Total Definitional Interpreters for Time and Space Complexity

Nils Anders Danielsson
University of Gothenburg and Chalmers University of Technology

Abstract

Definitional interpreters are sometimes used to specify the semantics of programming languages and to reason about the semantics of programs. This text presents one way in which total definitional interpreters defined using the delay monad can be used to specify and reason about time and space complexity. The approach works for non-terminating programs, and the text is supported by machine-checked proofs.

1 Introduction

Definitional interpreters [Reynolds 1972] are sometimes used to give the semantics of programming languages, also for languages with non-terminating programs. Several techniques are available to handle non-termination, including step-counting/fuel [Boyer and Moore 1997; Leroy and Grall 2009; Siek 2013; Owens et al. 2016; Amin and Rompf 2017; Bach Poulsen et al. 2018], the delay monad and other coinductive types [Capretta 2005; Nakata and Uustalu 2009; Paulin-Mohring 2009; Benton et al. 2009; Danielsson 2012], and guarded recursive types [Paviotti et al. 2015].

One application of total definitional interpreters has been to define operational semantics and prove compiler correctness [Young 1989; Danielsson 2012]; in particular, this approach is taken in the CakeML project [Owens et al. 2016]. In this work I show that it is possible to reason about time and space complexity (at least stack space usage) in this setting: I present a simple λ-calculus along with instrumented interpreters that make it possible to reason about the stack space usage and number of reduction steps for the corresponding compiled programs running on a virtual machine, without referring directly to the compiler or virtual machine. For stack usage I pay particular attention to programs that might fail to terminate: it can be important to be able to distinguish between a non-terminating program that runs in bounded stack space and one that does not.

The most closely related work is perhaps that of Young [1989], who discusses two languages and a compiler from one to the other. The semantics of the languages are given as definitional interpreters that use step-counting: each interpreter is defined by recursion on a number, and if this number becomes zero, then the interpreter stops. Programs in the target language can run out of stack space, and this is handled by also tracking stack space in the interpreter for the source language: if stack space runs out, then the interpreter returns an error value. Young proves compiler correctness under the assumption that the program terminates and stack space does not run out.

In this work I return a trace of stack sizes instead of crashing when stack space runs out (similarly to how Owens et al. [2016] return traces of input/output actions), and I prove compiler correctness for all programs, also those that require unbounded stack space. Furthermore, instead of using step-counting like Young and Owens et al., I follow Danielsson [2012] and use the coinductively defined delay monad (see Section 5) to model the effect of non-termination. See Section 11 for further discussion of related work.

The main technical contributions of this text are perhaps definitions of some (hopefully reusable) relations, and associated combinators, that are used to prove various properties:

- Two ways to relate upper bounds of potentially infinite lists of natural numbers (Sections 4 and 9).
- A variant of weak bisimilarity for the delay monad that can be used to quantify the difference in the number of steps in two computations (Section 10).

In order to make it convenient to use these relations I have defined them using sized types (Section 2), and I describe a number of combinators—in many cases size-preserving—that can be used to construct inhabitants of these relations.

All the main results and examples in this text have been formalised using Agda [Agda Team 2018], with the K rule turned off, and the code is available to inspect (at the time of writing from http://www.cse.chalmers.se/~nad/). The formalisation makes heavy use of sized types. In order to provide readers with some experience of what it is like to use sized types in practice, and because Agda is perhaps the only fairly mature proof assistant with support for sized types, I will use code that is close to actual Agda code in the text below. (There are some differences between the code presented below and the actual formalisation, but they are minor.)

2 Sized Types

Agda provides sized types as a mechanism to make it easier to write corecursive definitions. This section contains a quick introduction to (one kind of) sized types as implemented in Agda. Sized types can be used for both induction and coinduction, but in this text they are only used for coinduction.

I will introduce the concept by showing how to define the type of "conatural numbers"—the greatest fixpoint $\nu X.1 + X$, representing natural numbers extended with infinity—along with some related definitions.

The conatural numbers can be defined in the following rather verbose way using sized types:

```
data Conat (i : Size) : Set where
  zero : Conat i
  suc  : Conat′ i → Conat i

record Conat′ (i : Size) : Set where
  coinductive
  field force : {j : Size< i} → Conat j
```

The constructor zero stands for the conatural number zero, and suc n is the successor of n. An intuitive way to think about this definition is that the primed type *Conat′* is a suspended computation, and when you "force" such a computation you get a value, which can contain further suspended computations. (*Set* is a type of small types. The full code includes the **mutual** keyword as well, but I have omitted it in order to save some space.)

Sizes can be thought of as some kind of ordinals, and the type *Conat i* can be thought of as a type of possibly not fully defined conatural numbers of size at least i. The notation j : *Size*< i means (roughly) that j is strictly smaller than i. The type *Conat′ i* can be seen as standing for conatural numbers of size at least j, for any j : *Size*< i.

There is a special size ∞, and *Conat* ∞ can be seen as a type of fully defined conatural numbers of any size. The size ∞ can be thought of as a closure ordinal for which there is some kind of isomorphism between *Conat* ∞ and *Conat′* ∞: we have that i : *Size*< ∞ holds for every size i (including ∞).

Agda also supports a notion of subtyping: values of type *Conat i* ("conatural numbers of size at least i") can be used where values of type *Conat j* ("conatural numbers of size at least j") are expected, for any j : *Size*< i.

A basic method for defining values in coinductive types is to use corecursion. Agda supports corecursion with *copatterns* [Abel et al. 2013]. Here is one way to define "infinity":

```
infinity : ∀ {i} → Conat i        infinity′ : ∀ {i} → Conat′ i
infinity = suc infinity′          infinity′ .force = infinity
```

The code above states that *infinity* is the successor of *infinity′*. Agda treats *infinity′* as a value that is only unfolded if the force projection is applied to it, in which case the result is *infinity* (which can be unfolded further). A more compact notation with an anonymous copattern is also available:

```
infinity = suc λ { .force → infinity }
```

Notation like ∀ {i} → … means that the argument i is an implicit argument. Implicit arguments do not need to be given explicitly if Agda manages to infer them. However, it is possible to give a fully explicit definition of *infinity*:

```
infinity {i} = suc λ { .force {j} → infinity {j} }
```

There is no way to match on a size, sizes are only used to give information to the termination checker. The termination checker accepts the last definition of *infinity* above because, for every cycle in the call graph, there is a strict decrease in the size (if we ignore ∞), and the strictly smaller size (j : *Size*< i) is associated in a certain way to a copattern corresponding to a field (force) of the *coinductive* record type *Conat′*. If ∞ is ignored, then one can see *infinity* as being defined by some kind of transfinite recursion.

Note that the current, experimental Agda implementation of sized types is buggy. The fact that ∞ : *Size*< ∞ has led to problems, and a slightly different design has been discussed. I would be surprised if any bugs in Agda invalidated the main ideas presented below, but readers are of course free to be more sceptical.

The approach to sized types presented here is based on *deflationary iteration* [Abel 2012]. Abel and Pientka [2016] present a normalisation proof for this approach to sized types, but for a language without dependent types (and without ∞ : *Size*< ∞). Sacchini [2015] studies a language with dependent types, and sketches a normalisation proof, but his language is designed somewhat differently from Agda.

As an example of a coinductively defined *relation*, consider the following definition of "less than or equals":

```
data [_]_≤_ (i : Size) : (m n : Conat ∞) → Set where
  zero : ∀ {n} → [ i ] zero ≤ n
  suc  : ∀ {m n} →
         [ i ] m .force ≤′ n .force → [ i ] suc m ≤ suc n

record [_]_≤′_ (i : Size) (m n : Conat ∞) : Set where
  coinductive
  field force : {j : Size< i} → [ j ] m ≤ n
```

Note that Agda allows constructors to be overloaded. This definition states that the number zero is less than or equal to any conatural number, and that suc preserves the ordering relation (in a coinductive sense). Note also that the "primed" variant of the relation is defined in the same way as the primed variant of the conatural numbers. From now on most definitions of primed record types are omitted; all the omitted definitions have the same form.

For technical reasons Agda's equality type is not always appropriate to use with coinductive types. For the conatural numbers it often makes more sense to use the following notion of *bisimilarity*:

```
data [_]_∼N_ (i : Size) : (m n : Conat ∞) → Set where
  zero : [ i ] zero ∼N zero
  suc  : [ i ] m .force ∼N′ n .force → [ i ] suc m ∼N suc n
```

(Here and below I sometimes omit argument declarations, in this case for m and n, from type signatures.)

3 A Very Simple Language

Let us begin by studying a very basic programming language, where programs consist of potentially infinite lists of instructions, and there are only two instructions, alloc and dealloc:

$$\begin{array}{ll} \textbf{data}\ \mathit{Stmt} : \mathit{Set}\ \textbf{where} & \mathit{Program} : \mathit{Size} \to \mathit{Set} \\ \quad \mathsf{alloc}\ \mathsf{dealloc} : \mathit{Stmt} & \mathit{Program}\ i = \mathit{Colist}\ \mathit{Stmt}\ i \end{array}$$

Potentially infinite lists, or colists, are defined coinductively in the following way:

$$\begin{array}{l} \textbf{data}\ \mathit{Colist}\ (A : \mathit{Set})\ (i : \mathit{Size}) : \mathit{Set}\ \textbf{where} \\ \quad [\,] \quad : \mathit{Colist}\ A\ i \\ \quad \_{::}\_ : A \to \mathit{Colist}'\ A\ i \to \mathit{Colist}\ A\ i \end{array}$$

The semantics of a program is taken to be a trace of heap sizes. This makes the definitional interpreter very simple. The function *modify* computes the new heap size, given an instruction and the previous heap size:

$$\begin{array}{l} \mathit{modify} : \mathit{Stmt} \to \mathbb{N} \to \mathbb{N} \\ \mathit{modify}\ \mathsf{alloc} \quad\ = \mathsf{suc} \\ \mathit{modify}\ \mathsf{dealloc} = \mathit{pred} \end{array}$$

(Here suc is the overloaded successor constructor for the unary, inductive representation of natural numbers that I use, and *pred* is the predecessor function, with *pred* 0 defined to be 0.) The interpreter uses this function repeatedly, returning all the encountered heap sizes, including the initial one:

$$\begin{array}{l} [\![\_]\!] : \forall\ \{i\} \to \mathit{Program}\ i \to \mathbb{N} \to \mathit{Colist}\ \mathbb{N}\ i \\ [\![\ p\ ]\!]\ h = h :: [\![\ p\ ]\!]'\ h \end{array}$$

$$\begin{array}{l} [\![\_]\!]' : \forall\ \{i\} \to \mathit{Program}\ i \to \mathbb{N} \to \mathit{Colist}'\ \mathbb{N}\ i \\ [\![\ [\,] \quad]\!]'\ h\ .\mathsf{force} = [\,] \\ [\![\ s :: p\ ]\!]'\ h\ .\mathsf{force} = [\![\ p\ .\mathsf{force}\ ]\!]\ (\mathit{modify}\ s\ h) \end{array}$$

This may not seem like much of an interpreter. However, in Section 9 below a similar technique is used to instrument a definitional interpreter for a λ-calculus with information about stack sizes. The current section and Section 4 introduce some of the main ideas and definitions that will be used later.

