Total Definitional Interpreters for Looping Programs

NILS ANDERS DANIELSSON, University of Gothenburg & Chalmers University of Technology, Sweden

A question has been raised regarding how useful total definitional interpreters are in a setting in which programs run forever. For instance, is it possible to give a semantics in the form of a total definitional interpreter in such a way that one can distinguish between non-terminating programs that run in bounded space and those that do not? Here it is shown that this is possible.

The main object of study is an untyped λ-calculus with named recursive definitions. The semantics of this language is specified using a "big-step" definitional interpreter, using the delay monad to ensure that the definition is a total function. A (provably correct) compiler is given to the language of a virtual machine with tail calls, which is specified using a "small-step" definitional interpreter. A big-step definitional interpreter which is instrumented with information about stack sizes is also given for the λ-calculus, and these stack sizes are shown to agree with those encountered by the virtual machine. Two non-terminating programs are presented, and it is shown that one runs in bounded stack space, and that the other does not.

The development is accompanied by machine-checked proofs.

Additional Key Words and Phrases: coinduction, sized types, space complexity

1 INTRODUCTION

One can define the semantics of a programming language in many different ways. One approach is to use definitional interpreters [Reynolds 1972], i.e. interpreters that define the semantics of a language. If the aim is to use the semantics to produce machine-checked proofs about the language or programs written in the language, then it may be beneficial to define the interpreter using some kind of proof assistant. When the language that is interpreted has some kind of effect, and the internal programming language or function definition facility of the proof assistant that is used does not have direct support for that effect, then some kind of workaround is needed. For the effect of general (unrestricted) recursion several workarounds have been investigated, including the delay monad and other coinductive types [Capretta 2005; Nakata and Uustalu 2009; Paulin-Mohring 2009; Benton et al. 2009; Danielsson 2012], step-counting/fuel [Leroy and Grall 2009; Siek 2013; Owens et al. 2016; Amin and Rompf 2017; Bach Poulsen et al. 2018], and guarded recursive types [Paviotti et al. 2015].[1]

Recently Ancona et al. [2017b] introduced a method for defining the semantics of non-terminating languages based on coaxioms. They expressed some scepticism towards how well definitional interpreters can handle 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."

In this work I show that it is possible to handle this kind of situation using definitional interpreters. More concretely:

- In order to focus on the essentials I define a tiny programming language where programs are potentially infinite lists of the instructions "allocate" and "deallocate" (see Section 4). I define a definitional interpreter for this language using the delay monad (which is discussed in Section 3). The interpreter models a situation with bounded memory, so it takes the memory size as an input parameter, and if memory runs out, then it crashes. I provide an example of

[1] For the purpose of this set of references I count denotational semantics as definitional interpreters.

a program that provably crashes, and some examples of programs that provably run forever in bounded memory.

- In response to correspondence with Ancona et al. I define a second semantics of the programming language (Section 5). This semantics allows the memory to grow without limit, but instead I reason about the memory usage. I show that one non-terminating program requires an unbounded amount of memory, whereas other non-terminating programs only require bounded memory. I also define a simple program optimiser (Section 7), and show that this optimiser never increases maximum memory usage, but sometimes decreases it.
- As a somewhat more realistic example I present definitional interpreters for a small λ-calculus with booleans and recursive, unary definitions (Section 8) and a simple virtual machine with tail calls (Section 9), and a provably correct compiler from the former to the latter (Section 10). These languages and the compiler are based on work by Leroy and Grall [2009] and Danielsson [2012]. The interpreter for the λ-calculus is then instrumented with information about the stack size, and I prove that the compiler preserves maximum stack usage (Section 11). I give two examples of non-terminating programs, one that runs in bounded stack space, and one that does not (Section 12). I prove this by using the instrumented interpreter, not the lower-level virtual machine.

Related work is discussed in Section 13.

All the main results and examples in this paper have been formalised using Agda [Agda Team 2018], with the K rule turned off, and the code is available to inspect. The formalisation makes heavy use of sized types (see Section 2 for a quick introduction). 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 Agda code in the text below. (There are some differences between the text and the accompanying code, but they are minor. For instance, the accompanying code is sometimes more general due to the use of universe polymorphism.)

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 are used in several sections below.)

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

```
mutual

  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. (*Set* is a type of small types.)

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 $j : \mathit{Size}< i$ 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 $j : \mathit{Size}< i$.

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 $i : \mathit{Size}< \infty$ holds for every size *i*, including ∞ itself.

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 $j : \mathit{Size}< i$.

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":[2]

```
mutual

  infinity : ∀ {i} → Conat i
  infinity = suc infinity′

  infinity′ : ∀ {i} → Conat′ i
  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). The following, more compact notation with an anonymous copattern is also available:

```
infinity : ∀ {i} → Conat i
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} → Conat i
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 ($j : \mathit{Size}< i$) 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 $\infty : \mathit{Size}< \infty$ has led to problems, and a slightly different design has been discussed. However, I would be surprised if any bugs in Agda invalidated the main ideas presented below.

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. Sacchini [2015] studies a language with dependent types, and sketches a normalisation proof, but his language is designed somewhat differently from Agda.

[2] Agda does not by default make the projection force available unqualified. I do not include code that only manipulates what names are in scope in this text.

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

```
mutual

  data [_]_≤_ (i : Size) : Conat ∞ → 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).

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

```
mutual

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

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

3 THE DELAY MONAD

This section contains a brief presentation of the delay monad [Capretta 2005], which is used in several sections below. The delay monad represents potentially non-terminating computations:

```
mutual

  data Delay (A : Set) (i : Size) : Set where
    now  : A → Delay A i
    later : Delay′ A i → Delay A i

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

The constructor 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:

```
never : ∀ {A i} → Delay A i
never = later λ { .force → never }
```

The delay monad is a monad. The return and bind combinators can be defined in the following way:

```
return : ∀ {A i} → A → Delay A i
return x = now x

_>>=_ : ∀ {A B i} → Delay A i → (A → Delay B i) → Delay B i
now x >>= f = f x
later x >>= f = later λ { .force → x .force >>= f }
```

