner &= \ulcorner t_2 \urcorner \cdot \ulcorner t_1 \urcorner \\ \ulcorner \mathsf{force}\ t \urcorner &= \ulcorner t \urcorner \\ \ulcorner \mathsf{pay}\ n\ t \urcorner &= \ulcorner t \urcorner \end{aligned}$$

Erasure extends in a natural way to contexts, and term erasure can easily be verified to preserve types,

$$\Gamma \vdash t : \tau \quad \Rightarrow \quad \ulcorner \Gamma \urcorner \vdash \ulcorner t \urcorner : \ulcorner \tau \urcorner.$$

Free use of force or failure to insert ticks would invalidate all time complexity guarantees, so a subset of the thunked language is defined, the *run-time terms*:

$$\begin{aligned} e ::= {} & x \mid \lambda x.{}^{\checkmark}e \mid e_1 \cdot e_2 \\ & \mid (e_1, e_2) \mid \mathsf{uncurry}\ (\lambda xy.{}^{\checkmark}e) \\ & \mid \mathsf{z} \mid \mathsf{s}\ e \mid \mathsf{natrec}\ ({}^{\checkmark}e_1)\ (\lambda xy.{}^{\checkmark}e_2) \\ & \mid {}^{\checkmark}e \mid \mathsf{return}\ e \mid e_1 \gg\!= e_2 \mid \mathsf{pay}\ n\ e \end{aligned}$$

Note that all the conventions set up in Section 5 are satisfied by the run-time terms: every "right-hand side" starts with a tick, force is not used, and library functions cannot be used partially applied.

9.2 Operational semantics

Let us now define the operational semantics for the two languages. The semantics, which are inspired by Launchbury's semantics for lazy evaluation (1993), define how to evaluate a term to WHNF.

Simple semantics We begin with the semantics for the simple language. Terms are evaluated in heaps (or environments); lists of bindings of variables to terms:

$$\Sigma ::= \emptyset \mid \Sigma, x \mapsto t$$

Heaps have to be well-typed with respect to a context:

$$\epsilon \vdash \emptyset \qquad \frac{\Gamma \vdash \Sigma \qquad \Gamma \vdash t : \tau}{\Gamma, x : \tau \vdash \Sigma, x \mapsto t}\ (x \text{ fresh})$$

Note that these rules ensure that there are no cycles in the heap. This is OK since there is no recursive let allowing the definition of cyclic structures, and even if there were a recursive let the THUNK library would not be able to make use of the extra sharing anyway.

A subset of the terms are identified as being values:

$$\begin{aligned} v ::= {} & \lambda x.t \\ & \mid (x_1, x_2) \mid \mathsf{uncurry}\ (\lambda xy.t) \\ & \mid \mathsf{z} \mid \mathsf{s}\ x \mid \mathsf{natrec}\ x_1\ (\lambda xy.t_2) \end{aligned}$$

Note that these values are all in WHNF. Several of the constructors take variables as arguments; this is to increase sharing. Once again, THUNK cannot take advantage of this sharing, but I have tried to keep the semantics close to what a real-world lazy language would use. (Furthermore the generalised version of the library, described in Section 11, can take advantage of some of this sharing.)

The big-step operational semantics for the simple language is inductively defined in Figure 2. The notation $\Sigma_1 \mid t \Downarrow^n \Sigma_2 \mid v$ means that t, when evaluated in the heap Σ_1, reaches the WHNF v in n steps; the resulting heap is Σ_2. (Here it is assumed that Σ_1 and t are well-typed with respect to the *same* context.) In order to reduce duplication of antecedents an auxiliary relation is used to handle application: $\Sigma_1 \mid v_1 \bullet x_2 \Downarrow^n \Sigma_2 \mid v$ means that the application of the value v_1 to the variable x_2 evaluates to v in n steps, with initial heap Σ_1 and final heap Σ_2.

In the description of the semantics all variables are assumed to be globally unique (by renaming, if necessary). The mechanised version of the semantics uses de Bruijn indices, so name clashes are not an issue there.

The semantics is syntax-directed, and hence deterministic. Furthermore types are preserved,

$$\begin{aligned} & \Gamma_1 \vdash \Sigma_1 \ \wedge\ \Gamma_1 \vdash t : \tau \ \wedge\ \Sigma_1 \mid t \Downarrow^n \Sigma_2 \mid v \quad \Rightarrow \\ & \qquad \exists\, \Gamma_2.\ \Gamma_2 \vdash \Sigma_2 \ \wedge\ \Gamma_2 \vdash v : \tau. \end{aligned}$$

In the mechanisation this is true by construction. It is easy to make small mistakes when formalising languages, and working

Values

$$\frac{}{\Sigma \mid \lambda x.t \Downarrow^0 \Sigma \mid \lambda x.t}$$

$$\frac{}{\Sigma \mid (t_1, t_2) \Downarrow^0 \Sigma, x_1 \mapsto t_1, x_2 \mapsto t_2 \mid (x_1, x_2)}$$

$$\frac{}{\Sigma \mid \mathsf{uncurry}\ (\lambda xy.t) \Downarrow^0 \Sigma \mid \mathsf{uncurry}\ (\lambda xy.t)}$$

$$\frac{}{\Sigma \mid \mathsf{z} \Downarrow^0 \Sigma \mid \mathsf{z}} \qquad \frac{}{\Sigma \mid \mathsf{s}\ t \Downarrow^0 \Sigma, x \mapsto t \mid \mathsf{s}\ x}$$

$$\frac{}{\Sigma \mid \mathsf{natrec}\ t_1\ (\lambda xy.t_2) \Downarrow^0 \Sigma, x_1 \mapsto t_1 \mid \mathsf{natrec}\ x_1\ (\lambda xy.t_2)}$$