An upper bound predicate for colists can be defined in the following way (where $\ulcorner\_\urcorner$ maps natural numbers to the corresponding conatural numbers):

$$\begin{array}{l} [\_]\_\sqsubseteq\_ : \mathit{Size} \to \mathit{Colist}\ \mathbb{N}\ \infty \to \mathit{Conat}\ \infty \to \mathit{Set} \\ [\ i\ ]\ \mathit{ms} \sqsubseteq n = \Box\ i\ (\lambda\ m \to [\ \infty\ ]\ \ulcorner\ m\ \urcorner \le n)\ \mathit{ms} \end{array}$$

This says that the conatural number n is an upper bound of *ms* if every natural number in *ms* is bounded by n; $\Box\ \infty\ P\ \mathit{xs}$ means that the predicate P holds for every element in *xs*:

$$\begin{array}{l} \textbf{data}\ \Box\ (i : \mathit{Size})\ (P : A \to \mathit{Set}) : \mathit{Colist}\ A\ \infty \to \mathit{Set}\ \textbf{where} \\ \quad [\,] \quad : \Box\ i\ P\ [\,] \\ \quad \_{::}\_ : P\ x \to \Box'\ i\ P\ (\mathit{xs}\ .\mathsf{force}) \to \Box\ i\ P\ (x :: \mathit{xs}) \end{array}$$

Below a primed variant of $[\_]\_\sqsubseteq\_$, defined using $\Box'$ instead of $\Box$, will also be used.

A least upper bound is an upper bound that is bounded by every upper bound:

$$\begin{array}{l} \mathit{LUB} : \mathit{Colist}\ \mathbb{N}\ \infty \to \mathit{Conat}\ \infty \to \mathit{Set} \\ \mathit{LUB}\ \mathit{ns}\ n = [\ \infty\ ]\ \mathit{ns} \sqsubseteq n\ \times \\ \qquad\qquad\qquad (\forall\ n' \to [\ \infty\ ]\ \mathit{ns} \sqsubseteq n' \to [\ \infty\ ]\ n \le n') \end{array}$$

Least upper bounds are unique up to bisimilarity:

$$\mathit{LUB}\ \mathit{ns}\ n_1 \to \mathit{LUB}\ \mathit{ns}\ n_2 \to [\ i\ ]\ n_1 \sim_{\mathrm{N}} n_2$$

This follows from antisymmetry for conatural numbers.

Least upper bounds exist for every colist if and only if WLPO holds. WLPO is the classically valid "weak limited principle of omniscience", a constructive taboo that should neither be provable nor disprovable in Agda (in the absence of bugs). See the accompanying code for the formal statement and proof of this property, which was obtained in collaboration with Andreas Abel and Ulf Norell.

The maximum heap usage of a program that starts with an empty heap can be defined as follows:

$$\begin{array}{l} \mathit{Heap\text{-}usage} : \mathit{Program}\ \infty \to \mathit{Conat}\ \infty \to \mathit{Set} \\ \mathit{Heap\text{-}usage}\ p\ n = \mathit{LUB}\ ([\![\ p\ ]\!]\ 0)\ n \end{array}$$

Because least upper bounds are unique the maximum heap usage is also unique.

Let us now consider some examples. Here are three looping programs:

$$\begin{array}{l} \mathit{bounded}\ \mathit{bounded}_2\ \mathit{unbounded} : \mathit{Program}\ i \\ \mathit{bounded} \quad\ = \mathsf{alloc} ::' \mathsf{dealloc} :: \lambda\ \{\ .\mathsf{force} \to \mathit{bounded}\ \} \\ \mathit{bounded}_2 \quad = \mathsf{alloc} ::' \mathsf{alloc} ::' \mathsf{dealloc} ::' \mathsf{dealloc} :: \\ \qquad\qquad\quad \lambda\ \{\ .\mathsf{force} \to \mathit{bounded}_2\ \} \\ \mathit{unbounded} = \mathsf{alloc} :: \lambda\ \{\ .\mathsf{force} \to \mathit{unbounded}\ \} \end{array}$$

The first two definitions make use of $\_{::}'\_$, a variant of $\_{::}\_$ with an unprimed second argument, in order to avoid some clutter:

$$\begin{array}{l} \_{::}'\_ : A \to \mathit{Colist}\ A\ i \to \mathit{Colist}\ A\ i \\ x ::' \mathit{xs} = x :: \lambda\ \{\ .\mathsf{force} \to \mathit{xs}\ \} \end{array}$$

The three programs above are all non-terminating, in the sense that their traces are infinitely long. However, they have different space complexities. The programs *bounded* and $\mathit{bounded}_2$ run in bounded space, while *unbounded* requires unbounded space:

$$\begin{array}{ll} \mathit{Heap\text{-}usage}\ \mathit{bounded} & \ulcorner\ 1\ \urcorner \\ \mathit{Heap\text{-}usage}\ \mathit{bounded}_2 & \ulcorner\ 2\ \urcorner \\ \mathit{Heap\text{-}usage}\ \mathit{unbounded}\ \mathit{infinity} & \end{array}$$

The first two statements are easy to prove. For the last one I made use of the following lemma:

$$(\forall\ n \to \neg\ [\ \infty\ ]\ [\![\ p\ ]\!]\ 0 \sqsubseteq \ulcorner\ n\ \urcorner) \to \mathit{Heap\text{-}usage}\ p\ \mathit{infinity}$$

(The negation of A is the type of functions from A to the empty type.) If no natural number is an upper bound of the heap usage of p, then the maximum heap usage of p is infinity. This lemma can be proved by using the following lemma:

$$\begin{array}{l} (\forall\ n \to \neg\ [\ \infty\ ]\ \mathit{ms} \sqsubseteq \ulcorner\ n\ \urcorner) \to [\ \infty\ ]\ \mathit{ms} \sqsubseteq m \to \\ [\ \infty\ ]\ m \sim_{\mathrm{N}} \mathit{infinity} \end{array}$$

If no natural number is an upper bound of *ms*, but the conatural number m is, then m is bisimilar to infinity.

4 An Optimiser

In this section an optimiser is defined for the simple allocation language, and it is proved that this optimiser works as it should. The point of this exercise is to introduce a relation that will be used to prove compiler correctness in Section 9.

The optimiser takes subsequences consisting of alloc, alloc and dealloc, and replaces them with alloc:

$$\begin{array}{l}
\mathit{opt} : \forall\ \{i\} \to \mathit{Program}\ \infty \to \mathit{Program}\ i \\
\mathit{opt}\ []\qquad\qquad\quad = [] \\
\mathit{opt}\ (\mathrm{dealloc}\ \ :: p) = \mathrm{dealloc} :: \lambda\ \{\ .\mathrm{force} \to \mathit{opt}\ (p\ .\mathrm{force})\ \} \\
\mathit{opt}\ \{i\}\ (\mathrm{alloc} :: p) = \mathit{opt}_1\ (p\ .\mathrm{force}) \\
\quad \textbf{module}\ \mathit{Opt}\ \textbf{where} \\
\quad \mathit{default} : \mathit{Program}\ i \\
\quad \mathit{default} = \mathrm{alloc} :: \lambda\ \{\ .\mathrm{force} \to \mathit{opt}\ (p\ .\mathrm{force})\ \} \\
\\
\quad \mathit{opt}_2 : \mathit{Program}\ \infty \to \mathit{Program}\ i \\
\quad \mathit{opt}_2\ (\mathrm{dealloc} :: p'') = \mathrm{alloc} :: \\
\qquad\qquad\qquad\qquad\quad \lambda\ \{\ .\mathrm{force} \to \mathit{opt}\ (p''\ .\mathrm{force})\ \} \\
\quad \mathit{opt}_2\ _ \qquad\qquad\quad = \mathit{default} \\
\\
\quad \mathit{opt}_1 : \mathit{Program}\ \infty \to \mathit{Program}\ i \\
\quad \mathit{opt}_1\ (\mathrm{alloc} :: p') = \mathit{opt}_2\ (p'\ .\mathrm{force}) \\
\quad \mathit{opt}_1\ _ \qquad\qquad = \mathit{default}
\end{array}$$

The named **where** clause makes it possible to refer to the local definitions in the proof below. For instance, the name $\mathit{Opt}.\mathit{opt}_2$ refers to the local definition opt_2. Note that $\mathit{Opt}.\mathit{opt}_2$ takes two extra arguments, one implicit and one explicit, corresponding to the bound variables i and p from the left-hand side $\mathit{opt}\ \{i\}\ (\mathrm{alloc} :: p)$.

One might think that it would be better to replace subsequences consisting of alloc and dealloc with nothing, but if such an optimiser could be implemented, then it would not produce any output at all (not even []) for *bounded*.

The optimiser improves the space complexity of at least one program, because $\mathit{opt}\ \mathit{bounded}_2$ has the same semantics (up to bisimilarity) as *bounded*:

$$[\ i\ ]\ [\![\ \mathit{opt}\ \mathit{bounded}_2\ ]\!]\ n \sim_{\mathrm{L}} [\![\ \mathit{bounded}\ ]\!]\ n$$

Here $[\ \infty\ ]\ ms \sim_{\mathrm{L}} ns$ means that the colists ms and ns are bisimilar. Bisimilarity for colists is defined analogously to bisimilarity for conatural numbers.

It remains to prove that the maximum heap usage of an optimised program is at most as high as that of the original program (assuming that these maximums exist):

$$\mathit{Heap\text{-}usage}\ (\mathit{opt}\ p)\ m \to \mathit{Heap\text{-}usage}\ p\ n \to [\ i\ ]\ m \leq n$$

Proving this directly using corecursion might be tricky. Instead I will make use of the following relation, which states that every upper bound of the second colist is also an upper bound of the first:

$$\begin{array}{l}
[_]_\lesssim_ : \mathit{Size} \to \mathit{Colist}\ \mathbb{N}\ \infty \to \mathit{Colist}\ \mathbb{N}\ \infty \to \mathit{Set} \\
[\ i\ ]\ ms \lesssim ns = \forall\ \{n\} \to [\ \infty\ ]\ ns \sqsubseteq n \to [\ i\ ]\ ms \sqsubseteq n
\end{array}$$

Read $[\ \infty\ ]\ ms \lesssim ns$ as "ms is bounded by ns". If ms has the least upper bound m, and ns has the least upper bound n, then ms is bounded by ns if and only if m is bounded by n:

$$\mathit{LUB}\ ms\ m \to \mathit{LUB}\ ns\ n \to [\ \infty\ ]\ ms \lesssim ns \Leftrightarrow [\ \infty\ ]\ m \leq n$$

Thus the optimiser correctness property given above follows from the following statement:

$$[\ i\ ]\ [\![\ \mathit{opt}\ p\ ]\!]\ h \lesssim [\![\ p\ ]\!]\ h$$

I have defined four combinators which can be used to prove that one colist is bounded by another (I give their types but no names here; the implementations are straightforward and omitted):

$$\begin{array}{l}
[\ i\ ]\ []\lesssim ns \\
[\ i\ ]\ ms \lesssim ns\ .\mathrm{force} \to [\ i\ ]\ ms \lesssim n :: ns \\
\mathit{Bounded}\ m\ ns \to [\ i\ ]\ ms\ .\mathrm{force} \lesssim'\ ns \to [\ i\ ]\ m :: ms \lesssim ns \\
[\ i\ ]\ ms\ .\mathrm{force} \lesssim'\ ns\ .\mathrm{force} \to [\ i\ ]\ m :: ms \lesssim m :: ns
\end{array}$$

Here $\mathit{Bounded}\ m\ ns$ means that m is either less than or equal to some element in ns, or equal to zero. The last combinator is implemented using the previous two.