The monad laws can be proved up to strong bisimilarity, defined in the following way:

```
mutual

  data [_]_~D_ {A : Set} (i : Size) : Delay A ∞ → Delay A ∞ → Set where
    now   : ∀ {x}     →                          [ i ] now x ~D now x
    later : ∀ {x y} → [ i ] x .force ~D′ y .force → [ i ] later x ~D later y

  record [_]_~D′_ {A : Set} (i : Size) (x y : Delay A ∞) : Set where
    coinductive
    field force : {j : Size< i} → [ j ] x ~D y
```

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. Here is one way to define this relation [Danielsson and Altenkirch 2010; Danielsson 2018]:

```
mutual

  data [_]_≈D_ {A : Set} (i : Size) : Delay A ∞ → Delay A ∞ → Set where
    now    : ∀ {x}     →                          [ i ] now x ≈D now x
    later  : ∀ {x y} → [ i ] x .force ≈D′ y .force → [ i ] later x ≈D later y
    later⁽ : ∀ {x y} → [ i ] x .force ≈D y       → [ i ] later x ≈D y
    later⁾ : ∀ {x y} → [ i ] x       ≈D y .force → [ i ] x       ≈D later y

  record [_]_≈D′_ {A : Set} (i : Size) (x y : Delay A ∞) : Set where
    coinductive
    field force : {j : Size< i} → [ j ] x ≈D y
```

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

4 AN INTERPRETER WITH BOUNDED MEMORY

Now let us see one way to distinguish between looping programs that require bounded and unbounded memory, while using a definitional interpreter to give semantics to the programs. I will define an interpreter which models a machine with bounded memory. The interpreter keeps track of how much memory is used, and crashes if memory runs out.

In order to avoid cluttering the presentation with details I use a very basic programming language, where programs consist of potentially infinite lists of instructions, and there are only two instructions, allocate and deallocate:

```
data Stmt : Set where
  allocate deallocate : Stmt
```

```
Program : Size → Set
Program i = Colist Stmt i
```

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

```
mutual

  data Colist (A : Set) (i : Size) : Set where
    []  : Colist A i
    _::_ : A → Colist′ A i → Colist A i

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

The type *Heap limit* represents a heap with at most *limit* memory cells. In this simple setting the heap does not contain any data, it consists merely of the heap size and a proof that this number is at most *limit*:

```
record Heap (limit : ℕ) : Set where
  field size    : ℕ
        bounded : size ≤ limit
```

It is straightforward to shrink a heap. The following function shrinks the heap by one, unless it is empty, in which case the size is left unchanged (*pred* is the predecessor function on natural numbers; I have omitted the proof component of the heap record):

```
shrink : ∀ {l} → Heap l → Heap l
shrink h = record { size = pred (h .size) }
```

I have also defined a function that tries to increase the heap size by one. This function fails if the heap size limit would be exceeded. It makes use of the following proof-producing comparison operator (where $A \uplus B$ is the binary sum of A and B, with constructors $\mathsf{inj}_1 : A \to A \uplus B$ and $\mathsf{inj}_2 : B \to A \uplus B$):

```
_≤⊎>_ : ∀ m n → m ≤ n ⊎ n < m
```

If the current heap size is equal to or greater than the limit, then nothing is returned, and otherwise the heap size is increased (*Maybe A* has two constructors, nothing : *Maybe A* and just : $A \to$ *Maybe A*):

```
grow : ∀ {l} → Heap l → Maybe (Heap l)
grow {l} h with l ≤⊎> h .size
... | inj₁ _    = nothing
... | inj₂ h<l = just (record { size = suc (h .size) })
```

Here suc is the (overloaded) successor constructor for the unary, inductive representation of natural numbers that I use. (Note that ≤⊎> and h<l are single tokens with—hopefully—informative names.)

Now let us turn to the interpreter. The *step* function performs one step of computation. Deallocation always succeeds, whereas allocation can fail if memory runs out:

```
step : ∀ {l} → Stmt → Heap l → Maybe (Heap l)
step deallocate heap = just (shrink heap)
step allocate   heap = grow heap
```

The interpreter takes a program and a heap to a potentially non-terminating, potentially crashing computation that, if it is successful, returns the final heap. It uses *step* repeatedly until the program ends or crashes:

```
⟦_⟧ : ∀ {i l} → Program i → Heap l → Delay (Maybe (Heap l)) i
⟦ []     ⟧ heap = now (just heap)
⟦ s :: p ⟧ heap with step s heap
... | nothing          = crash
... | just new-heap = later λ { .force → ⟦ p .force ⟧ new-heap }
```

Here the *crash* function constructs crashing computations:

```
crash : ∀ {i l} → Delay (Maybe (Heap l)) i
crash = now nothing
```

Note that *crash* is not weakly bisimilar to *never*:[3]

```
¬ [ i ] now x ≈D never
```

(The negation of A is the type of functions from A to the empty type.)

Now let us consider some examples. Here are two looping programs that run in constant space, and one that does not run in bounded space:

```
constant-space : ∀ {i} → Program i
constant-space = allocate ::′ deallocate :: λ { .force → constant-space }

constant-space₂ : ∀ {i} → Program i
constant-space₂ = allocate    ::′ allocate    ::′
                  deallocate ::′ deallocate :: λ { .force → constant-space₂ }

unbounded-space : ∀ {i} → Program i
unbounded-space = allocate :: λ { .force → unbounded-space }
```

The definitions make use of _::′_, a variant of _::_ with an unprimed second argument, in order to avoid some clutter:

```
_::′_ : ∀ {A i} → A → Colist A i → Colist A i
x ::′ xs = x :: λ { .force → xs }
```

The first program, *constant-space*, is an endless sequence alternating between allocate and deallocate. This program crashes if the initial heap is full, but otherwise it runs forever. More precisely, in the former case the value returned by the interpreter is weakly bisimilar to *crash*, and otherwise it is weakly bisimilar to *never*:

```
(h : Heap l) → h .size ≡ l → [ i ] ⟦ constant-space ⟧ h ≈D crash
(h : Heap l) → h .size < l → [ i ] ⟦ constant-space ⟧ h ≈D never
```

The proofs are simple and omitted. It is also easy to prove that *constant-space*$_2$ runs forever if it is possible to allocate at least two more memory cells in the initial heap:

```
(h : Heap l) → 2 + h .size ≤ l → [ i ] ⟦ constant-space₂ ⟧ h ≈D never
```

However, because *unbounded-space* is an infinite sequence of allocate instructions it always runs out of memory and crashes:

