Thesis for the degree of Doctor of Philosophy

# Functional Program Correctness Through Types

**Nils Anders Danielsson**

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2007

Functional Program Correctness Through Types
Nils Anders Danielsson
ISBN 978-91-7385-034-6

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 2715
ISSN 0346-718X

Technical report 37D
Department of Computer Science and Engineering
Research groups: Programming Logic and Functional Programming

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed by Chalmers Reproservice
Göteborg, Sweden 2007

Functional Program Correctness Through Types
Nils Anders Danielsson
Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University

## Abstract

This thesis addresses the problem of avoiding errors in functional programs. The thesis has three parts, discussing different aspects of program correctness, with the unifying theme that types are an integral part of the methods used to establish correctness.

The first part validates a common, but not obviously correct, method for reasoning about functional programs. In this method, dubbed "fast and loose reasoning", programs written in a language with non-terminating functions are treated as if they were written in a total language. It is shown that fast and loose reasoning is sound when the programs are written in a given total subset of the language, and the resulting properties are translated back to the partial setting using certain partial equivalence relations which capture the concept of totality.

The second part discusses a method for ensuring that functions meet specified time bounds. The method is aimed at implementations of purely functional data structures, which often make essential use of lazy evaluation to ensure good time complexity in the presence of persistence. The associated complexity analyses are often complicated and hence error-prone, but by annotating the type of every function with its time complexity, using an annotated monad to combine time complexities of subexpressions, it is ensured that no details are forgotten.

The last part of the thesis is a case study in programming with strong invariants enforced by the type system. A dependently typed object language is represented in the meta language, which is also dependently typed, in such a way that it is impossible to form ill-typed terms. An interpreter is then implemented for the object language by using normalisation by evaluation. By virtue of the strong types used this implementation is a proof that every term has a normal form, and hence normalisation is proved. This also seems to be the first formal account of normalisation by evaluation for a dependently typed language.

**Keywords:** Program correctness, total languages, partial languages, time complexity, lazy evaluation, dependent types, strong invariants, well-typed syntax, normalisation by evaluation.

This thesis is based on the work contained in the following papers:

- Nils Anders Danielsson, Jeremy Gibbons, John Hughes and Patrik Jansson. *Fast and Loose Reasoning is Morally Correct.* In POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 206–217, 2006.
- Nils Anders Danielsson. *Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures.* Accepted for publication in POPL '08: Conference record of the 35th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2008.
- Nils Anders Danielsson. *A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family.* In Types for Proofs and Programs, International Workshop, TYPES 2006, Revised Selected Papers, volume 4502 of LNCS, pages 93–109, 2007.

# Contents

# Acknowledgements

I would like to thank everyone who has helped me on the path to this thesis.

A big thanks goes to my supervisor, Patrik Jansson, who has supported me throughout my PhD studies; and to my coauthors and colleagues, locally and internationally, who have provided a body of research to build on, participated in interesting discussions, and given me welcome feedback.

I would also like to thank my opponent, Martin Hofmann; the members of my grading committee, David Sands, Anton Setzer and Doaitse Swierstra; and Thierry Coquand and Devdatt Dubhashi, members of my advisory committee, for their feedback and time.

My current and former office mates and other fellow PhD students also deserve thanks, for providing me with entertaining discussions and distractions. More generally I appreciate the friendly working atmosphere provided by all the members of the CSE department.

Finally I am grateful to my parents, who have always supported me, and to Maria, for all her love and understanding.

# Chapter 1

# Introduction

Computer programs should not contain errors ("bugs"). Exactly what is meant by a bug may vary, and depending on the circumstances a certain amount of bugs may be acceptable to avoid overly high costs, but there are usually some properties which a given program must satisfy in order to be useful. For instance, the software running the financial transactions for a bank must not lose the customers' money; an aircraft running on autopilot must not crash; and an operating system must not crash either, at least not too often.

One part of the process of constructing a program is to ensure that these important properties are satisfied. There are a number of different ways of doing this, with various levels of associated assurance in the correctness of the software. These methods range from hoping that the programmers do everything right the first time, over well-engineered test suites, to various forms of mathematical proofs. This thesis focuses on the high-assurance end of this spectrum: proofs.

The thesis also focuses on a certain kind of programs: higher-order typed pure functional programs. Higher-order means that programs can be passed as arguments to other programs, and pure functional means that there are no side effects such as mutable variables or input/output. The reason for restricting this work to such languages is that I find that the structure given by types, the expressiveness of higher-order programs, and the absence of side-effects enable the writing of elegant and succinct programs, which it is relatively easy to reason about.

The thesis consists of three papers addressing different aspects of the theme outlined above. The rest of this chapter explains the three papers in more detail, giving some background information which is not included in the papers themselves.

## 1.1 Paper I

The first paper was published as:

> Nils Anders Danielsson, Jeremy Gibbons, John Hughes and Patrik Jansson. *Fast and Loose Reasoning is Morally Correct.* In POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 206–217, 2006.

This paper focuses on a method for reasoning about programs. In many programming languages it is possible to write programs which run forever (do not terminate). This phenomenon can be seen as a side-effect, and the possibility of non-termination often complicates reasoning about programs. The paper shows that a method of reasoning in which the possibility of non-termination is ignored is sometimes sound.

The following simple example will illustrate the kind of problems which can occur when non-termination is present. Let us define the data type of pairs, with first and second projection functions, as follows:

$$
\begin{array}{l}
\textbf{data}\ \_\times\_\ (a\ b : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\
\quad \langle \_, \_ \rangle : a \to b \to a \times b \\
\mathit{fst} : \forall \{a\ b\}.\ a \times b \to a \\
\mathit{fst}\ \langle x, y \rangle = x \\
\mathit{snd} : \forall \{a\ b\}.\ a \times b \to b \\
\mathit{snd}\ \langle x, y \rangle = y
\end{array}
$$

(This example, and the examples in Section 1.3, are defined in Agda (The Agda Team 2007; Norell 2007), with minor syntactic modifications to aid readability.) In a *total* language, i.e. a language in which all functions are guaranteed to terminate, the above definitions give rise to the following laws, assuming a suitably standard semantics:

$$
\begin{array}{l}
\forall x.\ \langle \mathit{fst}\ x, \mathit{snd}\ x \rangle = x \\
\forall x\ y.\ \mathit{fst}\ \ \langle x, y \rangle = x \\
\forall x\ y.\ \mathit{snd}\ \langle x, y \rangle = y
\end{array}
$$

Typical *partial* languages, i.e. languages allowing non-termination, only satisfy some of these laws; which laws depends on which evaluation strategy is used. If the pair constructor $\langle \_, \_ \rangle$ is *strict* (evaluates its arguments before returning a pair) the last two equations fail, since $x$ or $y$ could be non-terminating computations (typically written $\bot$):[1]

[1]This presentation assumes that $\mathit{fst}\ \bot = \bot = \mathit{snd}\ \bot$, which is usually the case.

$$\forall x \neq \bot.\ \mathit{fst}\ \ \langle x, \bot \rangle = \mathit{fst}\ \ \bot = \bot \neq x$$
$$\forall y \neq \bot.\ \mathit{snd}\ \langle \bot, y \rangle = \mathit{snd}\ \bot = \bot \neq y$$

On the other hand, if the pair constructor is *non-strict* (returns a pair without evaluating the arguments) the first law fails, since $\langle \mathit{fst}\ \bot, \mathit{snd}\ \bot \rangle = \langle \bot, \bot \rangle \neq \bot$.

Keeping track of bottoms is no fun, so in practice people often ignore the possibility of non-termination, hoping that the results so obtained are still valid. The paper dubs this proof method "fast and loose reasoning", and this form of reasoning is then justified (under certain conditions): it is explained how the results obtained using fast and loose reasoning can be reinterpreted, in a natural way, in a setting with possible non-termination. The basic idea is that, if the programs under consideration are total, the results will be valid as long as the inputs are also total.

This idea is very natural, so it did not come as a surprise that similar ideas had been published before. We brought these ideas to the attention of the functional programming community, though. A detailed discussion of related work is included in the paper.

Based on the description above it may seem that this work has little to do with the title of the thesis, Functional Program Correctness *Through Types*. However, types are used both to state and to prove the main results. More details are available in the paper itself.

The results are supported by a detailed set of proofs (Danielsson 2007). These proofs were developed in the traditional way: using pen and paper (or text editor and text file). This means that there are no guarantees whatsoever that the proofs are correct, unless you trust my theorem proving skills. The paper has been subject to peer review, but due to space considerations the paper only contains proof sketches.

If the proof had been mechanically checked, using a computer, it would have been easier to trust. The other papers in this thesis come with machine-checked proofs. Writing such proofs can require quite a lot of extra work, but I find that in many cases the higher level of trust, along with the assistance provided by the machine (you do not need to check all details manually), more than make up for this. Machine-checked proofs also have another advantage: you can come back to a proof at a later time and tinker with it, without worrying that there may be some hidden invariant which you have forgotten. If there is, and you break it, the computer will spot it for you.

## 1.2 Paper II

The second paper is:

> Nils Anders Danielsson. *Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures.* Accepted for publication in POPL '08: Conference record of the 35th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2008.

This paper is also concerned with improving the reliability of certain proofs, but this time the proofs do not concern the *values* obtained by evaluating a program (as in the previous paper), but rather the *time* required to evaluate a program to completion. One might believe that the execution time of a program can be established simply by measuring the time needed by a couple of test runs. While this method has its advantages, it also has some notable drawbacks:

- It is hard to draw qualitative conclusions about how the program will behave when other inputs (possibly much larger) are used.
- A program using the lazy evaluation strategy[2] (explained below) has the property that its execution time depends on how the results of parts of the program are used by other parts, making it even harder to draw qualitative conclusions about the running time of a certain part based only on measurements (Moss and Runciman 1998).

Note, however, that the notion of time can be hard to capture precisely in a formal setting, so instead an abstraction based on a notion of computation steps is often used; so also in this work.

The paper describes a simple library for establishing semiformal, mechanically checked time bounds for purely functional programs using lazy evaluation, in such a way that the bounds are valid no matter how the programs are used. (*Semi*formal since some annotations need to be inserted manually.) The focus is on a class of persistent data structures which often make essential use of lazy evaluation.

In general it can be quite expensive to mechanically prove the correctness of a program. However, a data structure is often optimised for speed, and accompanied by proofs (usually of the pen-and-paper kind) establishing the time bounds of its operations, so the step to a mechanically checked proof

[2] "Lazy evaluation" is an ambiguous expression. In this work it is taken as a synonym for call-by-need.

may not be very large. Furthermore, for data structures making essential use of lazy evaluation these proofs can be subtle, with lots of details to keep track of, so that mechanically checking the proofs is extra important.

### 1.2.1 Persistence

A data structure is *persistent* if previous versions of it can always be used, even if it is updated. (Note that an update of a persistent data structure typically yields a new data structure, possibly sharing some parts with the old one.) This property ensures that users of the data structure do not need to worry about aliasing: even if a data structure is passed to two unrelated parts of a program, and both parts modify the data structure, one part will not see the changes made by the other.

Since pure functional languages do not have mutable state *all* data structures are automatically persistent. Hence, from a correctness perspective, users of these languages have less to worry about. From an efficiency perspective the picture is less nice, though: partly because the absence of mutability can make it harder to construct efficient data structures, but also because different usage patterns can lead to different time complexities (execution times), making it harder to predict the running time of a program. For instance, as explained by Okasaki (1998), a common implementation of FIFO queues has the property that if it is used single-threadedly (i.e. if the output of one operation is always the input to the next), then every operation takes constant amortised time, whereas this can degenerate to linear time (for the tail function) for certain usage patterns.

Despite this a number of purely functional data structures exhibiting good performance no matter how they are used have been developed (see for instance Okasaki 1998; Kaplan and Tarjan 1999; Kaplan et al. 2000; Hinze and Paterson 2006). These data structures are often accompanied by rather subtle time complexity proofs. It is this kind of proof which the library described in the paper focuses on.

### 1.2.2 Laziness

One reason for the complexity of these complexity proofs is that many of the data structures make essential use of *laziness* in order to ensure good performance. Laziness is non-strictness with memoisation:

- Non-strictness means that an expression is only evaluated if its value is needed by another expression.

- Memoisation ensures that in an expression like **let** $x = e$ **in** $\langle x, x \rangle$ the expression $e$ is evaluated at most once, because when one use of $x$ is demanded the value of $e$ is memoised, and this value is returned immediately if the other use of $x$ is demanded.[3]

Memoisation provides a limited form of mutability, which implementations of data structures can take advantage of to improve the efficiency of operations. Laziness is a conservative optimisation of the more naive call-by-name evaluation strategy, so reasoning about functional properties (i.e. properties about the values of expressions) is unaffected. However, from the standpoint of time complexity proofs, the memoisation inherent in laziness introduces a side effect: the time it takes to evaluate an expression depends on other parts of the program (as mentioned above). Hence it is not surprising that time complexity proofs for lazy programs can be quite subtle.

### 1.2.3 Analysing lazy time complexity

A number of approaches to analysing lazy time complexity have been developed (Wadler 1988; Bjerner 1989; Bjerner and Holmström 1989; Sands 1995; Okasaki 1998; Moran and Sands 1999). Many of them are general, but have been described as complicated to use in practice (Okasaki 1998). The main technique in use, at least for analysing purely functional data structures, is probably Okasaki's banker's method.

This method is used to derive *amortised* time bounds. An amortised bound is not a real time bound, but something assigned by an analysis. If the operations being analysed have been assigned amortised upper time bounds $n_1$, $n_2$, etc., then this means that, if the (not necessarily distinct) operations $o_1$, $o_2$, …, $o_k$ are executed, the total number of steps actually taken by these operations is guaranteed to be at most $\sum_{i=1}^{k} n_{o_i}$. Note that this allows operation $i$ to take more than $n_i$ steps, as long as other operations take fewer steps than their time bounds allow. Amortisation can provide a more flexible setting for analysing time complexity, if the number of steps taken by any particular operation is not important, but only the total number of steps for a collection of operations.

Now, the basic idea of the banker's method is that when an operation creates an expression which is not immediately evaluated (a *suspension*), then a number of *debits* corresponding to the cost of evaluating this suspension is associated with it (in the analysis, not in the code). If a suspension contains further suspensions, then the corresponding debits can be associated with

[3]This is a simplified description, which assumes that all of $e$ is demanded at once; the same idea is still valid in the general case, though.

these suspensions instead of the outer one. Operations can then *discharge* (or pay off) debits, also in the analysis, and this has to be done in such a way that all debits associated with a suspension have been discharged before the suspension is *forced* (evaluated).

To simplify things a restriction is added: a debit only counts as discharged if it was paid off by the current operation or a logical predecessor of it (i.e. an operation which provided input, directly or indirectly, to the current operation). If two unrelated operations discharge the same debit, then the analysis overestimates the cost of them. This ensures that there is no need to reason about unrelated (and possibly unknown) parts of the program.

The amortised cost of an operation is then defined as the number of debits discharged, plus the work performed immediately (that which is not due to forcing suspensions created by other operations). The time needed to force a suspension which has already been paid off is not included in this cost, but this is OK, for two reasons:

1. The suspension has been paid off, and this cost covers the time needed to force the suspension once.

2. The first time the suspension is forced the resulting value will be memoised, so that any other operations which attempt to force the suspension only need to look up this value in order to make use of it.

Based on these observations the banker's method can be proved correct. The trick to analysing essential uses of laziness (where the side effect of memoisation is important) is then to set things up so that a suspension which will be used by several unrelated operations is paid off by common predecessors of these operations.

### 1.2.4 The library

A simplified but less general (although still useful, as demonstrated by some case studies) version of the banker's method underlies the library described in the paper. The library is introduced through a series of illustrative examples, which I refrain from repeating here. Suffice it to say that the library is based on annotating all functions with ticks, representing evaluation steps, and using an "annotated monad" to combine time complexities of subcomputations. As a result the types of functions become annotated with their time complexities, for instance as follows:

$$\mathit{minimum} : \forall a\ n.\ \mathit{Seq}\ a\ (1 + n) \rightarrow \mathit{Thunk}\ (8 + 5 * n)\ a$$

(This says that finding the minimum element in a sequence of length $1 + n$ takes at most $8 + 5n$ steps, using a certain implementation of *minimum*.) Finally a *pay* function, corresponding to the paying off of debits in the banker's method, makes it possible to improve the time bounds by taking advantage of memoisation. To keep the library simple only certain uses of memoisation are handled, though.

## 1.3 Paper III

The third paper was published as:

> Nils Anders Danielsson. *A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family.* In Types for Proofs and Programs, International Workshop, TYPES 2006, Revised Selected Papers, volume 4502 of LNCS, pages 93–109, 2007.

This paper also explores a proof method: it describes a case study in programming with very strong invariants enforced by the type system, i.e. "proof by construction".

There are (at least) two distinct ways to define the type system of a programming language. The traditional one is to first define "raw terms" through a context-free syntax, and then restrict attention to well-typed terms through a system of inference rules. Another approach is to define the well-typed terms directly. I find the second approach more elegant, since it avoids the need to introduce uninteresting things (ill-typed raw terms), also known as junk.

There are many different type systems, with various properties. One class of type systems are those with *dependent* types (Nordström et al. 1990), in which types can depend on values. This means that types can "talk about" values in a natural way. One example of dependent types was provided in the previous section, where the result type of *minimum* depends on the value of one of its inputs ($n$). A related concept is that of *inductive families* (Dybjer 1994), which are data types which can depend on values in a certain way.

For simply typed languages the second approach above, the one without junk, is quite easy to implement in a dependently typed host language with inductive families. In fact, this approach was used to mechanise the correctness proof for the second paper. If the type system to be implemented has dependent types it is considerably harder, though. The third paper shows that it can be done.

The two methods for defining type systems mentioned above will now be illustrated through a small example. A simple expression language with two types (integers and booleans) and three expression forms (literals, addition of integers, and conjunction of booleans) is defined, along with a simple denotational semantics.

## 1.3.1 Raw terms

Let us first define the set of raw terms:

```
data Value : Set where
  intVal  : Int  → Value
  boolVal : Bool → Value
data RawTerm : Set where
  lit  : Value → RawTerm
  plus : RawTerm → RawTerm → RawTerm
  and  : RawTerm → RawTerm → RawTerm
```

(*RawTerm* is a data type; plus is a constructor taking two *RawTerm*s as arguments, and yielding a *RawTerm* as result; and so on.) Using this definition nonsensical terms like plus (boolVal true) (intVal 5), i.e. true + 5, can be formed. A partial denotational semantics for the language can still be defined. The *Maybe* type is used to signal failure:

```
data Maybe (a : Set) : Set where
  nothing : Maybe a
  just    : a → Maybe a
```

The semantics is then defined by recursion over the structure of terms, using pattern matching to decompose the input:

```
⟦_⟧ : RawTerm → Maybe Value
⟦lit v⟧ = just v
⟦plus e₁ e₂⟧ with ⟦e₁⟧ | ⟦e₂⟧
... | just (intVal i₁) | just (intVal i₂) = just (intVal (i₁ + i₂))
... | _                | _                = nothing
⟦and e₁ e₂⟧ with ⟦e₁⟧ | ⟦e₂⟧
... | just (boolVal b₁) | just (boolVal b₂) = just (boolVal (b₁ ∧ b₂))
... | _                 | _                 = nothing
```

Under this semantics the evaluation of an ill-typed term does not result in a value; the result is nothing. (The **with** syntax is used to pattern match on the result of an intermediate computation.)

Let us now define a type system on top of the raw terms. First the types, integers and booleans, are defined:

```
data Ty : Set where
  int  : Ty
  bool : Ty
```

The inference rules of the type system can then be encoded as follows, where $e :: \sigma$ means that $e$ has type $\sigma$:

```
data _::_ : RawTerm → Ty → Set where
  tyLitInt  : (i : Int)  → lit (intVal i)  :: int
  tyLitBool : (b : Bool) → lit (boolVal b) :: bool
  tyPlus    : ∀e₁ e₂.  e₁ :: int  → e₂ :: int  → plus e₁ e₂ :: int
  tyAnd     : ∀e₁ e₂.  e₁ :: bool → e₂ :: bool → and  e₁ e₂ :: bool
```

Here $\_ :: \_$ is an inductive family; the types in the family (for instance $\mathsf{lit}\ (\mathsf{intVal}\ i) :: \mathsf{int}$) depend on values ($\mathsf{lit}\ (\mathsf{intVal}\ i)$ and $\mathsf{int}$). The definition corresponds to the relation inductively generated by the following inference rules:

$$\frac{i : \mathit{Int}}{\mathsf{lit}\ (\mathsf{intVal}\ i) :: \mathsf{int}} \qquad \frac{b : \mathit{Bool}}{\mathsf{lit}\ (\mathsf{boolVal}\ b) :: \mathsf{bool}}$$

$$\frac{e_1, e_2 : \mathit{RawTerm} \qquad e_1 :: \mathsf{int} \qquad e_2 :: \mathsf{int}}{\mathsf{plus}\ e_1\ e_2 :: \mathsf{int}}$$

$$\frac{e_1, e_2 : \mathit{RawTerm} \qquad e_1 :: \mathsf{bool} \qquad e_2 :: \mathsf{bool}}{\mathsf{and}\ e_1\ e_2 :: \mathsf{bool}}$$

Is the semantics well-formed? The next step is to prove that evaluation of well-typed terms does not get stuck ("progress"), and that the resulting value has the same type as the term ("preservation"). First the type *IsOK* is defined; $\mathit{IsOK}\ \sigma\ m$ means that $m = \mathsf{just}\ v$ for some value $v$ of type $\sigma$:

```
data IsOK : Ty → Maybe Value → Set where
  isOK : ∀{σ v}. lit v :: σ → IsOK σ (just v)
```

(Here $\{\ldots\}$ declares that one or more arguments are *hidden*, which means that they do not need to be given explicitly if Agda can infer them automatically.) The proof of progress and preservation then proceeds by induction over the structure of typing derivations:

```
evalOK : ∀{e σ}. e :: σ → IsOK σ ⟦e⟧
evalOK (tyLitInt i)   = isOK (tyLitInt i)
evalOK (tyLitBool b) = isOK (tyLitBool b)
evalOK (tyPlus e1 e2 t1 t2) with ⟦e1⟧ | ⟦e2⟧ | evalOK t1 | evalOK t2
... | ._ | ._ | isOK (tyLitInt i1) | isOK (tyLitInt i2) =
  isOK (tyLitInt (i1 + i2))
evalOK (tyAnd e1 e2 t1 t2) with ⟦e1⟧ | ⟦e2⟧ | evalOK t1 | evalOK t2
... | ._ | ._ | isOK (tyLitBool b1) | isOK (tyLitBool b2) =
  isOK (tyLitBool (b1 ∧ b2))
```

(The recursive invocations of $⟦e_i⟧$ above ensure that occurrences of $⟦e_i⟧$ are abstracted from the goals; this is the key to enabling pattern matching on the result of *evalOK* $t_i$ (Norell 2007; McBride and McKinna 2004). Feel free to ignore this technical detail, though.)

The development above is starting to look complicated. An alternative is to define the denotational semantics by recursion on typing predicates instead of raw terms, obviating the need for *evalOK*:

```
Sem : Ty → Set
Sem int   = Int
Sem bool = Bool

⟦_⟧ : ∀{e σ}. e :: σ → Sem σ
⟦tyLitInt i⟧         = i
⟦tyLitBool b⟧       = b
⟦tyPlus _ _ t1 t2⟧ = ⟦t1⟧ + ⟦t2⟧
⟦tyAnd _ _ t1 t2⟧  = ⟦t1⟧ ∧ ⟦t2⟧
```

Note that the type of ⟦_⟧ is a statement of progress and preservation. Note also that the presence of type information makes it unnecessary to tag the resulting values; the function *Sem* interprets the *Ty* types in terms of meta-level types.