Variables

$$\frac{\Sigma_1 \mid t \Downarrow^n \Sigma_2 \mid v}{\Sigma_1, x \mapsto t, \Sigma' \mid x \Downarrow^n \Sigma_2, x \mapsto v, \Sigma' \mid v}$$

Application

$$\frac{\Sigma_1 \mid t_1 \Downarrow^{n_1} \Sigma_2 \mid v_1 \qquad \Sigma_2, x_2 \mapsto t_2 \mid v_1 \bullet x_2 \Downarrow^{n_2} \Sigma_3 \mid v}{\Sigma_1 \mid t_1 \cdot t_2 \Downarrow^{n_1+n_2} \Sigma_3 \mid v}$$

$$\frac{\Sigma_1 \mid t_1[x := x_2] \Downarrow^n \Sigma_2 \mid v}{\Sigma_1 \mid (\lambda x.t_1) \bullet x_2 \Downarrow^{1+n} \Sigma_2 \mid v}$$

$$\frac{\begin{array}{c}\Sigma_1 \mid x_2 \Downarrow^{n_1} \Sigma_2 \mid (x_3, y_3) \\ \Sigma_2 \mid t_1[x := x_3, y := y_3] \Downarrow^{n_2} \Sigma_3 \mid v\end{array}}{\Sigma_1 \mid \mathsf{uncurry}\ (\lambda xy.t_1) \bullet x_2 \Downarrow^{1+n_1+n_2} \Sigma_3 \mid v}$$

$$\frac{\Sigma_1 \mid x_3 \Downarrow^{n_1} \Sigma_2 \mid \mathsf{z} \qquad \Sigma_2 \mid x_1 \Downarrow^{n_2} \Sigma_3 \mid v}{\Sigma_1 \mid \mathsf{natrec}\ x_1\ (\lambda xy.t_2) \bullet x_3 \Downarrow^{1+n_1+n_2} \Sigma_3 \mid v}$$

$$\frac{\begin{array}{c}\Sigma_1 \mid x_3 \Downarrow^{n_1} \Sigma_2 \mid \mathsf{s}\ x' \\ \Sigma_2, y \mapsto \mathsf{natrec}\ x_1\ (\lambda xy.t_2) \cdot x' \mid t_2[x := x'] \Downarrow^{n_2} \Sigma_3 \mid v\end{array}}{\Sigma_1 \mid \mathsf{natrec}\ x_1\ (\lambda xy.t_2) \bullet x_3 \Downarrow^{1+n_1+n_2} \Sigma_3 \mid v}$$

Figure 2. Operational semantics for the simple language. Note that only reductions (rules with $\bullet$ in the left-hand side) contribute to the cost of a computation.

with well-typed syntax is nice since many mistakes are caught early.

The cost model used by the semantics is that of the THUNK library: only reductions cost something. Nothing is charged for looking up variables (i.e. following pointers into the heap), for instance.

Thunked semantics Now on to the thunked semantics, which only applies to run-time terms. In order to be able to prove correctness the thunked semantics has more structure in the heap:

$$\Sigma ::= \emptyset \mid \Sigma, x \mapsto e \mid \Sigma, x \mapsto^n e$$

As before, only well-typed heaps are considered:

$$\epsilon \vdash \emptyset \qquad \frac{\Gamma \vdash \Sigma \qquad \Gamma \vdash e : \tau}{\Gamma, x : \tau \vdash \Sigma, x \mapsto e}\ (x \text{ fresh})$$

$$\frac{\Gamma \vdash \Sigma \qquad \Gamma \vdash e : \mathsf{Thunk}\ n\ \tau}{\Gamma, x : \tau \vdash \Sigma, x \mapsto^n e}\ (x \text{ fresh})$$

The $x \mapsto^n e$ bindings are used to keep track of terms which have already been paid off, but not yet evaluated. The *credit* associated with a heap is the total tick count of such bindings:

$$\begin{aligned} &\mathit{credit}\ \emptyset &&= 0 \\ &\mathit{credit}\ (\Sigma, x \mapsto e) &&= \mathit{credit}\ \Sigma \\ &\mathit{credit}\ (\Sigma, x \mapsto^n e) &&= \mathit{credit}\ \Sigma + n \end{aligned}$$

The credit will be used to state the correctness result later.

The thunked semantics uses the following values, which are all run-time:

$$\begin{aligned} v ::= {} & \lambda x.^{\checkmark} e \\ & \mid (x_1, x_2) \mid \mathsf{uncurry}\ (\lambda xy.^{\checkmark} e) \\ & \mid \mathsf{z} \mid \mathsf{s}\ x \mid \mathsf{natrec}\ (^{\checkmark} x_1)\ (\lambda xy.^{\checkmark} e_2) \\ & \mid \mathsf{return}^n\ v \end{aligned}$$

Here $\mathsf{return}^n\ v$ stands for n applications of $^{\checkmark}$ to $\mathsf{return}\ v$.

The thunked semantics, denoted by $\Sigma_1 \mid e \downdownarrows^n \Sigma_2 \mid v$, is given in Figure 3 (and presented with the same assumptions as the previous one). The thunked semantics preserves types, analogously to the simple one. Note that only binds ($\gg=$) introduce bindings of the form $x \mapsto^n e$, and that when a variable x bound like this is evaluated, it is updated with an unannotated binding; this memoisation is the one tracked by the library.

The following small example illustrates what can be derived using the thunked semantics:

$$\emptyset, x \mapsto^1 (\lambda y.^{\checkmark} \mathit{return}\ y) \cdot \mathsf{z} \mid x \downdownarrows^1 \emptyset, y \mapsto \mathsf{z}, x \mapsto \mathsf{z} \mid \mathsf{z}.$$

Note that $x : \mathsf{Nat}$ and $\mathit{time}\ \mathsf{Nat} = 0$, but the evaluation takes one step; the change in heap credit "pays" for this step (compare with the invariant in Section 9.3).