```
(h : Heap l) → [ i ] ⟦ unbounded-space ⟧ h ≈D crash
```

[3]Here and below I sometimes omit argument type declarations, in this case for i and x, from type signatures.

Before leaving this section, let me define a relation that relates programs that behave the same for all heaps with sufficient capacity:

```
_≃_ : Program ∞ → Program ∞ → Set
p ≃ q = ∃ λ c → ∀ l (h : Heap l) → c + h .size ≤ l → [ ∞ ] ⟦ p ⟧ h ≈ᴰ ⟦ q ⟧ h
```

Two programs p and q are related if there is a natural number c ("capacity") such that, for any heap which can be extended with at least c memory cells, the semantics of p is weakly bisimilar to the semantics of q. (If B has type $A \to Set$, then the type $\exists\, B$ consists of dependent pairs $(x\ ,\ y)$, where—at least if we ignore subtyping—x has type A and y has type $B\ x$.) This relation is an equivalence relation, and it is preserved by the cons operator:

```
p .force ≃ q .force → s :: p ≃ s :: q
```

One can prove that *constant-space* and $constant\text{-}space_2$ are related by letting the capacity be 2 and using some of the results mentioned above:

```
constant-space ≃ constant-space₂
```

However, *constant-space* and *unbounded-space* are not related:

```
¬ constant-space ≃ unbounded-space
```

In order to prove this, let us assume that we have *constant-space* ≃ *unbounded-space*. This means that there is some value c for which, for any limit l and heap h : *Heap* l such that $c + h$ .size is bounded by l, ⟦ *constant-space* ⟧ h is weakly bisimilar to ⟦ *unbounded-space* ⟧ h. If we let l be $c + 1$ and h be an empty heap, then we get a contradiction, because ⟦ *constant-space* ⟧ h is *never*, and ⟦ *unbounded-space* ⟧ h is *crash*.

5 AN INTERPRETER WITH UNBOUNDED MEMORY

As a follow-up to the development in the previous section I asked Ancona et al. for further examples of properties for which it is not clear to them if definitional interpreters work well. One part of the response was that they wondered if I could define a semantics that returns the largest heap size used by the program, or infinity if there is no bound on this size. A second step might be to show that some program transformation preserves or improves on the maximum heap size. In this section I will address the first part, and the second part in Section 7.

Definitional interpreters have the feature/drawback that they are computable functions. It is a hard ask to compute the maximum memory consumption of a program. (Under certain assumptions I can show that this is impossible in Agda, see Section 6.) Thus I will instead settle for returning a trace of the encountered heap sizes, and reason about this trace. Thus my solution does not quite match what Ancona et al. asked for.

The choice to just return a trace of heap sizes makes the definitional interpreter very simple. The function *modify* computes the new heap size, given an instruction and the previous heap size:

```
modify : Stmt → ℕ → ℕ
modify allocate   = suc
modify deallocate = pred
```

The interpreter uses this function repeatedly, returning all the encountered heap sizes (including the initial one):

```
mutual

  ⟦_⟧ : ∀ {i} → Program i → ℕ → Colist ℕ i
  ⟦ p ⟧ h = h :: ⟦ p ⟧′ h
```

```
⟦_⟧′ : ∀ {i} → Program i → ℕ → Colist′ ℕ i
⟦ []     ⟧′ h .force = []
⟦ s :: p ⟧′ h .force = ⟦ p .force ⟧ (modify s h)
```

This may not seem like much of an interpreter. However, in Section 11 below I will use a similar technique to instrument a definitional interpreter for a λ-calculus with information about stack sizes. The current section and Section 7 introduce some of the main ideas that will be employed later.

Given a colist (potentially infinite list) of natural numbers, what is its least upper bound? An upper bound predicate can be defined in the following way (where ⌜_⌝ maps natural numbers to the corresponding conatural numbers):

```
[_]_⊑_ : Size → Colist ℕ ∞ → Conat ∞ → Set
[ i ] ms ⊑ n = □ i (λ m → [ ∞ ] ⌜ m ⌝ ≤ n) ms
```

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

```
mutual

  data □ {A : Set} (i : Size) (P : A → Set) : Colist A ∞ → Set where
    []  : □ i P []
    _::_ : ∀ {x xs} → P x → □′ i P (xs .force) → □ i P (x :: xs)

  record □′ {A : Set} (i : Size) (P : A → Set) (xs : Colist A ∞) : Set where
    coinductive
    field force : {j : Size< i} → □ j P xs
```

Below the following primed variant of [_]_⊑_ will also be used:

```
[_]_⊑′_ : Size → Colist ℕ ∞ → Conat ∞ → Set
[ i ] ms ⊑′ n = □′ i (λ m → [ ∞ ] ⌜ m ⌝ ≤ n) ms
```

Given a definition of upper bounds it is easy to define a least upper bound predicate:

```
LUB : Colist ℕ ∞ → Conat ∞ → Set
LUB ns n = [ ∞ ] ns ⊑ n × (∀ n′ → [ ∞ ] ns ⊑ n′ → [ ∞ ] n ≤ n′)
```

A least upper bound is an upper bound that is bounded by every upper bound. Least upper bounds are unique up to bisimilarity:

```
LUB ns n₁ → LUB ns n₂ → [ i ] n₁ ∼ℕ n₂
```

This follows from antisymmetry for conatural numbers.

Now the maximum heap usage of a program that starts with an empty heap can be defined:

```
Maximum-heap-usage : Program ∞ → Conat ∞ → Set
Maximum-heap-usage p n = LUB (⟦ p ⟧ 0) n
```

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

Let us now revisit the example programs from Section 4. When the trace semantics from this section is used all three programs are non-terminating, in the sense that their traces are infinitely long. However, they have different space complexities: the maximum heap usage of *constant-space* is one, the maximum heap usage of $constant\text{-}space_2$ is two, and the maximum heap usage of *unbounded-space* is infinity:

Maximum-heap-usage constant-space ⌜ 1 ⌝
Maximum-heap-usage constant-space$_2$ ⌜ 2 ⌝
Maximum-heap-usage unbounded-space infinity

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

∀ p → (∀ n → ¬ [∞] ⟦ p ⟧ 0 ⊑ ⌜ n ⌝) → *Maximum-heap-usage p infinity*

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:

(∀ n → ¬ [∞] ms ⊑ ⌜ n ⌝) → [∞] ms ⊑ m → [∞] m ∼$_{\mathrm{N}}$ *infinity*

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

6 THE MAXIMUM HEAP USAGE CANNOT BE COMPUTED

It might be nice if one could always compute a maximum heap usage for a program. However, one can prove that (one form of) this statement implies the "weak limited principle of omniscience", a constructive taboo that should neither be provable nor disprovable in Agda (in the absence of bugs):

(∀ p → ∃ λ n → *Maximum-heap-usage p n*) → *WLPO*

WLPO is the statement that, for an arbitrary function from natural numbers to booleans, one can determine if this function returns false for all inputs:

```
Dec : Set → Set
Dec P = P ⊎ ¬ P

WLPO : Set
WLPO = (f : ℕ → Bool) → Dec (∀ n → f n ≡ false)
```

Note that *WLPO* follows from excluded middle (and, when excluded middle is formulated as in homotopy type theory, extensionality for functions).

Given a decision procedure for *Maximum-heap-usage* with the type given above one can prove *WLPO* by constructing a program whose maximum heap usage is zero if and only if the function always returns false:

```
p : ∀ {i} → (ℕ → Bool) → Program i
p f = if f 0 then allocate     ::′ []
              else deallocate :: λ { .force → p (λ n → f (1 + n)) }
```

It is also possible to go in the other direction. If *WLPO* holds, then a least upper bound can be determined for every colist:[4]

WLPO → (∀ ms → ∃ λ n → *LUB ms n*)

In order to prove this, let us define the following function:

```
>0 : Colist ℕ ∞ → ℕ → Bool
>0 []          _       = false
>0 (m     :: ms) (suc n) = >0 (ms .force) n
>0 (zero  :: ms) zero    = false
>0 (suc m :: ms) zero    = true
```

[4]This result was obtained in collaboration with Andreas Abel and Ulf Norell.

The boolean *>0 ms n* is true if and only if *ms* contains a positive number at position *n* (counting from zero). Now the least upper bound can be computed in the following way, by making use of *wlpo* : *WLPO*:

```
lub : ∀ {i} → Colist ℕ ∞ → Conat i
lub ms with wlpo (>0 ms)
... | inj₁ _ = zero
... | inj₂ _ = suc λ { .force → lub (map pred ms) }
```

If there is no number greater than zero in *ms*, then the least upper bound is zero. If there is some number greater than zero, then we know that the least upper bound must start with a successor constructor. We can emit this constructor, and continue by computing the least upper bound of a variant of *ms* where every positive number has been decreased by one.

Given the results above we can conclude that *WLPO* is logically equivalent to both of the computability statements for maximum heap usages and least upper bounds:

```
WLPO ⇔ (∀ p → ∃ λ n → Maximum-heap-usage p n)
WLPO ⇔ (∀ ms → ∃ λ n → LUB ms n)
```

7 AN OPTIMISER

In this section I will define an optimiser for the simple allocation language, and show that this optimiser works as it should. The optimiser takes subsequences consisting of allocate, allocate and deallocate, and replaces them with allocate:

```
optimise : ∀ {i} → Program ∞ → Program i
optimise     []                 = []
optimise     (deallocate :: p) = deallocate :: λ { .force → optimise (p .force) }
optimise {i} (allocate   :: p) = optimise₁ (p .force)
  module Optimise where
  default : Program i
  default = allocate :: λ { .force → optimise (p .force) }

  optimise₂ : Program ∞ → Program i
  optimise₂ (deallocate :: p′′) = allocate :: λ { .force → optimise (p′′ .force) }
  optimise₂ _                   = default

  optimise₁ : Program ∞ → Program i
  optimise₁ (allocate :: p′) = optimise₂ (p′ .force)
  optimise₁ _                = default
```

The named **where** clause makes it possible to refer to the local definitions in the proof below. For instance, *Optimise.optimise₂* refers to the local definition *optimise₂*. Note that *Optimise.optimise₂* takes two extra arguments, one implicit and one explicit, corresponding to the bound variables *i* and *p* from the clause *optimise* {*i*} (allocate :: *p*).

One might think that it would be better to replace subsequences consisting of allocate and deallocate with nothing, but if such an optimiser could be implemented, then it would not produce any output at all (not even []) for the program *constant-space*.

The optimiser improves the space complexity of at least one program, because the semantics of *optimise constant-space₂* matches that of *constant-space*:

```
[ i ] ⟦ optimise constant-space₂ ⟧ n ∼L ⟦ constant-space ⟧ n
```

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

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):

$$\mathit{Maximum\text{-}heap\text{-}usage}\ (\mathit{optimise}\ p)\ m \to \mathit{Maximum\text{-}heap\text{-}usage}\ p\ n \to [\ i\ ]\ m \leq n$$

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:

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

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 following statement implies that the optimiser never increases a program's maximum heap usage:

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

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):

$$\begin{aligned}&[\ i\ ]\ [] \lesssim ns\\&[\ i\ ]\ ms \lesssim ns\ .\mathsf{force} \to [\ i\ ]\ ms \lesssim n :: ns\\&\mathit{Bounded}\ m\ ns \to [\ i\ ]\ ms\ .\mathsf{force} \lesssim' ns \to [\ i\ ]\ m :: ms \lesssim ns\\&[\ i\ ]\ ms\ .\mathsf{force} \lesssim' ns\ .\mathsf{force} \to [\ i\ ]\ m :: ms \lesssim m :: ns\end{aligned}$$

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:

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

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 leads to a contradiction:

$$\neg\ (\forall\ \{i\ ms\ ns\ n\} \to [\ i\ ]\ ms \lesssim' ns\ .\mathsf{force} \to [\ i\ ]\ ms \lesssim n :: ns)$$

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):

$$\neg\ (\forall\ \{i\ ms\ ns\ os\} \to [\ \infty\ ]\ ms \lesssim ns \to [\ i\ ]\ ns \lesssim os \to [\ i\ ]\ ms \lesssim os)$$

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:

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

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 corresponding to the first clause of $\mathit{Optimise.optimise}_2$. I focus on that one. The goal is to prove the following statement (in the presence of some assumptions that are not needed):

$$[\ i\ ]\ [\![\ \mathit{Optimise.optimise}_2\ p\ (\mathsf{deallocate} :: p'')\ ]\!]\ h \lesssim h ::' 1 + h ::' [\![\ \mathsf{deallocate} :: p''\ ]\!]\ (2 + h)$$

The proof proceeds in the following way:

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

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 explanations inserted for each of the steps.

8 A SIMPLE LAMBDA CALCULUS

The language treated in the previous sections was admittedly very minimal. Let me now instead treat a somewhat more interesting language. The language that I present is 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:

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

As mentioned above I use a well-scoped representation: 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 mutually in the following way:

```
mutual

  Env : ℕ → Set
  Env n = Vec Value n

  data Value : Set where
    lam : ∀ {n} → Tm (suc n) → Env n → Value
    con : Bool → Value
```

A value of type *Env n* is a list of values of length n. The value lam t ρ 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 I combine the delay monad with the maybe monad transformer:

```
Delay-crash : Set → Size → Set
Delay-crash A i = Delay (Maybe A) i
```

In this section return, _>>=_ and **do** notation refers to this combined monad. An immediate crash (as opposed to one that happens later) is defined in the following way:

```
crash : ∀ {A i} → Delay-crash A i
crash = now nothing
```

The interpreter is parametrised by a definition of a term with at most one free variable for every name:

```
def : Name → 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:

```
_•_ : ∀ {i} → Value → Value → Delay-crash Value i
lam t ρ • v₂ = later λ { .force → ⟦ t ⟧ (v₂ :: ρ) }
con _   • _  = crash
```

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

```
⟦_⟧ : ∀ {i n} → Tm n → Env n → Delay-crash Value i
⟦ var x ⟧      ρ = return (index x ρ)
⟦ lam t ⟧      ρ = return (lam t ρ)
⟦ t₁ · t₂ ⟧    ρ = do v₁ ← ⟦ t₁ ⟧ ρ
                      v₂ ← ⟦ t₂ ⟧ ρ
                      v₁ • v₂
⟦ call f t ⟧   ρ = do v ← ⟦ t ⟧ ρ
                      lam (def f) [] • v
⟦ con b ⟧      ρ = return (con b)
⟦ if t₁ t₂ t₃ ⟧ ρ = do v₁ ← ⟦ t₁ ⟧ ρ
                       ⟦if⟧ v₁ t₂ t₃ ρ
```

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 _•_ to apply the first value to the second. Note that this means that the semantics is a call-by-value semantics. Calls to named functions are evaluated similarly to applications. Boolean literals are returned directly. If expressions are interpreted by interpreting the scrutinee, and then an auxiliary function determines how to proceed:

```
⟦if⟧ : ∀ {i n} → Value → Tm n → Tm n → Env n → Delay-crash Value i
⟦if⟧ (lam _ _)  _ _ _ = crash
⟦if⟧ (con true)  t₂ t₃ ρ = ⟦ t₂ ⟧ ρ
⟦if⟧ (con false) t₂ t₃ ρ = ⟦ t₃ ⟧ ρ
```

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 _•_, and otherwise the size argument is kept unchanged.

9 A VIRTUAL MACHINE WITH TAIL CALLS

This section presents a virtual machine, and Section 10 a compiler from the terms of the previous section to this virtual machine.

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 monad is some kind of combination of *Delay-crash* and a writer monad producing colists:

```
mutual

  data Delay-crash-trace (A B : Set) (i : Size) : Set where
    now   : B → Delay-crash-trace A B i
    crash : Delay-crash-trace A B i
    later : A → Delay-crash-trace′ A B i → Delay-crash-trace A B i
    tell  : A → Delay-crash-trace  A B i → Delay-crash-trace A B i
```

```
record Delay-crash-trace′ (A B : Set) (i : Size) : Set where
  coinductive
  field force : {j : Size< i} → Delay-crash-trace A B j
```

The type *Delay-crash-trace A B i* stands for computations that, in addition to perhaps terminating with a value of type *B*, also produce traces (colists) containing values of type *A*:

```
trace : ∀ {A B i} → Delay-crash-trace A B i → Colist A i
trace (now x)     = []
trace crash       = []
trace (later x m) = x :: λ { .force → trace (m .force) }
trace (tell x m)  = x :: λ { .force → 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. The traces can be removed from the computations, leading to trace-less computations of type *Delay-crash B i*:

```
delay-crash : ∀ {A B i} → Delay-crash-trace A B i → Delay-crash B i
delay-crash (now x)     = now (just x)
delay-crash crash       = now nothing
delay-crash (later x m) = later λ { .force → delay-crash (m .force) }
delay-crash (tell x m)  = delay-crash m
```

The *delay-crash* function removes the first argument from later constructors, and tell constructors are removed entirely.

Delay-crash-trace can be turned into a monad in much the same way as the delay monad. If (strong) bisimilarity is defined for *Delay-crash-trace* in the obvious way, then it is easy to prove that the monad laws hold up to bisimilarity (see the accompanying code for details).

The virtual machine (VM) uses the following instruction set:

```
mutual

  data Instr (n : ℕ) : Set where
    var  : Fin n → Instr n
    clo  : Code (suc n) → Instr n
    app  : Instr n
    ret  : Instr n
    cal  : Name → Instr n
    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), and with some changes to support booleans and calls to named functions. *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*. The individual instructions are explained when the semantics is given further down.

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 : ∀ {n} → Code n → VM-Env n → Stack-element

Stack : Set
Stack = List Stack-element
```

The VM's state consists of a piece of code (with at most n free variables for some $n : \mathbb{N}$), a stack, and an environment of size n:

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

The VM's semantics is parametrised by a function mapping names to pieces of code with at most one free variable:

```
def : Name → Code 1
```

As mentioned above the 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
  cont  : State → Result
  done  : VM-Value → Result
  crash : Result
```

The following function is similar to the single-step relation of a typical small-step semantics:

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

The different instructions are interpreted in the following way:

- var x (variable): Push the value corresponding to x in the environment onto the stack.
- clo c' (closure): Push a closure formed from c' and the environment onto the stack.
- app (application): If two values are on top of the stack, the second of which is a closure, pop these values, push a return frame consisting of the current code and environment onto the stack, form a new environment from the first value and the closure's environment, and continue with the closure's code.
- ret (return): If a value and a return frame are on top of the stack, pop the return frame, and continue with the return frame's code and environment.