### 1.3.2 Well-typed terms

The second variant of ⟦_⟧ does not make explicit use of the raw terms, which are only mentioned in the type signature, not in the body of the code. In the *well-typed* approach to representing type systems the raw terms are not included at all. Terms are identified with typing derivations, and the *Term* type depends on a *Ty* type (to aid a fair comparison the *Ty* and *Sem* definitions from above are repeated here):

```
data Ty : Set where
  int  : Ty
  bool : Ty

Sem : Ty → Set
Sem int  = Int
Sem bool = Bool

data Term : Ty → Set where
  lit  : ∀{σ}. Sem σ → Term σ
  plus : Term int  → Term int  → Term int
  and  : Term bool → Term bool → Term bool

⟦_⟧ : ∀{σ}. Term σ → Sem σ
⟦lit v⟧       = v
⟦plus t₁ t₂⟧ = ⟦t₁⟧ + ⟦t₂⟧
⟦and  t₁ t₂⟧ = ⟦t₁⟧ ∧ ⟦t₂⟧
```

In the well-typed approach it is impossible to form ill-typed terms like plus (boolVal true) (intVal 5); the type checker does not allow it.

### 1.3.3 Contributions

The simplicity of the well-typed approach has led to a number of publications formalising various aspects of different essentially simply typed programming languages (Coquand and Dybjer 1997; Altenkirch and Reus 1999; Coquand 2002; Xi et al. 2003; Pašalić and Linger 2004; McBride and McKinna 2004; Altenkirch and Chapman 2006; McKinna and Wright; Chlipala 2007). However, the type systems of the programming languages in which these formalisations have been done are often of a stronger kind than the simply typed languages being formalised. Almost no one has formalised a *dependently* typed language using this approach. Before my paper the only exception (known to me) was Dybjer's work (1996) on formalising so-called categories with families, which can be seen as the basic framework of dependent types. Furthermore Dybjer reported (personal communication) that his formalisation was quite hard to work with, stating that it led to a "dependent types nightmare". This provided a clear incentive to work on this problem.

The contributions of the paper is to provide a well-typed formalisation of a dependently typed language, and to show that it can be used by proving normalisation for the language (i.e. that there is an evaluation strategy under which the evaluation of every term terminates). In addition, this normalisation proof turned out to be the first formalised account of *normalisation by evaluation* (a specific proof technique for normalisation proofs) for a dependently typed language. The development is also a substantial case study in programming with very strong invariants enforced by the type system.

The formalisation uses *explicit substitutions*, i.e. substitutions are part of the term language being formalised. A more common approach is to use *implicit substitutions*, where substitution is a defined operation on the meta-level. This was my initial goal, but it turned out to be quite difficult to achieve (an explanation is provided in the paper). Hence the question of whether a dependently typed language with implicit substitutions can be formalised in a well-typed and usable way is still open.

## 1.4 Conclusion

This concludes the introduction. The rest of the thesis consists of the papers introduced above. More material, such as source code for machine-checked proofs, is available from my web page, currently `http://www.cs.chalmers.se/~nad/`.

# Bibliography

Thorsten Altenkirch and James Chapman. Tait in one big step. In *Workshop on Mathematically Structured Functional Programming, MSFP 2006*, 2006.

Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic, 13th International Workshop, CSL '99*, volume 1683 of *LNCS*, pages 453–468, 1999.

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

Bror Bjerner and Sören Holmström. A compositional approach to time analysis of first order lazy functional programs. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 157–165, 1989.

Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 54–65, 2007.

Catarina Coquand. A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. *Higher-Order and Symbolic Computation*, 15:57–90, 2002.

Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1): 75–94, 1997.

Nils Anders Danielsson. Proofs accompanying "Fast and loose reasoning is morally correct". Technical Report 2007:15, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

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

Peter Dybjer. Internal type theory. In *Types for Proofs and Programs, International Workshop, TYPES '95*, volume 1158 of *LNCS*, pages 120–134, 1996.

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

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

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

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

James McKinna and Joel Wright. A type-correct, stack-safe, provably correct expression compiler in Epigram. Accepted for publication in the Journal of Functional Programming.

Andrew Moran and David Sands. Improvement in a lazy context: an operational theory for call-by-need. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 43–56, 1999.

Graeme E. Moss and Colin Runciman. Automated benchmarking of functional data structures. In *Practical Aspects of Declarative Languages: First International Workshop, PADL'99*, volume 1551 of *LNCS*, pages 1–15, 1998.

Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory, An Introduction.* Oxford University Press, 1990.

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

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

Emir Pašalić and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering, Third International Conference, GPCE 2004*, volume 3286 of *LNCS*, pages 136–167, 2004.

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

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

Philip Wadler. Strictness analysis aids time analysis. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132, 1988.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL '03: Conference record of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–235, 2003.

# Chapter 2

# Fast and Loose Reasoning is Morally Correct

*This is a slightly edited version of a paper originally published as:*

Nils Anders Danielsson, Jeremy Gibbons, John Hughes and Patrik Jansson. *Fast and Loose Reasoning is Morally Correct.* In POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 206–217, 2006.

# Fast and Loose Reasoning is Morally Correct[*]

Nils Anders Danielsson John Hughes
Patrik Jansson
Chalmers University of Technology

Jeremy Gibbons
Oxford University Computing Laboratory

**Abstract**

Functional programmers often reason about programs as if they were written in a total language, expecting the results to carry over to non-total (partial) languages. We justify such reasoning.

Two languages are defined, one total and one partial, with identical syntax. The semantics of the partial language includes partial and infinite values, and all types are lifted, including the function spaces. A partial equivalence relation (PER) is then defined, the domain of which is the total subset of the partial language. For types not containing function spaces the PER relates equal values, and functions are related if they map related values to related values.

It is proved that if two closed terms have the same semantics in the total language, then they have related semantics in the partial language. It is also shown that the PER gives rise to a bicartesian closed category which can be used to reason about values in the domain of the relation.

## 1 Introduction

It is often claimed that functional programs are much easier to reason about than their imperative counterparts. Functional languages satisfy many pleasing equational laws, such as

$$curry \circ uncurry = id, \tag{1}$$

[*]This work is partially funded by the Swedish Foundation for Strategic Research as part of the research programme "Cover — Combining Verification Methods in Software Development," and by the Royal Swedish Academy of Sciences' funds.

$$(\mathit{fst}\ x, \mathit{snd}\ x) = x, \text{ and} \tag{2}$$
$$\mathit{fst}\ (x, y) = x, \tag{3}$$

and many others inspired by category theory. Such laws can be used to perform very pleasant proofs of program equality, and are indeed the foundation of an entire school of program transformation and derivation, the Squiggolers [BdM96, Jeu90, BdBH$^+$91, MFP91]. There is just one problem. In current real programming languages such as Haskell [PJ03] and ML [MTHM97], they are not generally valid.

The reason these laws fail is the presence of the undefined value $\perp$, and the fact that, in Haskell, $\perp$, $\lambda x.\perp$ and $(\perp, \perp)$ are all different (violating the first two laws above), while in ML, $\perp$, $(x, \perp)$ and $(\perp, y)$ are always the same (violating the third).

The fact that these laws are invalid does not prevent functional programmers from using them when developing programs, whether formally or informally. Squiggolers happily derive programs from specifications using them, and then transcribe the programs into Haskell in order to run them, confident that the programs will correctly implement the specification. Countless functional programmers happily curry or uncurry functions, confident that at worst they are changing definedness a little in obscure cases. Yet is this confidence justified? Reckless use of invalid laws can lead to patently absurd conclusions: for example, in ML, since $(x, \perp) = (y, \perp)$ for any $x$ and $y$, we can use the third law above to conclude that $x = y$, for any $x$ and $y$. How do we know that, when transforming programs using laws of this sort, we do not, for example, transform a correctly terminating program into an infinitely looping one?

This is the question we address in this paper. We call the unjustified reasoning with laws of this sort "fast and loose", and we show, under some mild and unsurprising conditions, that its conclusions are "morally correct". In particular, it is impossible to transform a terminating program into a looping one. Our results justify the hand reasoning that functional programmers already perform, and can be applied in proof checkers and automated provers to justify ignoring $\perp$-cases much of the time.

In the next section we give an example showing how it can be burdensome to keep track of all preconditions when one is only interested in finite and total values, but is reasoning about a program written in a partial language. Section 3 is devoted to defining the language that we focus on, its syntax and two different semantics: one set-theoretic and one domain-theoretic. Section 4 briefly discusses partial equivalence relations (PERs), and Section 5 introduces a PER on the domain-theoretic semantics. This PER is used to model totality. In Section 6 a partial surjective homomorphism from the set-theoretic semantics to the quotient of the domain-theoretic semantics given by the PER is exhibited, and in Section 7 we use this homomorphism to prove our main result: fast and loose reasoning is morally correct. Section 8 provides a more abstract result, showing how the PER gives rise to a category with many nice properties which can be used to reason about programs. We go back to our earlier example and show how it fits in with the theory in Section 9. We also exhibit another example where reasoning directly about the domain-theoretic semantics of a program may be preferable (Section 10). Section 11 recasts the theory for a strict language, Section 12 discusses related work, and Section 13 concludes with a discussion of the results and possible future extensions of the theory.

Most proofs needed for the development below are only sketched; full proofs are available as a technical report [Dan07].

## 2 Propagating preconditions

Let us begin with an example. Say that we need to prove that the function $map\ (\lambda x.y + x) \circ reverse :: [Nat] \to [Nat]$ has a left inverse $reverse \circ map\ (\lambda x.x - y)$. (All code in this section uses Haskell-like syntax.) In a total language we would do it more or less like this:

$$\begin{aligned}
& (reverse \circ map\ (\lambda x.x - y)) \circ (map\ (\lambda x.y + x) \circ reverse) \\
= \ & \{map\ f \circ map\ g = map\ (f \circ g),\ \circ \text{ associative}\} \\
& reverse \circ map\ \ ((\lambda x.x - y) \circ (\lambda x.y + x)) \circ reverse \\
= \ & \{(\lambda x.x - y) \circ (\lambda x.y + x) = id\} \\
& reverse \circ map\ id \circ reverse \\
= \ & \{map\ id = id\} \\
& reverse \circ id \circ reverse \\
= \ & \{id \circ f = f,\ \circ \text{ associative}\} \\
& reverse \circ reverse \\
= \ & \{reverse \circ reverse = id\} \\
& id.
\end{aligned}$$

Note the lemmas used for the proof, especially

$$(\lambda x.x - y) \circ (\lambda x.y + x) = id, \text{ and} \tag{4}$$
$$reverse \circ reverse = id. \tag{5}$$

Consider now the task of repeating this proof in the context of some language based on partial functions, such as Haskell. To be concrete, let us assume that the natural number data type *Nat* is defined in the usual way,

$$\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}. \tag{6}$$

Note that this type contains many properly partial values that do not correspond to any natural number, and also a total but infinite value. Let us also assume that $(+)$ and $(-)$ are defined by

$$(+) = \mathit{fold}\ \mathit{Succ}, \text{ and} \tag{7}$$
$$(-) = \mathit{fold}\ \mathit{pred}, \tag{8}$$

where $\mathit{fold} :: (a \to a) \to a \to \mathit{Nat} \to a$ is the fold over natural numbers ($\mathit{fold}\ s\ z\ n$ replaces all occurrences of *Succ* in $n$ with $s$, and *Zero* with $z$), and $\mathit{pred} :: \mathit{Nat} \to \mathit{Nat}$ is the predecessor function with $\mathit{pred}\ \mathit{Zero} = \mathit{Zero}$. The other functions and types are all standard [PJ03]; this implies that the list type also contains properly partial and infinite values.

Given these definitions the property proved above is no longer true. The proof breaks down in various places. More to the point, both lemmas (4) and (5) fail, and they fail due to both properly partial values, since

$$(\mathit{Succ}\ \mathit{Zero} + \mathit{Succ}\ \bot) - \mathit{Succ}\ \mathit{Zero} = \bot \neq \mathit{Succ}\ \bot \text{ and} \tag{9}$$
$$\mathit{reverse}\ (\mathit{reverse}\ (\mathit{Zero} : \bot)) = \bot \neq \mathit{Zero} : \bot, \tag{10}$$

and infinite values, since

$$(\mathit{fix}\ \mathit{Succ} + \mathit{Zero}) - \mathit{fix}\ \mathit{Succ} = \bot \neq \mathit{Zero} \text{ and} \tag{11}$$
$$\mathit{reverse}\ (\mathit{reverse}\ (\mathit{repeat}\ \mathit{Zero})) = \bot \neq \mathit{repeat}\ \mathit{Zero}. \tag{12}$$

(Here *fix* is the fixpoint combinator, i.e. *fix Succ* is the "infinite" lazy natural number. The application *repeat* $x$ yields an infinite list containing only $x$.) Note that $\mathit{id} \circ f = f$ also fails, since we have lifted function spaces and $\mathit{id} \circ \bot = \lambda x.\bot \neq \bot$, but that does not affect this example since $\mathit{reverse} \neq \bot$.

These problems are not surprising; they are the price you pay for partiality. Values that are properly partial and/or infinite have different properties than their total, finite counterparts. A reasonable solution is to stay in the partial language but restrict our inputs to total, finite values.

Let us see what the proof looks like then. We have to $\eta$-expand our property, and assume that $\mathit{xs} :: [\mathit{Nat}]$ is a total, finite list and that $y :: \mathit{Nat}$ is a total, finite natural number. (Note the terminology used here: if a list is said to be total, then all elements in the list are assumed to be total as well, and similarly for finite values. The concepts of totality and finiteness are discussed in more detail in Sections 5 and 9, respectively.) We get

$$
\begin{array}{ll}
& ((\mathit{reverse} \circ \mathit{map}\ (\lambda x.x - y)) \\
& \quad \circ (\mathit{map}\ (\lambda x.y + x) \circ \mathit{reverse}))\ \mathit{xs} \\
= & \{\mathit{map}\ f \circ \mathit{map}\ g = \mathit{map}\ (f \circ g), \text{definition of } \circ\} \\
& \mathit{reverse}\ (\mathit{map}\ ((\lambda x.x - y) \circ (\lambda x.y + x))\ (\mathit{reverse}\ \mathit{xs})) \\
= & \left\{ \begin{array}{l} \bullet\ \mathit{map}\ f\ \mathit{xs} = \mathit{map}\ g\ \mathit{xs} \text{ if } \mathit{xs} \text{ is total and } f\ x = g\ x \text{ for all total } x, \\ \bullet\ \mathit{reverse}\ \mathit{xs} \text{ is total, finite if } \mathit{xs} \text{ is,} \\ \bullet\ ((\lambda x.x - y) \circ (\lambda x.y + x))\ x = \mathit{id}\ x \text{ for total } x \text{ and total, finite } y \end{array} \right\} \\
& \mathit{reverse}\ (\mathit{map}\ \mathit{id}\ (\mathit{reverse}\ \mathit{xs})) \\
= & \{\mathit{map}\ \mathit{id} = \mathit{id}\} \\
& \mathit{reverse}\ (\mathit{id}\ (\mathit{reverse}\ \mathit{xs})) \\
= & \{\text{definition of } \mathit{id}\} \\
& \mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) \\
= & \{\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) = \mathit{xs} \text{ for total, finite } \mathit{xs}\} \\
& \mathit{xs}.
\end{array}
$$

Comparing to the previous proof we see that all steps are more or less identical, using similar lemmas, except for the second step, where two new lemmas are required. How did that step become so unwieldy? The problem is that, although we know that $(\lambda x.x - y) \circ (\lambda x.y + x) = \mathit{id}$ given total input, and also that $\mathit{xs}$ only contains total natural numbers, we have to manually *propagate* this precondition through *reverse* and *map*.

One view of the problem is that the type system used is too weak. If there were a type for total, finite natural numbers, and similarly for lists, then the propagation would be handled by the types of *reverse* and *map*. The imaginary total language used for the first proof effectively has such a type system.

On the other hand, note that the two versions of the program are written using identical syntax, and the semantic rules for the total language and the partial language are probably more or less identical when only total, finite (or even total, infinite) values are considered. Does not this imply that we can get the second result above, with all preconditions, by using the first proof? The answer is yes, with a little extra effort, and proving this is what many of the sections below will be devoted to. In Section 9 we come back to this example and spell out in full detail what "a little extra effort" boils down to in this case.

## 3 Language

This section defines the main language discussed in the text. It is a strongly typed, monomorphic functional language with recursive (polynomial) types and their corresponding fold and unfold operators. Having only folds and unfolds is not a serious limitation; it is e.g. easy to implement primitive recursion over lists or natural numbers inside the language.

Since we want our results to be applicable to reasoning about Haskell programs we include the explicit strictness operator seq, which forces us to have lifted function spaces in the domain-theoretic semantics given below (since seq can be used to distinguish between $\perp$ and $\lambda x.\perp$; see Figure 6). We discuss the most important differences between Haskell and this language in Section 13.

### 3.1 Static semantics

The term syntax of the language, $\mathcal{L}_1$, is inductively defined by

$$\begin{aligned} t ::= {} & x \mid t_1\ t_2 \mid \lambda x.t \\ & \mid \mathsf{seq} \mid \star \\ & \mid (,) \mid \mathsf{fst} \mid \mathsf{snd} \\ & \mid \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{case} \\ & \mid \mathsf{in}_{\mu F} \mid \mathsf{out}_{\mu F} \mid \mathsf{in}_{\nu F} \mid \mathsf{out}_{\nu F} \mid \mathsf{fold}_F \mid \mathsf{unfold}_F. \end{aligned} \tag{13}$$

The pairing function (,) can be used in a distfix style, as in $(t_1, t_2)$. The type syntax is defined by

$$\sigma,\, \tau,\, \gamma ::= \sigma \to \tau \mid \sigma \times \tau \mid \sigma + \tau \mid 1 \mid \mu F \mid \nu F \tag{14}$$

and

$$F,\, G ::= Id \mid K_\sigma \mid F \times G \mid F + G. \tag{15}$$

The letters $F$ and $G$ range over functors; $Id$ is the identity functor and $K_\sigma$ is the constant functor with $K_\sigma\ \tau = \sigma$ (informally). The types $\mu F$ and $\nu F$ are inductive and coinductive types, respectively. As an example, in the set-theoretic semantics introduced below $\mu(K_1 + Id)$ represents finite natural numbers, and $\nu(K_1 + Id)$ represents natural numbers extended with infinity. The type constructor $\to$ is sometimes used right associatively, without explicit parentheses.

In order to discuss general recursion we define the language $\mathcal{L}_2$ to be $\mathcal{L}_1$ extended with

$$t ::= \ldots \mid \mathsf{fix}. \tag{16}$$

However, whenever fix is not explicitly mentioned, the language discussed is $\mathcal{L}_1$ (or the restriction $\mathcal{L}_1'$ of $\mathcal{L}_1$ introduced below).

We only consider well-typed terms according to the typing rules in Figure 1. To ease the presentation we also introduce some syntactic sugar for terms and types, see Figures 2 and 3.

## 3.2 Dynamic semantics

Before we define the semantics of the languages we need to introduce some notation. We will use both sets and pointed $\omega$-complete partial orders (CPOs). For CPOs $\cdot_\perp$ is the lifting operator, and $\langle \cdot \to \cdot \rangle$ is the continuous function space constructor. Furthermore, for both sets and CPOs, $\times$ is cartesian product, and $+$ is separated sum; $A + B$ contains elements of the form $\mathit{inl}(a)$ with $a \in A$ and $\mathit{inr}(b)$ with $b \in B$. The one-point set/CPO is denoted by 1, with $\star$ as the only element. The constructor for the (set- or domain-theoretic) semantic domain of the recursive type $T$, where $T$ is $\mu F$ or $\nu F$, is denoted by $\mathit{in}_T$, and the corresponding destructor is denoted by $\mathit{out}_T$ (often with omitted indices). For more details about $\mathit{in}$ and $\mathit{out}$, see below. Since we use lifted function spaces, we use special notation for lifted function application,

$$f @ x = \begin{cases} \perp, & f = \perp, \\ f\ x, & \text{otherwise.} \end{cases} \tag{17}$$

(This operator is left associative with the same precedence as ordinary function application.) Many functions used on the meta-level are not lifted, though, so @ is not used very much below. Finally note that we are a little sloppy, in that we do not write out liftings explicitly; we write $(x, y)$ for a non-bottom element of $(A \times B)_\perp$, for instance.

Now, two different denotational semantics are defined for the languages introduced above, one domain-theoretic ($[\![\cdot]\!]$) and one set-theoretic ($\langle\!\langle\cdot\rangle\!\rangle$). (Note that when $t$ is closed we sometimes use $[\![t]\!]$ as a shorthand for $[\![t]\!]\,\rho$, and similarly for $\langle\!\langle\cdot\rangle\!\rangle$.) The domain-theoretic semantics is modelled on languages like Haskell and can handle general recursion. The set-theoretic semantics is modelled on total languages and is only defined for terms in $\mathcal{L}_1$. In Section 7 we will show how results obtained using the set-theoretic semantics can be transformed into results on the domain-theoretic side.