Equivalence The thunked semantics is both sound,

$$\Sigma_1 \mid e \downdownarrows^n \Sigma_2 \mid v \quad \Rightarrow \quad \ulcorner \Sigma_1 \urcorner \mid \ulcorner e \urcorner \Downarrow^n \ulcorner \Sigma_2 \urcorner \mid \ulcorner v \urcorner,$$

and complete,

$$\begin{gathered} \ulcorner \Sigma_1 \urcorner \mid \ulcorner e \urcorner \Downarrow^n \Sigma_2 \mid v \quad \Rightarrow \\ \exists\ \Sigma_2', v'.\ \ulcorner \Sigma_2' \urcorner = \Sigma_2 \ \wedge\ \ulcorner v' \urcorner = v \ \wedge\ \Sigma_1 \mid e \downdownarrows^n \Sigma_2' \mid v', \end{gathered}$$

with respect to the simple one. (Here erasure has been extended in the obvious way to heaps.) These properties are almost trivial, since the rules for the two semantics are identical up to erasure, and can be proved by induction over the structure of derivations. Some auxiliary lemmas, such as $\ulcorner e[x := y] \urcorner = \ulcorner e \urcorner [x := y]$, need to be proved as well.

9.3 Time complexity guarantees

Now that we know that the two semantics are equivalent the only thing remaining is to verify the time complexity guarantees for the thunked semantics. It is straightforward to prove by induction over the structure of derivations that the following invariant holds:

$$\frac{\Gamma \vdash e : \tau \qquad \Sigma_1 \mid e \downdownarrows^n \Sigma_2 \mid v}{\mathit{credit}\ \Sigma_2 + n \leq \mathit{credit}\ \Sigma_1 + \mathit{time}\ \tau}$$

(The *time* function was introduced in Section 3.) Note that when a computation is started in an empty heap this invariant implies that $n \leq \mathit{time}\ \tau$, i.e. the time bound given by the type is an upper bound on the actual number of computation steps. In the general case the inequality says that $\mathit{time}\ \tau$ is an upper bound on the actual number of steps plus the increase in heap credit (which may sometimes be negative), i.e. $\mathit{time}\ \tau$ is an upper bound on the amortised time complexity with respect to the heap credit.

By using completeness the invariant above can be simplified:

$$\frac{\Gamma \vdash e : \tau \qquad \ulcorner \Sigma_1 \urcorner \mid \ulcorner e \urcorner \Downarrow^n \Sigma_2 \mid v}{n \leq \mathit{credit}\ \Sigma_1 + \mathit{time}\ \tau}$$

Values

$$\frac{}{\Sigma \mid \lambda x.{}^{\checkmark} e \downdownarrows^{0} \Sigma \mid \lambda x.{}^{\checkmark} e} \qquad \frac{}{\Sigma \mid (e_1, e_2) \downdownarrows^{0} \Sigma, x_1 \mapsto e_1, x_2 \mapsto e_2 \mid (x_1, x_2)} \qquad \frac{}{\Sigma \mid \mathsf{uncurry}\ (\lambda xy.{}^{\checkmark} e) \downdownarrows^{0} \Sigma \mid \mathsf{uncurry}\ (\lambda xy.{}^{\checkmark} e)}$$

$$\frac{}{\Sigma \mid \mathsf{z} \downdownarrows^{0} \Sigma \mid \mathsf{z}} \qquad \frac{}{\Sigma \mid \mathsf{s}\, e \downdownarrows^{0} \Sigma, x \mapsto e \mid \mathsf{s}\, x} \qquad \frac{}{\Sigma \mid \mathsf{natrec}\ ({}^{\checkmark} e_1)\ (\lambda xy.{}^{\checkmark} e_2) \downdownarrows^{0} \Sigma, x_1 \mapsto e_1 \mid \mathsf{natrec}\ ({}^{\checkmark} x_1)\ (\lambda xy.{}^{\checkmark} e_2)}$$

Variables

$$\frac{\Sigma_1 \mid e \downdownarrows^{n} \Sigma_2 \mid v}{\Sigma_1, x \mapsto e, \Sigma' \mid x \downdownarrows^{n} \Sigma_2, x \mapsto v, \Sigma' \mid v} \qquad \frac{\Sigma_1 \mid e \downdownarrows^{n} \Sigma_2 \mid \mathsf{return}^{m}\ v}{\Sigma_1, x \mapsto^{m} e, \Sigma' \mid x \downdownarrows^{n} \Sigma_2, x \mapsto v, \Sigma' \mid v}$$

Library primitives

$$\frac{\Sigma_1 \mid e \downdownarrows^{n} \Sigma_2 \mid \mathsf{return}^{m}\ v}{\Sigma_1 \mid {}^{\checkmark} e \downdownarrows^{n} \Sigma_2 \mid \mathsf{return}^{1+m}\ v} \qquad \frac{\Sigma_1 \mid e \downdownarrows^{n} \Sigma_2 \mid v}{\Sigma_1 \mid \mathsf{return}\ e \downdownarrows^{n} \Sigma_2 \mid \mathsf{return}^{0}\ v} \qquad \frac{\Sigma_1 \mid e \downdownarrows^{n} \Sigma_2 \mid \mathsf{return}^{m_1}\ v}{\Sigma_1 \mid \mathsf{pay}\ m_2\ e \downdownarrows^{n} \Sigma_2 \mid \mathsf{return}^{m_2}\ (\mathsf{return}^{m_1 - m_2}\ v)}$$

$$\frac{\Sigma_1 \mid e_2 \downdownarrows^{n_1} \Sigma_2 \mid v_2 \qquad \Sigma_2, x_1 \mapsto^{m_1} e_1 \mid v_2 \bullet x_1 \downdownarrows^{n_2} \Sigma_3 \mid \mathsf{return}^{m_2}\ v}{\Sigma_1 \mid e_1 \gg\!= e_2 \downdownarrows^{n_1+n_2} \Sigma_3 \mid \mathsf{return}^{m_1+m_2}\ v}\ (\text{if } \Gamma_1 \vdash e_1 : \mathit{Thunk}\ m_1\ \tau_1)$$