- cal f (call): If a value is on top of the stack, pop this value and put it into a new environment with a single binding. Push a return frame with the remainder of the code and the old environment onto the stack, and continue with *def* f.
- tcl f (tail call): Like cal f, but do not push a return frame onto the stack. The idea is that there should already be a return frame on the stack that can be reused.
- con b (constructor): Push the boolean b onto the stack.
- bra c_1 c_2 (branch): If there is a boolean on top of the stack, pop this boolean and continue with either c_1 or c_2, depending on the boolean, followed by the remainder of the old code.

If the machine encounters a state with an empty piece of code and an empty environment, and a single value on the stack, then the machine terminates with this value. If no rule matches, then the machine crashes.

The full semantics is defined by iterating *step* corecursively:

```
mutual

  exec⁺ : ∀ {i} → State → Delay-crash-trace State VM-Value i
  exec⁺ s = later s λ { .force → exec⁺′ (step s) }

  exec⁺′ : ∀ {i} → Result → Delay-crash-trace State VM-Value i
  exec⁺′ (cont s)  = exec⁺ s
  exec⁺′ (done v) = now v
  exec⁺′ crash     = crash
```

Note that the resulting trace contains all the encountered states. A semantics without states can be obtained by using *delay-crash*:

```
exec : ∀ {i} → State → Delay-crash VM-Value i
exec = delay-crash ∘ exec⁺
```

The following function returns the length of a state's stack component:

```
stack-size : State → ℕ
stack-size ⟨ _ , s , _ ⟩ = length s
```

This function can be used to construct a trace of all the encountered stack sizes:

```
stack-sizes : ∀ {i} → State → Colist ℕ i
stack-sizes = map stack-size ∘ trace ∘ exec⁺
```

10 A CORRECT COMPILER

Let us now see how one can construct a provably correct compiler from the λ-calculus in Section 8 to the language of the virtual machine in Section 9. 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:

```
In-tail-context : Set
In-tail-context = Bool
```

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. Note the use of a code continuation:

```
mutual

  comp : ∀ {n} → 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)

  comp-body : ∀ {n} → Tm (suc n) → Code (suc n)
  comp-body t = comp true t (ret :: [])
```

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 definition for each name:

```
  def : Name → Tm 1
```

The following function compiles such definitions:

```
  comp-name : Name → Code 1
  comp-name f = comp-body (def f)
```

When compiler correctness is stated below the function *comp-name* is used to provide an implementation of *def* for the virtual machine. For the purposes of stating compiler correctness I also define functions that compile environments and values:

```
mutual

  comp-env : ∀ {n} → Env n → VM-Env n
  comp-env []      = []
  comp-env (v :: ρ) = comp-val v :: comp-env ρ

  comp-val : Value → VM-Value
  comp-val (lam t ρ) = lam (comp-body t) (comp-env ρ)
  comp-val (con b)   = con b
```

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

```
  comp₀ : Tm 0 → Code 0
  comp₀ t = comp false t []
```

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]):

```
[ ∞ ] exec ⟨ comp₀ t , [] , [] ⟩ ≈D ⟦ t ⟧ [] >>= λ v → return (comp-val v)
```

This statement says that the result of running the code obtained by compiling the program t on the virtual machine (with an empty stack) is weakly bisimilar to the semantics of t, 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:

```
Stack-OK i k tc s →
Cont-OK i ⟨ c , s , comp-env ρ ⟩ k →
[ i ] exec ⟨ comp tc t c , s , comp-env ρ ⟩ ≈D ⟦ t ⟧ ρ >>= k
```

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

```
Cont-OK : Size → State → (Value → Delay-crash VM-Value ∞) → Set
Cont-OK i ⟨ c , s , ρ ⟩ k = ∀ v → [ i ] exec ⟨ c , val (comp-val v) :: s , ρ ⟩ ≈D k v
```

The first assumption is targeted at tail-calls:

```
data Stack-OK (i : Size) (k : Value → Delay-crash VM-Value ∞) :
                In-tail-context → Stack → Set where
  unrestricted : ∀ {s} → Stack-OK i k false s
  restricted   : ∀ {s n} {c : Code n} {ρ : VM-Env n} →
                 Cont-OK i ⟨ c , s , ρ ⟩ k → Stack-OK i k true (ret c ρ :: s)
```

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.

11 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:

```
[_, _]_•I_ : ∀ {i} → (ℕ → ℕ) → (ℕ → ℕ) → Value → Value →
             Delay-crash-trace (ℕ → ℕ) Value i
[ f₁ , f₂ ] lam t₁ ρ •I v₂ = later f₁ λ { .force → do
                                 v ← ⟦ t₁ ⟧I (v₂ :: ρ) true
                                 tell f₂ (return v) }
[ _ , _ ] con _    •I _    = crash
```

The function is used in different ways, so it is parametrised by two stack 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 instrumented main function is defined in the following way. Note that tail context information is passed around:

```
⟦_⟧I : ∀ {i n} → Tm n → Env n → In-tail-context → Delay-crash-trace (ℕ → ℕ) Value i
⟦ var x ⟧I    ρ _ = tell suc (return (index x ρ))
⟦ lam t ⟧I    ρ _ = tell suc (return (lam t ρ))
```