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma[x \mapsto \sigma] \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau}$$

$$\frac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau}$$

$$\begin{array}{ll}
\Gamma \vdash \mathsf{seq} : \sigma \to \tau \to \tau & \Gamma \vdash \mathsf{fix} : (\sigma \to \sigma) \to \sigma \\
\Gamma \vdash \star : 1 & \Gamma \vdash (,) : \sigma \to \tau \to (\sigma \times \tau) \\
\Gamma \vdash \mathsf{fst} : (\sigma \times \tau) \to \sigma & \Gamma \vdash \mathsf{snd} : (\sigma \times \tau) \to \tau \\
\Gamma \vdash \mathsf{inl} : \sigma \to (\sigma + \tau) & \Gamma \vdash \mathsf{inr} : \tau \to (\sigma + \tau) \\
\multicolumn{2}{l}{\Gamma \vdash \mathsf{case} : (\sigma + \tau) \to (\sigma \to \gamma) \to (\tau \to \gamma) \to \gamma} \\
\Gamma \vdash \mathsf{in}_{\mu F} : F\ \mu F \to \mu F & \Gamma \vdash \mathsf{out}_{\mu F} : \mu F \to F\ \mu F \\
\Gamma \vdash \mathsf{in}_{\nu F} : F\ \nu F \to \nu F & \Gamma \vdash \mathsf{out}_{\nu F} : \nu F \to F\ \nu F \\
\multicolumn{2}{l}{\Gamma \vdash \mathsf{fold}_F : (F\ \sigma \to \sigma) \to \mu F \to \sigma} \\
\multicolumn{2}{l}{\Gamma \vdash \mathsf{unfold}_F : (\sigma \to F\ \sigma) \to \sigma \to \nu F}
\end{array}$$

**Figure 1:** Typing rules for $\mathcal{L}_1$ and $\mathcal{L}_2$.

$$\begin{aligned}
\circ &\mapsto \lambda f\, g\, x.f\ (g\ x) \\
Id &\mapsto \lambda f\, x.f\ x \\
K_\sigma &\mapsto \lambda f\, x.x \\
F \times G &\mapsto \lambda f\, x.\mathsf{seq}\ x\ (F\ f\ (\mathsf{fst}\ x), G\ f\ (\mathsf{snd}\ x)) \\
F + G &\mapsto \lambda f\, x.\mathsf{case}\ x\ (\mathsf{inl} \circ F\ f)\ (\mathsf{inr} \circ G\ f)
\end{aligned}$$

**Figure 2:** Syntactic sugar for terms.

$$\begin{aligned}
Id\ \sigma &\mapsto \sigma \\
K_\tau\ \sigma &\mapsto \tau \\
(F \times G)\ \sigma &\mapsto F\ \sigma \times G\ \sigma \\
(F + G)\ \sigma &\mapsto F\ \sigma + G\ \sigma
\end{aligned}$$

**Figure 3:** Syntactic sugar for types.

The semantic domains for all types are defined in Figure 4. We define the semantics of recursive types by appealing to category-theoretic work [BdM96, FM91]. For instance, the set-theoretic semantic domain of $\mu F$ is the codomain of the initial object in $F$-$\mathsf{Alg}(\mathsf{SET})$. Here $\mathsf{SET}$ is the category of sets and total functions, and $F$-$\mathsf{Alg}(\mathsf{SET})$ is the category of $F$-algebras (in $\mathsf{SET}$) and homomorphisms between them. The initial object, which is known to exist given our limitations on $F$, is a function $in_{\mu F} \in \langle\!\langle F\ \mu F \rightarrow \mu F \rangle\!\rangle$. The inverse of $in_{\mu F}$ exists and, as noted above, is denoted by $out_{\mu F}$. Initiality of $in_{\mu F}$ implies that for any function $f \in \langle\!\langle F\ \sigma \rightarrow \sigma \rangle\!\rangle$ there is a unique function $fold_F\ f \in \langle\!\langle \mu F \rightarrow \sigma \rangle\!\rangle$ satisfying the universal property

$$\forall h \in \langle\!\langle \mu F \rightarrow \sigma \rangle\!\rangle\,.\ h = fold_F\ f \ \Leftrightarrow\ h \circ in_{\mu F} = f \circ F\ h. \tag{18}$$

This is how $\langle\!\langle \mathsf{fold}_F \rangle\!\rangle$ is defined. To define $\langle\!\langle \mathsf{unfold}_F \rangle\!\rangle$ we go via the *final* object $out_{\nu F}$ in $F$-$\mathsf{Coalg}(\mathsf{SET})$ (the category of $F$-coalgebras) instead. The semantics of all terms are given in Figure 6.

The domain-theoretic semantics lives in the category $\mathsf{CPO}$ of CPOs and continuous functions. To define $[\![\mu F]\!]$, the category $\mathsf{CPO}_\perp$ of CPOs and *strict* continuous functions is also used. We want all types in the domain-theoretic semantics to be lifted (like in Haskell). To model this we lift all functors using $L$, which is defined in Figure 5.

If we were to define $[\![\mathsf{fold}_F]\!]$ using the same method as for $\langle\!\langle \mathsf{fold}_F \rangle\!\rangle$, then that would restrict its arguments to be strict functions. An explicit fixpoint is used instead. The construction still satisfies the universal property associated with folds if all functions involved are strict [FM91]. For symmetry we also define $[\![\mathsf{unfold}_F]\!]$ using an explicit fixpoint; that does not affect its universality property.

The semantics of $\mathsf{fix}$ is, as usual, given by a least fixpoint construction.

We have been a little sloppy above, in that we have not defined the action of the functor $K_\sigma$ on objects. When working in $\mathsf{SET}$ we let $K_\sigma\ A = \langle\!\langle \sigma \rangle\!\rangle$, and in $\mathsf{CPO}$ and $\mathsf{CPO}_\perp$ we let $K_\sigma\ A = [\![\sigma]\!]$. Otherwise the functors have their usual meanings.

## 4 Partial equivalence relations

In what follows we will use *partial equivalence relations*, or PERs for short.

A PER on a set $S$ is a symmetric and transitive binary relation on $S$. For a PER $R$ on $S$, and some $x \in S$ with $xRx$, define the *equivalence class* of $x$ as

$$[x]_R = \{\, y \mid y \in S,\ xRy \,\}\,. \tag{19}$$

$$
\begin{aligned}
[\![\sigma \to \tau]\!] &= \langle [\![\sigma]\!] \to [\![\tau]\!] \rangle_\perp & \langle\!\langle \sigma \to \tau \rangle\!\rangle &= \langle\!\langle \sigma \rangle\!\rangle \to \langle\!\langle \tau \rangle\!\rangle \\
[\![\sigma \times \tau]\!] &= ([\![\sigma]\!] \times [\![\tau]\!])_\perp & \langle\!\langle \sigma \times \tau \rangle\!\rangle &= \langle\!\langle \sigma \rangle\!\rangle \times \langle\!\langle \tau \rangle\!\rangle \\
[\![\sigma + \tau]\!] &= ([\![\sigma]\!] + [\![\tau]\!])_\perp & \langle\!\langle \sigma + \tau \rangle\!\rangle &= \langle\!\langle \sigma \rangle\!\rangle + \langle\!\langle \tau \rangle\!\rangle \\
[\![1]\!] &= 1_\perp & \langle\!\langle 1 \rangle\!\rangle &= 1
\end{aligned}
$$

$$
\begin{aligned}
[\![\mu F]\!] &= \text{The codomain of the initial object in } L(F)\text{-}\mathsf{Alg}(\mathsf{CPO}_\perp). \\
\langle\!\langle \mu F \rangle\!\rangle &= \text{The codomain of the initial object in } F\text{-}\mathsf{Alg}(\mathsf{SET}). \\
[\![\nu F]\!] &= \text{The domain of the final object in } L(F)\text{-}\mathsf{Coalg}(\mathsf{CPO}). \\
\langle\!\langle \nu F \rangle\!\rangle &= \text{The domain of the final object in } F\text{-}\mathsf{Coalg}(\mathsf{SET}).
\end{aligned}
$$

**Figure 4:** Semantic domains for types.

$$
\begin{aligned}
L(Id) &= Id \\
L(K_\sigma) &= K_\sigma \\
L(F \times G) &= (L(F) \times L(G))_\perp \\
L(F + G) &= (L(F) + L(G))_\perp
\end{aligned}
$$

**Figure 5:** Lifting of functors.

(The index $R$ is omitted below.) Note that the equivalence classes partition $\operatorname{dom}(R) = \{\, x \in S \mid xRx \,\}$, the *domain* of $R$. Let $[R]$ denote the set of equivalence classes of $R$.

For convenience we will use the notation $\{c\}$ for an arbitrary (but fix) element $x \in c$, where $c$ is an equivalence class of some PER $R \subseteq S^2$. Note that the choice of element $x$ is irrelevant in many contexts. For example, given the PER defined in Section 5, we have that $[\mathit{inl}(\{c\})]$ denotes the same equivalence class no matter which element in $c$ is chosen.

## 5 Moral equality

We will now inductively define a family of PERs $\sim_\sigma$ on the domain-theoretic semantic domains; with $\mathit{Rel}(\sigma) = \wp\big([\![\sigma]\!]^2\big)$ we will have $\sim_\sigma \in \mathit{Rel}(\sigma)$. (Here $\wp(X)$ is the power set of $X$. The index $\sigma$ will sometimes be omitted.)

If two values are related by $\sim$, then we say that they are *morally equal*.

$$
\begin{array}{rlrl}
\llbracket x \rrbracket\,\rho &= \rho(x) & \langle\!\langle x \rangle\!\rangle\,\rho &= \rho(x) \\
\llbracket t_1 t_2 \rrbracket\,\rho &= (\llbracket t_1 \rrbracket\,\rho)\;@\;(\llbracket t_2 \rrbracket\,\rho) & \langle\!\langle t_1 t_2 \rangle\!\rangle\,\rho &= (\langle\!\langle t_1 \rangle\!\rangle\,\rho)\;(\langle\!\langle t_2 \rangle\!\rangle\,\rho) \\
\llbracket \lambda x.t \rrbracket\,\rho &= \lambda v.\,\llbracket t \rrbracket\,\rho[x \mapsto v] & \langle\!\langle \lambda x.t \rangle\!\rangle\,\rho &= \lambda v.\,\langle\!\langle t \rangle\!\rangle\,\rho[x \mapsto v] \\
\llbracket \mathsf{seq} \rrbracket &= \lambda v_1\, v_2.\begin{cases}\bot, & v_1 = \bot \\ v_2, & \text{otherwise}\end{cases} & \langle\!\langle \mathsf{seq} \rangle\!\rangle &= \lambda v_1\, v_2.v_2 \\
\llbracket \mathsf{fix} \rrbracket &= \lambda f.\,\bigsqcup_{i=0}^{\infty} f^i @ \bot & \langle\!\langle \mathsf{fix} \rangle\!\rangle & \text{is not defined.} \\
\llbracket \star \rrbracket &= \star & \langle\!\langle \star \rangle\!\rangle &= \star \\
\llbracket (,) \rrbracket &= \lambda v_1\, v_2.(v_1, v_2) & \langle\!\langle (,) \rangle\!\rangle &= \lambda v_1\, v_2.(v_1, v_2) \\
\llbracket \mathsf{fst} \rrbracket &= \lambda v.\begin{cases}\bot, & v = \bot \\ v_1, & v = (v_1, v_2)\end{cases} & \langle\!\langle \mathsf{fst} \rangle\!\rangle &= \lambda(v_1, v_2).v_1 \\
\llbracket \mathsf{snd} \rrbracket &= \lambda v.\begin{cases}\bot, & v = \bot \\ v_2, & v = (v_1, v_2)\end{cases} & \langle\!\langle \mathsf{snd} \rangle\!\rangle &= \lambda(v_1, v_2).v_2 \\
\llbracket \mathsf{inl} \rrbracket &= \lambda v.\mathit{inl}(v) & \langle\!\langle \mathsf{inl} \rangle\!\rangle &= \lambda v.\mathit{inl}(v) \\
\llbracket \mathsf{inr} \rrbracket &= \lambda v.\mathit{inr}(v) & \langle\!\langle \mathsf{inr} \rangle\!\rangle &= \lambda v.\mathit{inr}(v) \\
\llbracket \mathsf{case} \rrbracket &= \lambda v\, f_1\, f_2.\begin{cases}\bot, & v = \bot \\ f_1 @ v_1, & v = \mathit{inl}(v_1) \\ f_2 @ v_2, & v = \mathit{inr}(v_2)\end{cases} & \langle\!\langle \mathsf{case} \rangle\!\rangle &= \lambda v\, f_1\, f_2.\begin{cases} f_1\; v_1, & v = \mathit{inl}(v_1) \\ f_2\; v_2, & v = \mathit{inr}(v_2)\end{cases}
\end{array}
$$

$$
\begin{aligned}
\llbracket \mathsf{in}_{\mu F} \rrbracket &= \begin{cases}\text{The initial object in } L(F)\text{-}\mathsf{Alg}(\mathsf{CPO}_\bot)\text{, viewed as a} \\ \text{morphism in } \mathsf{CPO}.\end{cases} \\
\langle\!\langle \mathsf{in}_{\mu F} \rangle\!\rangle &= \begin{cases}\text{The initial object in } F\text{-}\mathsf{Alg}(\mathsf{SET})\text{, viewed as a morphism} \\ \text{in } \mathsf{SET}.\end{cases} \\
\llbracket \mathsf{out}_{\nu F} \rrbracket &= \begin{cases}\text{The final object in } L(F)\text{-}\mathsf{Coalg}(\mathsf{CPO})\text{, viewed as a} \\ \text{morphism in } \mathsf{CPO}.\end{cases} \\
\langle\!\langle \mathsf{out}_{\nu F} \rangle\!\rangle &= \begin{cases}\text{The final object in } F\text{-}\mathsf{Coalg}(\mathsf{SET})\text{, viewed as a} \\ \text{morphism in } \mathsf{SET}.\end{cases} \\
\llbracket \mathsf{fold}_F \rrbracket &= \lambda f.\,\llbracket \mathsf{fix} \rrbracket @ (\lambda g.f \circ \llbracket F \rrbracket @ g \circ \llbracket \mathsf{out}_{\mu F} \rrbracket) \\
\langle\!\langle \mathsf{fold}_F \rangle\!\rangle &= \lambda f.\begin{cases}\text{The unique morphism in } F\text{-}\mathsf{Alg}(\mathsf{SET}) \text{ from } \langle\!\langle \mathsf{in}_{\mu F} \rangle\!\rangle \\ \text{to } f\text{, viewed as a morphism in } \mathsf{SET}.\end{cases} \\
\llbracket \mathsf{unfold}_F \rrbracket &= \lambda f.\,\llbracket \mathsf{fix} \rrbracket @ (\lambda g.\,\llbracket \mathsf{in}_{\nu F} \rrbracket \circ \llbracket F \rrbracket @ g \circ f) \\
\langle\!\langle \mathsf{unfold}_F \rangle\!\rangle &= \lambda f.\begin{cases}\text{The unique morphism in } F\text{-}\mathsf{Coalg}(\mathsf{SET}) \text{ from } f \text{ to} \\ \langle\!\langle \mathsf{out}_{\nu F} \rangle\!\rangle\text{, viewed as a morphism in } \mathsf{SET}.\end{cases}
\end{aligned}
$$

**Figure 6:** Semantics of well-typed terms, for some context $\rho$ mapping variables to semantic values. The semantics of $\mathsf{in}_{\nu F}$ and $\mathsf{out}_{\mu F}$ are the inverses of the semantics of $\mathsf{out}_{\nu F}$ and $\mathsf{in}_{\mu F}$, respectively.

We use moral equality to formalise totality: a value $x \in [\![\sigma]\!]$ is said to be *total* iff $x \in \operatorname{dom}(\sim_\sigma)$. The intention is that if $\sigma$ does not contain function spaces, then we should have $x \sim_\sigma y$ iff $x$ and $y$ are equal, total values. For functions we will have $f \sim g$ iff $f$ and $g$ map (total) related values to related values.

The definition of totality given here should correspond to basic intuition. Sometimes another definition is used instead, where $f \in [\![\sigma \to \tau]\!]$ is total iff $f@x = \bot$ implies that $x = \bot$. That definition is not suitable for non-strict languages where most semantic domains are not flat. As a simple example, consider $[\![\mathsf{fst}]\!]$; we will have $[\![\mathsf{fst}]\!] \in \operatorname{dom}(\sim)$, so $[\![\mathsf{fst}]\!]$ is total according to our definition, but $[\![\mathsf{fst}]\!]@(\bot, \bot) = \bot$ and $(\bot, \bot) \neq \bot$.

Given the family of PERs $\sim$ we can relate the set-theoretic semantic values with the total values of the domain-theoretic semantics; see Sections 6 and 7.

## 5.1 Non-recursive types

The PER $\sim_{\sigma\to\tau}$ is a logical relation, i.e. we have the following definition for function spaces:

$$\begin{aligned} f \sim_{\sigma\to\tau} g \iff& \\ & f \neq \bot \ \wedge\ g \neq \bot \ \wedge \\ & \forall x, y \in [\![\sigma]\!] \,.\ x \sim_\sigma y \ \Rightarrow\ f@x \sim_\tau g@y. \end{aligned} \tag{20}$$

We need to ensure explicitly that $f$ and $g$ are non-bottom because some of the PERs will turn out to have $\bot \in \operatorname{dom}(\sim)$ or $\operatorname{dom}(\sim) = \emptyset$.

Pairs are related if corresponding components are related:

$$\begin{aligned} x \sim_{\sigma\times\tau} y \iff \exists x_1, y_1 \in [\![\sigma]\!]\,,\ x_2, y_2 \in [\![\tau]\!]\,. \\ x = (x_1, x_2) \ \wedge\ y = (y_1, y_2) \ \wedge \\ x_1 \sim_\sigma y_1 \ \wedge\ x_2 \sim_\tau y_2. \end{aligned} \tag{21}$$

Similarly, sums are related if they are of the same kind with related components:

$$\begin{aligned} & x \sim_{\sigma+\tau} y \iff \\ & \quad (\exists x_1, y_1 \in [\![\sigma]\!] \,.\ x = \mathit{inl}(x_1) \ \wedge\ y = \mathit{inl}(y_1) \ \wedge\ x_1 \sim_\sigma y_1) \ \vee \\ & \quad (\exists x_2, y_2 \in [\![\tau]\!] \,.\ x = \mathit{inr}(x_2) \ \wedge\ y = \mathit{inr}(y_2) \ \wedge\ x_2 \sim_\tau y_2)\,. \end{aligned} \tag{22}$$

The value $\star$ of the unit type is related to itself and $\bot$ is not related to anything:

$$x \sim_1 y \iff x = y = \star. \tag{23}$$

It is easy to check that what we have so far yields a family of PERs.

## 5.2 Recursive types

The definition for recursive types is trickier. Consider lists. When should one list be related to another? Given the intentions above it seems reasonable for $xs$ to be related to $ys$ whenever they have the same, total list structure (spine), and elements at corresponding positions are recursively related. In other words, something like

$$\begin{aligned}
&xs \sim_{\mu(K_1+(K_\sigma \times Id))} ys \;\Leftrightarrow \\
&\quad (xs = in\ (inl(\star)) \;\wedge\; ys = in\ (inl(\star))) \\
&\quad \vee \Big(\exists x, y \in [\![\sigma]\!]\,,\; xs', ys' \in [\![\mu(K_1 + (K_\sigma \times Id))]\!]\,. \\
&\qquad xs = in\ (inr((x, xs'))) \;\wedge\; ys = in\ (inr((y, ys'))) \\
&\qquad \wedge\, x \sim_\sigma y \;\wedge\; xs' \sim_{\mu(K_1+(K_\sigma \times Id))} ys'\Big).
\end{aligned} \tag{24}$$

We formalise the intuition embodied in (24) by defining a relation transformer $RT(F)$ for each functor $F$,

$$\begin{aligned}
&RT(F) \in Rel(\mu F) \to Rel(\mu F) \\
&RT(F)(X) = \{\, (in\ x, in\ y) \,|\, (x, y) \in RT'_{\mu F}(F)(X) \,\}\,.
\end{aligned} \tag{25}$$

The helper $RT'_\sigma(F)$ is defined by

$$\begin{aligned}
&RT'_\sigma(F) \in Rel(\sigma) \to Rel(F\ \sigma) \\
&RT'_\sigma(Id)(X) &&= X \\
&RT'_\sigma(K_\tau)(X) &&= \sim_\tau \\
&RT'_\sigma(F_1 \times F_2)(X) &&= \left\{ ((x_1, x_2), (y_1, y_2)) \;\middle|\; \begin{array}{l} (x_1, y_1) \in RT'_\sigma(F_1)(X), \\ (x_2, y_2) \in RT'_\sigma(F_2)(X) \end{array} \right\} \\
&RT'_\sigma(F_1 + F_2)(X) &&= \{\, (inl(x_1)\,, inl(y_1)) \,|\, (x_1, y_1) \in RT'_\sigma(F_1)(X) \,\} \\
& &&\quad \cup \\
& &&\quad \{\, (inr(x_2)\,, inr(y_2)) \,|\, (x_2, y_2) \in RT'_\sigma(F_2)(X) \,\}\,.
\end{aligned} \tag{26}$$

The relation transformer $RT(F)$ is defined for inductive types. However, replacing $\mu F$ with $\nu F$ in the definition is enough to yield a transformer suitable for coinductive types.

Now, note that $RT(F)$ is a monotone operator on the complete lattice $(Rel(\mu F)\,, \subseteq)$. This implies that it has both least and greatest fixpoints [Pri02], which leads to the following definitions:

$$x \sim_{\mu F} y \;\Leftrightarrow\; (x, y) \in \mu RT(F) \;\text{ and} \tag{27}$$

$$x \sim_{\nu F} y \;\Leftrightarrow\; (x, y) \in \nu RT(F)\,. \tag{28}$$

These definitions may not be entirely transparent. If we go back to the list example and expand the definition of $RT(K_1 + (K_\sigma \times Id))$ we get

$$\begin{aligned} & RT(K_1 + (K_\sigma \times Id))(X) = \\ & \quad \{ \, (in \; (inl(\star)), in \; (inl(\star))) \, \} \; \cup \\ & \quad \left\{ \begin{array}{l} (in \; (inr((x, xs))), \\ \;\; in \; (inr((y, ys)))) \end{array} \;\middle|\; x, y \in [\![\sigma]\!] \, , \; x \sim y, \; (xs, ys) \in X \right\} . \end{aligned} \tag{29}$$

The least and greatest fixpoints of this operator correspond to our original aims for $\sim_{\mu(K_1+(K_\sigma \times Id))}$ and $\sim_{\nu(K_1+(K_\sigma \times Id))}$. (Note that we never consider an infinite inductive list as being total.)

After adding recursive types it is still possible to show that what we have defined actually constitutes a family of PERs, but it takes a little more work. First note the two proof principles given by the definitions above: induction,

$$\forall X \subseteq [\![\mu F]\!]^2 . \;\; RT(F)(X) \subseteq X \;\Rightarrow\; \mu RT(F) \subseteq X, \tag{30}$$

and coinduction,

$$\forall X \subseteq [\![\nu F]\!]^2 . \;\; X \subseteq RT(F)(X) \;\Rightarrow\; X \subseteq \nu RT(F) \, . \tag{31}$$

Many proofs needed for this paper proceed according to a scheme similar to the following one, named IIICI below (Induction-Induction-Induction-Coinduction-Induction):

- First induction over the type structure.
- For inductive types, induction according to (30) and then induction over the functor structure.
- For coinductive types, coinduction according to (31) and then induction over the functor structure.

Using this scheme it is proved that $\sim$ is a family of PERs [Dan07].

## 5.3 Properties

We can prove that $\sim$ satisfies a number of other properties as well. Before leaving the subject of recursive types, we note that

$$x \sim_{F \, \mu F} y \;\Leftrightarrow\; in \; x \sim_{\mu F} in \; y \tag{32}$$

and

$$x \sim_{\nu F} y \;\Leftrightarrow\; \mathit{out}\; x \sim_{F\,\nu F} \mathit{out}\; y \tag{33}$$

hold, as well as the symmetric statements where $\mu F$ is replaced by $\nu F$ and vice versa. This is proved using a method similar to IIICI, but not quite identical. Another method similar to IIICI is used to verify that $\sim$ satisfies one of our initial goals: if $\sigma$ does not contain function spaces, then $x \sim_\sigma y$ iff $x, y \in \mathrm{dom}(\sim_\sigma)$ and $x = y$.

Continuing with order related properties, it is proved using induction over the type structure that $\sim_\sigma$ is monotone when seen as a function $\sim_\sigma \in [\![\sigma]\!]^2 \rightarrow 1_\perp$. This implies that all equivalence classes are upwards closed. We also have (by induction over the type structure) that $\perp \notin \mathrm{dom}(\sim_\sigma)$ for almost all types $\sigma$. The only exceptions are given by the grammar

$$\chi ::= \nu \mathit{Id} \,|\, \mu K_\chi \,|\, \nu K_\chi. \tag{34}$$

Note that $[\![\chi]\!] = \{ \perp \}$ for all these types.