The last two combinators take a primed variant of the relation as an argument:

$$\begin{array}{l}
[_]_\lesssim'_ : \mathit{Size} \to \mathit{Colist}\ \mathbb{N}\ \infty \to \mathit{Colist}\ \mathbb{N}\ \infty \to \mathit{Set} \\
[\ i\ ]\ ms \lesssim'\ ns = \forall\ \{n\} \to [\ \infty\ ]\ ns \sqsubseteq n \to [\ i\ ]\ ms \sqsubseteq'\ n
\end{array}$$

However, the second combinator takes the unprimed variant of the relation as an argument instead. This means that, while the second combinator can be used in corecursive proofs, it does not introduce a size change, whereas the others do.

If the second combinator had taken the primed variant of the relation as an argument instead, then one could have proved that any colist was bounded by any infinite colist by using this combinator repeatedly in a corecursive proof. This implies that the type of such a combinator is contradictory:

$$\begin{array}{l}
\neg\ (\forall\ \{i\ ms\ ns\ n\} \to \\
\qquad [\ i\ ]\ ms \lesssim'\ ns\ .\mathrm{force} \to [\ i\ ]\ ms \lesssim n :: ns)
\end{array}$$

The "bounded by" relation is a preorder. However, the transitivity proof is only size-preserving in the first argument:

$$[\ i\ ]\ ms \lesssim ns \to [\ \infty\ ]\ ns \lesssim os \to [\ i\ ]\ ms \lesssim os$$

One can derive a contradiction from the assumption that the transitivity proof is size-preserving in the other argument (see the accompanying code for details):

$$\begin{array}{l}
\neg\ (\forall\ \{i\ ms\ ns\ os\} \to \\
\qquad [\ \infty\ ]\ ms \lesssim ns \to [\ i\ ]\ ns \lesssim os \to [\ i\ ]\ ms \lesssim os)
\end{array}$$

This means, roughly speaking, that one can only use corecursive calls in the first argument of transitivity. As a workaround one can sometimes use the following variant of transitivity, which takes a bisimilarity proof as the first argument:

$$[\ i\ ]\ ms \sim_{\mathrm{L}} ns \to [\ i\ ]\ ns \lesssim os \to [\ i\ ]\ ms \lesssim os$$

For more discussion of transitivity proofs that are not size-preserving, see Danielsson [2018].

Let me now show how I finished the correctness proof ($[\ i\ ]\ [\![\ \mathit{opt}\ p\ ]\!]\ h \lesssim [\![\ p\ ]\!]\ h$). The proof is corecursive, based on the call structure of the optimiser, and uses the combinators discussed above. The most interesting case is perhaps the one for the first clause of $\mathit{Opt}.\mathit{opt}_2$. I focus on that one. The goal is to prove the following statement (in the presence of some assumptions that are not needed):

$$\begin{array}{l}[\ i\ ]\ [\![\ \mathit{Opt}.\mathit{opt}_2\ p\ (\mathsf{dealloc} :: p'')\ ]\!]\ h \lesssim \\ \qquad h ::' 1 + h ::' [\![\ \mathsf{dealloc} :: p''\ ]\!]\ (2 + h)\end{array}$$

The proof proceeds in the following way:

$$\begin{array}{ll}
[\![\ \mathit{Opt}.\mathit{opt}_2\ p\ (\mathsf{dealloc} :: p'')\ ]\!]\ h & \sim \\
h ::' [\![\ \mathit{opt}\ (p''\ .\mathsf{force})\ ]\!]\ (1 + h) & \lesssim \\
h ::' [\![\ p''\ .\mathsf{force}\ ]\!]\ (1 + h) & \lesssim \\
h ::' 1 + h ::' 2 + h ::' [\![\ p''\ .\mathsf{force}\ ]\!]\ (1 + h) & \sim \\
h ::' 1 + h ::' [\![\ \mathsf{dealloc} :: p''\ ]\!]\ (2 + h) &
\end{array}$$

The first and last steps basically amount to unfolding of definitions. The second step uses the last proof combinator mentioned above (which—importantly—has a primed argument), followed by a corecursive call to the top-level correctness proof. The third step uses three applications of the proof combinators mentioned above (first the last one, and then two applications of the second one), followed by a use of reflexivity.

The formal proof in the accompanying source code is written using equational reasoning combinators (based on an idea due to Norell [2007]) that allow it to be formatted like the chain of reasoning steps above, but with an explanation inserted for every step.

5 The Delay Monad

This section contains a brief presentation of the delay monad [Capretta 2005], which is used to define an interpreter in Section 6. The delay monad represents computations that are potentially non-terminating:

$$\begin{array}{l}
\mathbf{data}\ \mathit{Delay}\ (A : \mathit{Set})\ (i : \mathit{Size}) : \mathit{Set}\ \mathbf{where} \\
\quad \mathsf{now} : A \to \mathit{Delay}\ A\ i \\
\quad \mathsf{later} : \mathit{Delay}'\ A\ i \to \mathit{Delay}\ A\ i
\end{array}$$

The application now x represents a situation in which a computation terminates immediately with the value x, and later x stands for a program that may or may not terminate later. The computation *never* represents non-termination:

$$\begin{array}{l}
\mathit{never} : \mathit{Delay}\ A\ i \\
\mathit{never} = \mathsf{later}\ \lambda\ \{\ .\mathsf{force} \to \mathit{never}\ \}
\end{array}$$

The delay monad is a monad (the definition is omitted). The monad laws can be proved up to strong bisimilarity ($[\_]\_\sim_{\mathrm{D}}\_$), which can be defined analogously to bisimilarity for conatural numbers.

In many cases strong bisimilarity is too strong, because it only relates terminating computations if they terminate in the same number of steps (later constructors). An alternative is to use weak bisimilarity, which relates any computations that terminate with the same value, and can be defined in the following way [Danielsson and Altenkirch 2010; Danielsson 2018]:

$$\begin{array}{l}
\mathbf{data}\ [_]_\approx_{\mathrm{D}}_\ (i : \mathit{Size}) : (x\ y : \mathit{Delay}\ A\ \infty) \to \mathit{Set}\ \mathbf{where} \\
\quad \mathsf{now} \quad : [\ i\ ]\ \mathsf{now}\ x \approx_{\mathrm{D}} \mathsf{now}\ x \\
\quad \mathsf{later} \quad : [\ i\ ]\ x\ .\mathsf{force} \approx_{\mathrm{D}}'\ y\ .\mathsf{force} \to \\
\qquad\qquad\ [\ i\ ]\ \mathsf{later}\ x\ \approx_{\mathrm{D}}\ \mathsf{later}\ y \\
\quad \mathsf{later}^{\mathsf{l}} : [\ i\ ]\ x\ .\mathsf{force} \approx_{\mathrm{D}} y \to [\ i\ ]\ \mathsf{later}\ x \approx_{\mathrm{D}} y \\
\quad \mathsf{later}^{\mathsf{r}} : [\ i\ ]\ x \approx_{\mathrm{D}} y\ .\mathsf{force} \to [\ i\ ]\ x \approx_{\mathrm{D}} \mathsf{later}\ y
\end{array}$$

Note that the definition above uses a mixture of induction and coinduction: the later constructor is "coinductive", because it takes a primed argument, whereas $\mathsf{later}^{\mathsf{l}}$ and $\mathsf{later}^{\mathsf{r}}$ are "inductive", because they take unprimed arguments. The definition should be read as an inductive definition nested inside a coinductive one.

6 A Simple Lambda Calculus

The language treated in Sections 3–4 was very minimal. Let us now switch attention to a somewhat more interesting language, based on the one used in Leroy and Grall's study of coinductive big-step semantics [2009]. Danielsson [2012] later used the same language to study the use of the delay monad to define total definitional interpreters; he used a well-scoped representation, and I take the same approach here. I have replaced the infinite set of uninterpreted constants used in those previous works with booleans, and added calls to unary, named definitions.

The syntax is defined in the following way:

$$\begin{array}{l}
\mathbf{data}\ \mathit{Tm}\ (n : \mathbb{N}) : \mathit{Set}\ \mathbf{where} \\
\quad \mathsf{var}\ \ : \mathit{Fin}\ n \to \mathit{Tm}\ n \\
\quad \mathsf{lam} : \mathit{Tm}\ (\mathsf{suc}\ n) \to \mathit{Tm}\ n \\
\quad _\cdot_\ \ : \mathit{Tm}\ n \to \mathit{Tm}\ n \to \mathit{Tm}\ n \\
\quad \mathsf{call}\ : \mathit{Name} \to \mathit{Tm}\ n \to \mathit{Tm}\ n \\
\quad \mathsf{con}\ : \mathit{Bool} \to \mathit{Tm}\ n \\
\quad \mathsf{if}\quad : \mathit{Tm}\ n \to \mathit{Tm}\ n \to \mathit{Tm}\ n \to \mathit{Tm}\ n
\end{array}$$

A well-scoped representation is used: the term data type is parametrised by an upper bound on the number of free variables, and uses de Bruijn indices. The term var x is a variable; *Fin n* stands for natural numbers strictly less than n. The term lam t stands for a lambda abstraction, and $t_1 \cdot t_2$ is an application. The term call $f\ t$ is a call to the named, unary function f: I assume that a type of names, *Name*, is given. Finally we have con b, a literal boolean, and if $t_1\ t_2\ t_3$, which stands for "if t_1 then t_2 else t_3".

The interpreter uses closures (following Leroy and Grall). Environments and values are defined in the following way:

$Env : \mathbb{N} \to Set$
$Env\ n = Vec\ Value\ n$

$\textbf{data}\ Value : Set\ \textbf{where}$
$\quad \mathsf{lam} : Tm\ (\mathsf{suc}\ n) \to Env\ n \to Value$
$\quad \mathsf{con} : Bool \to Value$

A value of type $Env\ n$ is a list of values of length n. The value $\mathsf{lam}\ t\ \rho$ is a closure, a combination of a term with at most $1 + n$ free variables, and an environment containing values for n of those variables. Boolean literals are turned into values by the constructor con.

I define a total, definitional interpreter for the syntax above by using the delay monad, following Danielsson [2012] (who used the term "partiality monad" for what is now commonly called the delay monad). However, unlike Danielsson I use sized types, which makes the definition a little easier.