Application

$$\frac{\Sigma_1 \mid e_1 \downdownarrows^{n_1} \Sigma_2 \mid v_1 \qquad \Sigma_2, x_2 \mapsto e_2 \mid v_1 \bullet x_2 \downdownarrows^{n_2} \Sigma_3 \mid v}{\Sigma_1 \mid e_1 \cdot e_2 \downdownarrows^{n_1+n_2} \Sigma_3 \mid v} \qquad \frac{\Sigma_1 \mid e_1[x := x_2] \downdownarrows^{n} \Sigma_2 \mid \mathsf{return}^{m}\ v}{\Sigma_1 \mid (\lambda x.{}^{\checkmark} e_1) \bullet x_2 \downdownarrows^{1+n} \Sigma_2 \mid \mathsf{return}^{1+m}\ v}$$

$$\frac{\Sigma_1 \mid x_2 \downdownarrows^{n_1} \Sigma_2 \mid (x_3, y_3) \qquad \Sigma_2 \mid e_1[x := x_3, y := y_3] \downdownarrows^{n_2} \Sigma_3 \mid \mathsf{return}^{m}\ v}{\Sigma_1 \mid \mathsf{uncurry}\ (\lambda xy.{}^{\checkmark} e_1) \bullet x_2 \downdownarrows^{1+n_1+n_2} \Sigma_3 \mid \mathsf{return}^{1+m}\ v}$$

$$\frac{\Sigma_1 \mid x_3 \downdownarrows^{n_1} \Sigma_2 \mid \mathsf{z} \qquad \Sigma_2 \mid x_1 \downdownarrows^{n_2} \Sigma_3 \mid \mathsf{return}^{m}\ v}{\Sigma_1 \mid \mathsf{natrec}\ ({}^{\checkmark} x_1)\ (\lambda xy.{}^{\checkmark} e_2) \bullet x_3 \downdownarrows^{1+n_1+n_2} \Sigma_3 \mid \mathsf{return}^{1+m}\ v}$$

$$\frac{\Sigma_1 \mid x_3 \downdownarrows^{n_1} \Sigma_2 \mid \mathsf{s}\, x' \qquad \Sigma_2, y \mapsto \mathsf{natrec}\ ({}^{\checkmark} x_1)\ (\lambda xy.{}^{\checkmark} e_2) \cdot x' \mid e_2[x := x'] \downdownarrows^{n_2} \Sigma_3 \mid \mathsf{return}^{m}\ v}{\Sigma_1 \mid \mathsf{natrec}\ ({}^{\checkmark} x_1)\ (\lambda xy.{}^{\checkmark} e_2) \bullet x_3 \downdownarrows^{1+n_1+n_2} \Sigma_3 \mid \mathsf{return}^{1+m}\ v}$$

Figure 3. Operational semantics for the thunked language. If the right-hand side of an antecedent is $\mathsf{return}^{m}\ v$, then the only possible type-correct values are $\mathsf{return}^{m}\ v$ (for suitable m and v); this can be seen as a form of pattern matching.

This statement does not refer to the thunked semantics, but is not compositional, since Σ_2 does not carry any credit.

10. Extensions

This section discusses some possible extensions of the simple languages used to prove the library correct. These extensions are meant to indicate that the correctness proof also applies to a full-scale language such as Agda.

Partial applications Partial applications of library primitives were disallowed in Section 5. Other partial applications are allowed, though. As an example, two-argument lambdas can be introduced (with the obvious typing rules):

$$t ::= \ldots \mid \lambda xy.t \qquad v ::= \ldots \mid \lambda xy.t$$

$$e ::= \ldots \mid \lambda xy.{}^{\checkmark} e \qquad v ::= \ldots \mid \lambda xy.{}^{\checkmark} e$$

The operational semantics are extended as follows:

$$\Sigma \mid \lambda xy.t \Downarrow^{0} \Sigma \mid \lambda xy.t$$

$$\Sigma \mid (\lambda xy.t_1) \bullet x_2 \Downarrow^{0} \Sigma \mid \lambda y.t_1[x := x_2]$$

$$\Sigma \mid \lambda xy.{}^{\checkmark} e \downdownarrows^{0} \Sigma \mid \lambda xy.{}^{\checkmark} e$$

$$\Sigma \mid (\lambda xy.{}^{\checkmark} e_1) \bullet x_2 \downdownarrows^{0} \Sigma \mid \lambda y.{}^{\checkmark} e_1[x := x_2]$$

The proofs of equivalence and correctness go through easily with these rules.

Note that nothing is charged for the applications above; only when all arguments have been supplied (and hence evaluation of the right-hand side can commence) is something charged. If this cost measure is too coarse for a certain application, then two-argument lambdas should not be used. (Note that $\lambda x.{}^{\checkmark}\lambda y.{}^{\checkmark} e$ still works.)

Partial applications of constructors can be treated similarly; in the mechanised correctness proof the successor constructor s is a function of type $\mathsf{Nat} \to \mathsf{Nat}$.

Inductive types The examples describing the use of THUNK made use of various data types. Extending the languages with strictly positive inductive data types or families (Dybjer 1994) should be straightforward, following the examples of natural numbers and products.