The (near-complete) absence of bottoms in $\mathrm{dom}(\sim)$ gives us an easy way to show that related values are not always equal: at most types $[\![\mathsf{seq}]\!] \sim [\![\lambda x.\lambda y.y]\!]$ but $[\![\mathsf{seq}]\!] \neq [\![\lambda x.\lambda y.y]\!]$. This example breaks down when $\mathsf{seq}$ is used at type $\chi \rightarrow \sigma \rightarrow \sigma$ (unless $\mathrm{dom}(\sim_\sigma) = \emptyset$). To be able to prove the fundamental theorem below, let $\mathcal{L}_1'$ denote the language consisting of all terms from $\mathcal{L}_1$ which contain no uses of $\mathsf{seq}$ at type $\chi \rightarrow \sigma \rightarrow \sigma$.

Now, by using induction over the term structure instead of the type structure, and then following the rest of IIICI, it is shown that the fundamental theorem of logical relations holds for any term $t$ in $\mathcal{L}_1'$: if $\rho(x) \sim \rho'(x)$ for all free variables $x$ in a term $t$, then

$$[\![t]\!]\,\rho \sim [\![t]\!]\,\rho'. \tag{35}$$

The fundamental theorem is important because it implies that $[\![t]\!] \in \mathrm{dom}(\sim_\sigma)$ for all closed terms $t : \sigma$ in $\mathcal{L}_1'$. In other words, all closed terms in $\mathcal{L}_1'$ denote total values. Note, however, that $[\![\mathsf{fix}]\!] \notin \mathrm{dom}(\sim)$ (at most types) since $[\![\lambda x.x]\!] \in \mathrm{dom}(\sim)$ and $[\![\mathsf{fix}]\!] \,@\, [\![\lambda x.x]\!] = \perp$.

## 5.4 Examples

With moral equality defined we can prove a number of laws for $\sim$ which are not true for $=$. As an example, consider $\eta$-equality (combined with extensionality):

$$\begin{aligned} &\forall f, g \in [\![\sigma \rightarrow \tau]\!]\,. \\ &\qquad (\forall x \in [\![\sigma]\!]\,.\; f @ x = g @ x) \;\Leftrightarrow\; f = g. \end{aligned} \tag{36}$$

This law is not valid, since the left hand side is satisfied by the distinct values $f = \bot$ and $g = \lambda v.\bot$. On the other hand, the following variant follows immediately from the definition of $\sim$:

$$\begin{aligned}&\forall f, g \in \mathrm{dom}(\sim_{\sigma\to\tau})\,.\\ &\qquad (\forall x \in \mathrm{dom}(\sim_\sigma)\,.\; f @ x \sim g @ x) \;\Leftrightarrow\; f \sim g.\end{aligned} \tag{37}$$

As another example, consider currying (1). The corresponding statement, $[\![\mathit{curry} \circ \mathit{uncurry}]\!] \sim [\![\mathit{id}]\!]$, is easily proved using the fundamental theorem (35) and the $\eta$-law (37) above. We can also prove surjective pairing. Since $p \in \mathrm{dom}(\sim_{\sigma\times\tau})$ implies that $p = (x, y)$ for some $x \in \mathrm{dom}(\sim_\sigma)$ and $y \in \mathrm{dom}(\sim_\tau)$ we get $[\![(\mathsf{fst}\ t, \mathsf{snd}\ t)]\!]\,\rho = [\![t]\!]\,\rho$, given that $[\![t]\!]\,\rho \in \mathrm{dom}(\sim)$.

## 6 Partial surjective homomorphism

For the main theorem (Section 7) we need to relate values in $\langle\!\langle\sigma\rangle\!\rangle$ to values in $[\sim_\sigma]$, the set of equivalence classes of $\sim_\sigma$. Due to cardinality issues there is in general no total bijection between these sets; consider $\sigma = (\mathit{Nat} \to \mathit{Nat}) \to \mathit{Nat}$ with $\mathit{Nat} = \mu(K_1 + \mathit{Id})$, for instance. We can define a *partial surjective homomorphism* [Fri75] from $\langle\!\langle\sigma\rangle\!\rangle$ to $[\sim_\sigma]$, though. This means that for each type $\sigma$ there is a partial, surjective function $j_\sigma \in \langle\!\langle\sigma\rangle\!\rangle \tilde{\to} [\sim_\sigma]$, which for function types satisfies

$$(j_{\tau_1\to\tau_2}\ f)\ (j_{\tau_1}\ x) = j_{\tau_2}\ (f\ x) \tag{38}$$

whenever $f \in \mathrm{dom}(j_{\tau_1\to\tau_2})$ and $x \in \mathrm{dom}(j_{\tau_1})$. (Here $\tilde{\to}$ is the partial function space constructor and $\mathrm{dom}(f)$ denotes the domain of the partial function $f$. Furthermore we define $[f]\ \ [x] = [f@x]$, which is well-defined.)

The functions $j_\sigma \in \langle\!\langle\sigma\rangle\!\rangle \tilde{\to} [\sim_\sigma]$ are simultaneously proved to be well-defined (except where explicitly partial) and surjective by induction over the type structure plus some other techniques for the recursive cases. The following basic cases are easy:

$$\begin{aligned}&j_{\sigma\times\tau} \in \langle\!\langle\sigma\times\tau\rangle\!\rangle \tilde{\to} [\sim_{\sigma\times\tau}]\\ &j_{\sigma\times\tau}\ (x, y) = [(\{j_\sigma\ x\}\,, \{j_\tau\ y\})]\,,\end{aligned} \tag{39}$$

$$\begin{aligned}&j_{\sigma+\tau} \in \langle\!\langle\sigma+\tau\rangle\!\rangle \tilde{\to} [\sim_{\sigma+\tau}]\\ &j_{\sigma+\tau}\ \mathit{inl}(x) = [\mathit{inl}(\{j_\sigma\ x\})]\\ &j_{\sigma+\tau}\ \mathit{inr}(y) = [\mathit{inr}(\{j_\tau\ y\})]\,,\end{aligned} \tag{40}$$

and

$$\begin{aligned}&j_1 \in \langle\!\langle 1\rangle\!\rangle \tilde{\to} [\sim_1]\\ &j_1\ \star = [\star]\,.\end{aligned} \tag{41}$$

Note the use of $\{\cdot\}$ to ease the description of these functions.

It turns out to be impossible in general to come up with a total definition of $j$ for function spaces. Consider the function $\mathit{isInfinite} \in \langle\!\langle \mathit{CoNat} \to \mathit{Bool} \rangle\!\rangle$ (with $\mathit{CoNat} = \nu(K_1 + \mathit{Id})$ and $\mathit{Bool} = 1 + 1$) given by

$$\mathit{isInfinite}\ n = \begin{cases} \mathit{True}, & j\ n = [\omega]\,, \\ \mathit{False}, & \text{otherwise.} \end{cases} \tag{42}$$

(Here $\omega = [\![\mathsf{unfold}_{K_1+\mathit{Id}}\ \mathsf{inr}\ \star]\!]$ is the infinite "natural number", $\mathit{True} = \mathit{inl}(\star)$ and $\mathit{False} = \mathit{inr}(\star)$.) Any surjective homomorphism $j$ must be undefined for *isInfinite*.

Instead we settle for a partial definition. We employ a technique, originating from Friedman [Fri75], which makes it easy to prove that $j$ is homomorphic: if possible, let $j_{\tau_1 \to \tau_2}\ f$ be the element $g \in [\sim_{\tau_1 \to \tau_2}]$ satisfying

$$\forall x \in \mathrm{dom}(j_{\tau_1})\,.\ g\ (j_{\tau_1}\ x) = j_{\tau_2}\ (f\ x). \tag{43}$$

If a $g$ exists, then it can be shown to be unique (using surjectivity of $j_{\tau_1}$). If no such $g$ exists, then let $j_{\tau_1 \to \tau_2}\ f$ be undefined. To show that $j_{\tau_1 \to \tau_2}$ is surjective we use a lemma stating that $\langle\!\langle \sigma \rangle\!\rangle$ is empty iff $[\sim_\sigma]$ is.

The definition of $j$ for inductive and coinductive types follows the idea outlined for the basic cases above, but is more involved. For inductive types we use the following definition of $j$:

$$\begin{aligned} & j_{\mu F} \in \langle\!\langle \mu F \rangle\!\rangle \xrightarrow{\sim} [\sim_{\mu F}] \\ & j_{\mu F}\ x = [\mathit{in}\ \ \{J_{F,F}\ (\mathit{out}\ x)\}]\,, \end{aligned} \tag{44}$$

where

$$\begin{aligned} & J_{F,G} \in \langle\!\langle G\ \mu F \rangle\!\rangle \xrightarrow{\sim} [\sim_{G\ \mu F}] \\ & J_{F,\mathit{Id}}\ x && = j_{\mu F}\ x \\ & J_{F,K_\sigma}\ x && = j_\sigma\ x \\ & J_{F,G_1 \times G_2}\ (x, y) && = [(\{J_{F,G_1}\ x\}\,, \{J_{F,G_2}\ y\})] \\ & J_{F,G_1 + G_2}\ \mathit{inl}(x) && = [\mathit{inl}(\{J_{F,G_1}\ x\})] \\ & J_{F,G_1 + G_2}\ \mathit{inr}(y) && = [\mathit{inr}(\{J_{F,G_2}\ y\})]\,. \end{aligned} \tag{45}$$

Note the recursive invocation of $j_{\mu F}$ above. For this to be well-defined we extend the induction used to prove that $j$ is well-defined and surjective to lexicographic induction on first the type and then the *size* of values of inductive type. This size can be defined using the *fold* operator in the category SET.

Finally we define $j$ for coinductive types. Since some coinductive values are infinite we cannot use simple recursion like for inductive types. Instead we define a total helper function using *unfold* from the category CPO directly,

$$\begin{aligned} &j'_{\nu F} \in \langle \langle\!\langle \nu F \rangle\!\rangle_\perp \to [\![\nu F]\!] \rangle \\ &j'_{\nu F} = \mathit{unfold}_F \ \bigl(J'_{\nu F, F} \circ \mathit{lift\ out}\bigr) \end{aligned} \tag{46}$$

(where $\mathit{lift} \in \langle A \to B \rangle \to \langle A_\perp \to B_\perp \rangle$ lifts functions by making them strict), and then wrap up the result whenever possible:

$$\begin{aligned} &j_{\nu F} \in \langle\!\langle \nu F \rangle\!\rangle \tilde{\to} [\sim_{\nu F}] \\ &j_{\nu F}\ x = \begin{cases} [j'_{\nu F}\ x]\,, & j'_{\nu F}\ x \in \mathrm{dom}(\sim_{\nu F})\,, \\ \text{undefined}, & \text{otherwise.} \end{cases} \end{aligned} \tag{47}$$

Above we use $J'_{\sigma,G}$, defined by

$$\begin{aligned} &J'_{\sigma,G} \in \langle \langle\!\langle G\ \sigma \rangle\!\rangle_\perp \to L(G)\ \ \langle\!\langle \sigma \rangle\!\rangle_\perp \rangle \\ &J'_{\sigma,G}\ \perp &&= \perp \\ &J'_{\sigma,\mathit{Id}}\ x &&= x \\ &J'_{\sigma,K_\tau}\ x &&= \begin{cases} \{j_\tau\ x\}\,, & x \in \mathrm{dom}(j_\tau)\,, \\ \perp, & \text{otherwise} \end{cases} \\ &J'_{\sigma,G_1 \times G_2}\ (x,y) &&= (J'_{\sigma,G_1}\ x, J'_{\sigma,G_2}\ y) \\ &J'_{\sigma,G_1+G_2}\ \mathit{inl}(x) &&= \mathit{inl}\bigl(J'_{\sigma,G_1}\ x\bigr) \\ &J'_{\sigma,G_1+G_2}\ \mathit{inr}(y) &&= \mathit{inr}\bigl(J'_{\sigma,G_2}\ y\bigr)\,. \end{aligned} \tag{48}$$

Note that the recursive invocation of $j_\tau$ in $J'_{\sigma,K_\tau}$ is OK since $\tau$ is structurally smaller than the type $\nu F$ above.

## 7 Main theorem

Now we get to our main theorem. Assume that $t$ is a term in $\mathcal{L}'_1$ with contexts $\rho$ and $\rho'$ satisfying

$$\rho(x) \in \mathrm{dom}(\sim)\ \wedge\ j\ \rho'(x) = [\rho(x)] \tag{49}$$

for all variables $x$ free in $t$. Then we have that $j\ (\langle\!\langle t \rangle\!\rangle\, \rho')$ is well-defined and

$$j\ \ (\langle\!\langle t \rangle\!\rangle\ \ \rho') = [[\![t]\!]\ \ \rho]\,. \tag{50}$$

This result can be proved by induction over the structure of $t$, induction over the size of values of inductive type and coinduction for coinductive types. The case where $t$ is an application relies heavily on $j$ being homomorphic. Note that the proof depends on the particular definition of $j$ given in Section 6; if we wanted to use a different partial surjective homomorphism then the proof would need to be modified.

As a corollary to the main theorem we get, for any two terms $t_1,\ t_2 : \sigma$ in $\mathcal{L}_1'$ with two pairs of contexts $\rho_1$, $\rho_1'$ and $\rho_2$, $\rho_2'$ both satisfying the conditions of (49) (for $t_1$ and $t_2$, respectively), that

$$\langle\!\langle t_1 \rangle\!\rangle\, \rho_1' = \langle\!\langle t_2 \rangle\!\rangle\, \rho_2' \ \Rightarrow\ [\![ t_1 ]\!]\, \rho_1 \sim [\![ t_2 ]\!]\, \rho_2. \tag{51}$$

In other words, if we can prove that two terms are equal in the world of sets, then they are morally equal in the world of domains. When formalised like this the reasoning performed using set-theoretic methods, "fast and loose" reasoning, is no longer loose.

If $j$ had been injective, then (51) would have been an equivalence. That would mean that we could handle unequalities ($\neq$). The particular $j$ defined here is not injective, which can be shown using the function *isIsInfinite* $\in \langle\!\langle (\mathit{CoNat} \to \mathit{Bool}) \to \mathit{Bool} \rangle\!\rangle$ given by

$$\mathit{isIsInfinite}\ f = \begin{cases} \mathit{True}, & f = \mathit{isInfinite}, \\ \mathit{False}, & \text{otherwise.} \end{cases} \tag{52}$$

(*CoNat*, *isInfinite* etc. are defined in Section 6.) We get $j\ \mathit{isIsInfinite} = j\ (\lambda f.\mathit{False})$ (both defined), so $j$ is not injective. In fact, no $j$ which satisfies the main theorem (50) and uses the definition above for function spaces (43) can be injective.

## 8 Category-theoretic approach

Equation (51) above is useful partly because $\mathsf{SET}$ is a well-understood category. For those who prefer to work abstractly instead of working in $\mathsf{SET}$, the following result may be a useful substitute. We define the category $\mathsf{PER}_\sim$ as follows:

**Objects** The objects are types $\sigma$ (without any restrictions).

**Morphisms** The morphisms of type $\sigma \to \tau$ are the elements of $[\sim_{\sigma\to\tau}]$, i.e. equivalence classes of total functions.

**Composition** $[f] \circ [g] = [\lambda v.f @ (g @ v)]$.

This category is bicartesian closed, with initial algebras and final coalgebras for (at least) polynomial functors. All the laws that follow from this statement can be used to reason about programs. For instance, it should not be hard to repeat the total proofs from this paper using such laws.

For this method to be immediately useful, the various constructions involved should correspond closely to those in the underlying language. And they do:

**Initial object** The initial object is $\mu Id$, with the unique morphism of type $\mu Id \to \sigma$ given by $[\lambda v.\bot]$.

**Final object** The final object is 1, with the unique morphism of type $\sigma \to 1$ given by $[\lambda v.\star]$. Note that $\nu Id$ is isomorphic to 1.

**Products** The product of $\sigma$ and $\tau$ is $\sigma \times \tau$. The projections are $[[\![\mathsf{fst}]\!]]$ and $[[\![\mathsf{snd}]\!]]$, and given $[f] : \gamma \to \sigma$ and $[g] : \gamma \to \tau$ the unique morphism which "makes the diagram commute" is $[\lambda v.(f @ v, g @ v)]$.

**Coproducts** The coproduct of $\sigma$ and $\tau$ is $\sigma + \tau$. The injections are $[[\![\mathsf{inl}]\!]]$ and $[[\![\mathsf{inr}]\!]]$, and given $[f] : \sigma \to \gamma$ and $[g] : \tau \to \gamma$ the unique morphism which "makes the diagram commute" is $[\lambda v.\, [\![\mathsf{case}]\!] @ v @ f @ g]$.

**Exponentials** The exponential of $\tau$ and $\sigma$ is $\sigma \to \tau$. The apply morphism is $[\lambda(f, x).f @ x]$, and currying is given by the morphism $[\lambda f\, x\, y.f @ (x, y)]$.

**Initial algebras** For a polynomial functor $F$ the corresponding initial $F$-algebra is $(\mu F, [[\![\mathsf{in}_{\mu F}]\!]])$. Given the $F$-algebra $[f] : F\ \sigma \to \sigma$, the unique homomorphism from $[[\![\mathsf{in}_{\mu F}]\!]]$ is $[[\![\mathsf{fold}_F]\!] @ f]$.

**Final coalgebras** For a polynomial functor $F$ the corresponding final $F$-coalgebra is $(\nu F, [[\![\mathsf{out}_{\nu F}]\!]])$. Given the $F$-coalgebra $[f] : \sigma \to F\ \sigma$, the unique homomorphism to $[[\![\mathsf{out}_{\nu F}]\!]]$ is $[[\![\mathsf{unfold}_F]\!] @ f]$.

The proofs of these properties are rather easy, and do not require constructions like $j$.

The partial surjective homomorphism $j$ fits into the category-theoretic picture anyway: it can be extended to a partial functor to $\mathsf{PER}_\sim$ from the category which has types $\sigma$ as objects, total functions between the corresponding set-theoretic domains $\langle\!\langle \sigma \rangle\!\rangle$ as morphisms, and ordinary function composition as composition of morphisms. The object part of this functor is the identity, and the morphism part is given by the function space case of $j$.

## 9 Review of example

After having introduced the main theoretic body, let us now revisit the example from Section 2.

We verified that $\mathit{revMap} = \mathit{reverse} \circ \mathit{map}\ (\lambda x.x - y)$ is the left inverse of $\mathit{mapRev} = \mathit{map}\ (\lambda x.y + x) \circ \mathit{reverse}$ in a total setting. Let us express this result using the language introduced in Section 3. The type of the functions becomes $\mathit{ListNat} \to \mathit{ListNat}$, where $\mathit{ListNat}$ is the inductive type $\mu(K_1 + (K_{\mathit{Nat}} \times \mathit{Id}))$ of lists of natural numbers, and $\mathit{Nat}$ is the inductive type $\mu(K_1 + \mathit{Id})$. Note also that the functions $\mathit{reverse}$, $\mathit{map}$, $(+)$ and $(-)$ can be expressed using folds, so the terms belong to $\mathcal{L}_1$. Finally note that $\mathsf{seq}$ is not used at a type $\chi \to \sigma \to \sigma$ with $\bot \in \mathrm{dom}(\sim_\chi)$, so the terms belong to $\mathcal{L}_1'$, and we can make full use of the theory.

Our earlier proof in effect showed that

$$\langle\!\langle \mathit{revMap} \circ \mathit{mapRev} \rangle\!\rangle\ [y \mapsto n] = \langle\!\langle \mathit{id} \rangle\!\rangle \tag{53}$$

for an arbitrary $n \in \langle\!\langle \mathit{Nat} \rangle\!\rangle$, which by (51) implies that

$$[\![ \mathit{revMap} \circ \mathit{mapRev} ]\!]\ [y \mapsto n'] \sim_{\mathit{ListNat} \to \mathit{ListNat}} [\![ \mathit{id} ]\!] \tag{54}$$

whenever $n' \in \mathrm{dom}(\sim_{\mathit{Nat}})$ and $[n'] = j\ n$ for some $n \in \langle\!\langle \mathit{Nat} \rangle\!\rangle$. By the fundamental theorem (35) and the main theorem (50) we have that $[\![t]\!]$ satisfies the conditions for $n'$ for any closed term $t \in \mathcal{L}_1'$ of type $\mathit{Nat}$. This includes all total, finite natural numbers.

It remains to interpret $\sim_{\mathit{ListNat} \to \mathit{ListNat}}$. Denote the left hand side of (54) by $f$. The equation implies that $f @ \mathit{xs} \sim \mathit{ys}$ whenever $\mathit{xs} \sim \mathit{ys}$. By using the fact (mentioned in Section 5.3) that $\mathit{xs} \sim_{\mathit{ListNat}} \mathit{ys}$ iff $\mathit{xs} \in \mathrm{dom}(\sim_{\mathit{ListNat}})$ and $\mathit{xs} = \mathit{ys}$, we can restate the equation as $f @ \mathit{xs} = \mathit{xs}$ whenever $\mathit{xs} \in \mathrm{dom}(\sim_{\mathit{ListNat}})$.

We want $\mathit{xs} \in \mathrm{dom}(\sim_{\mathit{ListNat}})$ to mean the same as "$\mathit{xs}$ is total and finite". We defined totality to mean "related according to $\sim$" in Section 5, so $\mathit{xs} \in \mathrm{dom}(\sim_{\mathit{ListNat}})$ iff $\mathit{xs}$ is total. We have not defined finiteness, though. However, with any reasonable definition we can be certain that $x \in \mathrm{dom}(\sim_\sigma)$ is finite if $\sigma$ does not contain function spaces or coinductive types; in the absence of such types we can define a function $\mathit{size}_\sigma \in \mathrm{dom}(\sim_\sigma) \to \mathbb{N}$ which has the property that $\mathit{size}\ x' < \mathit{size}\ x$ whenever $x'$ is a structurally smaller part of $x$, such as with $x = \mathit{inl}(x')$ or $x = \mathit{in}\ (x', x'')$.

Hence we have arrived at the statement proved by the more elaborate proof in Section 2: for all total and finite lists $\mathit{xs}$ and all total and finite natural numbers $n$,

$$[\![ (\mathit{revMap} \circ \mathit{mapRev})\ \mathit{xs} ]\!]\ [y \mapsto n] = [\![ \mathit{xs} ]\!]\,. \tag{55}$$

This means that we have proved a result about the partial language without having to manually propagate preconditions. At first glance it may seem as if the auxiliary arguments spelled out in this section make using total methods more expensive than we first indicated. However, note that the various parts of these arguments only need to be carried out at most once for each type. They do not need to be repeated for every new proof.

## 10 Partial reasoning is sometimes preferable

This section discusses an example of a different kind from the one given in Section 2; an example where partial reasoning (i.e. reasoning using the domain-theoretic semantics directly) seems to be more efficient than total reasoning.

We define two functions *sums* and *diffs*, both of type $\mathit{ListNat} \to \mathit{ListNat}$, with the inductive types *ListNat* and *Nat* defined just like in Section 9. The function *sums* takes a list of numbers and calculates their running sum, and *diffs* performs the left inverse operation, along the lines of

$$\begin{aligned} \mathit{sums}\ [3,1,4,1,5] &= [3,4,8,9,14], \text{ and} \quad (56)\\ \mathit{diffs}\ [3,4,8,9,14] &= [3,1,4,1,5] \quad (57)\end{aligned}$$

(using standard syntactic sugar for lists and natural numbers). The aim is to prove that

$$\langle\!\langle \mathit{diffs} \circ \mathit{sums} \rangle\!\rangle = \langle\!\langle \mathit{id} \rangle\!\rangle\,. \quad (58)$$

We do that in Section 10.1. Alternatively, we can implement the functions in $\mathcal{L}_2$ and prove that

$$[\![ \mathit{diffs} \circ \mathit{sums} ]\!] @ \mathit{xs} = \mathit{xs} \quad (59)$$

for all total, finite lists $\mathit{xs} \in [\![ \mathit{ListNat} ]\!]$ containing total, finite natural numbers. That is done in Section 10.2. We then compare the experiences in Section 10.3.