The interpreter can both crash and fail to terminate, so the delay monad is combined with the maybe monad transformer ($Maybe\ A$ has two constructors, $\mathsf{nothing} : Maybe\ A$ and $\mathsf{just} : A \to Maybe\ A$):

$Delay_C : Set \to Size \to Set$
$Delay_C\ A\ i = Delay\ (Maybe\ A)\ i$

Here C stands for "crash". In this section return and **do** notation refer to this monad. An immediate crash—as opposed to one that happens later—is defined in the following way (note that *crash* is not weakly bisimilar to *never*):

$crash : Delay_C\ A\ i$
$crash = \mathsf{now}\ \mathsf{nothing}$

The interpreter is parametrised by a function mapping names to terms with at most one free variable, $def : Name \to Tm\ 1$. The interpreter is defined in a "big-step" way using three mutually (co)recursive functions. The first one tries to apply one value to another. If the first value is a boolean literal this leads to a crash. In the case of a closure the interpreter proceeds with the evaluation of the body of the closure in the closure's environment, extended with the second value:

$\_\bullet\_ : Value \to Value \to Delay_C\ Value\ i$
$\mathsf{lam}\ t_1\ \rho \bullet v_2 = \mathsf{later}\ \lambda\ \{\ .\mathsf{force} \to [\![\ t_1\ ]\!]\ (v_2 :: \rho)\ \}$
$\mathsf{con}\ \_ \quad \bullet \_ \ = crash$

The main function interprets a term in an environment of matching size:

$[\![\_]\!] : Tm\ n \to Env\ n \to Delay_C\ Value\ i$
$[\![\ \mathsf{var}\ x\ ]\!] \quad \rho = \mathsf{return}\ (index\ x\ \rho)$
$[\![\ \mathsf{lam}\ t\ ]\!] \quad \rho = \mathsf{return}\ (\mathsf{lam}\ t\ \rho)$
$[\![\ t_1 \cdot t_2\ ]\!] \quad \rho = \textbf{do}\ v_1 \leftarrow [\![\ t_1\ ]\!]\ \rho$
$\qquad\qquad\qquad\quad v_2 \leftarrow [\![\ t_2\ ]\!]\ \rho$
$\qquad\qquad\qquad\quad v_1 \bullet v_2$
$[\![\ \mathsf{call}\ f\ t\ ]\!] \quad \rho = \textbf{do}\ v \leftarrow [\![\ t\ ]\!]\ \rho$
$\qquad\qquad\qquad\quad \mathsf{lam}\ (def\ f)\ []\bullet v$
$[\![\ \mathsf{con}\ b\ ]\!] \quad \rho = \mathsf{return}\ (\mathsf{con}\ b)$
$[\![\ \mathsf{if}\ t_1\ t_2\ t_3\ ]\!]\ \rho = \textbf{do}\ v_1 \leftarrow [\![\ t_1\ ]\!]\ \rho$
$\qquad\qquad\qquad\quad [\![if]\!]\ v_1\ t_2\ t_3\ \rho$

The value of a variable is the corresponding entry in the environment, and the value of a lambda abstraction is a closure. Applications are interpreted by first interpreting the function, then (if the first computation terminates with a value) the argument, and finally (if also the second computation terminates with a value) using $\_\bullet\_$ to apply the first value to the second. Note that this is a call-by-value semantics. Calls to named functions are evaluated similarly to applications. Boolean literals are returned directly. Conditionals are interpreted by first interpreting the scrutinee, and then letting the auxiliary function $[\![if]\!]$ determine how to proceed:

$[\![if]\!] : Value \to Tm\ n \to Tm\ n \to Env\ n \to Delay_C\ Value\ i$
$[\![if]\!]\ (\mathsf{lam}\ \_\ \_) \quad \_\ \_\ \_ = crash$
$[\![if]\!]\ (\mathsf{con}\ \mathsf{true}) \quad t_2\ t_3\ \rho = [\![\ t_2\ ]\!]\ \rho$
$[\![if]\!]\ (\mathsf{con}\ \mathsf{false})\ t_2\ t_3\ \rho = [\![\ t_3\ ]\!]\ \rho$

The definitions above are total. They are defined using an outer corecursion and an inner recursion on the structure of terms. A decrease of the size argument is introduced in $\_\bullet\_$, and otherwise the size argument is kept unchanged.

7 A Virtual Machine with Tail Calls

This section presents a virtual machine, or VM, and Section 8 a compiler from the terms of the previous section to this VM.

The virtual machine is defined corecursively in a "small-step" way, following Leroy and Grall [2009] and Danielsson [2012]. Danielsson used the monad of the previous section (defined without sized types) to define the semantics. Later I want to analyse the stack usage of compiled programs, so I use a variant of this monad that allows the virtual machine to return a trace of all states that it encounters.

The type family underlying the monad is defined in the following way:

$\textbf{data}\ Delay_{CT}\ (A\ B : Set)\ (i : Size) : Set\ \textbf{where}$
$\quad \mathsf{now} \quad : B \to Delay_{CT}\ A\ B\ i$
$\quad \mathsf{crash} : Delay_{CT}\ A\ B\ i$
$\quad \mathsf{later} \ : A \to Delay_{CT}'\ A\ B\ i \to Delay_{CT}\ A\ B\ i$
$\quad \mathsf{tell} \quad : A \to Delay_{CT}\ \ A\ B\ i \to Delay_{CT}\ A\ B\ i$

Here T stands for "trace". The type $Delay_{CT}\ A\ B\ \infty$ represents a class of computations that, in addition to perhaps terminating with a value of type B, also produce traces (colists) containing values of type A:

$trace : Delay_{CT}\ A\ B\ i \to Colist\ A\ i$
$trace\ (\mathsf{now}\ x) \quad = []$
$trace\ \mathsf{crash} \qquad = []$
$trace\ (\mathsf{later}\ x\ m) = x :: \lambda\ \{\ .\mathsf{force} \to trace\ (m\ .\mathsf{force})\ \}$
$trace\ (\mathsf{tell}\ x\ m) \quad = x :: \lambda\ \{\ .\mathsf{force} \to trace\ m\ \}$

Note that the later constructor takes a first argument of type A, which is used when constructing colists. There is also a tell constructor which is an inductive variant of later.

Traces can be removed from computations:

$$delay_{\mathrm{C}} : Delay_{\mathrm{CT}}\ A\ B\ i \to Delay_{\mathrm{C}}\ B\ i$$

The $delay_{\mathrm{C}}$ function removes the first argument from later constructors, and tell constructors are removed entirely.

$Delay_{\mathrm{CT}}$ can be turned into a monad, and strong bisimilarity can be defined, in much the same way as for the delay monad. See the accompanying code for details, including proofs showing that the monad laws hold up to bisimilarity.

The virtual machine uses the following instruction set:

```
data Instr (n : ℕ) : Set where
  var     : Fin n → Instr n
  clo     : Code (suc n) → Instr n
  app ret : Instr n
  cal tcl : Name → Instr n
  con     : Bool → Instr n
  bra     : Code n → Code n → Instr n

Code : ℕ → Set
Code n = List (Instr n)
```

This definition is based on Leroy and Grall's, but well-scoped (following Danielsson): *Instr n* represents instructions with at most n free variables, and *Code n* stands for lists of arbitrary length of instructions of type *Instr n*. There are also some changes to support booleans and calls to named functions: the con instruction takes a boolean instead of a natural number, the new instructions cal and tcl are used for regular calls and tail calls, respectively, and the new instruction bra is used for branching.

Environments and values are defined as for the interpreter, but using *Code* instead of *Tm*. The interpreter is stack-based, with two kinds of stack elements, values and return frames:

```
data Stack-element : Set where
  val : VM-Value → Stack-element
  ret : Code n → VM-Env n → Stack-element
```

A *Stack* is a list of stack elements. The VM's state consists of a piece of code, a stack, and an environment:

```
data State : Set where
  ⟨_,_,_⟩ : Code n → Stack → VM-Env n → State
```

As mentioned above the VM's semantics is given in a small-step way. The result of running the VM one step is either a new state, a final value, or an indication that the VM has crashed:

```
data Result : Set where
  continue : State → Result
  done     : VM-Value → Result
  crash    : Result
```

The VM's semantics is parametrised by a function mapping names to pieces of code, $def : Name \to Code\ 1$. The *step* function given in Figure 1 is similar to the single-step relation of a typical small-step semantics. Note that execution of the cal instruction involves pushing a return frame onto the stack, but that this is not done when the tcl instruction is executed. The idea is that there should already be a return frame on the stack that can be reused.

The semantics is defined by iterating *step* corecursively:

```
exec⁺ : State → Delay_CT State VM-Value i
exec⁺ s = later s λ { .force → exec⁺′ (step s) }

exec⁺′ : Result → Delay_CT State VM-Value i
exec⁺′ (continue s) = exec⁺ s
exec⁺′ (done v)     = now v
exec⁺′ crash        = crash
```

Note that the resulting trace contains every encountered state. A semantics without states can also be obtained:

```
exec : State → Delay_C VM-Value i
exec = delay_C ∘ exec⁺
```

Furthermore it is possible to construct a trace of all the encountered stack sizes:

```
stack-sizes : State → Colist ℕ i
stack-sizes =
  map (λ { ⟨ _ , s , _ ⟩ → length s }) ∘ trace ∘ exec⁺
```

8 A Correct Compiler

Let us now see how one can construct a provably correct compiler from the λ-calculus in Section 6 to the language of the virtual machine in Section 7. The compiler and its correctness proof are based on the work by Leroy and Grall [2009] and Danielsson [2012].

Unlike these previous works I provide support for tail calls. The compiler takes an argument with information about whether the compiled term should be treated as if it is in a tail context. I have based the definition of tail contexts on the one used by Kelsey et al. [1998].