Equality When THUNK is implemented using a dependently typed language such as Agda, one inductive family deserves further scrutiny: the equality (or identity) type. In practice it is likely that users of the THUNK library need to prove that various equalities hold. As an example, in the append example given in Section 3 the equality $1 + ((1 + 2 * m) + (1 + 0)) = 1 + 2 * (1 + m)$ must be established in order for the program to type check. If the host language type checker is smart enough certain such equalities may well be handled automatically using various decision procedures, but in the general case the user cannot expect all equalities to be solved automatically.

One way to supply equality proofs to the type checker is to use the equality type $(\equiv)$ together with the *subst* function, which expresses substitutivity of $(\equiv)$:

$$
\begin{array}{l}
\textbf{data}\ (\equiv)\ (x : a) : a \to \star\ \textbf{where} \\
\quad \mathsf{refl} : x \equiv x \\
\mathit{subst} : (P : a \to \star) \to x \equiv y \to P\ x \to P\ y \\
\mathit{subst}\ P\ \mathsf{refl}\ p = p
\end{array}
$$

Assuming a proof

$$
\begin{array}{l}
\mathit{lemma} : (m : \mathbb{N}) \to 1 + ((1 + 2 * m) + (1 + 0)) \equiv \\
\qquad\qquad\qquad\quad 1 + 2 * (1 + m)
\end{array}
$$

the definition of $(+\!\!+)$ can be corrected:

$$
\begin{array}{l}
(+\!\!+) \ :\ \{a : \star\} \to \{m, n : \mathbb{N}\} \\
\qquad \to \mathit{Seq}\ a\ m \to \mathit{Seq}\ a\ n \\
\qquad \to \mathit{Thunk}\ (1 + 2 * m)\ (\mathit{Seq}\ a\ (m + n)) \\
(+\!\!+) \qquad\qquad \mathsf{nil} \qquad\ \ ys = {}^{\checkmark}\ \mathit{return}\ ys \\
(+\!\!+)\ \{a\}\ \{\mathsf{suc}\ m\}\ \{n\}\ (x :: xs)\ ys = \\
\quad \mathit{subst}\ (\lambda x \to \mathit{Thunk}\ x\ (\mathit{Seq}\ a\ (\mathsf{suc}\ m + n)))\ (\mathit{lemma}\ m) \\
\quad ({}^{\checkmark}\ xs +\!\!+ ys \gg\!\!= \lambda xsys \to {}^{\checkmark} \\
\qquad \mathit{return}\ (x :: xsys))
\end{array}
$$

However, now *subst* and *lemma* interfere with the evaluation of $(+\!\!+)$, so the stated time complexity is no longer correct.

One way to address this problem would be to let *subst* cost one tick, and also pay for the equality proofs, just as if $(\equiv)$ was any other inductive family. However, this is not what we want to do. We just want to use the proofs to show that the program is type correct, we do not want to evaluate them.

A better solution is to erase all equality proofs and inline the identity function resulting from *subst* (and do the same for similar functions derived from the eliminator of $(\equiv)$). This is type safe as long as the underlying logical theory is consistent and only closed terms are evaluated, since then the only term of type $x \equiv y$ is refl (and only if $x = y$). Then *subst* (fully applied) can be used freely by the user of the library, without having to worry about overheads not tracked by the thunk monad.

Implementing proof erasure just for this library goes against the spirit of the project, though, since modifying a compiler is not lightweight. Fortunately proof erasure is an important, general problem in the compilation of dependently typed languages (see for instance Brady et al. 2004; Paulin-Mohring 1989), so it is not unreasonable to expect a good compiler to guarantee that the erasure outlined above will always take place.

Functions like *subst* have been used in the case studies accompanying this paper.

Fixpoints The simple languages introduced above are most likely terminating, since they are very similar to Gödel's System T. However, nothing stops us from adding a fixpoint combinator:

$$
t ::= \ldots \mid \mathsf{fix}\ (\lambda x.t) \qquad\qquad e ::= \ldots \mid \mathsf{fix}\ (\lambda x.{}^{\checkmark}\ e)
$$

$$
\frac{\Gamma, x : \tau\ \vdash t : \tau}{\Gamma\ \vdash \mathsf{fix}\ t : \tau \to \tau}
$$

$$
\frac{\Sigma_1, x \mapsto \mathsf{fix}\ (\lambda x.t) \mid t\ \Downarrow^{n}\ \Sigma_2 \mid v}{\Sigma_1 \mid \mathsf{fix}\ (\lambda x.t)\ \Downarrow^{1+n}\ \Sigma_2 \mid v}
$$

$$
\frac{\Sigma_1, x \mapsto \mathsf{fix}\ (\lambda x.{}^{\checkmark} e) \mid e\ \Downarrow\!\!\Downarrow^{n}\ \Sigma_2 \mid v}{\Sigma_1 \mid \mathsf{fix}\ (\lambda x.{}^{\checkmark} e)\ \Downarrow\!\!\Downarrow^{1+n}\ \Sigma_2 \mid v}
$$

The mechanised correctness proof also includes fixpoint operators, and they do not complicate the development at all.

There is one problem with unrestricted fixpoint operators, though: they make logical systems inconsistent. This invalidates the equality proof erasure optimisation discussed above, and hence including an unrestricted fix may not be a good idea, at least not in the context of Agda.

11. Paying for deeply embedded thunks

THUNK, as described above, has an important limitation: it is impossible to pay for thunks embedded deep in a data structure, without a large overhead. This section describes the problem and outlines a solution.