### 10.1 Using total reasoning

First let us implement the functions in $\mathcal{L}_1'$. To make the development easier to follow, we use some syntax borrowed from Haskell. We also use the function *foldr*, modelled on its Haskell namesake:

$$\begin{aligned} &\mathit{foldr} : (\sigma \to (\tau \to \tau)) \to \tau \to \mu(K_1 + (K_\sigma \times \mathit{Id})) \to \tau \\ &\mathit{foldr}\ f\ x = \mathsf{fold}_{K_1+(K_\sigma\times \mathit{Id})}\ (\lambda y.\mathsf{case}\ y\ (\lambda\_.x)\ (\lambda p.f\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p)))\,. \end{aligned} \quad (60)$$

The following is a simple, albeit inefficient, recursive implementation of *sums*:

$$\begin{aligned}&\mathit{sums} : \mathit{ListNat} \to \mathit{ListNat}\\ &\mathit{sums} = \mathit{foldr}\ \mathit{add}\ [\,],\end{aligned} \tag{61}$$

where

$$\begin{aligned}&\mathit{add} : \mathit{Nat} \to \mathit{ListNat} \to \mathit{ListNat}\\ &\mathit{add}\ x\ \mathit{ys} = x : \mathit{map}\ (\lambda y.x + y)\ \mathit{ys}.\end{aligned} \tag{62}$$

Here $(+)$ and $(-)$ (used below) are implemented as folds, in a manner analogous to (7) and (8) in Section 2. The function *map* can be implemented using *foldr*:

$$\begin{aligned}&\mathit{map} : (\sigma \to \tau) \to \mu(K_1 + (K_\sigma \times \mathit{Id})) \to \mu(K_1 + (K_\tau \times \mathit{Id}))\\ &\mathit{map}\ f = \mathit{foldr}\ (\lambda x\ \mathit{ys}.f\ x : \mathit{ys})\ [\,].\end{aligned} \tag{63}$$

The definition of *diffs* uses similar techniques:

$$\begin{aligned}&\mathit{diffs} : \mathit{ListNat} \to \mathit{ListNat}\\ &\mathit{diffs} = \mathit{foldr}\ \mathit{sub}\ [\,],\end{aligned} \tag{64}$$

where

$$\begin{aligned}&\mathit{sub} : \mathit{Nat} \to \mathit{ListNat} \to \mathit{ListNat}\\ &\mathit{sub}\ x\ \mathit{ys} = x : \mathit{toHead}\ (\lambda y.y - x)\ \mathit{ys}.\end{aligned} \tag{65}$$

The helper function *toHead* applies a function to the first element of a non-empty list, and leaves empty lists unchanged:

$$\begin{aligned}&\mathit{toHead} : (\mathit{Nat} \to \mathit{Nat}) \to \mathit{ListNat} \to \mathit{ListNat}\\ &\mathit{toHead}\ f\ (y : \mathit{ys}) = f\ y : \mathit{ys}\\ &\mathit{toHead}\ f\ [\,] \qquad\ \ = [\,].\end{aligned} \tag{66}$$

Now let us prove (58). We can use fold fusion [BdM96],

$$\begin{aligned}&g \circ \mathit{foldr}\ f\ e = \mathit{foldr}\ f'\ e'\\ &\qquad \Leftarrow\ g\ e = e' \ \wedge\ \forall x, y.\ g\ (f\ x\ y) = f'\ x\ (g\ y).\end{aligned} \tag{67}$$

(For simplicity we do not write out the semantic brackets $\langle\!\langle\cdot\rangle\!\rangle$, or any contexts.) We have

$$\begin{aligned}
& \mathit{diffs} \circ \mathit{sums} = \mathit{id} \\
\Leftrightarrow\ & \{\text{definition of } \mathit{sums},\ \mathit{id} = \mathit{foldr}\ (:)\ [\,]\} \\
& \mathit{diffs} \circ \mathit{foldr}\ \mathit{add}\ [\,] = \mathit{foldr}\ (:)\ [\,] \\
\Leftarrow\ & \{\text{fold fusion}\} \\
& \mathit{diffs}\ [\,] = [\,] \wedge \\
& \forall x, ys.\ \ \mathit{diffs}\ (\mathit{add}\ x\ ys) = x : \mathit{diffs}\ ys.
\end{aligned}$$

The first conjunct is trivial, and the second one can be proved by using the lemmas

$$(\lambda y.y - x) \circ (\lambda y.x + y) = \mathit{id} \tag{68}$$

and

$$\mathit{diffs} \circ \mathit{map}\ (\lambda y.x + y) = \mathit{toHead}\ (\lambda y.x + y) \circ \mathit{diffs}. \tag{69}$$

To prove the second lemma we use fold-map fusion [BdM96],

$$\mathit{foldr}\ f\ e \circ \mathit{map}\ g = \mathit{foldr}\ (f \circ g)\ e. \tag{70}$$

We have

$$\begin{aligned}
& \mathit{diffs} \circ \mathit{map}\ (\lambda y.x + y) \\
=\ & \{\text{definition of } \mathit{diffs}\} \\
& \mathit{foldr}\ \mathit{sub}\ [\,] \circ \mathit{map}\ (\lambda y.x + y) \\
=\ & \{\text{fold-map fusion}\} \\
& \mathit{foldr}\ (\mathit{sub} \circ (\lambda y.x + y))\ [\,] \\
=\ & \{\text{fold fusion, see below}\} \\
& \mathit{toHead}\ (\lambda y.x + y) \circ \mathit{foldr}\ \mathit{sub}\ [\,] \\
=\ & \{\text{definition of } \mathit{diffs}\} \\
& \mathit{toHead}\ (\lambda y.x + y) \circ \mathit{diffs}.
\end{aligned}$$

To finish up we have to verify that the preconditions for fold fusion are satisfied above,

$$\mathit{toHead}\ (\lambda y.x + y)\ [\,] = [\,], \tag{71}$$

and

$$\begin{aligned}\forall y, ys.\ \ &\mathit{toHead}\ (\lambda y.x + y)\ (\mathit{sub}\ y\ \mathit{ys}) = \\ &(\mathit{sub} \circ (\lambda y.x + y))\ y\ (\mathit{toHead}\ (\lambda y.x + y)\ \mathit{ys}).\end{aligned} \tag{72}$$

The first one is yet again trivial, and the second one can be proved by using the lemma

$$\lambda z.z - y = (\lambda z.z - (x + y)) \circ (\lambda y.x + y). \tag{73}$$

## 10.2 Using partial reasoning

Let us now see what we can accomplish when we are not restricted to a total language. Yet again we borrow some syntax from Haskell; most notably we do not use fix directly, but define functions using recursive equations instead.

The definitions above used structural recursion. The programs below instead use structural corecursion, as captured by the function *unfoldr*, which is based on the standard unfold for lists as given by the Haskell Report [PJ03]:

$$\begin{aligned}&\mathit{unfoldr} : (\tau \to (1 + (\sigma \times \tau))) \to \tau \to \mu(K_1 + (K_\sigma \times \mathit{Id})) \\ &\mathit{unfoldr}\ f\ b = \mathsf{case}\ (f\ b)\ (\lambda\_.[\,])\ (\lambda p.\mathsf{fst}\ p : \mathit{unfoldr}\ f\ (\mathsf{snd}\ p)).\end{aligned} \tag{74}$$

Note that we cannot use unfold here, since it has the wrong type. We can write *unfoldr* with the aid of fix, though. In total languages inductive and coinductive types cannot easily be mixed; we do not have the same problem in partial languages.

The corecursive definition of *sums*,

$$\begin{aligned}&\mathit{sums} : \mathit{ListNat} \to \mathit{ListNat} \\ &\mathit{sums}\ \mathit{xs} = \mathit{unfoldr}\ \mathit{next}\ (0, \mathit{xs}),\end{aligned} \tag{75}$$

with helper *next*,

$$\begin{aligned}&\mathit{next} : (\mathit{Nat} \times \mathit{ListNat}) \to (1 + (\mathit{Nat} \times (\mathit{Nat} \times \mathit{ListNat}))) \\ &\mathit{next}\ (e, [\,]) \qquad\ = \mathsf{inl}\ \star \\ &\mathit{next}\ (e, x : \mathit{xs}) = \mathsf{inr}\ (e + x, (e + x, \mathit{xs})),\end{aligned} \tag{76}$$

should be just as easy to follow as the recursive one, if not easier. Here we have used the same definitions of $(+)$ and $(-)$ as above, and 0 is shorthand for $\mathsf{in}_{\mathit{Nat}}\ (\mathsf{inl}\ \star)$. The definition of *diffs*,

$$\begin{aligned}&\mathit{diffs} : \mathit{ListNat} \to \mathit{ListNat} \\ &\mathit{diffs}\ \mathit{xs} = \mathit{unfoldr}\ \mathit{step}\ (0, \mathit{xs}),\end{aligned} \tag{77}$$

with *step*,

$$\begin{aligned}
&\mathit{step} : (\mathit{Nat} \times \mathit{ListNat}) \to (1 + (\mathit{Nat} \times (\mathit{Nat} \times \mathit{ListNat})))\\
&\mathit{step}\ (e, [\,]) \qquad = \mathsf{inl}\ \star \\
&\mathit{step}\ (e, x : \mathit{xs}) = \mathsf{inr}\ (x - e, (x, \mathit{xs})),
\end{aligned} \tag{78}$$

is arguably more natural than the previous one.

Now we can prove (59) for all total lists containing total, finite natural numbers; we do not need to restrict ourselves to finite lists. To do that we use the approximation lemma [HG01],

$$\mathit{xs} = \mathit{ys} \quad \Leftrightarrow \quad \forall n \in \mathbb{N}.\ \mathit{approx}\ n\ \mathit{xs} = \mathit{approx}\ n\ \mathit{ys}, \tag{79}$$

where the function *approx* is defined by

$$\begin{aligned}
&\mathit{approx} \in \mathbb{N} \to [\![\mu(K_1 + (K_\sigma \times \mathit{Id}))]\!] \to [\![\mu(K_1 + (K_\sigma \times \mathit{Id}))]\!]\\
&\mathit{approx}\ 0 \qquad\quad \_ \qquad\ = \bot\\
&\mathit{approx}\ (n+1)\ \bot \qquad = \bot\\
&\mathit{approx}\ (n+1)\ [\,] \qquad = [\,]\\
&\mathit{approx}\ (n+1)\ (x : \mathit{xs}) = x : \mathit{approx}\ n\ \mathit{xs}.
\end{aligned} \tag{80}$$

Note that this definition takes place on the meta-level, since the natural numbers $\mathbb{N}$ do not correspond to any type in our language.

We have the following (yet again ignoring semantic brackets and contexts and also all uses of @):

$$\begin{aligned}
&\quad \forall \text{ total } \mathit{xs} \text{ containing total, finite numbers.}\\
&\quad\quad (\mathit{diffs} \circ \mathit{sums})\ \mathit{xs} = \mathit{xs}\\
\Leftrightarrow\ &\{\text{approximation lemma}\}\\
&\quad \forall \text{ total } \mathit{xs} \text{ containing total, finite numbers.}\\
&\quad\quad \forall n \in \mathbb{N}.\ \mathit{approx}\ n\ ((\mathit{diffs} \circ \mathit{sums})\ \mathit{xs}) = \mathit{approx}\ n\ \mathit{xs}\\
\Leftrightarrow\ &\{\text{predicate logic, definition of } \mathit{diffs}, \mathit{sums} \text{ and } \circ\}\\
&\quad \forall n \in \mathbb{N}.\ \forall \text{ total } \mathit{xs} \text{ containing total, finite numbers.}\\
&\quad\quad \mathit{approx}\ n\ (\mathit{unfoldr}\ \mathit{step}\ (0, \mathit{unfoldr}\ \mathit{next}\ (0, \mathit{xs}))) =\\
&\quad\quad \mathit{approx}\ n\ \mathit{xs}\\
\Leftarrow\ &\{\text{generalise, 0 is total and finite}\}\\
&\quad \forall n \in \mathbb{N}.\ \forall \text{ total } \mathit{xs} \text{ containing total, finite numbers.}\\
&\quad\quad \forall \text{ total and finite } y.\\
&\quad\quad\quad \mathit{approx}\ n\ (\mathit{unfoldr}\ \mathit{step}\ (y, \mathit{unfoldr}\ \mathit{next}\ (y, \mathit{xs}))) =\\
&\quad\quad\quad \mathit{approx}\ n\ \mathit{xs}.
\end{aligned}$$

We proceed by induction on the natural number $n$. The $n = 0$ case is trivial. For $n = k + 1$ we have two cases, $\mathit{xs} = [\,]$ and $\mathit{xs} = z : \mathit{zs}$ (with $z$ being a total, finite natural number, etc.). The first case is easy, whereas the second one requires a little more work:

$$\begin{aligned}
& \mathit{approx}\ (k+1) \\
& \qquad (\mathit{unfoldr}\ \mathit{step}\ (y, \mathit{unfoldr}\ \mathit{next}\ (y, z : \mathit{zs}))) \\
= {} & \{\text{definition of } \mathit{unfoldr}, \mathit{next} \text{ and } \mathit{step}\} \\
& \mathit{approx}\ (k+1) \\
& \qquad ((y+z) - y : \\
& \qquad\quad \mathit{unfoldr}\ \mathit{step}\ (y+z, \mathit{unfoldr}\ \mathit{next}\ (y+z, \mathit{zs}))) \\
= {} & \{(y+z) - y = z \text{ for } y,\ z \text{ total and finite}\} \\
& \mathit{approx}\ (k+1) \\
& \qquad (z : \mathit{unfoldr}\ \mathit{step}\ (y+z, \mathit{unfoldr}\ \mathit{next}\ (y+z, \mathit{zs}))) \\
= {} & \{\text{definition of } \mathit{approx}\} \\
& z : \mathit{approx}\ k \\
& \qquad (\mathit{unfoldr}\ \mathit{step}\ (y+z, \mathit{unfoldr}\ \mathit{next}\ (y+z, \mathit{zs}))) \\
= {} & \{\text{inductive hypothesis, } y+z \text{ is total and finite}\} \\
& z : \mathit{approx}\ k\ \mathit{zs} \\
= {} & \{\text{definition of } \mathit{approx}\} \\
& \mathit{approx}\ (k+1)\ (z : \mathit{zs}).
\end{aligned}$$

Note that we need a lemma stating that $y + z$ is total and finite whenever $y$ and $z$ are.

## 10.3 Comparison

The last proof above, based on reasoning using domain-theoretic methods, is arguably more concise than the previous one, especially considering that it is more detailed. It also proves a stronger result, since it is not limited to finite lists.

In the short example in Section 2 we had to explicitly propagate preconditions through both *map* and *reverse*. In comparison, in this longer example we only had to propagate preconditions through $(+)$ (in the penultimate step of the last proof). Notice especially the second step of the last case above. The variables $y$ and $z$ were assumed to be finite and total, and hence the lemma $(y + z) - y = z$ could immediately be applied.

There is of course the possibility that the set-theoretic implementation and proof are unnecessarily complex.[1] Note for instance that the domain-theoretic variants work equally well in the set-theoretic world, if we go for coinductive instead of inductive lists, and replace the approximation lemma with the take lemma [HG01]. Using such techniques in a sense leads to more robust results, since they never require preconditions of the kind above to be propagated manually.

However, since inductive and coinductive types are not easily mixed we cannot always go this way. If, for example, we want to process the result of *sums* using a fold, then we cannot use coinductive lists. In general we cannot use hylomorphisms [MFP91], unfolds followed by folds, in a total setting. If we want or need to use a hylomorphism, then we have to use a partial language.

## 11 Strict languages

We can treat strict languages (at least the somewhat odd language introduced below) using the framework developed so far by modelling strictness using seq, just like strict data type fields are handled in the Haskell Report [PJ03]. For simplicity we reuse the previously given set-theoretic semantics, and also all of the domain-theoretic semantics, except for one rule, the one for application.

More explicitly, we define the domain-theoretic, strict semantics $[\![\cdot]\!]_\perp$ by $[\![\sigma]\!]_\perp = [\![\sigma]\!]$ for all types. For terms we let application be strict,

$$[\![t_1 t_2]\!]_\perp \rho = \begin{cases} ([\![t_1]\!]_\perp \rho) \mathbin{@} ([\![t_2]\!]_\perp \rho)\,, & [\![t_2]\!]_\perp \rho \neq \perp, \\ \perp, & \text{otherwise.} \end{cases} \tag{81}$$

Abstractions are treated just as before,

$$[\![\lambda x.t]\!]_\perp \rho = \lambda v.\, [\![t]\!]_\perp \rho[x \mapsto v]\,, \tag{82}$$

and whenever $t$ is not an application or abstraction we let $[\![t]\!]_\perp \rho = [\![t]\!] \rho$.

We then define a type-preserving syntactic translation $^*$ on $\mathcal{L}_1$, with the

[1]Indeed, as pointed out to us after publication by Anton Setzer. If the definitions are changed so that *sums xs* = *foldr next* ($\lambda\_.[\,]$) *xs* 0, where *next x h* = $\lambda e.e + x : h\ (e + x)$, and *diffs xs* = *foldr step* ($\lambda\_.[\,]$) *xs* 0, where *step x h* = $\lambda e.x - e : h\ x$, then the domain-theoretic proof carries over to a total setting with inductive lists (by using the take lemma). However, the domain-theoretic proof is only marginally more complicated, and still has the advantage that it applies to both inductive and coinductive lists.

intention of proving that $[\![t]\!]_\perp\,\rho = [\![t^*]\!]\,\rho$. The translation is as follows:

$$t^* = \begin{cases} \mathsf{seq}\ {t_2}^*\ ({t_1}^*\ {t_2}^*), & t = t_1\ t_2, \\ \lambda x.{t_1}^*, & t = \lambda x.t_1, \\ t, & \text{otherwise.} \end{cases} \tag{83}$$

The desired property follows easily by induction over the structure of terms. It is also easy to prove that $\langle\!\langle t \rangle\!\rangle\,\rho = \langle\!\langle t^* \rangle\!\rangle\,\rho$.

Given these properties we can easily prove the variants of the main theorem (50) and its corollary (51) that result from replacing $[\![\cdot]\!]$ with $[\![\cdot]\!]_\perp$. The category-theoretic results from Section 8 immediately transfer to this new setting since the category is the same and $[\![t]\!]_\perp = [\![t]\!]$ for all closed terms $t$ not containing applications.

## 12 Related work

The notion of totality used above is very similar to that used by Scott [Sco76]. Aczel's interpretation of Martin-Löf type theory [Acz77] is also based on similar ideas, but types are modelled as predicates instead of PERs. That work has been extended by Smith [Smi84], who interprets a polymorphic variant of Martin-Löf type theory in an untyped language which shares many properties with our partial language $\mathcal{L}_2$; he does not consider coinductive types or seq, though. Beeson considers a variant of Martin-Löf type theory with $W$-types [Bee82]. $W$-types can be used to model strictly positive inductive and coinductive types [Dyb97, AAG05]. Modelling coinductive types can also be done in other ways [Hal87], and the standard trick of coding non-strict evaluation using function spaces (*force* and *delay*) may also be applicable. Furthermore it seems as if Per Martin-Löf, in unpublished work, considered lifted function spaces in a setting similar to [Smi84].

The method we use to relate the various semantic models is basically that of Friedman [Fri75]; his method is more abstract, but defined for a language with only base types, natural numbers and functions.

Scott mentions that a category similar to $\mathsf{PER}_\sim$ is bicartesian closed [Sco76].

There is a vast body of literature written on the subject of Aczel interpretations, PER models of types, and so on, and some results may be known as folklore without having been published. This text can be seen as a summary of some results, most of them previously known in one form or another, that we consider important for reasoning about functional programs. By writing down the results we make the details clear. Furthermore we apply the ideas to the problem of reasoning about programs, instead of using them only to interpret one theory in another. This is quite a natural idea, so it is not unreasonable to expect that others have made similar attempts. We know about [Dyb85], in which Dybjer explains how one can reason about an untyped partial language using total methods for expressions that are typeable (corresponding to our total values). We have not found any work that discusses fast and loose reasoning for strict languages.

Another angle on the work presented here is that we want to get around the fact that many category-theoretic isomorphisms are missing in categories like CPO. For instance, one cannot have a cartesian closed category with coproducts and fixpoints for all morphisms [HP90]. In this work we disallow all fixpoints except well-behaved ones that can be expressed as folds and unfolds. Another approach is to disallow recursion explicitly for sum types [BB91]. The MetaSoft project (see e.g. [BT83, Bli87]) advocated using "naive denotational semantics", a denotational semantics framework that does not incorporate reflexive domains. This means that some fixpoints (both on the type and the value level) are disallowed, with the aim that the less complex denotational structures may instead simplify understanding.

A case can be made for sometimes reasoning using a conservative approximate semantics, obtaining answers that are not always exactly correct [DJ04, discussion]. Programmers using languages like Haskell often ignore issues related to partiality anyway, so the spirit of many programs can be captured without treating all corner cases correctly. The methods described in this paper in a sense amount to using an approximate semantics, but with the ability to get exactly correct results by translating results involving $\sim$ to equalities with preconditions.

There is some correspondence between our approach and that of total and partial correctness reasoning for imperative programs, for example with Dijkstra's wp and wlp predicate transformers [Dij76]. In both cases, simpler approximate methods can be used to prove slightly weaker results than "what one really wants". However, in the w(l)p case, conjoining partial correctness with termination yields total correctness. In contrast, in our case, there is in general no (known) simple adjustment of a fast and loose proof to make a true proof of the same property. Nevertheless, the fast and loose proof already yields a true proof of a related property.

Sometimes it is argued that total functional programming should be used to avoid the problems with partial languages. Turner does that in the context of a language similar to the total one described here [Tur96], and discusses methods for circumventing the limitations associated with mandatory totality.

## 13 Discussion and future work

We have justified reasoning about functional languages containing partial and infinite values and lifted types, including lifted functions, using total methods. Two total methods were described, one based on a set-theoretic semantics and one based on a bicartesian closed category, both using a partial equivalence relation to interpret results in the context of a domain-theoretic semantics.

We have focused on equational reasoning. However, note that, by adding and verifying some extra axioms, the category-theoretic approach can be used to reason about unequalities ($\neq$). Given this ability it should be possible to handle more complex logical formulas as well; we have not examined this possibility in detail, though. Since the method of Section 7 (without injective $j$) cannot handle unequalities, it is in some sense weaker.

The examples in Sections 2 and 10 indicate that total methods are sometimes cheaper than partial ones, and sometimes more expensive. We have not performed any quantitative measurements, so we cannot judge the relative frequency of these two outcomes. One reasonable conclusion is that it would be good if total and partial methods could be mixed without large overheads. We have not experimented with this, but can still make some remarks.

First it should be noted that $\sim$ is not a congruence: we can have $x \sim y$ but still have $f @ x \not\sim f @ y$ (if $f \notin \mathrm{dom}(\sim)$). We can still use an established fact like $x \sim y$ by translating the statement into a form using preconditions and equality, like we did in Section 9. This translation is easy, but may result in many nontrivial preconditions, perhaps more preconditions than partial reasoning would lead to. When this is not the case it seems as if using total reasoning in some leaves of a proof, and then partial reasoning on the top-level, should work out nicely.