$$
\begin{aligned}
&[\![\ t_1 \cdot t_2\ ]\!]_{\mathrm{I}} \quad \rho\ \_ = \mathbf{do}\ v_1 \leftarrow [\![\ t_1\ ]\!]_{\mathrm{I}}\ \rho\ \mathsf{false}\\
&\qquad\qquad\qquad\qquad\ \ v_2 \leftarrow [\![\ t_2\ ]\!]_{\mathrm{I}}\ \rho\ \mathsf{false}\\
&\qquad\qquad\qquad\qquad\ \ [\ \mathit{pred}\ ,\ \mathit{pred}\ ]\ v_1 \bullet_{\mathrm{I}} v_2\\
&[\![\ \mathsf{call}\ f\ t\ ]\!]_{\mathrm{I}} \quad \rho\ \mathit{tc} = \mathbf{do}\ v \leftarrow [\![\ t\ ]\!]_{\mathrm{I}}\ \rho\ \mathsf{false}\\
&\qquad\qquad\qquad\qquad\ \ [\ \delta_1\ \mathit{tc}\ ,\ \delta_2\ \mathit{tc}\ ]\ \mathsf{lam}\ (\mathit{def}\ f)\ [\,] \bullet_{\mathrm{I}} v\\
&[\![\ \mathsf{con}\ b\ ]\!]_{\mathrm{I}} \quad \rho\ \_ = \mathsf{tell}\ \mathsf{suc}\ (\mathsf{return}\ (\mathsf{con}\ b))\\
&[\![\ \mathsf{if}\ t_1\ t_2\ t_3\ ]\!]_{\mathrm{I}}\ \rho\ \mathit{tc} = \mathbf{do}\ v_1 \leftarrow [\![\ t_1\ ]\!]_{\mathrm{I}}\ \rho\ \mathsf{false}\\
&\qquad\qquad\qquad\qquad\ \ [\![\mathit{if}]\!]_{\mathrm{I}}\ v_1\ t_2\ t_3\ \rho\ \mathit{tc}
\end{aligned}
$$

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 constructor, and by one for the ret constructor. 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):

$$
\begin{aligned}
&\delta_1 : \mathit{In\text{-}tail\text{-}context} \to \mathbb{N} \to \mathbb{N}\\
&\delta_1\ \mathsf{true}\ = \mathit{pred}\\
&\delta_1\ \mathsf{false} = \mathit{id}
\end{aligned}
$$

$$
\begin{aligned}
&\delta_2 : \mathit{In\text{-}tail\text{-}context} \to \mathbb{N} \to \mathbb{N}\\
&\delta_2\ \mathsf{true}\ = \mathit{id}\\
&\delta_2\ \mathsf{false} = \mathit{pred}
\end{aligned}
$$

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

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

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):

$$
\begin{aligned}
&\mathit{numbers} : \forall\ \{A\ i\} \to \mathit{Delay\text{-}crash\text{-}trace}\ (\mathbb{N} \to \mathbb{N})\ A\ i \to \mathbb{N} \to \mathit{Colist}\ \mathbb{N}\ i\\
&\mathit{numbers}\ x\ n = \mathit{scanl}\ (\lambda\ m\ f \to f\ m)\ n\ (\mathit{trace}\ x)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{stack\text{-}sizes}_{\mathrm{I}} : \forall\ \{i\} \to \mathit{Tm}\ 0 \to \mathit{Colist}\ \mathbb{N}\ i\\
&\mathit{stack\text{-}sizes}_{\mathrm{I}}\ t = \mathit{numbers}\ ([\![\ t\ ]\!]_{\mathrm{I}}\ [\,]\ \mathsf{false})\ 0
\end{aligned}
$$

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 8:

$$[\ i\ ]\ \mathit{delay\text{-}crash}\ ([\![\ t\ ]\!]_{\mathrm{I}}\ \rho\ \mathit{tc}) \sim_{\mathrm{D}} [\![\ t\ ]\!]\ \rho$$

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:

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

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:

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

The relation used here states that the two colists are bounded by each other, and is defined using the "bounded by" relation from Section 7:

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

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

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

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 7. 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:

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

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

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

The relation is an equivalence relation. I provide three transitivity-like results, two of which preserve the size of one argument:

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

The correctness proof has a similar structure to the correctness proof given in Section 10. Here is the type of the key lemma:

$$\mathit{Stack\text{-}OK}\ i\ k\ tc\ s \to$$
$$\mathit{Cont\text{-}OK}\ i\ \langle\ c\ ,\ s\ ,\ \mathit{comp\text{-}env}\ \rho\ \rangle\ k \to$$
$$[\ i\ ]\ \mathit{stack\text{-}sizes}\ \langle\ \mathit{comp}\ tc\ t\ c\ ,\ s\ ,\ \mathit{comp\text{-}env}\ \rho\ \rangle \approx \mathit{numbers}\ (\llbracket\ t\ \rrbracket_{\mathrm{I}}\ \rho\ tc \gg\!= k)\ (\mathit{length}\ s)$$

The *Cont-OK* and *Stack-OK* predicates are defined in the following way:

```
Cont-OK : Size → State → (Value → Delay-crash-trace (ℕ → ℕ) VM-Value ∞) → Set
Cont-OK i ⟨ c , s , ρ ⟩ k =
  ∀ v → [ i ] stack-sizes ⟨ c , val (comp-val v) :: s , ρ ⟩ ≈ numbers (k v) (1 + length s)

data Stack-OK (i : Size) (k : Value → Delay-crash-trace (ℕ → ℕ) VM-Value ∞) :
                In-tail-context → Stack → Set where
  unrestricted : ∀ {s} → Stack-OK i k false s
  restricted   : ∀ {s n} {c : Code n} {ρ : VM-Env n} →
                 (∀ v → [ i ] 2 + length s ::′ stack-sizes ⟨ c , val (comp-val v) :: s , ρ ⟩ ≈
                              numbers (k v) (2 + length s)) →
                 Stack-OK i k true (ret c ρ :: s)
```

For full details of the correctness proof, see the accompanying code.

12 TWO EXAMPLES

This section contains two examples of non-terminating programs, one that runs in bounded stack space, and one that does not. I prove this without reasoning directly about the compiler or virtual machine, instead using the instrumented semantics.

The first example is the term $(\lambda x.x\ x)\ (\lambda x.x\ x)$, implemented in the following way (where fzero : *Fin* 1 stands for zero):

```
ω : Tm 0
ω = lam (var fzero · var fzero)

Ω : Tm 0
Ω = ω · ω
```

It is easy to show that this program is non-terminating:

```
Ω-loops : ∀ {i} → [ i ] ⟦ Ω ⟧ [] ∼D never
Ω-loops = later λ { .force → Ω-loops }
```

The stack-sizes encountered when interpreting Ω are captured by the colist *Ω-sizes* 0 (which expands to 0, 1, 2, 1, 2, 3, 2, 3, 4…):

```
Ω-sizes : ∀ {i} → ℕ → Colist ℕ i
Ω-sizes n = n ::′ 1 + n ::′ 2 + n :: λ { .force → Ω-sizes (1 + n) }
```

This can be proved straightforwardly using corecursion:

```
[ i ] stack-sizesI Ω ∼L Ω-sizes 0
```