The problem Let us generalise the lazy sequences from Section 7 by letting the cost needed to force tails vary throughout the sequence:

$$
\begin{array}{l}
\mathit{CostSeq} : \mathbb{N} \to \star \\
\mathit{CostSeq}\ n = \mathit{Seq}\ \mathbb{N}\ n \\
\textbf{data}\ \mathit{Seq}_L\ (a : \star) : (n : \mathbb{N}) \to \mathit{CostSeq}\ n \to \star\ \textbf{where} \\
\quad \mathsf{nil}_L \quad :\ \mathit{Seq}_L\ a\ 0\ \mathsf{nil} \\
\quad (::_L)\ \ :\ a \to \mathit{Thunk}\ c\ (\mathit{Seq}_L\ a\ n\ cs) \\
\qquad\qquad \to \mathit{Seq}_L\ a\ (1 + n)\ (c :: cs)
\end{array}
$$

Here $\mathit{Seq}_L\ a\ n\ cs$ stands for a lazy sequence of length n, containing elements of type a, where the costs needed to force the tails of the sequence is given by the elements of cs.

Now, assume that $xs : \mathit{Seq}_L\ a\ n\ cs$, where

$$
cs = 0 :: 0 :: \ldots :: 0 :: 2 :: 4 :: \ldots :: \mathsf{nil}.
$$

Assume further that the analysis of an algorithm requires that the first debit in xs is paid off, resulting in $xs' : \mathit{Seq}_L\ a\ n\ cs'$ where

$$
cs' = 0 :: 0 :: \ldots :: 0 :: 1 :: 4 :: \ldots :: \mathsf{nil}.
$$

In order to accomplish this the type of a tail embedded deep down in xs needs to be changed. This requires a recursive function, which does not take constant time to execute, and this is likely to ruin the analysis of the algorithm.

The solution A way around this problem is to generalise the type of *pay*:

$$
\begin{array}{l}
\mathit{pay}_g\ \ :\ (C : \mathit{Ctxt}) \to (m : \mathbb{N}) \\
\qquad \to C\ [\mathit{Thunk}\ n\ a] \to \mathit{Thunk}\ m\ (C\ [\mathit{Thunk}\ (n - m)\ a])
\end{array}
$$

Here C is a *context* enabling payments deep down in the data structure. These contexts have to be quite restrictive, to ensure correctness of the analysis. For instance, they should have at most one hole, to ensure that only one thunk is paid off. Similarly one should not be allowed to pay off the codomain of a function, or the element type of a list; the following types are clearly erroneous:

$$
\begin{array}{l}
\mathit{payFun}\ \ :\ (m : \mathbb{N}) \to (a \to \mathit{Thunk}\ n\ b) \\
\qquad\quad \to \mathit{Thunk}\ m\ (a \to \mathit{Thunk}\ (n - m)\ b) \\
\mathit{payList}\ \ :\ (m : \mathbb{N}) \to \mathit{List}\ (\mathit{Thunk}\ n\ a) \\
\qquad\quad \to \mathit{Thunk}\ m\ (\mathit{List}\ (\mathit{Thunk}\ (n - m)\ a))
\end{array}
$$

For instance, if *payList* were allowed one could take a list with n elements, all of type $\mathit{Thunk}\ 1\ \mathbb{N}$, and obtain a $\mathit{List}\ (\mathit{Thunk}\ 0\ \mathbb{N})$ by just paying for one computation step, instead of n.

To avoid such problems the following type of contexts is defined:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Ctxt} : \star_1\ \textbf{where} \\
\quad \bullet \qquad\quad : \mathit{Ctxt} \\
\quad \mathit{const}\bullet\ \ : \star\ \to \mathit{Ctxt} \\
\quad \mathit{Thunk}\bullet : \mathbb{N} \to \mathit{Ctxt} \to \mathit{Ctxt}
\end{array}
$$

$$
\begin{aligned}
&(\bullet\times) &&: \mathit{Ctxt} \to \star &&\to \mathit{Ctxt}\\
&(\times\bullet) &&: \star &&\to \mathit{Ctxt} \to \mathit{Ctxt}
\end{aligned}
$$

(The type $\star_1$ is a type of large types.) These contexts can be turned into types by instantiating the holes:

$$
\begin{aligned}
&\cdot\,[\,\cdot\,] : \mathit{Ctxt} \to \star \to \star\\
&\bullet\,[\,b\,] &&= b\\
&(\mathit{const}\bullet\ a)\,[\,b\,] &&= a\\
&(\mathit{Thunk}\bullet\ n\ C)\,[\,b\,] &&= \mathit{Thunk}\ n\ (C\,[\,b\,])\\
&(C\ \bullet\!\times\ a)\,[\,b\,] &&= C\,[\,b\,] \times a\\
&(a\ \times\!\bullet\ C)\,[\,b\,] &&= a \times C\,[\,b\,]
\end{aligned}
$$

This definition of contexts may at first seem rather restrictive, since no recursive type constructors are included. However, when using dependent types one can define new types by explicit recursion. A variant of Seq_L can for instance be defined as follows:

$$
\begin{aligned}
&\mathit{Seq}_L : \star \to \mathit{CostSeq}\ n \to \star\\
&\mathit{Seq}_L\ a\ \mathsf{nil} &&= \mathit{Unit}\\
&\mathit{Seq}_L\ a\ (c :: cs) &&= a \times \mathit{Thunk}\ c\ (\mathit{Seq}_L\ a\ cs)
\end{aligned}
$$