Another observation is that, even if some term $t$ is written in a partial style (using $\mathsf{fix}$), we may still have $[\![t]\!] \in \mathrm{dom}(\sim)$. This would for example be the case if we implemented *foldr* (see Section 10.1) using $\mathsf{fix}$ instead of $\mathsf{fold}$. Hence, if we explicitly prove that $[\![t]\!] \in \mathrm{dom}(\sim)$ then we can use $t$ in a total setting. This proof may be expensive, but enables us to use total reasoning on the top-level, with partial reasoning in some of the leaves.

Now on to other issues. An obvious question is whether one can extend the results to more advanced languages incorporating stronger forms of recursive types, polymorphism, or type constructors. Adding polymorphism would give us an easy way to transform free theorems [Rey83, Wad89] from the set-theoretic side (well, perhaps not set-theoretic [Rey84]) to the domain-theoretic one. It should be interesting to compare those results to other work involving free theorems and seq [JV04].

However, the main motivation for treating a more advanced type system is that we want the results to be applicable to languages like Haskell, and matching more features of Haskell's type system is important for that goal. Still, the current results should be sufficient to reason about monomorphic Haskell programs using only polynomial recursive types, with one important caveat: Haskell uses the sums-of-products style of data type definitions. When simulating such definitions using binary type constructors, extra bottoms are introduced. As an example, $[\![\mu(K_1 + \mathit{Id})]\!]$ contains the different values $\mathit{in}\ (\mathit{inl}(\bot))$ and $\mathit{in}\ (\mathit{inl}(\star))$, but since the constructor *Zero* is nullary the Haskell data type *Nat* from Section 2 does not contain an analogue of $\mathit{in}\ (\mathit{inl}(\bot))$. One simple but unsatisfactory solution to this problem is to restrict the types used on the Haskell side to analogues of those discussed in this paper. Another approach is of course to rework the theory using sums-of-products style data types. We foresee no major problems with this, but some details may require special attention.

## Acknowledgements

Thanks to Thierry Coquand and Peter Dybjer for pointing out related work and giving us feedback. Thanks also to Koen Claessen, Jörgen Gustavsson, Ulf Norell, Ross Paterson, K. V. S. Prasad, David Sands and Josef Svenningsson for participating in discussions and/or giving feedback.

## References

[AAG05] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[Acz77] Peter Aczel. The strength of Martin-Löf's intuitionistic type theory with one universe. In *Proceedings of Symposia in Mathematical Logic, Oulu, 1974, and Helsinki, 1975*, pages 1–32, University of Helsinki, Department of Philosophy, 1977.

[BB91] Marek A. Bednarczyk and Andrzej M. Borzyszkowski. Cpo's do not form a cpo and yet recursion works. In *VDM '91*, volume 551 of *LNCS*, pages 268–278. Springer-Verlag, 1991.

[BdBH+91] R.C. Backhouse, P.J. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J.C.S.P. van der Woude. Relational catamorphisms. In *Constructing Programs from Specifications*, pages 287–318. North-Holland, 1991.

[BdM96] Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice Hall, 1996.

[Bee82] M. Beeson. Recursive models for constructive set theories. *Annals of Mathematical Logic*, 23:127–178, 1982.

[Bli87] Andrzej Blikle. *MetaSoft Primer, Towards a Metalanguage for Applied Denotational Semantics*, volume 288 of *LNCS*. Springer-Verlag, 1987.

[BT83] Andrzej Blikle and Andrzej Tarlecki. Naive denotational semantics. In *Information Processing 83*, pages 345–355. North-Holland, 1983.

[Dan07] Nils Anders Danielsson. Proofs accompanying "Fast and loose reasoning is morally correct". Technical Report 2007:15, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DJ04] Nils Anders Danielsson and Patrik Jansson. Chasing bottoms, a case study in program verification in the presence of partial and infinite values. In *MPC 2004*, volume 3125 of *LNCS*, pages 85–109. Springer-Verlag, 2004.

[Dyb85] Peter Dybjer. Program verification in a logical theory of constructions. In *FPCA'85*, volume 201 of *LNCS*, pages 334–349. Springer-Verlag, 1985. Appears in revised form as Programming Methodology Group Report 26, University of Göteborg and Chalmers University of Technology, 1986.

[Dyb97] Peter Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical Computer Science*, 176:329–335, 1997.

[FM91] Maarten M Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1991.

[Fri75] Harvey Friedman. Equality between functionals. In *Logic Colloquium: Symposium on Logic held at Boston, 1972-73*, number 453 in Lecture Notes in Mathematics, pages 22–37. Springer, 1975.

[Hal87] Lars Hallnäs. An intensional characterization of the largest bisimulation. *Theoretical Computer Science*, 53(2–3):335–343, 1987.

[HG01] Graham Hutton and Jeremy Gibbons. The generic approximation lemma. *Information Processing Letters*, 79(4):197–201, 2001.

[HP90] Hagen Huwig and Axel Poigné. A note on inconsistencies caused by fixpoints in a cartesian closed category. *Theoretical Computer Science*, 73(1):101–112, 1990.

[Jeu90] J. Jeuring. Algorithms from theorems. In *Programming Concepts and Methods*, pages 247–266. North-Holland, 1990.

[JV04] Patricia Johann and Janis Voigtländer. Free theorems in the presence of *seq*. In *POPL'04*, pages 99–110. ACM Press, 2004.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA'91*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, 2003.

[Pri02] Hilary A. Priestley. Ordered sets and complete lattices, a primer for computer science. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, chapter 2, pages 21–78. Springer-Verlag, 2002.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier, 1983.

[Rey84] John C. Reynolds. Polymorphism is not set-theoretic. In *Semantics of Data Types*, volume 173 of *LNCS*, pages 145–156. Springer-Verlag, 1984.

[Sco76] Dana Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–587, 1976.

[Smi84] Jan Smith. An interpretation of Martin-Löf's type theory in a type-free theory of propositions. *Journal of Symbolic Logic*, 49(3):730–753, 1984.

[Tur96] David Turner. Elementary strong functional programming. In *FPLE'95*, volume 1022 of *LNCS*, pages 1–13. Springer-Verlag, 1996.

[Wad89] Philip Wadler. Theorems for free! In *FPCA'89*, pages 347–359. ACM Press, 1989.

# Chapter 3

# Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures

*This is a slightly edited version of a paper which will be published as:*

Nils Anders Danielsson. *Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures.* In POPL '08: Conference record of the 35th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2008.

# Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures

Nils Anders Danielsson
Chalmers University of Technology

**Abstract**

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

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

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

## 1 Introduction

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

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

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

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

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

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

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

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

- Linear-time minimum using insertion sort, the standard example for time complexity analysis of call-by-name programs (see Section 7).

- Implicit queues (Okasaki 1998), which make essential use of laziness (see Section 8).

- The deque operations of Hinze and Paterson's finger trees (2006).[1]
- Banker's queues (Okasaki 1998), by using the generalised method described in Section 11.[1]

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

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

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

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

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

## 2 Host language

This section discusses some aspects of Agda (Norell 2007; The Agda Team 2007), the language used for the examples in the paper, in order to make it easier to follow the text. Agda is a dependently typed functional language, and for the purposes of this paper it may be useful to think of it as a total

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

variant of Haskell (Peyton Jones 2003) with dependent types and generalised algebraic data types, but no infinite values or coinduction.

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

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

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

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

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

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

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

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

```
data ℕ : ⋆ where
  zero : ℕ
  suc  : ℕ → ℕ
```

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

```
data Seq (a : ⋆) : ℕ → ⋆ where
  nil  : Seq a zero
  (::) : {n : ℕ} → a → Seq a n → Seq a (suc n)
```

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

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

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

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

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

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

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

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

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

## 3 Library basics

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

```
$(+\!\!+) : Seq\ a\ m \to Seq\ a\ n \to Thunk\ (1 + 2 * m)\ (Seq\ a\ (m + n))$
$\mathsf{nil}$        $+\!\!+\ ys = {}^{\checkmark}\ return\ ys$
$(x :: xs) +\!\!+\ ys = {}^{\checkmark}$
  $xs +\!\!+ ys \ggg\!= \lambda xsys \to {}^{\checkmark}$
  $return\ (x :: xsys)$
```

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

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

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

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

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

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

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

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

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

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

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

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

$$\omega : \mathbb{N} \qquad\qquad ticks : Thunk\ \omega\ a$$
$$\omega = 1 + \omega \qquad\qquad ticks = {}^{\checkmark}\ ticks$$

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

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

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

$$force : Thunk\ n\ a \rightarrow a$$

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

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

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

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

## 4 Implementation

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

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

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

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

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

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

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

$$
\begin{array}{l}
\mathit{force} : \mathit{Thunk}\ n\ a \to a \\
\mathit{force}\ x = x
\end{array}
$$

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

It may be possible to implement a variant of the library in a strict language with explicit support for laziness (with memoisation). The correctness statement and proof would probably need to be modified a bit, and some type signatures may need to be changed to make it possible to ensure that code is not evaluated prematurely.

## 5 Conventions

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

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

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

The rest of this section discusses and motivates the conventions.

**Run-time vs. compile-time** It would be very awkward to have to deal with thunks in the *types* of functions, so the rules for $^{\checkmark}$ only apply to terms that will actually be executed at run-time. The function *force* may obviously not be used in run-time terms, since it can be used to discard annotations.

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

$$\begin{aligned}
&\mathit{build} : (n : \mathbb{N}) \rightarrow \mathit{Thunk}\ (1 + 2 * n)\ (\mathbb{N} \rightarrow \mathit{Thunk}\ 1\ \mathbb{N}) \\
&\mathit{build}\ \mathsf{zero} \quad\ = {}^{\checkmark}\, \mathit{return}\ (\lambda n \rightarrow {}^{\checkmark}\, \mathit{return}\ n) \\
&\mathit{build}\ (\mathsf{suc}\ n) = {}^{\checkmark} \\
&\quad \mathit{build}\ n \ggg\!= \lambda f \rightarrow {}^{\checkmark} \\
&\quad \mathit{return}\ (\lambda n \rightarrow f\ (\mathsf{suc}\ n))
\end{aligned}$$

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

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

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

## 6 Some utility functions

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

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

$$\begin{aligned}
&\mathit{wait} : (n : \mathbb{N}) \rightarrow \mathit{Thunk}\ m\ a \rightarrow \mathit{Thunk}\ (1 + n + m)\ a \\
&\mathit{wait}\ \mathsf{zero} \quad\ \ x = {}^{\checkmark}\, x \\
&\mathit{wait}\ (\mathsf{suc}\ n)\ x = {}^{\checkmark}\, \mathit{wait}\ n\ x \\
&\mathit{return}_w : (n : \mathbb{N}) \rightarrow a \rightarrow \mathit{Thunk}\ (1 + n)\ a \\
&\mathit{return}_w\ \mathsf{zero} \quad\ \ x = {}^{\checkmark}\, \mathit{return}\ x \\
&\mathit{return}_w\ (\mathsf{suc}\ n)\ x = {}^{\checkmark}\, \mathit{return}_w\ n\ x
\end{aligned}$$

Note that $\mathit{return}_w$ cannot be defined in terms of *wait*; the extra tick would give rise to a different type:

$$\begin{aligned}
&\mathit{return}_w : (n : \mathbb{N}) \rightarrow a \rightarrow \mathit{Thunk}\ (2 + n)\ a \\
&\mathit{return}_w\ n\ x = {}^{\checkmark}\, \mathit{wait}\ n\ (\mathit{return}\ x)
\end{aligned}$$

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

**abstract**
  $wait : (n : \mathbb{N}) \to Thunk\ m\ a \to Thunk\ (1 + n + m)\ a$
  $wait\ \_\ x = x$
  $return_w : (n : \mathbb{N}) \to a \to Thunk\ (1 + n)\ a$
  $return_w\ \_\ x = x$

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

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

$(=\!\!\ll) : (a \to Thunk\ m\ b) \to Thunk\ n\ a \to Thunk\ (n + m)\ b$
$f =\!\!\ll c = c \gg\!\!= f$

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

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

**if_then_else_** $: Bool \to a \to a \to Thunk\ 1\ a$
**if** true **then** $x$ **else** $y =^{\checkmark} return\ x$
**if** false **then** $x$ **else** $y =^{\checkmark} return\ y$

## 7 Non-strict data types

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

First lazy sequences are defined:

**data** $Seq_L\ (a : \star)\ (c : \mathbb{N}) : \mathbb{N} \to \star$ **where**
  $\mathsf{nil}_L\ : Seq_L\ a\ c\ 0$
  $(::_L) : a \to Thunk\ c\ (Seq_L\ a\ c\ n) \to Seq_L\ a\ c\ (1 + n)$

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

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

$$
\begin{array}{l}
\mathit{insert}\ :\ \{c : \mathbb{N}\} \to a \to \mathit{Seq}_L\ a\ c\ n \\
\qquad \to \mathit{Thunk}\ 4\ (\mathit{Seq}_L\ a\ (4 + c)\ (1 + n)) \\
\mathit{insert}\ \{c\}\ x\ \mathsf{nil}_L \qquad = {}^{\checkmark}\, \mathit{return}_w\ 2\ (x ::_L \mathit{return}_w\ (3 + c)\ \mathsf{nil}_L) \\
\mathit{insert}\ \{c\}\ x\ (y ::_L \mathit{ys}) = {}^{\checkmark} \\
\quad x \leq y \ggg\!= \lambda b \to {}^{\checkmark} \\
\quad \mathbf{if}\ b\ \mathbf{then}\ x ::_L \mathit{wait}\ (2 + c)\ (\mathit{wait}_L\ 2\ (y ::_L \mathit{ys})) \\
\qquad\quad \mathbf{else}\ \ y ::_L (\mathit{insert}\ x =\!\ll \mathit{ys})
\end{array}
$$

When $x \leq y$ the function $\mathit{wait}_L$ is used to ensure that the resulting sequence has the right type:

$$
\begin{array}{l}
\mathit{wait}_L : (c : \mathbb{N}) \to \mathit{Seq}_L\ a\ c'\ n \to \mathit{Thunk}\ 1\ (\mathit{Seq}_L\ a\ (2 + c + c')\ n) \\
\mathit{wait}_L\ c\ \mathsf{nil}_L \qquad = {}^{\checkmark}\, \mathit{return}\ \mathsf{nil}_L \\
\mathit{wait}_L\ c\ (x ::_L \mathit{xs}) = {}^{\checkmark}\, \mathit{return}\ (x ::_L \mathit{wait}\ c\ (\mathit{wait}_L\ c =\!\ll \mathit{xs}))
\end{array}
$$

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

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

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

$$
\begin{array}{l}
\mathit{sort} : \mathit{Seq}\ a\ n \to \mathit{Thunk}\ (1 + 5 * n)\ (\mathit{Seq}_L\ a\ (4 * n)\ n) \\
\mathit{sort}\ \mathsf{nil} \qquad = {}^{\checkmark}\, \mathit{return}\ \mathsf{nil}_L \\
\mathit{sort}\ (x :: \mathit{xs}) = {}^{\checkmark}\, \mathit{insert}\ x =\!\ll \mathit{sort}\ \mathit{xs}
\end{array}
$$

Note that the time needed to access the first element of the result is linear in the length of the input, whereas the time needed to force the entire result is

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

quadratic. Using *sort* and *head* the *minimum* function can easily be defined for non-empty sequences:

```
$head : Seq_L\ a\ c\ (1 + n) \to Thunk\ 1\ a$
$head\ (x ::_L xs) =^{\checkmark} return\ x$
$minimum : Seq\ a\ (1 + n) \to Thunk\ (8 + 5 * n)\ a$
$minimum\ xs =^{\checkmark} head \mathrel{=\!\!\ll} sort\ xs$
```

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

```
$last : Seq_L\ a\ c\ (1 + n) \to Thunk\ (1 + (1 + n) * (1 + c))\ a$
$last\ (x ::_L xs) =^{\checkmark} last'\ x \mathrel{=\!\!\ll} xs$
  where
  $last' : a \to Seq_L\ a\ c\ n \to Thunk\ (1 + n * (1 + c))\ a$
  $last'\ x\ \mathsf{nil}_L$        $=^{\checkmark} return\ x$
  $last'\ x\ (y ::_L ys) =^{\checkmark} last'\ y \mathrel{=\!\!\ll} ys$
$maximum : Seq\ a\ (1 + n) \to Thunk\ (13 + 14 * n + 4 * n\hat{\ }2)\ a$
$maximum\ xs =^{\checkmark} last \mathrel{=\!\!\ll} sort\ xs$
```

Fortunately there are better ways to implement this function.

## 8 Essential laziness

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

```
abstract
  $pay : (m : \mathbb{N}) \to Thunk\ n\ a \to Thunk\ m\ (Thunk\ (n - m)\ a)$
  $pay\ \_\ x = x$
```

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

The correctness of *pay* is obvious, since

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

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

1. When *pay m t* is executed (as part of a sequence of binds) the thunk *t* is executed for *m* steps and then suspended again.

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

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

## 8.1 Implicit queues

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

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

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

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

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

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

```
$snoc : Queue\ a \to a \to Thunk\ 5\ (Queue\ a)$
$snoc\ \mathsf{empty}$                 $x_1 = {}^{\checkmark}\ return_w\ 3\ (\mathsf{single}\ x_1)$
$snoc\ (\mathsf{single}\ x_1)$          $x_2 = {}^{\checkmark}$
  $return_w\ 3\ (\mathsf{twoZero}\ x_1\ x_2\ (return_w\ 4\ \mathsf{empty}))$
$snoc\ (\mathsf{twoZero}\ x_1\ x_2\ xs_3)$    $x_4 = {}^{\checkmark}$
  $pay\ 2\ xs_3 \gg\!= \lambda xs'_3 \to {}^{\checkmark}$
  $return_w\ 0\ (\mathsf{twoOne}\ x_1\ x_2\ xs'_3\ x_4)$
$snoc\ (\mathsf{twoOne}\ x_1\ x_2\ xs_3\ x_4)\ x_5 = {}^{\checkmark}$
  $xs_3 \gg\!= \lambda xs'_3 \to {}^{\checkmark}$
  $return\ (\mathsf{twoZero}\ x_1\ x_2\ (snoc\ xs'_3\ (x_4, x_5)))$
$snoc\ (\mathsf{oneZero}\ x_1\ xs_2)$       $x_3 = {}^{\checkmark}$
  $xs_2 \gg\!= \lambda xs'_2 \to {}^{\checkmark}$
  $return_w\ 0\ (\mathsf{oneOne}\ x_1\ xs'_2\ x_3)$
$snoc\ (\mathsf{oneOne}\ x_1\ xs_2\ x_3)$     $x_4 = {}^{\checkmark}$
  $pay\ 3\ (snoc\ xs_2\ (x_3, x_4)) \gg\!= \lambda xs_{234} \to {}^{\checkmark}$
  $return\ (\mathsf{oneZero}\ x_1\ xs_{234})$
```

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

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

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

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

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

```
$\mathbf{data}\ View_{\prec}\ (a : \star) : \star\ \mathbf{where}$
  $\mathsf{nil}_{\prec}\ \ : View_{\prec}\ a$
  $\mathsf{cons}_{\prec} : a \to Thunk\ 4\ (Queue\ a) \to View_{\prec}\ a$
```

The function itself is defined as follows:

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

## 8.2 Calculating invariants

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

1. Make all the subqueues thunked, also the one in the oneOne constructor.

2. Denote the time bounds and the number of debits on the various subqueues by variables. For instance:

$$\begin{array}{ll}\mathsf{oneZero} : a \to \mathit{Thunk}\ d_{10}\ (\mathit{Queue}\ (a \times a)) & \to \mathit{Queue}\ a \\ \mathsf{oneOne}\ \ : a \to \mathit{Thunk}\ d_{11}\ (\mathit{Queue}\ (a \times a)) \to a & \to \mathit{Queue}\ a\end{array}$$

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

$$\begin{array}{l} \mathit{snoc}\ (\mathsf{oneOne}\ x_1\ xs_2\ x_3)\ x_4 =^{\checkmark} \\ \quad \begin{array}{lll} xs_2 & \ggg\!= \lambda xs_2' & \to^{\checkmark} \\ \mathit{pay}\ (s - d_{10})\ (\mathit{snoc}\ xs_2'\ (x_3, x_4)) & \ggg\!= \lambda xs_{234} & \to^{\checkmark} \\ \mathit{return}?\ (\mathsf{oneZero}\ x_1\ xs_{234}) & & \end{array}\end{array}$$

(The function *return*? could be either *return* or $\mathit{return}_w\ n$, for some $n$, depending on the outcome of the analysis.)

4. The basic structure of the code now gives rise to a number of inequalities which have to be satisfied in order for the code to be well-typed. For instance, the oneOne case of *snoc* gives rise to the inequality $3 + d_{11} + (s - d_{10}) \leq s$. Solve these inequalities. If no solution can be found, then one of the assumptions above was incorrect.

5. Optionally: If, for instance, a *pay* annotation was unnecessary, then it may be possible to tighten the time bounds a little, since a tick can be removed.

## 9 Correctness

The correctness of the library is established as follows:

1. Two small languages are defined: the *simple* one without the *Thunk* type, and the *thunked* one with the library functions as primitives. An erasure function $\ulcorner\cdot\urcorner$ converts thunked terms to simple ones.

2. A lazy operational semantics is defined for the simple language, and another operational semantics is defined for the thunked language. It is shown that, under erasure, the thunked semantics is equivalent to the simple one.

3. The THUNK library guarantees (see Section 3) are established for the thunked semantics. Since the two semantics are equivalent this implies that the guarantees hold for erased terms evaluated using the simple semantics.

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

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

## 9.1 Languages

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

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

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

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

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

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

*Common typing rules*

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

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

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

*Extra typing rules for the thunked language*

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

$$\frac{\Gamma \vdash t_1 : \mathsf{Thunk}\ n_1\ \tau_1 \qquad \Gamma \vdash t_2 : \tau_1 \to \mathsf{Thunk}\ n_2\ \tau_2}{\Gamma \vdash t_1 \ggg t_2 : \mathsf{Thunk}\ (n_1 + n_2)\ \tau_2} \qquad \frac{\Gamma \vdash t : \mathsf{Thunk}\ n\ \tau}{\Gamma \vdash \mathsf{force}\ t : \tau}$$

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

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

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

Here $n$ stands for a natural number, *not* a term of type $\mathsf{Nat}$.

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

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

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

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

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

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

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

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

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

## 9.2 Operational semantics

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

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

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

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

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

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

A subset of the terms are identified as being values:

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

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

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

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

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

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

In the mechanisation this is true by construction. It is easy to make small mistakes when formalising languages, and working with well-typed syntax is nice since many mistakes are caught early.

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

*Values*

$$\frac{}{\Sigma \mid \lambda x.t \Downarrow^0 \Sigma \mid \lambda x.t} \qquad \frac{}{\Sigma \mid (t_1, t_2) \Downarrow^0 \Sigma, x_1 \mapsto t_1, x_2 \mapsto t_2 \mid (x_1, x_2)}$$

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

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

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

*Variables*

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

*Application*

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

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

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

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

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

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

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

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

As before, only well-typed heaps are considered:

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

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

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

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

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

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

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

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

The thunked semantics, denoted by $\Sigma_1 \mid e \Downarrow^n \Sigma_2 \mid v$, is given in Figures 3–4 (and presented with the same assumptions as the previous one). The thunked semantics preserves types, analogously to the simple one. Note that only binds ($\gg=$) introduce bindings of the form $x \mapsto^n e$, and that when a variable $x$ bound like this is evaluated, it is updated with an unannotated binding; this memoisation is the one tracked by the library.