The main compilation function is defined in the following way (with *In-tail-context* equal to *Bool*). Note the use of a code continuation:

```
comp : In-tail-context → Tm n → Code n → Code n
comp _     (var x)      c = var x :: c
comp _     (lam t)      c = clo (comp-body t) :: c
comp _     (t₁ · t₂)    c = comp false t₁
                              (comp false t₂ (app :: c))
comp true  (call f t)   c = comp false t (tcl f :: c)
comp false (call f t)   c = comp false t (cal f :: c)
comp _     (con b)      c = con b :: c
comp tc    (if t₁ t₂ t₃) c =
  comp false t₁ (bra (comp tc t₂ []) (comp tc t₃ []) :: c)
```

```
step : State → Result
step ⟨ var x     :: c ,                         s  , ρ ⟩ = continue ⟨ c       , val (index x ρ) :: s ,      ρ  ⟩
step ⟨ clo c′    :: c ,                         s  , ρ ⟩ = continue ⟨ c       , val (lam c′ ρ)  :: s ,      ρ  ⟩
step ⟨ app       :: c , val v :: val (lam c′ ρ′) :: s  , ρ ⟩ = continue ⟨ c′      , ret c ρ         :: s , v :: ρ′ ⟩
step ⟨ ret       :: c , val v :: ret c′ ρ′       :: s  , ρ ⟩ = continue ⟨ c′      , val v           :: s ,      ρ′ ⟩
step ⟨ cal f     :: c , val v                    :: s  , ρ ⟩ = continue ⟨ def f   , ret c ρ         :: s , v :: [] ⟩
step ⟨ tcl f     :: c , val v                    :: s  , ρ ⟩ = continue ⟨ def f   ,                    s , v :: [] ⟩
step ⟨ con b     :: c ,                         s  , ρ ⟩ = continue ⟨ c       , val (con b)     :: s ,      ρ  ⟩
step ⟨ bra c₁ c₂ :: c , val (con true)           :: s  , ρ ⟩ = continue ⟨ c₁ ++ c ,                    s ,      ρ  ⟩
step ⟨ bra c₁ c₂ :: c , val (con false)          :: s  , ρ ⟩ = continue ⟨ c₂ ++ c ,                    s ,      ρ  ⟩
step ⟨ []             , val v                    :: [] , [] ⟩ = done v
step _                                                      = crash
```

Figure 1. The step function used to define the semantics of the virtual machine.

$$\begin{array}{l} \mathit{comp\text{-}body} : \mathit{Tm}\ (\mathsf{suc}\ n) \to \mathit{Code}\ (\mathsf{suc}\ n) \\ \mathit{comp\text{-}body}\ t = \mathit{comp}\ \mathsf{true}\ t\ (\mathsf{ret} :: []) \end{array}$$

The body of an abstraction is compiled in a tail context, but the two arguments to application, the single argument to a call, and the scrutinee of an if-then-else expression are not. The two branches of if-then-else are compiled in a tail context if the if-then-else expression as a whole is.

Just like the interpreter the compiler is parametrised by a function mapping names to terms ($\mathit{def} : \mathit{Name} \to \mathit{Tm}\ 1$). The following function compiles such definitions:

$$\begin{array}{l} \mathit{comp\text{-}name} : \mathit{Name} \to \mathit{Code}\ 1 \\ \mathit{comp\text{-}name}\ f = \mathit{comp\text{-}body}\ (\mathit{def}\ \ f) \end{array}$$

All compiler correctness statements below are given in settings in which the interpreter and compiler are instantiated with the same function *def*, and the function *comp-name* is used to provide an implementation of *def* for the virtual machine. For the purpose of stating compiler correctness I also define functions that compile environments and values (see the accompanying code for details):

$$\begin{array}{l} \mathit{comp\text{-}env} : \mathit{Env}\ n \to \mathit{VM\text{-}Env}\ n \\ \mathit{comp\text{-}val}\ \ : \mathit{Value} \to \mathit{VM\text{-}Value} \end{array}$$

The following function is the top-level entry point to the compiler:

$$\begin{array}{l} \mathit{comp}_0 : \mathit{Tm}\ 0 \to \mathit{Code}\ 0 \\ \mathit{comp}_0\ t = \mathit{comp}\ \mathsf{false}\ t\ [] \end{array}$$

The top-level expression is not compiled in a tail context, because when the VM starts the stack is empty, so there is no return frame that can be reused on the stack.

Now compiler correctness can be stated (following Danielsson [2012]):

$$\begin{array}{l} [\ \infty\ ]\ \mathit{exec}\ \langle\ \mathit{comp}_0\ t\ , []\ , []\ \rangle \approx_{\mathrm{D}} \\ \qquad [\![\ t\ ]\!]\ []\ \gg\!=\ \lambda\ v \to \mathsf{return}\ (\mathit{comp\text{-}val}\ v) \end{array}$$

This says that the result of running the virtual machine with an initial state containing the code obtained by compiling the program t, an empty stack, and an empty environment, is weakly bisimilar to the semantics of t according to the interpreter, provided that if the interpreter produces a value, then this value is compiled before it is returned. Note that this correctness statement applies to programs that terminate with a value, programs that crash, and programs that fail to terminate.

The correctness proof is rather similar to the one given by Danielsson; use of sized types makes the proof a little easier. Here is the type of the key lemma:

$$\begin{array}{l} \mathit{Stack\text{-}OK}\ i\ k\ \mathit{tc}\ s \to \\ \mathit{Cont\text{-}OK}\ i\ \langle\ c\ , s\ , \mathit{comp\text{-}env}\ \rho\ \rangle\ k \to \\ [\ i\ ]\ \mathit{exec}\ \langle\ \mathit{comp}\ \mathit{tc}\ t\ c\ , s\ , \mathit{comp\text{-}env}\ \rho\ \rangle \approx_{\mathrm{D}} [\![\ t\ ]\!]\ \rho \gg\!= k \end{array}$$

Note the two assumptions. The second assumption relates the code continuation c, the stack s and the environment ρ to the monadic continuation k:

$$\begin{array}{l} \mathit{Cont\text{-}OK} : \mathit{Size} \to \mathit{State} \to \\ \qquad\qquad (\mathit{Value} \to \mathit{Delay}_{\mathrm{C}}\ \mathit{VM\text{-}Value}\ \infty) \to \mathit{Set} \\ \mathit{Cont\text{-}OK}\ i\ \langle\ c\ , s\ , \rho\ \rangle\ k = \\ \quad \forall\ v \to [\ i\ ]\ \mathit{exec}\ \langle\ c\ , \mathsf{val}\ (\mathit{comp\text{-}val}\ v) :: s\ , \rho\ \rangle \approx_{\mathrm{D}} k\ v \end{array}$$

The first assumption is targeted at tail-calls:

$$\begin{array}{l} \textbf{data}\ \mathit{Stack\text{-}OK} \\ \qquad (i : \mathit{Size})\ (k : \mathit{Value} \to \mathit{Delay}_{\mathrm{C}}\ \mathit{VM\text{-}Value}\ \infty) : \\ \qquad \mathit{In\text{-}tail\text{-}context} \to \mathit{Stack} \to \mathit{Set}\ \textbf{where} \\ \ \ \mathsf{unrestricted} : \mathit{Stack\text{-}OK}\ i\ k\ \mathsf{false}\ s \\ \ \ \mathsf{restricted}\ \ \ \ : \mathit{Cont\text{-}OK}\ i\ \langle\ c\ , s\ , \rho\ \rangle\ k \to \\ \qquad\qquad\qquad \mathit{Stack\text{-}OK}\ i\ k\ \mathsf{true}\ (\mathsf{ret}\ c\ \rho :: s) \end{array}$$

For programs compiled in a tail context the stack has to start with a return frame, and it has to satisfy a certain assumption that also involves the monadic continuation. The *Stack-OK* predicate is perhaps the main addition to Danielsson's correctness proof.

9 An Instrumented Interpreter

Let me now show an instrumented interpreter that makes it possible to reason about a program's stack usage without reasoning directly about compiled programs and the virtual machine. I want to emphasise that it was not immediately obvious to me how to construct this instrumented semantics, it was developed together with its correctness proof.

The interpreter produces a trace of size change functions that is then turned into a trace of sizes. Here is the instrumented application function (S stands for space):

$$\begin{array}{l}
[_,_]_\bullet_{\mathrm{S}}_ : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \to \mathit{Value} \to \mathit{Value} \to \\
\qquad \mathit{Delay}_{\mathrm{CT}}\ (\mathbb{N} \to \mathbb{N})\ \mathit{Value}\ i \\
[\ f_1\ ,\ f_2\ ]\ \mathsf{lam}\ t_1\ \rho \bullet_{\mathrm{S}} v_2 = \mathsf{later}\ f_1\ \lambda\ \{\ .\mathsf{force} \to \mathbf{do} \\
\qquad\qquad v \leftarrow [\![\ t_1\ ]\!]_{\mathrm{S}}\ (v_2 :: \rho)\ \mathsf{true} \\
\qquad\qquad \mathsf{tell}\ f_2\ (\mathsf{return}\ v)\ \} \\
[\ _\ ,\ _\ ]\ \mathsf{con}\ _ \quad \bullet_{\mathrm{S}}\ _ \ = \mathsf{crash}
\end{array}$$

The function is used in different ways, so it is parametrised by two size change functions. One is used before the body of the closure (if any) is evaluated, and one is used after the body has been evaluated successfully (if ever).

The main function is defined in the following way. Note that tail context information is passed around:

$$\begin{array}{l}
[\![_]\!]_{\mathrm{S}} : \mathit{Tm}\ n \to \mathit{Env}\ n \to \mathit{In\text{-}tail\text{-}context} \to \\
\qquad \mathit{Delay}_{\mathrm{CT}}\ (\mathbb{N} \to \mathbb{N})\ \mathit{Value}\ i \\
[\![\ \mathsf{var}\ x\ ]\!]_{\mathrm{S}} \quad \rho\ _ \ = \mathsf{tell}\ \mathsf{suc}\ (\mathsf{return}\ (\mathit{index}\ x\ \rho)) \\
[\![\ \mathsf{lam}\ t\ ]\!]_{\mathrm{S}} \quad \rho\ _ \ = \mathsf{tell}\ \mathsf{suc}\ (\mathsf{return}\ (\mathsf{lam}\ t\ \rho)) \\
[\![\ t_1 \cdot t_2\ ]\!]_{\mathrm{S}} \quad \rho\ _ \ = \mathbf{do}\ v_1 \leftarrow [\![\ t_1\ ]\!]_{\mathrm{S}}\ \rho\ \mathsf{false} \\
\qquad\qquad v_2 \leftarrow [\![\ t_2\ ]\!]_{\mathrm{S}}\ \rho\ \mathsf{false} \\
\qquad\qquad [\ \mathit{pred}\ ,\ \mathit{pred}\ ]\ v_1 \bullet_{\mathrm{S}} v_2 \\
[\![\ \mathsf{call}\ f\ t\ ]\!]_{\mathrm{S}} \quad \rho\ \mathit{tc} = \mathbf{do}\ v \leftarrow [\![\ t\ ]\!]_{\mathrm{S}}\ \rho\ \mathsf{false} \\
\qquad\qquad [\ \delta\ \mathit{tc}\ ,\ \delta\ (\mathit{not}\ \mathit{tc})\ ] \\
\qquad\qquad\ \ \mathsf{lam}\ (\mathit{def}\ f)\ []\ \bullet_{\mathrm{S}}\ v \\
[\![\ \mathsf{con}\ b\ ]\!]_{\mathrm{S}} \quad \rho\ _ \ = \mathsf{tell}\ \mathsf{suc}\ (\mathsf{return}\ (\mathsf{con}\ b)) \\
[\![\ \mathsf{if}\ t_1\ t_2\ t_3\ ]\!]_{\mathrm{S}}\ \rho\ \mathit{tc} = \mathbf{do}\ v_1 \leftarrow [\![\ t_1\ ]\!]_{\mathrm{S}}\ \rho\ \mathsf{false} \\
\qquad\qquad [\![\mathit{if}]\!]_{\mathrm{S}}\ v_1\ t_2\ t_3\ \rho\ \mathit{tc}
\end{array}$$