(Here *Unit* is the unit type.) By using this type and pay_g it is now possible to pay off any of the tails in the sequence with only constant overhead:

```
pay_L  :  {a : ⋆} → (cs_1 : CostSeq n_1) → (c' : ℕ)
       → Seq_L a (cs_1 ++ c :: cs_2)
       → Thunk (1 + c') (Seq_L a (cs_1 ++ (c − c') :: cs_2))
pay_L {a} cs_1 c' xs = ✓
  cast lemma_2 (pay_g (C a cs_1) c' (cast lemma_1 xs))
  where
  C : ⋆ → CostSeq n → Ctxt
  C a nil       = a ×• •
  C a (c :: cs) = a ×• Thunk• c (C a cs)

  lemma_1 : Seq_L a (cs_1 ++ c :: cs_2) ≡⋆
              (C a cs_1) [ Thunk c (Seq_L a cs_2) ]
  lemma_2 :
    Thunk c' ((C a cs_1) [ Thunk (c − c') (Seq_L a cs_2) ]) ≡⋆
    Thunk c' (Seq_L a (cs_1 ++ (c − c') :: cs_2))
```

The equality ($\overset{\star}{\equiv}$) used above is a variant of the one introduced in Section 10, but this one relates types:

$$(\overset{\star}{\equiv}) : \star \to \star \to \star \qquad \mathit{cast} : a \overset{\star}{\equiv} b \to a \to b$$

The generalised *pay* can be used to analyse banker's queues (Okasaki 1998), which exhibit the problem with deep payments mentioned above. Space considerations preclude further discussion of this analysis, though. Interested readers are referred to the source code of the analysis for details.

Finally note that pay_g has been proved correct, using the same (mechanised) approach as above; for details see Danielsson (2007).

12. Limitations

This section discusses some limitations of THUNK.

Dependent bind One thing which may have bothered users familiar with dependently typed languages is that the second (function) argument to bind is non-dependent. This could be fixed by replacing (≫=) with a more general function:

$$
\begin{aligned}
\mathit{bind}\ :\ &(f : a \to \mathbb{N}) \to (b : a \to \star)\\
&\to (x : \mathit{Thunk}\ m\ a) \to ((y : a) \to \mathit{Thunk}\ (f\ y)\ (b\ y))\\
&\to \mathit{Thunk}\ (m + f\ (\mathit{force}\ x))\ (b\ (\mathit{force}\ x))
\end{aligned}
$$

However, this would not in itself be very useful: *force* is abstract (see Section 4), so *force x* would not evaluate in the type of *bind*.

One way to work around this is by providing the library user with a number of axioms specifying how the library primitives evaluate. This can be useful anyway, since it makes it possible to prove ordinary functional properties of annotated code inside Agda. This solution is rather complicated, though.

A better approach is perhaps to avoid the dependently typed bind, and this can often be achieved by using indexed types. Consider the following two variants of the append function:

$$
\begin{aligned}
(+\!\!+)\ :\ &(xs : \mathit{List}\ a) \to \mathit{List}\ a\\
&\to \mathit{Thunk}\ (1 + 2 * \mathit{length}\ xs)\ (\mathit{List}\ a)\\
(+\!\!+)\ :\ &\mathit{Seq}\ a\ m \to \mathit{Seq}\ a\ n\\
&\to \mathit{Thunk}\ (1 + 2 * m)\ (\mathit{Seq}\ a\ (m + n))
\end{aligned}
$$

The result type of the first function depends on the value of the first list. This is not the case for the second function, where the result type instead depends on the index *m*. Putting enough information in type indices is often a good way to avoid the need for a dependent bind.

Aliasing Another limitation is that the library cannot track thunk aliases, except in the limited way captured by *pay* and the $\mapsto^n$ bindings of the thunked operational semantics. If $x, y : \mathit{Thunk}\ n\ a$ are aliases for each other, and *x* is forced, then the library has no way of knowing that *y* is also forced; the type of *y* does not change just because *x* is forced. Okasaki (1998) uses aliases in this way to eliminate amortisation, through a technique called *scheduling*.

Interface stability If thunked types are exposed to external users of a data structure library then another problem shows up: the types of functions analysed using THUNK are not robust against small changes in the implementation. A function such as *maximum*, introduced in Section 7, has a rather precise type:

$$\mathit{maximum} : \mathit{Seq}\ a\ (1 + n) \to \mathit{Thunk}\ (13 + 14 * n + 4 * n\hat{\ }2)\ a$$

A type based on big O notation would be more stable:

$$\mathit{maximum} : \mathit{Seq}\ a\ (1 + n) \to \mathit{Thunk}\ \mathcal{O}(n\hat{\ }2)\ a$$

However, expressing big O notation in a sound way without a dedicated type system seems to be hard. It is probably a good idea to use *force* to avoid exporting thunked types.

13. Related work

Time complexity for lazy evaluation Several approaches to analysing lazy (call-by-need) time complexity have been developed (Wadler 1988; Bjerner 1989; Bjerner and Holmström 1989; Sands 1995; Okasaki 1998; Moran and Sands 1999). Many of them are general, but have been described as complicated to use in practice (Okasaki 1998).

It seems to be rather uncommon to actually use these techniques to analyse non-trivial programs. The main technique in use is probably Okasaki's banker's method (1998), which is mainly used for analysing purely functional data structures, and which this work is based on. As described in Section 12 the banker's method is more general than the one described here, but it can also be seen as more complicated, since it distinguishes between several kinds of cost (shared and unshared) which are collapsed in this work.

Ross Paterson (personal communication) has independently sketched an analysis similar to the one developed here, but without dependent types or the annotated monad.

The ticks used in this work are related to those used by Moran and Sands (1999), but their theory does not require ticks to be inserted to ensure that computation steps are counted; ticks are instead used to represent and prove improvements and cost equivalences, with the help of a tick algebra.

Benzinger (2004) describes a system for automated complexity analysis of higher-order functional programs. The system is parametrised on an annotated operational semantics, so it may be able to handle a call-by-need evaluation strategy. No such experiments seem to have been performed, though, so it is unclear how practical it would be.