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

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

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

*Values*

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

$$\frac{}{\Sigma \mid \mathsf{uncurry}\ (\lambda xy.^{\checkmark} e) \ \downdownarrows^{0}\ \Sigma \mid \mathsf{uncurry}\ (\lambda xy.^{\checkmark} e)} \qquad \frac{}{\Sigma \mid \mathsf{z} \ \downdownarrows^{0}\ \Sigma \mid \mathsf{z}}$$

$$\frac{}{\Sigma \mid \mathsf{s}\ e \ \downdownarrows^{0}\ \Sigma, x \mapsto e \mid \mathsf{s}\ x}$$

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

*Variables*

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

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

*Library primitives*

$$\frac{\Sigma_1 \mid e \ \downdownarrows^{n}\ \Sigma_2 \mid \mathsf{return}^{m}\ v}{\Sigma_1 \mid {}^{\checkmark} e \ \downdownarrows^{n}\ \Sigma_2 \mid \mathsf{return}^{1+m}\ v} \qquad \frac{\Sigma_1 \mid e \ \downdownarrows^{n}\ \Sigma_2 \mid v}{\Sigma_1 \mid \mathsf{return}\ e \ \downdownarrows^{n}\ \Sigma_2 \mid \mathsf{return}^{0}\ v}$$

$$\frac{\Sigma_1 \mid e_2 \ \downdownarrows^{n_1}\ \Sigma_2 \mid v_2 \qquad \Sigma_2, x_1 \mapsto^{m_1} e_1 \mid v_2 \bullet x_1 \ \downdownarrows^{n_2}\ \Sigma_3 \mid \mathsf{return}^{m_2}\ v}{\Sigma_1 \mid e_1 \gg\!= e_2 \ \downdownarrows^{n_1+n_2}\ \Sigma_3 \mid \mathsf{return}^{m_1+m_2}\ v}$$

$$\frac{\Sigma_1 \mid e \ \downdownarrows^{n}\ \Sigma_2 \mid \mathsf{return}^{m_1}\ v}{\Sigma_1 \mid \mathsf{pay}\ m_2\ e \ \downdownarrows^{n}\ \Sigma_2 \mid \mathsf{return}^{m_2}\ (\mathsf{return}^{m_1 - m_2}\ v)}$$

**Figure 3:** Operational semantics for the thunked language. In the rule for bind ($\gg\!=$) it is assumed that $\Gamma_1 \vdash e_1 : \mathit{Thunk}\ m_1\ \tau_1$ for some $\Gamma_1$ and $\tau_1$. Furthermore, if the right-hand side of an antecedent is $\mathsf{return}^{m}\ v$, then the only possible type-correct values are $\mathsf{return}^{m}\ v$ (for suitable $m$ and $v$); this can be seen as a form of pattern matching.

*Application*

$$\frac{\Sigma_1 \mid e_1 \Downarrow^{n_1} \Sigma_2 \mid v_1 \qquad \Sigma_2, x_2 \mapsto e_2 \mid v_1 \bullet x_2 \Downarrow^{n_2} \Sigma_3 \mid v}{\Sigma_1 \mid e_1 \cdot e_2 \Downarrow^{n_1+n_2} \Sigma_3 \mid v}$$

$$\frac{\Sigma_1 \mid e_1[x := x_2] \Downarrow^{n} \Sigma_2 \mid \mathsf{return}^m\ v}{\Sigma_1 \mid (\lambda x.^{\checkmark} e_1) \bullet x_2 \Downarrow^{1+n} \Sigma_2 \mid \mathsf{return}^{1+m}\ v}$$

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

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

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

**Figure 4:** Operational semantics for the thunked language (continued).

**Equivalence** The thunked semantics is both sound,

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

and complete,

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

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

## 9.3 Time complexity guarantees

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

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

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

By using completeness the invariant above can be simplified:

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

This statement does not refer to the thunked semantics, but is not compositional, since $\Sigma_2$ does not carry any credit.

# 10 Extensions

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

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

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

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

The operational semantics are extended as follows:

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

$$\Sigma \mid \lambda xy.^{\checkmark}\, e \Downarrow\!\!\Downarrow^{0} \Sigma \mid \lambda xy.^{\checkmark}\, e \qquad \Sigma \mid (\lambda xy.^{\checkmark}\, e_1) \bullet x_2 \Downarrow\!\!\Downarrow^{0} \Sigma \mid \lambda y.^{\checkmark} e_1[x := x_2]$$

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

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

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

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

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

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

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

Assuming a proof

$$\mathit{lemma} : (m : \mathbb{N}) \to 1 + ((1 + 2 * m) + (1 + 0)) \equiv 1 + 2 * (1 + m)$$

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

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

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

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

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

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

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

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

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

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

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

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

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

## 11 Paying for deeply embedded thunks

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned} \mathit{payFun}\ :\ & (m : \mathbb{N}) \to (a \to \mathit{Thunk}\ n\ b) \\ & \to \mathit{Thunk}\ m\ (a \to \mathit{Thunk}\ (n - m)\ b) \\ \mathit{payList}\ :\ & (m : \mathbb{N}) \to \mathit{List}\ (\mathit{Thunk}\ n\ a) \\ & \to \mathit{Thunk}\ m\ (\mathit{List}\ (\mathit{Thunk}\ (n - m)\ a)) \end{aligned}$$

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

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

$$\begin{array}{lllll} \textbf{data}\ \mathit{Ctxt} : \star_1\ \textbf{where} \\ \quad \bullet & : \mathit{Ctxt} \\ \quad \mathit{const}\bullet & : \star & \to \mathit{Ctxt} \\ \quad \mathit{Thunk}\bullet & : \mathbb{N} & \to \mathit{Ctxt} & \to \mathit{Ctxt} \\ \quad (\bullet\times) & : \mathit{Ctxt} & \to \star & \to \mathit{Ctxt} \\ \quad (\times\bullet) & : \star & \to \mathit{Ctxt} & \to \mathit{Ctxt} \end{array}$$

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{\cdot\,[\,\cdot\,] : \mathit{Ctxt} \to \star \to \star} \\
\bullet\,[\,b\,] & = b \\
(\mathit{const}\bullet\ a)\,[\,b\,] & = a \\
(\mathit{Thunk}\bullet\ n\ C)\,[\,b\,] & = \mathit{Thunk}\ n\ (C\,[\,b\,]) \\
(C\ \bullet\!\times\ a)\,[\,b\,] & = C\,[\,b\,] \times a \\
(a\ \times\!\bullet\ C)\,[\,b\,] & = a \times C\,[\,b\,]
\end{array}
$$

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{Seq}_L : \star \to \mathit{CostSeq}\ n \to \star} \\
\mathit{Seq}_L\ a\ \mathsf{nil} & = \mathit{Unit} \\
\mathit{Seq}_L\ a\ (c :: cs) & = a \times \mathit{Thunk}\ c\ (\mathit{Seq}_L\ a\ cs)
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{pay}_L\ :\ \{\,a : \star\} \to (cs_1 : \mathit{CostSeq}\ n_1) \to (c' : \mathbb{N}) \\
\qquad \to \mathit{Seq}_L\ a\ (cs_1 +\!\!+\ c :: cs_2) \\
\qquad \to \mathit{Thunk}\ (1 + c')\ (\mathit{Seq}_L\ a\ (cs_1 +\!\!+\ (c - c') :: cs_2)) \\
\mathit{pay}_L\ \{\,a\}\ cs_1\ c'\ xs =^{\checkmark} \\
\quad \mathit{cast}\ \mathit{lemma}_2\ (\mathit{pay}_g\ (C\ a\ cs_1)\ c'\ (\mathit{cast}\ \mathit{lemma}_1\ xs)) \\
\quad \textbf{where} \\
\quad C : \star \to \mathit{CostSeq}\ n \to \mathit{Ctxt} \\
\quad C\ a\ \mathsf{nil} \qquad\ = a\ \times\!\bullet\ \bullet \\
\quad C\ a\ (c :: cs) = a\ \times\!\bullet\ \mathit{Thunk}\bullet\ c\ (C\ a\ cs) \\
\quad \mathit{lemma}_1 : \mathit{Seq}_L\ a\ (cs_1 +\!\!+\ c :: cs_2) \overset{\star}{\equiv} \\
\qquad\qquad (C\ a\ cs_1)\,[\,\mathit{Thunk}\ c\ (\mathit{Seq}_L\ a\ cs_2)\,] \\
\quad \mathit{lemma}_2 : \mathit{Thunk}\ c'\ ((C\ a\ cs_1)\,[\,\mathit{Thunk}\ (c - c')\ (\mathit{Seq}_L\ a\ cs_2)\,]) \overset{\star}{\equiv} \\
\qquad\qquad \mathit{Thunk}\ c'\ (\mathit{Seq}_L\ a\ (cs_1 +\!\!+\ (c - c') :: cs_2))
\end{array}
$$

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

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

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

Finally note that $\mathit{pay}_g$ has been proved correct, using the same (mechanised) approach as above; for details see Danielsson (2007).

## 12 Limitations

This section discusses some limitations of Thunk.

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

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

However, this would not in itself be very useful: *force* is abstract (see Section 4), so *force x* would not evaluate in the type of *bind*. One way to work around this is by providing the library user with a number of axioms specifying how the library primitives evaluate. This can be useful anyway, since it makes it possible to prove ordinary functional properties of annotated code inside Agda. This solution is rather complicated, though.

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

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

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

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

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

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

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

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

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

## 13 Related work

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

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

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

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

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

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

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

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

$$
\begin{aligned}
&\mathbf{codata}\ \cdot^{\nu}\ (a : \star) : \star\ \mathbf{where}\\
&\quad \mathsf{return} : a \quad \to a^{\nu}\\
&\quad \mathsf{step} \quad : a^{\nu} \to a^{\nu}
\end{aligned}
$$

The following definition of bind turns it into a monad:

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

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

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

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

## 14 Conclusions

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

## Acknowledgements

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

## References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Chapter 4

# A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family

*This is a slightly edited version of a paper originally published as:*

Nils Anders Danielsson. *A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family.* In Types for Proofs and Programs, International Workshop, TYPES 2006, Revised Selected Papers, volume 4502 of LNCS, pages 93–109, 2007.

# A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family

Nils Anders Danielsson
Chalmers University of Technology

**Abstract**

It is demonstrated how a dependently typed lambda calculus (a logical framework) can be formalised inside a language with inductive-recursive families. The formalisation does not use raw terms; the well-typed terms are defined directly. It is hence impossible to create ill-typed terms.

As an example of programming with strong invariants, and to show that the formalisation is usable, normalisation is proved. Moreover, this proof seems to be the first formal account of normalisation by evaluation for a dependently typed language.

## 1 Introduction

Programs can be verified in many different ways. One difference lies in how invariants are handled. Consider a type checker, for instance. The typing rules of the language being type checked are important invariants of the resulting abstract syntax. In the *external* approach to handling invariants the type checker works with raw terms. Only later, when verifying the soundness of the type checker, is it necessary to verify that the resulting, supposedly well-typed terms satisfy the invariants (typing rules). In the *internal* approach the typing rules are instead represented directly in the abstract syntax data types, and soundness thus follows automatically from the type of the type checking function, possibly at the cost of extra work in the implementation. For complicated invariants the internal approach requires strong forms of data types, such as inductive families or generalised algebraic data types.

Various aspects of many different essentially simply typed programming languages have been formalised using the internal approach [CD97, AR99, Coq02, XCC03, PL04, MM04, AC06, MW06, McBb]. Little work has been done on formalising *dependently* typed languages using this approach, though; Dybjer's work [Dyb96] on formalising so-called categories with families, which can be seen as the basic framework of dependent types, seems to be the only exception. The present work attempts to fill this gap.

This paper describes a formalisation of the type system, and a proof of normalisation, for a dependently typed lambda calculus (basically the logical framework of Martin-Löf's monomorphic type theory [NPS90] with explicit substitutions). Moreover, the proof of normalisation seems to be the first formal implementation of normalisation by evaluation [ML75, BS91] for a dependently typed language. The ultimate goal of this work is to have a formalised implementation of the core type checker for a full-scale implementation of type theory.

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

- A fully typed representation of a dependently typed language (Sect. 3).
- A proof of normalisation (Sect. 5). This proof seems to be the first account of a formal implementation of normalisation by evaluation for a dependently typed language.
- Everything is implemented and type checked in the proof checker AgdaLight [Nor07]. The code can be downloaded from the author's web page [Dan07].

## 2 Meta Language

Let us begin by introducing the meta language in which the formalisation has been carried out, AgdaLight [Nor07], a prototype of a dependently typed programming language. It is in many respects similar to Haskell, but, naturally, deviates in some ways.

One difference is that AgdaLight lacks polymorphism, but has *hidden arguments*, which in combination with dependent types compensate for this loss. For instance, the ordinary list function *map* could be given the following type signature:

$$\mathit{map} : \{\, a, b : \mathit{Set} \,\} \rightarrow (a \rightarrow b) \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ b$$

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

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

AgdaLight also has inductive-recursive families [DS06], illustrated by the following example (which is not recursive, just inductive). Data types are introduced by listing the constructors and giving their types; natural numbers can for instance be defined as follows:

$$\begin{array}{l} \textbf{data}\ Nat : Set\ \textbf{where} \\ \quad zero : Nat \\ \quad suc\ \ : Nat \to Nat \end{array}$$

Vectors, lists of a given fixed length, may be more interesting:

$$\begin{array}{l} \textbf{data}\ Vec\ (a : Set) : Nat \to Set\ \textbf{where} \\ \quad nil\ \ \ : Vec\ a\ zero \\ \quad cons : \{n : Nat\} \to a \to Vec\ a\ n \to Vec\ a\ (suc\ n) \end{array}$$

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

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

$$\begin{array}{l} tail : \{a : Set\} \to \{n : Nat\} \to Vec\ a\ (suc\ n) \to Vec\ a\ n \\ tail\ (cons\ x\ xs) = xs \end{array}$$

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

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

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

Note that the hidden arguments of $(+\!\!+)$ were not declared in its type signature. This is not allowed by AgdaLight, but often done in the paper to reduce notational noise. Some other details of the formalisation are also ignored, to make the paper easier to follow. The actual code can be downloaded for inspection [Dan07].

Note also that the inductive-recursive families in this formalisation are inductive-recursive in the sense that they consist of inductive families and recursive functions which depend on each other. However, some of them do not quite meet the requirements of [DS06]; see Sects. 3.2 and 5.2. Furthermore [DS06] only deals with functions defined using elimination rules. The functions in this paper are defined using pattern matching and structural recursion.

AgdaLight currently lacks (working) facilities for checking that the code is terminating and that all pattern matching definitions are exhaustive. However, for the formalisation presented here this has been verified manually. Unless some mistake has been made all data types are strictly positive (with the exception of *Val*; see Sect. 5.2), all definitions are exhaustive, and every function uses structural recursion of the kind accepted by the termination checker foetus [AA02].

## 3 Object Language

The object language that is formalised is a simple dependently typed lambda calculus with explicit substitutions. Its type system is sketched in Fig. 1. The labels on the rules correspond to constructors introduced in the formalisation. Note that $\Gamma \Rightarrow \Delta$ is the type of substitutions taking terms with variables in $\Gamma$ to terms with variables in $\Delta$, and that the symbol $=_\star$ stands for $\beta\eta$-equality between types. Some things are worth noting about the language:

- It has explicit substitutions in the sense that the application of a substitution to a term is an explicit construction in the language. However, the application of a substitution to a *type* is an implicit operation.
- There does not seem to be a "standard" choice of basic substitutions. The set chosen here is the following:
  - $[x \mapsto t]$ is the substitution mapping $x$ to $t$ and every other variable to itself.
  - $\mathit{wk}\ x\ \tau$ extends the context with a new, unused variable.
  - $\mathit{id}\ \Gamma$ is the identity substitution on $\Gamma$.
  - $\rho \uparrow_x \tau$ is a lifting; variable $x$ is mapped to itself, and the other variables are mapped by $\rho$.
  - $\rho_1\ \rho_2$ is composition of substitutions.

*Contexts*

$$\frac{}{\varepsilon\ \mathit{context}}\ (\varepsilon) \qquad \frac{\Gamma\ \mathit{context} \qquad \Gamma \vdash \tau\ \mathit{type}}{\Gamma, x : \tau\ \mathit{context}}\ (\triangleright)$$

*Types*

$$\frac{}{\Gamma \vdash \star\ \mathit{type}}\ (\star) \qquad \frac{\Gamma \vdash \tau_1\ \mathit{type} \qquad \Gamma, x : \tau_1 \vdash \tau_2\ \mathit{type}}{\Gamma \vdash \Pi(x : \tau_1)\, \tau_2\ \mathit{type}}\ (\Pi) \qquad \frac{\Gamma \vdash t : \star}{\Gamma \vdash \mathit{El}\ t\ \mathit{type}}\ (\mathit{El})$$

*Terms*

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ (\mathit{var}) \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . t : \Pi(x : \tau_1)\, \tau_2}\ (\lambda)$$

$$\frac{\Gamma \vdash t : \tau \qquad \rho\ :\ \Gamma \Rightarrow \Delta}{\Delta \vdash t\ \rho : \tau\ \rho}\ (/\!\vdash) \qquad \frac{\Gamma \vdash t_1 : \Pi(x : \tau_1)\, \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2\ [x \mapsto t_2]}\ (@)$$

$$\frac{\Gamma \vdash t : \tau_1 \qquad \tau_1 =_\star \tau_2}{\Gamma \vdash t : \tau_2}\ (::\overset{\equiv}{\vdash})$$

*Substitutions*

$$\frac{\Gamma \vdash t : \tau}{[x \mapsto t]\ :\ \Gamma, x : \tau \Rightarrow \Gamma}\ (\mathit{sub}) \qquad \frac{}{\mathit{wk}\ x\ \tau\ :\ \Gamma \Rightarrow \Gamma, x : \tau}\ (\mathit{wk})$$

$$\frac{}{\mathit{id}\ \Gamma\ :\ \Gamma \Rightarrow \Gamma}\ (\mathit{id}) \qquad \frac{\rho\ :\ \Gamma \Rightarrow \Delta}{\rho \uparrow_x \tau\ :\ \Gamma, x : \tau \Rightarrow \Delta, x : \tau\ \rho}\ (\uparrow)$$

$$\frac{\rho_1\ :\ \Gamma \Rightarrow \Delta \qquad \rho_2\ :\ \Delta \Rightarrow X}{\rho_1\ \rho_2\ :\ \Gamma \Rightarrow X}\ (\odot)$$

**Figure 1:** Sketch of the type system that is formalised. If a rule mentions $\Gamma \vdash t : \tau$, then it is implicitly assumed that $\Gamma$ *context* and $\Gamma \vdash \tau$ *type*; similar assumptions apply to the other judgements as well. All freshness side conditions have been omitted.

- Heterogeneous equality is used. Two types can be equal ($\tau_1 =_\star \tau_2$) even though their contexts are not definitionally equal in the meta-theory. Contexts of equal types are always provably equal in the object-theory, though (see Sect. 3.6).

The following subsections describe the various parts of the formalisation: contexts, types, terms, variables, substitutions and equalities. Section 3.7 discusses some of the design choices made. The table below summarises the types defined; the concept being defined, typical variable names used for elements of the type, and the type name (fully applied):

| | | |
|---|---|---|
| Contexts | $\Gamma$, $\Delta$, $X$ | $\mathit{Ctxt}$ |
| Types | $\tau$, $\sigma$ | $\mathit{Ty}\ \Gamma$ |
| Terms | $t$ | $\Gamma \vdash \tau$ |
| Variables | $v$ | $\Gamma \ni \tau$ |
| Substitutions | $\rho$ | $\Gamma \Rightarrow \Delta$ |
| Equalities | $eq$ | $\Gamma_1 =_{\mathit{Ctxt}} \Gamma_2,\ \ \tau_1 =_\star \tau_2,\ \ldots$ |

Note that all the types in this section are part of the same mutually recursive definition, together with the function (/) (see Sect. 3.2).

## 3.1 Contexts

Contexts are represented in a straight-forward way. The empty context is written $\varepsilon$, and $\Gamma \rhd \tau$ is the context $\Gamma$ extended with the type $\tau$. Variables are represented using de Bruijn indices, so there is no need to mention variables here:

```
data Ctxt : Set where
  ε   : Ctxt
  (▷) : (Γ : Ctxt) → Ty Γ → Ctxt
```

$\mathit{Ty}\ \Gamma$ is the type, introduced below, of object-language types with variables in $\Gamma$.

## 3.2 Types

The definition of the type family $\mathit{Ty}$ of object-level types follows the type system sketch in Fig. 1:

```
data Ty : Ctxt → Set where
  ⋆  : {Γ : Ctxt} → Ty Γ
  Π  : (τ : Ty Γ) → Ty (Γ ▷ τ) → Ty Γ
  El : Γ ⊢ ⋆ → Ty Γ
```

The type $\Gamma \vdash \tau$ stands for terms of type $\tau$ with variables in $\Gamma$, so terms can only be viewed as types if they have type $\star$.

Note that types are indexed on the context to which their variables belong, and similarly terms are indexed on both contexts and types ($\Gamma \vdash \tau$). The meta-theory behind indexing a type by a type family defined in the *same* mutually recursive definition has not been worked out properly yet. It is, however, crucial to this formalisation.

Let us now define the function $(/)$, which applies a substitution to a type (note that postfix application is used). The type $\Gamma \Rightarrow \Delta$ stands for a substitution which, when applied to something in context $\Gamma$ (a type, for instance), transforms this into something in context $\Delta$:

$$\begin{array}{lcl}
(/) : \mathit{Ty}\ \Gamma \ \rightarrow\ \Gamma \Rightarrow \Delta \ \rightarrow\ \mathit{Ty}\ \Delta & & \\
\star \qquad\quad / \rho & = & \star \\
\Pi\ \tau_1\ \tau_2\ / \ \rho & = & \Pi\ (\tau_1\ /\ \rho)\ (\tau_2\ /\ \rho \uparrow \tau_1) \\
\mathit{El}\ t \qquad / \rho & = & \mathit{El}\ (t \ /_{\vdash}\ \rho)
\end{array}$$

The constructor $(/_{\vdash})$ is the analogue of $(/)$ for terms (see Sect. 3.3). The substitution transformer $(\uparrow)$ is used when going under binders; $\rho \uparrow \tau_1$ behaves as $\rho$, except that the new variable zero in the original context is mapped to the new variable zero in the resulting context:

$$(\uparrow) : (\rho : \Gamma \Rightarrow \Delta) \ \rightarrow\ (\sigma : \mathit{Ty}\ \Gamma) \ \rightarrow\ \Gamma \triangleright \sigma \ \Rightarrow\ \Delta \triangleright (\sigma\ /\ \rho)$$

Substitutions are defined in Sect. 3.5.