The stack size is increased for variables, abstractions and literal booleans (which correspond to pushing something onto the stack). When an application is evaluated the application function is used with *pred* and *pred*: the stack size is reduced by one for the app instruction, and by one for the ret instruction. If a tail call is evaluated, then the stack size is decreased before the call is made, and when a non-tail call is evaluated, then the stack size is decreased after the call has completed successfully (if ever):

$$\begin{array}{l}
\delta : \mathit{In\text{-}tail\text{-}context} \to \mathbb{N} \to \mathbb{N} \\
\delta\ \mathit{tc} = \mathit{if}\ \ \mathit{tc}\ \mathit{then}\ \mathit{pred}\ \mathit{else}\ \mathit{id}
\end{array}$$

The stack size is also decreased when the scrutinee of an if-then-else expression has been evaluated successfully to a boolean literal:

$$\begin{array}{l}
[\![\mathit{if}]\!]_{\mathrm{S}} : \mathit{Value} \to \mathit{Tm}\ n \to \mathit{Tm}\ n \to \mathit{Env}\ n \to \\
\qquad \mathit{In\text{-}tail\text{-}context} \to \mathit{Delay}_{\mathrm{CT}}\ (\mathbb{N} \to \mathbb{N})\ \mathit{Value}\ i \\
[\![\mathit{if}]\!]_{\mathrm{S}}\ (\mathsf{lam}\ _\ _) \quad _\ _\ _\ _ = \mathsf{crash} \\
[\![\mathit{if}]\!]_{\mathrm{S}}\ (\mathsf{con}\ \mathsf{true})\ \ t_2\ t_3\ \rho\ \mathit{tc} = \mathsf{tell}\ \mathit{pred}\ ([\![\ t_2\ ]\!]_{\mathrm{S}}\ \rho\ \mathit{tc}) \\
[\![\mathit{if}]\!]_{\mathrm{S}}\ (\mathsf{con}\ \mathsf{false})\ t_2\ t_3\ \rho\ \mathit{tc} = \mathsf{tell}\ \mathit{pred}\ ([\![\ t_3\ ]\!]_{\mathrm{S}}\ \rho\ \mathit{tc})
\end{array}$$

Given a computation yielding a trace of stack size functions, and an initial stack size, it is easy to construct a trace of stack sizes by starting with the inital value, and then applying the functions, one after another. This is captured by the following application of the standard *scanl* function (implemented for colists):

$$\begin{array}{l}
\mathit{numbers} : \mathit{Delay}_{\mathrm{CT}}\ (\mathbb{N} \to \mathbb{N})\ A\ i \to \mathbb{N} \to \mathit{Colist}\ \mathbb{N}\ i \\
\mathit{numbers}\ x\ n = \mathit{scanl}\ (\lambda\ m\ f \to f\ m)\ n\ (\mathit{trace}\ x)
\end{array}$$

The stack sizes encountered when evaluating a (closed) program can then be defined in the following way:

$$\begin{array}{l}
\mathit{stack\text{-}sizes}_{\mathrm{S}} : \mathit{Tm}\ 0 \to \mathit{Colist}\ \mathbb{N}\ i \\
\mathit{stack\text{-}sizes}_{\mathrm{S}}\ t = \mathit{numbers}\ ([\![\ t\ ]\!]_{\mathrm{S}}\ []\ \mathsf{false})\ 0
\end{array}$$

Note that false is used as the *In-tail-context* argument, matching the use of false in comp_0.

If the traces are removed, then the instrumented semantics produces computations that are strongly bisimilar to those produced by the semantics given in Section 6:

$$[\ i\ ]\ \mathit{delay}_{\mathrm{C}}\ ([\![\ t\ ]\!]_{\mathrm{S}}\ \rho\ \mathit{tc}) \sim_{\mathrm{D}} [\![\ t\ ]\!]\ \rho$$

Perhaps more interestingly, if the trace of stack sizes produced by the instrumented semantics has the least upper bound i, and the corresponding trace produced by the virtual machine has the least upper bound v, then i and v are bisimilar:

$$\begin{array}{l}
\mathit{LUB}\ (\mathit{stack\text{-}sizes}_{\mathrm{S}}\ t)\ i \to \\
\mathit{LUB}\ (\mathit{stack\text{-}sizes}\ \langle\ \mathit{comp}_0\ t\ ,\ []\ ,\ []\ \rangle)\ v \to [\ \infty\ ]\ i \sim_{\mathrm{N}} v
\end{array}$$

However, the traces are not necessarily bisimilar (see the accompanying code for a counterexample). I had a previous version of the instrumented interpreter for which the traces were bisimilar, but I decided to simplify the interpreter a little. In a more complicated setting it might be useful not to couple the instrumented semantics too closely to the lower-level semantics.

Instead of proving that the traces are bisimilar I have proved the following property:

$$[\ \infty\ ]\ \mathit{stack\text{-}sizes}\ \langle\ \mathit{comp}_0\ t\ ,\ []\ ,\ []\ \rangle \approx \mathit{stack\text{-}sizes}_{\mathrm{S}}\ t$$

The relation used here states that the two colists are bounded by each other, and is defined using the "bounded by" relation from Section 4:

$$\begin{array}{l}
[_]_\approx_ : \mathit{Size} \to \mathit{Colist}\ \mathbb{N}\ \infty \to \mathit{Colist}\ \mathbb{N}\ \infty \to \mathit{Set} \\
[\ i\ ]\ \mathit{ms} \approx \mathit{ns} = [\ i\ ]\ \mathit{ms} \lesssim \mathit{ns} \times [\ i\ ]\ \mathit{ns} \lesssim \mathit{ms}
\end{array}$$

Colists that are related by this relation have the same least upper bounds (if any):

$$[\ \infty\ ]\ \mathit{ms} \approx \mathit{ns} \to \mathit{LUB}\ \mathit{ms}\ n \Leftrightarrow \mathit{LUB}\ \mathit{ns}\ n$$

One approach to proving that two colists are upper bounds of each other would be to prove that one is an upper bound of the other, and vice versa, for instance by using some combinators from Section 4. As a possibly more direct alternative I provide some combinators that work directly with the "bounded by each other" relation, as well as the following primed variant:

$$[\_]\_{\approx}'\_ : \mathit{Size} \to \mathit{Colist}\ \mathbb{N}\ \infty \to \mathit{Colist}\ \mathbb{N}\ \infty \to \mathit{Set}$$
$$[\ i\ ]\ \mathit{ms} \approx' \mathit{ns} = [\ i\ ]\ \mathit{ms} \lesssim' \mathit{ns} \times [\ i\ ]\ \mathit{ns} \lesssim' \mathit{ms}$$

I give the combinators' types but no names here (the implementations are straightforward and omitted):

$$\mathit{Bounded}\ n\ \mathit{ms} \to [\ i\ ]\ \mathit{ms} \approx \mathit{ns}\ .\mathsf{force} \to [\ i\ ]\ \mathit{ms} \approx n :: \mathit{ns}$$
$$\mathit{Bounded}\ m\ \mathit{ns} \to [\ i\ ]\ \mathit{ms}\ .\mathsf{force} \approx \mathit{ns} \to [\ i\ ]\ m :: \mathit{ms} \approx \mathit{ns}$$
$$\mathit{Bounded}\ m\ (n :: \mathit{ns}) \to \mathit{Bounded}\ n\ (m :: \mathit{ms}) \to$$
$$\quad [\ i\ ]\ \mathit{ms}\ .\mathsf{force} \approx' \mathit{ns}\ .\mathsf{force} \to [\ i\ ]\ m :: \mathit{ms} \approx n :: \mathit{ns}$$
$$[\ i\ ]\ \mathit{ms}\ .\mathsf{force} \approx' \mathit{ns}\ .\mathsf{force} \to [\ i\ ]\ m :: \mathit{ms} \approx m :: \mathit{ns}$$

The relation is an equivalence relation. Along with a proof of transitivity I provide two transitivity-like results that preserve the size of one argument:

$$[\ \infty\ ]\ \mathit{ms} \approx \mathit{ns} \to [\ \infty\ ]\ \mathit{ns} \approx \mathit{os} \to [\ i\ ]\ \mathit{ms} \approx \mathit{os}$$
$$[\ \infty\ ]\ \mathit{ms} \sim_{\mathrm{L}} \mathit{ns} \to [\ i\ ]\ \mathit{ns} \approx \mathit{os} \to [\ i\ ]\ \mathit{ms} \approx \mathit{os}$$
$$[\ i\ ]\ \mathit{ms} \approx \mathit{ns} \to [\ \infty\ ]\ \mathit{ns} \sim_{\mathrm{L}} \mathit{os} \to [\ i\ ]\ \mathit{ms} \approx \mathit{os}$$

Note that working with $[\_]\_{\approx}'\_$ directly can be a little awkward, because (as discussed in Section 2) Agda sometimes requires the force projection to be written explicitly in the code, and $[\_]\_{\approx}'\_$ is defined in terms of *two* types with force fields. To avoid some plumbing a trick can be used:

$$\mathbf{record}\ [\_]\_{\approx}''\_$$
$$\qquad (i : \mathit{Size})\ (\mathit{ms}\ \mathit{ns} : \mathit{Colist}\ \mathbb{N}\ \infty) : \mathit{Set}\ \mathbf{where}$$
$$\quad \mathbf{coinductive}$$
$$\quad \mathbf{field}\ \mathsf{force} : \{j : \mathit{Size}{<}\ i\} \to [\ j\ ]\ \mathit{ms} \approx \mathit{ns}$$

This relation contains just a single force field, and it is a drop-in replacement for $[\_]\_{\approx}'\_$, in the sense that the relations are pointwise logically equivalent (in a size-preserving way).

The correctness proof has a similar structure to the correctness proof given in Section 8. Here is the type of the key lemma:

$$\mathit{Stack\text{-}OK}\ i\ k\ \mathit{tc}\ s \to$$
$$\mathit{Cont\text{-}OK}\ i\ \langle\ c\ ,\ s\ ,\ \mathit{comp\text{-}env}\ \rho\ \rangle\ k \to$$
$$[\ i\ ]\ \mathit{stack\text{-}sizes}\ \langle\ \mathit{comp}\ \mathit{tc}\ t\ c\ ,\ s\ ,\ \mathit{comp\text{-}env}\ \rho\ \rangle \approx$$
$$\qquad \mathit{numbers}\ ([\![\ t\ ]\!]_{\mathrm{S}}\ \rho\ \mathit{tc} \ggg k)\ (\mathit{length}\ s)$$

The *Cont-OK* and *Stack-OK* predicates are omitted due to lack of space. For full details of the correctness proof, see the accompanying code.

Now let us consider some examples. They are only outlined briefly. The first example is a program which in more usual notation may be written as $(\lambda x.x\ x)\ (\lambda x.x\ x)$. It is straightforward to show that this program is non-terminating, and that it requires unbounded stack space. The second example is f true, with the definition $f\ x\ =\ f$ true. This program gets compiled into code using a tail call. It is straightforward to show that this program is non-terminating, and that it runs in *bounded* stack space. Details about these examples can be found in the accompanying code, but I want to note that the statements regarding stack usage are proved without reasoning directly about the compiler or virtual machine. Instead the instrumented semantics is used.

10 Time Complexity

Let us now consider how time complexity can be handled in this setting. An obvious idea is to use the delay monad to keep track of the number of steps in an execution, with one later constructor standing for one step. This does not work in situations where a later constructor is necessary to establish that a definition is productive, but a corresponding time step is not wanted. Here I will focus on situations in which this is not the case.