Note that this is a statement about the instrumented semantics, not about the virtual machine. It should come as no surprise that the least upper bound of *Ω-sizes* 0 is infinity:

```
LUB (Ω-sizes 0) infinity
```

Thus Ω does not run in bounded stack space:

```
LUB (stack-sizesI Ω) infinity
```

The second example is a non-terminating program that gets compiled into code using a tail call. Here the *Name* parameter is the unit type ⊤, with a single inhabitant tt. The only definition is a term that ignores its argument and calls itself repeatedly:

```
def : ⊤ → Tm 1
def tt = call tt (con true)
```

In the remainder of this section the interpreter and the instrumented interpreter are instantiated with this definition of *def*. The following top-level program is used to get things going:

```
go : Tm 0
go = call tt (con true)
```

It is easy to verify that *go* does not terminate:

```
go-loops : ∀ {i} → [ i ] ⟦ go ⟧ [] ∼D never
go-loops = later λ { .force → go-loops }
```

The stack sizes encountered when interpreting *go* are captured by *go-sizes* (which expands to 0, 1, 1, 2, 1, 2, 1, 2…):

```
loop-sizes : ∀ {i} → Colist ℕ i
loop-sizes = 1 ::′ 2 :: λ { .force → loop-sizes }

go-sizes : Colist ℕ ∞
go-sizes = 0 ::′ 1 ::′ loop-sizes
```

The corecursive proof is straightforward and omitted:

```
[ i ] stack-sizesI go ∼L go-sizes
```

The least upper bound of *go-sizes* is two:

```
LUB go-sizes ⌜ 2 ⌝
```

Thus *go* runs in bounded stack space:

```
LUB (stack-sizesI go) ⌜ 2 ⌝
```

13 RELATED WORK

I have already mentioned some parts of the work of Leroy and Grall [2009] above. They also provide an example of how one might get unintended consequences when defining coinductive big-step semantics, showing that if one interprets the inductive inference rules of a simple big-step semantics coinductively, then the resulting relation is not what one might have naively expected.

Nakata and Uustalu [2009] define trace-based, functional and relational, small-step and big-step semantics for a while language in Coq. Traces are non-empty, potentially infinite lists of states (mappings of variables to integers), and the functional semantics are defined using corecursion and structural recursion. The use of traces makes it possible to distinguish between different non-terminating computations (but memory usage is not tracked).

Nakata and Uustalu provide several examples of how alternative definitions of their coinductive big-step relational semantics can lead to subtle problems. Danielsson [2012] sees Nakata and Uustalu's *relational* big-step semantics as technical and brittle, noting that the problems discussed by Nakata and Uustalu are avoided in the definition of the *functional* big-step semantics, which (because Coq is a constructive type theory) has to be a productive function from terms and initial states to traces.

Danielsson [2012] and Owens et al. [2016] argue that total definitional interpreters (in the former case defined using the delay monad, in the latter case using step counters) have some desirable properties compared to relational big-step semantics, especially if one is interested in

giving semantics to programs that can crash or fail to terminate, and not only to programs that terminate successfully.

Ancona et al. [2017a, 2018] have developed an approach to inference systems which is a kind of mixture of induction and coinduction: the inference rules are read coinductively, with the condition that every node in a proof tree has to be provable *inductively*, using the regular inference rules plus some extra rules called corules. They provide some examples of trace-based semantics defined in their framework [2018]. These semantics make it possible to distinguish between different non-terminating programs. In private communication they have also showed me a big-step semantics for a kind of λ-calculus with references. The judgements of this semantics include information about the maximum heap size (potentially ∞) required for a (potentially non-terminating) computation.

My observation about this approach is that I find it hard to understand the presented semantics. I can see that the corules work in certain examples, but it is harder to see that the semantics matches the intentions. Are there enough corules? Too many? In my opinion an advantage of total definitional interpreters is that they provide a format which makes it harder to make mistakes: you are forced to explain exactly what the semantics is for every piece of abstract syntax. Ancona et al. [2018] state that they do not claim that their approach is easier than using a more standard approach (a labelled small-step semantics), but they claim that it allows more direct reasoning.

The CerCo project [Amadio et al. 2014] included the development of an optimising compiler from a subset of C to machine code. 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 my instrumented interpreter 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.

The CerCo compiler seems to have been partly verified in Matita (I found a number of axioms in the source code). Campbell et al. [2013] state that the main correctness theorem does not guarantee directly that the compiled program has the same result or termination behaviour as the source program, but it is suggested that a stronger result could have been proved with more effort. Other than this partial proof (in a setting which is much more ambitious than my little case study) I am not aware of any other machine-checked compiler correctness proofs that are applicable to non-terminating programs and involve guarantees about time or space complexity. However, there are a number of machine-checked compiler correctness proofs that are applicable to non-terminating programs, and I would not be surprised if there are also machine-checked proofs showing that non-terminating programs satisfy properties related to resource consumption.

14 CONCLUSIONS

I have shown that one can give a semantics, using a total definitional interpreter, to at least one programming language in such a way that non-terminating programs that require different amounts of stack space can be distinguished. I also defined a compiler that preserves the stack space guarantees and proved that it is correct.

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

ACKNOWLEDGEMENTS

I would like to thank Davide Ancona, Francesco Dagnino and Elena Zucca for providing the starting point of this work. I would also like to thank Andreas Abel and Ulf Norell for helping me improve a result that is presented (in improved form) in Section 5, and Robin Adams and Ulf Norell for providing other 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. 2017a. Generalizing Inference Systems by Coaxioms. In *Programming Languages and Systems, 26th European Symposium on Programming, ESOP 2017*. https://doi.org/10.1007/978-3-662-54434-1_2

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

Davide Ancona, Francesco Dagnino, and Elena Zucca. 2018. Modeling Infinite Behaviour by Corules. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018*. https://doi.org/10.4230/LIPIcs.ECOOP.2018.21

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

Brian Campbell, Ilias Garnier, James McKinna, and Ian Stark. 2013. *Project FP7-ICT-2009-C-243881 CerCo, Report n. D3.4 Front-end Correctness Proofs, Version 1.0*. Technical Report. Retrieved 2018-07-11 from https://cordis.europa.eu/docs/projects/cnect/1/243881/080/deliverables/001-D34.pdf

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

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 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