Tracking resource usage using types Several frameworks for tracking resource usage using types have been developed. Usually these frameworks do not address lazy evaluation (for instance Crary and Weirich 2000; Constable and Crary 2002; Rebón Portillo et al. 2003; Brady and Hammond 2006; Hofmann and Jost 2006). There can still be similarities; for instance, the system of Hofmann and Jost uses amortised analysis to bound heap space usage, with potential tracked by types.

Hughes, Pareto, and Sabry (1996) have constructed a type system which keeps track of bounds on the sizes of values in a lazy language with data and codata. This information is used to guarantee termination or productivity of well-typed terms; more precise time bounds are not handled.

The use of an annotated monad to combine time complexities of subexpressions appears to be novel. However, there is a close connection to Capretta's partiality monad (2005), which is a coinductive type constructor $\cdot^{\nu}$ defined roughly as follows:

$$\begin{array}{l} \textbf{codata}\ \cdot^{\nu}\ (a : \star) : \star\ \textbf{where} \\ \quad \mathsf{return} : a \quad \to a^{\nu} \\ \quad \mathsf{step} \quad : a^{\nu} \to a^{\nu} \end{array}$$

The following definition of bind turns it into a monad:

$$\begin{array}{l} (\gg\!=) : a^{\nu} \to (a \to b^{\nu}) \to b^{\nu} \\ \mathsf{return}\ x \gg\!= f = f\ x \\ \mathsf{step}\ x \quad \gg\!= f = \mathsf{step}\ (x \gg\!= f) \end{array}$$

Compare the definitions above to the following shallow embedding of the thunk monad:

$$\begin{array}{l} \textbf{data}\ \mathit{Thunk}\ (a : \star) : \mathbb{N} \to \star\ \textbf{where} \\ \quad \mathit{return} : a \qquad\quad \to \mathit{Thunk}\ a\ 0 \\ \quad \checkmark \qquad : \mathit{Thunk}\ a\ n \to \mathit{Thunk}\ a\ (1+n) \\ (\gg\!=) : \mathit{Thunk}\ a\ m \to (a \to \mathit{Thunk}\ b\ n) \to \mathit{Thunk}\ b\ (m+n) \\ \mathit{return}\ x \ \gg\!= f = f\ x \\ (\checkmark x) \qquad \gg\!= f = \checkmark x \gg\!= f \end{array}$$

The only difference is that the thunk monad is inductive, and annotated with the number of ticks. This indicates that it may be interesting to explore the consequences of making the thunk monad coinductive, annotated with the coinductive natural numbers ($\mathbb{N}$ extended with ω).

14. Conclusions

A simple, lightweight library for semiformal verification of the time complexity of purely functional data structures has been described. The usefulness of the library has been demonstrated and its limitations discussed. Furthermore the semantics of the library has been precisely defined, the time complexity guarantees have been verified with respect to the semantics, and the correctness proof has been checked using a proof assistant.

Acknowledgments

I would like to thank Ulf Norell, who has taught me a lot about dependently typed programming, discussed many aspects of this work, and fixed many of the bugs in Agda and AgdaLight that I have reported. I am also grateful to Ross Paterson, who showed me his analysis of implicit queues which turned out to yield a nice and compact example. Other people who deserve thanks are Jeremy Gibbons, Martin Hofmann, Patrik Jansson, Geraint Jones, David Sands, Anton Setzer, and some anonymous reviewers.

References

Ralph Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318:79–103, 2004.

Bror Bjerner and Sören Holmström. A compositional approach to time analysis of first order lazy functional programs. In *FPCA '89*, pages 157–165, 1989.

Bror Bjerner. *Time Complexity of Programs in Type Theory*. PhD thesis, Department of Computer Science, University of Göteborg, 1989.

Edwin Brady and Kevin Hammond. A dependently typed framework for static analysis of program execution costs. In *IFL 2005*, volume 4015 of *LNCS*, pages 74–90, 2006.

Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *TYPES 2003: Types for Proofs and Programs*, volume 3085 of *LNCS*, pages 115–129, 2004.

Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–28, 2005.

Robert L. Constable and Karl Crary. *Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman*, chapter Computational Complexity and Induction for Partial Computable Functions in Type Theory. A K Peters Ltd, 2002.

Karl Crary and Stephanie Weirich. Resource bound certification. In *POPL '00*, pages 184–198, 2000.

Nils Anders Danielsson. A formalisation of the correctness result from "Lightweight semiformal time complexity analysis for purely functional data structures". Technical Report 2007:16, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.

Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *ESOP 2006*, volume 3924 of *LNCS*, pages 22–37, 2006.

John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96*, pages 410–423, 1996.

Haim Kaplan and Robert E. Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM*, 46(5):577–603, 1999.

Haim Kaplan, Chris Okasaki, and Robert E. Tarjan. Simple confluently persistent catenable lists. *SIAM Journal on Computing*, 30(3):965–977, 2000.

John Launchbury. A natural semantics for lazy evaluation. In *POPL '93*, pages 144–154, 1993.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain-Specific Languages (DSL '99)*, pages 109–122, 1999.

Andrew Moran and David Sands. Improvement in a lazy context: an operational theory for call-by-need. In *POPL '99*, pages 43–56, 1999.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

Christine Paulin-Mohring. Extracting F_{ω}'s programs from proofs in the calculus of constructions. In *POPL '89*, pages 89–104, 1989.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

Álvaro J. Rebón Portillo, Kevin Hammond, Hans-Wolfgang Loidl, and Pedro Vasconcelos. Cost analysis using automatic size and time inference. In *IFL 2002*, volume 2670 of *LNCS*, pages 232–247, 2003.

David Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.

The Agda Team. The Agda Wiki. Available at `http://www.cs.chalmers.se/~ulfn/Agda/`, 2007.

Philip Wadler. Strictness analysis aids time analysis. In *POPL '88*, pages 119–132, 1988.