The virtual machine in Section 7 produces one later constructor for each step of the computation, and the interpreter in Section 6 produces one later constructor for each application of a closure to a value. However, the compiler is not cost-preserving for these cost measures. Consider programs of the form con true $\cdot$ ($\cdots$ (con true $\cdot$ con true) $\cdots$), with $1 + n$ $\_\cdot\_$ constructors. The result of applying the interpreter to one of these programs is an immediate crash, without any later constructors, whereas the corresponding compiled program takes $3 + n$ steps to execute on the virtual machine.

To address this issue I have defined another instrumented interpreter, which is just the regular interpreter from Section 6 with some extra "ticks" inserted into the code:

$$\checkmark\_ : \mathit{Delay}_{\mathrm{C}}\ A\ i \to \mathit{Delay}_{\mathrm{C}}\ A\ i$$
$$\checkmark\ x = \mathsf{later}\ \lambda\ \{\ .\mathsf{force} \to x\ \}$$

A tick is just an application of the later constructor, but I use a special function to highlight the fact that the ticks are not needed to establish that the definition is productive. For the main function ticks have been inserted for variables, abstractions and literals (T stands for time):

$$[\![\_]\!]_{\mathrm{T}} : \mathit{Tm}\ n \to \mathit{Env}\ n \to \mathit{Delay}_{\mathrm{C}}\ \mathit{Value}\ i$$
$$[\![\ \mathsf{var}\ x\ ]\!]_{\mathrm{T}}\quad \rho = \checkmark\ \mathsf{return}\ (\mathit{index}\ x\ \rho)$$
$$[\![\ \mathsf{lam}\ t\ ]\!]_{\mathrm{T}}\quad \rho = \checkmark\ \mathsf{return}\ (\mathsf{lam}\ t\ \rho)$$
$$[\![\ t_1 \cdot t_2\ ]\!]_{\mathrm{T}}\quad \rho = \mathbf{do}\ v_1 \leftarrow [\![\ t_1\ ]\!]_{\mathrm{T}}\ \rho$$
$$\qquad\qquad v_2 \leftarrow [\![\ t_2\ ]\!]_{\mathrm{T}}\ \rho$$
$$\qquad\qquad v_1 \bullet_{\mathrm{T}} v_2$$
$$[\![\ \mathsf{call}\ f\ t\ ]\!]_{\mathrm{T}}\quad \rho = \mathbf{do}\ v \leftarrow [\![\ t\ ]\!]_{\mathrm{T}}\ \rho$$
$$\qquad\qquad \mathsf{lam}\ (\mathit{def}\ f)\ []\ \bullet_{\mathrm{T}}\ v$$
$$[\![\ \mathsf{con}\ b\ ]\!]_{\mathrm{T}}\quad \rho = \checkmark\ \mathsf{return}\ (\mathsf{con}\ b)$$
$$[\![\ \mathsf{if}\ t_1\ t_2\ t_3\ ]\!]_{\mathrm{T}}\ \rho = \mathbf{do}\ v_1 \leftarrow [\![\ t_1\ ]\!]_{\mathrm{T}}\ \rho$$
$$\qquad\qquad [\![\mathit{if}]\!]_{\mathrm{T}}\ v_1\ t_2\ t_3\ \rho$$

The application function is unchanged and ticks have been inserted for the then and else branches of conditionals (the code is omitted). Again I want to emphasise that it was not immediately obvious to me how to construct the instrumented semantics, it was developed together with its correctness proof.

This instrumented interpreter provides a suitable cost measure, in the sense that the cost of running a compiled program on the virtual machine is linear in the cost of running the corresponding source program on the interpreter (and vice versa):

$$[\ \infty\ ]\ \mathit{steps}\,(⟦\ t\ ⟧_{\mathrm{T}}\ [])\ \leq\ \mathit{steps}\,(\mathit{exec}\ \langle\ \mathit{comp}_0\ t\ ,\ []\ ,\ []\ \rangle)\ \times$$
$$[\ \infty\ ]\ \mathit{steps}\,(\mathit{exec}\ \langle\ \mathit{comp}_0\ t\ ,\ []\ ,\ []\ \rangle)\ \leq\ \ulcorner\ 1\ \urcorner \oplus \ulcorner\ 2\ \urcorner \otimes \mathit{steps}\,(⟦\ t\ ⟧_{\mathrm{T}}\ [])$$

Here the *steps* function gives the (conatural) number of later constructors in a computation, and _⊕_ and _⊗_ are addition and multiplication, respectively, of conatural numbers. Note that this statement applies to programs that terminate successfully, programs that crash, and programs that fail to terminate.

I have proved the compiler correctness result above by using a variant of weak bisimilarity that I call quantitative weak bisimilarity. This relation can be used to quantify the difference in the number of steps in two computations (compare to the definition of regular weak bisimilarity in Section 5):

$$\begin{array}{l}
\textbf{data}\ [_|_|_|_|_]_\approx_{\mathrm{D}}_\ (i : \mathit{Size})\ (m^{\mathrm{l}}\ m^{\mathrm{r}} : \mathit{Conat}\ \infty) : \\
\quad (n^{\mathrm{l}}\ n^{\mathrm{r}} : \mathit{Conat}\ \infty)\ (x\ y : \mathit{Delay}\ A\ \infty) \to \mathit{Set}\ \textbf{where} \\
\quad \mathsf{now}\ \ : [\ i \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid n^{\mathrm{l}} \mid n^{\mathrm{r}}\ ]\ \mathsf{now}\ x \approx_{\mathrm{D}} \mathsf{now}\ x \\
\quad \mathsf{later}\ : [\ i \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid n^{\mathrm{l}} \oplus m^{\mathrm{l}} \mid n^{\mathrm{r}} \oplus m^{\mathrm{r}}\ ] \\
\qquad\quad x\ .\mathsf{force} \approx_{\mathrm{D}}{}'\ y\ .\mathsf{force} \to \\
\qquad\quad [\ i \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid n^{\mathrm{l}} \mid n^{\mathrm{r}}\ ]\ \mathsf{later}\ x \approx_{\mathrm{D}} \mathsf{later}\ y \\
\quad \mathsf{later}^{\mathrm{l}} : [\ i \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid n^{\mathrm{l}}\ .\mathsf{force} \mid n^{\mathrm{r}}\ ]\ x\ .\mathsf{force} \approx_{\mathrm{D}} y \to \\
\qquad\quad [\ i \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid \mathsf{suc}\ n^{\mathrm{l}} \mid n^{\mathrm{r}}\ ]\ \mathsf{later}\ x \approx_{\mathrm{D}} y \\
\quad \mathsf{later}^{\mathrm{r}} : [\ i \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid n^{\mathrm{l}} \mid n^{\mathrm{r}}\ .\mathsf{force}\ ]\ x \approx_{\mathrm{D}} y\ .\mathsf{force} \to \\
\qquad\quad [\ i \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid n^{\mathrm{l}} \mid \mathsf{suc}\ n^{\mathrm{r}}\ ]\ x \approx_{\mathrm{D}} \mathsf{later}\ y
\end{array}$$

Note that, when the $\mathsf{later}^{\mathrm{l}}$ or $\mathsf{later}^{\mathrm{r}}$ constructors are used, one of the "n" indices is incremented, and that when the later constructor is used each of the "n" indices is decremented by the corresponding "m" parameter. The following property provides a characterisation of the relation (and is used in the compiler correctness proof):

$$\begin{array}{l}
[\ \infty \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid n^{\mathrm{l}} \mid n^{\mathrm{r}}\ ]\ x \approx_{\mathrm{D}} y \Leftrightarrow \\
[\ \infty\ ]\ x \approx_{\mathrm{D}} y\ \times \\
[\ \infty\ ]\ \mathit{steps}\ x \leq n^{\mathrm{l}} \oplus (\ulcorner\ 1\ \urcorner \oplus m^{\mathrm{l}}) \otimes \mathit{steps}\ y\ \times \\
[\ \infty\ ]\ \mathit{steps}\ y \leq n^{\mathrm{r}} \oplus (\ulcorner\ 1\ \urcorner \oplus m^{\mathrm{r}}) \otimes \mathit{steps}\ x
\end{array}$$

The relation holds if and only if the two computations are weakly bisimilar and each of them is bounded by a linear function of the other (assuming that m^{l}, m^{r}, n^{l} and n^{r} are constants). Note that this characterisation is not stated as a size-preserving function (with an arbitrary size i instead of ∞). The left-to-right direction of this characterisation can be made size-preserving, but—assuming that Agda is free of bugs—the right-to-left direction can be made size-preserving if and only if the carrier type A is uninhabited (see the accompanying code for a proof).

A form of weakening can be proved for the relation:

$$\begin{array}{l}
[\ \infty\ ]\ m^{\mathrm{l}} \leq m^{\mathrm{l}\prime} \to [\ \infty\ ]\ m^{\mathrm{r}} \leq m^{\mathrm{r}\prime} \to \\
[\ \infty\ ]\ n^{\mathrm{l}}\ \leq n^{\mathrm{l}\prime}\ \to [\ \infty\ ]\ n^{\mathrm{r}}\ \leq n^{\mathrm{r}\prime}\ \to \\
[\ i \mid m^{\mathrm{l}}\ \mid m^{\mathrm{r}}\ \mid n^{\mathrm{l}}\ \mid n^{\mathrm{r}}\ ]\ x \approx_{\mathrm{D}} y \to \\
[\ i \mid m^{\mathrm{l}\prime} \mid m^{\mathrm{r}\prime} \mid n^{\mathrm{l}\prime} \mid n^{\mathrm{r}\prime}\ ]\ x \approx_{\mathrm{D}} y
\end{array}$$

It is also possible to prove the following three transitivity-like results:

$$\begin{array}{l}
[\ i \mid m \mid \ulcorner\ 0\ \urcorner \mid n \mid \ulcorner\ 0\ \urcorner\ ]\ x \approx_{\mathrm{D}} y \to [\ i\ ]\ y \sim_{\mathrm{D}} z \to \\
\quad [\ i \mid m \mid \ulcorner\ 0\ \urcorner \mid n \mid \ulcorner\ 0\ \urcorner\ ]\ x \approx_{\mathrm{D}} z \\
[\ i \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid n^{\mathrm{l}} \mid n^{\mathrm{r}}\ ]\ x \approx_{\mathrm{D}} y \to [\ \infty\ ]\ y \sim_{\mathrm{D}} z \to \\
\quad [\ i \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid n^{\mathrm{l}} \mid n^{\mathrm{r}}\ ]\ x \approx_{\mathrm{D}} z \\
[\ \infty\ ]\ x \sim_{\mathrm{D}} y \to [\ i \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid n^{\mathrm{l}} \mid n^{\mathrm{r}}\ ]\ y \approx_{\mathrm{D}} z \to \\
\quad [\ i \mid m^{\mathrm{l}} \mid m^{\mathrm{r}} \mid n^{\mathrm{l}} \mid n^{\mathrm{r}}\ ]\ x \approx_{\mathrm{D}} z
\end{array}$$

Weakening and the first transitivity-like result are used in the compiler correctness proof. This proof has a structure which is similar to those of the proofs in Sections 8 and 9. The key lemma has the following type:

$$\begin{array}{l}
\mathit{Stack\text{-}OK}\ i\ k\ \delta\ \mathit{tc}\ s \to \\
\mathit{Cont\text{-}OK}\ i\ \langle\ c\ ,\ s\ ,\ \mathit{comp\text{-}env}\ \rho\ \rangle\ k\ \delta \to \\
[\ i \mid \ulcorner\ 1\ \urcorner \mid \ulcorner\ 0\ \urcorner \mid \mathit{max}\ \ulcorner\ 1\ \urcorner\ \delta \mid \ulcorner\ 0\ \urcorner\ ] \\
\quad \mathit{exec}\ \langle\ \mathit{comp}\ \mathit{tc}\ t\ c\ ,\ s\ ,\ \mathit{comp\text{-}env}\ \rho\ \rangle \approx_{\mathrm{D}} ⟦\ t\ ⟧_{\mathrm{T}}\ \rho \gg\!\!= k
\end{array}$$

Here *max* is the binary maximum function for conatural numbers, and the *Cont-OK* and *Stack-OK* predicates are again omitted. The δ parameter corresponds to what I thought was the most difficult part of the correctness proof. Note that there is only one later constructor in the closure case of $\_\bullet_{\mathrm{T}}\_$. This corresponds to (up to) two steps in the virtual machine: one for app, cal or tcl, and one for the ret instruction. However, there can be an arbitrary delay between these two steps—the intermediate computation can even fail to terminate. The δ parameter was introduced to address this delay. It is related to the number of pending ret instructions.

11 Related Work

The work of Young [1989, 1988] is closely related to this work. As was mentioned in the introduction Young defines interpreters using the approach with fuel, tracks the space consumption of compiled programs in a source-level interpreter which "crashes" if it runs out of stack space, and proves compiler correctness for programs that terminate without running out of stack space. The main methodological differences between this work and the work of Young are perhaps the following ones:

- I use the delay monad instead of fuel.

- I return a trace of stack sizes, instead of crashing when stack space runs out.
- I prove compiler correctness for all programs, not only those that terminate successfully.
- I treat a seemingly useful notion of “time”. Young writes that his fuel counter “bears a rather complicated and unintuitive relation to the number of ‘steps’ executed” [1989]. His compiler correctness result implies that, if the source program terminates successfully in n steps, then the target program terminates successfully in a number of steps that is a function of n as well as some other arguments, including the current execution environment. The code listing for this function spans more than two pages [Young 1988]. Young defined this function because the logic he used (the Boyer-Moore logic) does not support existential quantifiers.

I want to note that Young treats languages that are much more complicated than the ones treated here.

The CerCo project [Amadio et al. 2014] included the development of an optimising compiler from a subset of C to machine code, that was partly verified (I found a number of axioms in the source code). I think the project used small-step definitional interpreters (one or more fuel-based, and one or more that produce coinductive resumptions) to give the semantics of languages. It was argued (roughly) that, in a setting where the aim is to establish upper bounds on (non-asymptotic) worst-case execution times, *uniform* cost models for high-level source code may not be sufficiently precise, because one piece of source code can have different performance characteristics depending on how it is used. This project took a different approach: the compiler produced a source program annotated with cost information (processor cycles and stack space usage). Note that the instrumented interpreter given in Section 9 does not provide a cost model that is uniform in this sense, because the stack space usage for a call depends on whether or not it is compiled to a tail call. However, I do not claim that the approach I have taken scales to precise analysis of optimised machine code. Perhaps it could be used for reasoning about *asymptotic* time and/or space complexity.

I am not aware of any other mechanically checked compiler correctness result involving resource guarantees for languages defined using total definitional interpreters. The CakeML project (a verified compiler for a subset of Standard ML) uses definitional interpreters [Owens et al. 2016], but as far as I know it does not treat time or space complexity. Owens et al. [2016] model input and output by returning a trace of input/output actions. This is similar to the use of a trace of stack sizes in this work. A difference is that Owens et al. use the approach with fuel, so a single run of their interpreter can only produce a finite trace. They handle this by taking the least upper bound of a chain of finite traces (ordered by a prefix relation). When the approach described in this work is used there is no need to use least upper bounds in this way (as opposed to the way described in Section 3 and later), because potentially infinite traces are produced directly by the interpreter.

There are mechanically checked compiler correctness results involving resource guarantees for languages defined using something other than total definitional interpreters. Blazy et al. [2014] have extended the C compiler CompCert [Leroy 2009] with (formally verified) loop bound estimation. The semantics used are small-step, and the guarantees are given for terminating programs. Carbonneaux et al. [2014a] have developed Quantitative CompCert, a variant of CompCert. Quantitative CompCert gives guarantees about stack space usage that hold also in the presence of non-termination, and comes with mechanically checked proofs. The source and target languages are specified using small-step semantics, and the target language has a finite stack. Carbonneaux et al. [2014b] also discuss tail calls.

Ancona et al. [2017] expressed some scepticism towards how well definitional interpreters capture certain properties of non-terminating programs, and gave a concrete example: “For instance, if a program consists of an infinite loop that allocates new heap space at each step without releasing it, one would like to conclude that it will eventually crash even though a definitional interpreter returns timeout for all possible values of the step counter.” The development in Section 3 can be seen as evidence that definitional interpreters (at least those using the delay monad rather than step counters) can be used to state and prove this kind of property.

12 Conclusions

I have shown one way in which time and space complexity can be handled when the semantics of a programming language is defined using a total definitional interpreter implemented using the delay monad. I have also presented some techniques that can be used to work with the resulting semantics. I want to emphasise that the presented approach works for non-terminating programs.

I have only treated toy examples in this text, but I hope that the text provides guidance to others who want to try the same approach.

Acknowledgements

I would like to thank Andreas Abel, Robin Adams, Thorsten Altenkirch, Davide Ancona, Francesco Dagnino, Ulf Norell, Andrea Vezzosi, Elena Zucca and some anonymous reviewers for useful feedback. This work has been supported by a grant from the Swedish Research Council (621-2013-4879).

References

Andreas Abel. 2012. Type-Based Termination, Inflationary Fixed-Points, and Mixed Inductive-Coinductive Types. In *Proceedings 8th Workshop on Fixed Points in Computer Science*. https://doi.org/10.4204/EPTCS.77.1

Andreas Abel and Brigitte Pientka. 2016. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming* (2016). https://doi.org/10.1017/S0956796816000022

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *POPL '13, Proceedings of 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* https://doi.org/10.1145/2429069.2429075

The Agda Team. 2018. The Agda Wiki. Retrieved 2018-07-11 from http://wiki.portal.chalmers.se/agda/

Roberto M. Amadio, Nicolas Ayache, Francois Bobot, Jaap P. Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, and Claudio Sacerdoti Coen. 2014. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis, Third International Workshop, FOPARA 2013.* https://doi.org/10.1007/978-3-319-12466-7_1

Nada Amin and Tiark Rompf. 2017. Type Soundness Proofs with Definitional Interpreters. In *POPL'17, Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3009837.3009866

Davide Ancona, Francesco Dagnino, and Elena Zucca. 2017. Reasoning on Divergent Computations with Coaxioms. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017). https://doi.org/10.1145/3133905

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proceedings of the ACM on Programming Languages* 2, POPL (2018). https://doi.org/10.1145/3158104

Nick Benton, Andrew Kennedy, and Carsten Varming. 2009. Some Domain Theory and Denotational Semantics in Coq. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009.* https://doi.org/10.1007/978-3-642-03359-9_10

Sandrine Blazy, André Maroneze, and David Pichardie. 2014. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Verified Software: Theories, Tools, Experiments, 5th International Conference, VSTTE 2013.* https://doi.org/10.1007/978-3-642-54108-7_15

Robert S. Boyer and J Strother Moore. 1997. Mechanized Formal Reasoning about Programs and Computing Machines. In *Automated Reasoning and Its Applications, Essays in Honor of Larry Wos.* The MIT Press.

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* (2005). https://doi.org/10.2168/LMCS-1(2:1)2005

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014a. End-to-End Verification of Stack-Space Bounds for C Programs. In *PLDI'14, Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation.* https://doi.org/10.1145/2594291.2594301

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014b. *End-to-End Verification of Stack-Space Bounds for C Programs.* Technical Report YALEU/DCS/TR-1487. Yale University, Department of Computer Science.

Nils Anders Danielsson. 2012. Operational Semantics Using the Partiality Monad. In *ICFP'12, Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming.* https://doi.org/10.1145/2364527.2364546

Nils Anders Danielsson. 2018. Up-to Techniques using Sized Types. *Proceedings of the ACM on Programming Languages* 2, POPL (2018). https://doi.org/10.1145/3158131

Nils Anders Danielsson and Thorsten Altenkirch. 2010. Subtyping, Declaratively: An Exercise in Mixed Induction and Coinduction. In *Mathematics of Program Construction, 10th International Conference, MPC 2010.* https://doi.org/10.1007/978-3-642-13321-3_8

Richard Kelsey, William Clinger, and Jonathan Rees (Eds.). 1998. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* (1998). https://doi.org/10.1023/A:1010051815785

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* (2009). https://doi.org/10.1145/1538788.1538814

Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Information and Computation* (2009). https://doi.org/10.1016/j.ic.2007.12.004

Keiko Nakata and Tarmo Uustalu. 2009. Trace-Based Coinductive Operational Semantics for While: Big-step and Small-step, Relational and Functional Styles. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009.* https://doi.org/10.1007/978-3-642-03359-9_26

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Chalmers University of Technology and Göteborg University.

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Programming Languages and Systems, 25th European Symposium on Programming, ESOP 2016.* https://doi.org/10.1007/978-3-662-49498-1_23

Christine Paulin-Mohring. 2009. A constructive denotational semantics for Kahn networks in Coq. In *From Semantics to Computer Science: Essays in Honour of Gilles Kahn.* Cambridge University Press.

Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. 2015. A Model of PCF in Guarded Type Theory. In *The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).* https://doi.org/10.1016/j.entcs.2015.12.020

John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *ACM '72, Proceedings of the ACM annual conference.* https://doi.org/10.1145/800194.805852

Jorge Luis Sacchini. 2015. Well-Founded Sized Types in the Calculus of (Co)Inductive Constructions. Draft. Retrieved 2018-07-11 from http://web.archive.org/web/20160531152811/http://www.qatar.cmu.edu:80/~sacchini/well-founded/well-founded.pdf

Jeremy Siek. 2013. Type Safety in Three Easy Lemmas. Blog post. Retrieved 2018-07-11 from http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html

William D. Young. 1988. *A Verified Code Generator for a Subset of Gypsy.* Technical Report 33. Computational Logic Inc. Retrieved 2018-10-13 from http://www.computationallogic.com/reports/files/033.ps

William D. Young. 1989. A Mechanically Verified Code Generator. *Journal of Automated Reasoning* (1989). https://doi.org/10.1007/BF00243134