## 3.3 Terms

The types $\Gamma \vdash \tau$ and $\Gamma \ni \tau$ stand for terms and variables, respectively, of type $\tau$ in context $\Gamma$. Note that what is customarily written $\Gamma \vdash t : \tau$, like in Fig. 1, is now written $t : \Gamma \vdash \tau$. There are five kinds of terms: variables (*var*), abstractions ($\lambda$), applications ($@$), casts ($::^{\equiv}_{\vdash}$) and substitution applications ($/_{\vdash}$):

$$\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ (\vdash) : (\Gamma : \mathit{Ctxt}) \ \rightarrow\ \mathit{Ty}\ \Gamma \ \rightarrow\ \mathit{Set}\ \textbf{where}} \\
\quad \mathit{var} & : \Gamma \ni \tau & & \rightarrow \Gamma \vdash \tau \\
\quad \lambda & : \Gamma \triangleright \tau_1 \vdash \tau_2 & & \rightarrow \Gamma \vdash \Pi\ \tau_1\ \tau_2 \\
\quad (@) & : \Gamma \vdash \Pi\ \tau_1\ \tau_2 & \rightarrow\ (t_2 : \Gamma \vdash \tau_1) & \rightarrow \Gamma \vdash \tau_2\ /\ \mathit{sub}\ t_2 \\
\quad (::^{\equiv}_{\vdash}) & : \Gamma \vdash \tau_1 & \rightarrow\ \tau_1 =_\star \tau_2 & \rightarrow \Gamma \vdash \tau_2 \\
\quad (/_{\vdash}) & : \Gamma \vdash \tau & \rightarrow\ (\rho : \Gamma \Rightarrow \Delta) & \rightarrow \Delta \ \vdash\ \tau\ /\ \rho
\end{array}$$

Notice the similarity to the rules in Fig. 1. The substitution *sub* $t_2$ used in the definition of $(@)$ replaces *vz* with $t_2$, and lowers the index of all other variables by one:

$$\mathit{sub} : \Gamma \vdash \tau \;\rightarrow\; \Gamma \rhd \tau \Rightarrow \Gamma$$

The conversion rule defined here ($::^{\equiv}_{\vdash}$) requires the two contexts to be definitionally equal in the meta-theory. A more general version of the rule would lead to increased complexity when functions that pattern match on terms are defined. However, we can *prove* a general version of the conversion rule, so no generality is lost:

$$(::_{\vdash}) : \Gamma_1 \vdash \tau_1 \;\rightarrow\; \tau_1 =_{\star} \tau_2 \;\rightarrow\; \Gamma_2 \vdash \tau_2$$

In this formalisation, whenever a cast constructor named ($::^{\equiv}_{\bullet}$) is introduced (where $\bullet$ can be $\vdash$ or $\ni$, for instance), a corresponding generalised variant ($::_{\bullet}$) is always proved.

Before moving on to variables, note that all typing information is present in a term, including casts (the conversion rule). Hence this type family actually represents typing derivations.

## 3.4 Variables

Variables are represented using de Bruijn indices (the notation ($\ni$) is taken from [McBb]):

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{data}\ (\ni) : (\Gamma : \mathit{Ctxt}) \;\rightarrow\; \mathit{Ty}\ \Gamma \;\rightarrow\; \mathit{Set}\ \mathbf{where}} \\
\quad \mathit{vz} & : & \{\sigma : \mathit{Ty}\ \Gamma\} \rightarrow \Gamma \rhd \sigma \ni \sigma \,/\, \mathit{wk}\ \sigma \\
\quad \mathit{vs} & : \Gamma \ni \tau \;\rightarrow & \{\sigma : \mathit{Ty}\ \Gamma\} \rightarrow \Gamma \rhd \sigma \ni \tau \,/\, \mathit{wk}\ \sigma \\
\quad (::^{\equiv}_{\ni}) & : \Gamma \ni \tau_1 \;\rightarrow & \tau_1 =_{\star} \tau_2 \quad\;\; \rightarrow \Gamma \ni \tau_2
\end{array}$$

The rightmost variable in the context is denoted by *vz* ("variable zero"), and *vs* $v$ is the variable to the left of $v$. The substitution *wk* $\sigma$ is a weakening, taking something in context $\Gamma$ to the context $\Gamma \rhd \sigma$:

$$\mathit{wk} : (\sigma : \mathit{Ty}\ \Gamma) \;\rightarrow\; \Gamma \Rightarrow \Gamma \rhd \sigma$$

The use of weakening is necessary since, for instance, $\sigma$ is a type in $\Gamma$, whereas *vz* creates a variable in $\Gamma \rhd \sigma$.

The constructor ($::^{\equiv}_{\ni}$) is a variant of the conversion rule for variables. It might seem strange that the conversion rule is introduced twice, once for variables and once for terms. However, note that *var* $v$ $::^{\equiv}_{\vdash}$ *eq* is a term and not a variable, so if the conversion rule is needed to show that a variable has a certain type, then ($::^{\equiv}_{\vdash}$) cannot be used.

## 3.5 Substitutions

Substitutions are defined as follows:

$$\begin{array}{l}
\mathbf{data}\ (\Rightarrow) : \mathit{Ctxt} \to \mathit{Ctxt} \to \mathit{Set}\ \mathbf{where} \\
\quad \begin{array}{lllll}
\mathit{sub} & : \Gamma \vdash \tau & & \to & \Gamma \rhd \tau \Rightarrow \Gamma \\
\mathit{wk} & : (\sigma : \mathit{Ty}\ \Gamma) & & \to & \Gamma \Rightarrow \Gamma \rhd \sigma \\
(\uparrow) & : (\rho : \Gamma \Rightarrow \Delta) & \to (\sigma : \mathit{Ty}\ \Gamma) & \to & \Gamma \rhd \sigma \Rightarrow \Delta \rhd (\sigma \mathbin{/} \rho) \\
\mathit{id} & : & & & \Gamma \Rightarrow \Gamma \\
(\odot) & : \Gamma \Rightarrow \Delta & \to \Delta \Rightarrow X & \to & \Gamma \Rightarrow X
\end{array}
\end{array}$$

Single-term substitutions (*sub*), weakenings (*wk*) and liftings ($\uparrow$) have been introduced above. The remaining constructors denote the identity substitution (*id*) and composition of substitutions ($\odot$). The reasons for using this particular definition of ($\Rightarrow$) are outlined in Sect. 3.7.

## 3.6 Equality

The following equalities are defined:

$$\begin{array}{lllll}
(=_{\mathit{Ctxt}}) & : \mathit{Ctxt} & \to \mathit{Ctxt} & \to \mathit{Set} \\
(=_{\star}) & : \mathit{Ty}\ \Gamma_1 & \to \mathit{Ty}\ \Gamma_2 & \to \mathit{Set} \\
(=_{\vdash}) & : \Gamma_1 \vdash \tau_1 & \to \Gamma_2 \vdash \tau_2 & \to \mathit{Set} \\
(=_{\ni}) & : \Gamma_1 \ni \tau_1 & \to \Gamma_2 \ni \tau_2 & \to \mathit{Set} \\
(=_{\Rightarrow}) & : \Gamma_1 \Rightarrow \Delta_1 & \to \Gamma_2 \Rightarrow \Delta_2 & \to \mathit{Set}
\end{array}$$

As mentioned above heterogeneous equality is used. As a sanity check every equality is associated with one or more lemmas like the following one, which states that equal terms have equal types:

$$\mathit{eq}_{\vdash}\mathit{eq}_{\star} : \{t_1 : \Gamma_1 \vdash \tau_1\} \to \{t_2 : \Gamma_2 \vdash \tau_2\} \to t_1 =_{\vdash} t_2 \to \tau_1 =_{\star} \tau_2$$

The context and type equalities are the obvious congruences. For instance, type equality is defined as follows:

$$\begin{array}{l}
\mathbf{data}\ (=_{\star}) : \mathit{Ty}\ \Gamma_1 \to \mathit{Ty}\ \Gamma_2 \to \mathit{Set}\ \mathbf{where} \\
\quad \begin{array}{ll}
\star_{\mathit{Cong}} & : \Gamma_1 =_{\mathit{Ctxt}} \Gamma_2 \to \star\{\Gamma_1\} =_{\star} \star\{\Gamma_2\} \\
\Pi_{\mathit{Cong}} & : \tau_{11} =_{\star} \tau_{12} \to \tau_{21} =_{\star} \tau_{22} \to \Pi\ \tau_{11}\ \tau_{21} =_{\star} \Pi\ \tau_{12}\ \tau_{22} \\
\mathit{El}_{\mathit{Cong}} & : t_1 =_{\vdash} t_2 \to \mathit{El}\ t_1 =_{\star} \mathit{El}\ t_2
\end{array}
\end{array}$$

In many presentations of type theory it is also postulated that type equality is an equivalence relation. This introduces an unnecessary amount of constructors into the data type; when proving something about a data type one typically needs to pattern match on all its constructors. Instead I have chosen to *prove* that every equality (except $(=_\vdash)$) is an equivalence relation:

$$\begin{array}{ll} \mathit{refl}_\star & : \tau \;=_\star\; \tau \\ \mathit{sym}_\star & : \tau_1 =_\star \tau_2 \;\rightarrow\; \tau_2 =_\star \tau_1 \\ \mathit{trans}_\star & : \tau_1 =_\star \tau_2 \;\rightarrow\; \tau_2 =_\star \tau_3 \;\rightarrow\; \tau_1 =_\star \tau_3 \end{array}$$

(And so on for the other equalities.)

The semantics of a variable should not change if a cast is added, so the variable equality is a little different. In order to still be able to prove that the relation is an equivalence the following definition is used:

$$\begin{array}{llllll} \multicolumn{6}{l}{\textbf{data}\ (=_\ni) : \Gamma_1 \ni \tau_1 \;\rightarrow\; \Gamma_2 \ni \tau_2 \;\rightarrow\; \mathit{Set}\ \textbf{where}} \\ \quad \mathit{vz}_{\mathit{Cong}} & : & \sigma_1 =_\star \sigma_2 \;\rightarrow & \mathit{vz}\ \ \{\sigma_1\} & =_\ni & \mathit{vz}\ \ \{\sigma_2\} \\ \quad \mathit{vs}_{\mathit{Cong}} & : v_1 =_\ni v_2 \;\rightarrow & \sigma_1 =_\star \sigma_2 \;\rightarrow & \mathit{vs}\ v_1\,\{\sigma_1\} & =_\ni & \mathit{vs}\ v_2\,\{\sigma_2\} \\ \quad \mathit{castEq}^\ell_\ni & : v_1 =_\ni v_2 & \rightarrow & v_1 ::^\equiv_\ni \mathit{eq} & =_\ni & v_2 \\ \quad \mathit{castEq}^r_\ni & : v_1 =_\ni v_2 & \rightarrow & v_1 & =_\ni & v_2 ::^\equiv_\ni \mathit{eq} \end{array}$$

For substitutions extensional equality is used:

$$\begin{array}{l} \textbf{data}\ (=_\Rightarrow)\ (\rho_1 : \Gamma_1 \Rightarrow \Delta_1)\ (\rho_2 : \Gamma_2 \Rightarrow \Delta_2) : \mathit{Set}\ \textbf{where} \\ \quad \mathit{extEq}\ \ :\ \ \Gamma_1 =_{\mathit{Ctxt}} \Gamma_2 \;\rightarrow\; \Delta_1 =_{\mathit{Ctxt}} \Delta_2 \\ \qquad\quad \rightarrow\; (\forall v_1\ v_2.\ v_1 =_\ni v_2 \;\rightarrow\; \mathit{var}\ v_1 \mathbin{/\!\!\vdash} \rho_1 =_\vdash \mathit{var}\ v_2 \mathbin{/\!\!\vdash} \rho_2) \\ \qquad\quad \rightarrow\; \rho_1 =_\Rightarrow \rho_2 \end{array}$$

Note that this data type contains negative occurrences of $\mathit{Ty}$, $(\ni)$ and $(=_\ni)$, which are defined in the same mutually recursive definition as $(=_\Rightarrow)$. In order to keep this definition strictly positive a first-order variant of $(=_\Rightarrow)$ is used, which simulates the higher-order version by explicitly enumerating all the variables. The first-order variant is later proved equivalent to the definition given here.

Term equality is handled in another way than the other equalities. The presence of the $\beta$ and $\eta$ laws makes it hard to prove that $(=_\vdash)$ is an equivalence relation, and hence this is postulated:

$$\begin{array}{ll} \multicolumn{2}{l}{\textbf{data}\ (=_\vdash) : \Gamma_1 \vdash \tau_1 \;\rightarrow\; \Gamma_2 \vdash \tau_2 \;\rightarrow\; \mathit{Set}\ \textbf{where}} \\ \multicolumn{2}{l}{\qquad \text{-- Equivalence.}} \\ \quad \mathit{refl}_\vdash & : (t : \Gamma \vdash \tau) \;\rightarrow\; t =_\vdash t \\ \quad \mathit{sym}_\vdash & : t_1 =_\vdash t_2 \;\rightarrow\; t_2 =_\vdash t_1 \\ \quad \mathit{trans}_\vdash & : t_1 =_\vdash t_2 \;\rightarrow\; t_2 =_\vdash t_3 \;\rightarrow\; t_1 =_\vdash t_3 \\ \multicolumn{2}{l}{\qquad \text{-- Congruence.}} \\ \quad \mathit{var}_{\mathit{Cong}} & : v_1 =_\ni v_2 \;\rightarrow\; \mathit{var}\ v_1 =_\vdash \mathit{var}\ v_2 \end{array}$$

$$\begin{array}{ll}
\lambda_{Cong} & : t_1 =_{\vdash} t_2 \rightarrow \lambda\ t_1 =_{\vdash} \lambda\ t_2 \\
(@_{Cong}) & : t_{11} =_{\vdash} t_{12} \rightarrow t_{21} =_{\vdash} t_{22} \rightarrow t_{11} @ t_{21} =_{\vdash} t_{12} @ t_{22} \\
(/\!\!\vdash_{Cong}) & : t_1 =_{\vdash} t_2 \rightarrow \rho_1 =_{\Rightarrow} \rho_2 \rightarrow t_1 /\!\!\vdash \rho_1 =_{\vdash} t_2 /\!\!\vdash \rho_2 \\
\quad \text{-- Cast, } \beta \text{ and } \eta \text{ equality.} & \\
castEq_{\vdash} & : t ::^{\equiv}_{\vdash} eq =_{\vdash} t \\
\beta & : (\lambda\ t_1) @ t_2 =_{\vdash} t_1 /\!\!\vdash sub\ t_2 \\
\eta & : \{ t : \Gamma \vdash \Pi\ \tau_1\ \tau_2 \} \rightarrow \lambda\ ((t /\!\!\vdash wk\ \tau_1) @ var\ vz) =_{\vdash} t \\
\quad \text{-- Substitution application axioms.} & \\
\ldots &
\end{array}$$

The $\eta$ law basically states that, if $x$ is not free in $t$, and $t$ is of function type, then $\lambda x.t\ x = t$. The first precondition on $t$ is handled by explicitly weakening $t$, though.

The behaviour of $(/\!\!\vdash)$ also needs to be postulated. The abstraction and application cases are structural; the *id* case returns the term unchanged, and the $(\odot)$ case is handled by applying the two substitutions one after the other; a variable is weakened by applying *vs* to it; substituting $t$ for variable zero results in $t$, and otherwise the variable's index is lowered by one; and finally lifted substitutions need to be handled appropriately:

$$\begin{array}{lllll}
\multicolumn{5}{l}{\textbf{data}\ (=_{\vdash}) : \Gamma_1 \vdash \tau_1 \rightarrow \Gamma_2 \vdash \tau_2 \rightarrow Set\ \textbf{where}} \\
\quad \ldots & & & & \\
\quad \text{-- Substitution application axioms.} & & & & \\
\quad substLam & : \lambda\ t & /\!\!\vdash \rho & =_{\vdash} & \lambda\ (t /\!\!\vdash \rho \uparrow \tau_1) \\
\quad substApp & : (t_1 @ t_2) & /\!\!\vdash \rho & =_{\vdash} & (t_1 /\!\!\vdash \rho) @ (t_2 /\!\!\vdash \rho) \\
\quad idVanishesTm & : t & /\!\!\vdash id & =_{\vdash} & t \\
\quad compSplitsTm & : t & /\!\!\vdash (\rho_1 \odot \rho_2) & =_{\vdash} & t /\!\!\vdash \rho_1 /\!\!\vdash \rho_2 \\
\quad substWk & : var\ v & /\!\!\vdash wk\ \sigma & =_{\vdash} & var\ (vs\ v) \\
\quad substVzSub & : var\ vz & /\!\!\vdash sub\ t & =_{\vdash} & t \\
\quad substVsSub & : var\ (vs\ v) & /\!\!\vdash sub\ t & =_{\vdash} & var\ v \\
\quad substVzLift & : var\ vz & /\!\!\vdash (\rho \uparrow \sigma) & =_{\vdash} & var\ vz \\
\quad substVsLift & : var\ (vs\ v) & /\!\!\vdash (\rho \uparrow \sigma) & =_{\vdash} & var\ v /\!\!\vdash \rho /\!\!\vdash wk\ (\sigma\ /\ \rho)
\end{array}$$

## 3.7 Design Choices

Initially I tried to formalise a language with implicit substitutions, i.e. I tried to implement $(/\!\!\vdash)$ as a function instead of as a constructor. This turned out to be difficult, since when $(/\!\!\vdash)$ is defined as a function many substitution lemmas need to be proved in the initial mutually recursive definition containing all the type families above, and when the mutual dependencies become too complicated it is hard to prove that the code is terminating.

As an example of why substitution lemmas are needed, take the axiom *substApp* above. If *substApp* is considered as a pattern matching equation in the definition of $(/\!\!\vdash)$, then it needs to be modified in order to type check:

$$(t_1 @ t_2) \mathbin{/\!\!\vdash} \rho \;=\; (t_1 \mathbin{/\!\!\vdash} \rho) @ (t_2 \mathbin{/\!\!\vdash} \rho) \mathbin{::^{\equiv}_{\vdash}} \mathit{subCommutes}_{\star}$$

Here $\mathit{subCommutes}_{\star}$ states that, in certain situations, *sub* commutes with other substitutions:

$$\mathit{subCommutes}_{\star} : \tau \mathbin{/} (\rho \uparrow \sigma) \mathbin{/} \mathit{sub}\ (t \mathbin{/\!\!\vdash} \rho) =_{\star} \tau \mathbin{/} \mathit{sub}\ t \mathbin{/} \rho$$

Avoidance of substitution lemmas is also the reason for making the equalities heterogeneous. It would be possible to enforce directly that, for instance, two terms are only equal if their respective types are equal. It suffices to add the type equality as an index to the term equality:

$$(=_{\vdash}) : \{\tau_1 =_{\star} \tau_2\} \;\to\; \Gamma_1 \vdash \tau_1 \;\to\; \Gamma_2 \vdash \tau_2 \;\to\; \mathit{Set}$$

However, in this case *substApp* could not be defined without a lemma like $\mathit{subCommutes}_{\star}$. Furthermore this definition of $(=_{\vdash})$ easily leads to a situation where two equality *proofs* need to be proved equal. These problems are avoided by, instead of enforcing equality directly, proving that term equality implies type equality ($\mathit{eq}_{\vdash}\mathit{eq}_{\star}$) and so on. These results also require lemmas like $\mathit{subCommutes}_{\star}$, but the lemmas can be proved after the first, mutually recursive definition.

The problems described above could be avoided in another way, by postulating the substitution lemmas needed, i.e. adding them as type equality constructors. This approach has not been pursued, as I have tried to minimise the amount of "unnecessary" postulates and definitions.

The postulate *substApp* discussed above also provides motivation for defining $(/)$ as a function, even though $(/\!\!\vdash)$ is a constructor: if $(/)$ were a constructor then $t_1 \mathbin{/\!\!\vdash} \rho$ would not have a $\Pi$ type as required by $(@)$ (the type would be $\Pi\ \tau_1\ \tau_2 \mathbin{/} \rho$), and hence a cast would be required in the definition of *substApp*. I have not examined this approach in detail, but I suspect that it would be harder to work with.

Another important design choice is the basic set of substitutions. The following definition is a natural candidate for this set:

$$\begin{array}{l}
\textbf{data}\ (\tilde{\Rightarrow}) : \mathit{Ctxt} \;\to\; \mathit{Ctxt} \;\to\; \mathit{Set}\ \textbf{where} \\
\quad \emptyset \quad\;\; : \varepsilon \mathrel{\tilde{\Rightarrow}} \Delta \\
\quad (\blacktriangleright) : (\rho : \Gamma \mathrel{\tilde{\Rightarrow}} \Delta) \;\to\; \Delta \vdash \tau \mathbin{/} \rho \;\to\; \Gamma \rhd \tau \mathrel{\tilde{\Rightarrow}} \Delta
\end{array}$$

This type family encodes simultaneous (parallel) substitutions; for every variable in the original context a term in the resulting context is given. So far so good, but the substitutions used in the type signatures above (*sub* and *wk*, for instance) need to be implemented in terms of $\emptyset$ and $(\blacktriangleright)$, and these implementations seem to require various substitution lemmas, again leading to the problems described above.

Note that, even though $(\hat{\Rightarrow})$ is not used to define what a substitution is, the substitutions $\emptyset$ and $(\blacktriangleright)$ can be defined in terms of the other substitutions, and they are used in Sect. 5.3 when value environments are defined.

## 4 Removing Explicit Substitutions

In Sect. 5 a normalisation proof for the lambda calculus introduced in Sect. 3 is presented. The normalisation function defined there requires terms without explicit substitutions ("implicit terms"). This section defines a data type $\mathit{Tm}^{-}$ representing such terms.

The type $\mathit{Tm}^{-}$ provides a view of the $(\vdash)$ terms (the "explicit terms"). Other views will be introduced later, for instance normal forms (Sect. 5.1), and they will all follow the general scheme employed by $\mathit{Tm}^{-}$, with minor variations.

Implicit terms are indexed on explicit terms to which they are, in a sense, $\beta\eta$-equal; the function $\mathit{tm}^{-}\mathit{ToTm}$ converts an implicit term to the corresponding explicit term, and $\mathit{tm}^{-}\mathit{ToTm}\ t^{-} =_{\vdash} t$ for every implicit term $t^{-} : \mathit{Tm}^{-}\ t$:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Tm}^{-} : \Gamma \vdash \tau \ \rightarrow\ \mathit{Set}\ \textbf{where} \\
\quad \begin{array}{lllll}
\mathit{var}^{-} & : (v : \Gamma \ni \tau) & & \rightarrow & \mathit{Tm}^{-}\ (\mathit{var}\ v) \\
\lambda^{-} & : \mathit{Tm}^{-}\ t & & \rightarrow & \mathit{Tm}^{-}\ (\lambda\ t) \\
(@^{-}) & : \mathit{Tm}^{-}\ t_1 & \rightarrow\ \mathit{Tm}^{-}\ t_2 & \rightarrow & \mathit{Tm}^{-}\ (t_1 @ t_2) \\
(::^{\equiv}_{\vdash^{-}}) & : \mathit{Tm}^{-}\ t_1 & \rightarrow\ t_1 =_{\vdash} t_2 & \rightarrow & \mathit{Tm}^{-}\ t_2
\end{array} \\
\mathit{tm}^{-}\mathit{ToTm} : \{\, t : \Gamma \vdash \tau \,\} \ \rightarrow\ \mathit{Tm}^{-}\ t \ \rightarrow\ \Gamma \vdash \tau \\
\begin{array}{ll}
\mathit{tm}^{-}\mathit{ToTm}\ (\mathit{var}^{-}\ v) & = \mathit{var}\ v \\
\mathit{tm}^{-}\mathit{ToTm}\ (\lambda^{-}\ t^{-}) & = \lambda\ (\mathit{tm}^{-}\mathit{ToTm}\ t^{-}) \\
\mathit{tm}^{-}\mathit{ToTm}\ (t_1^{-}\ @^{-}\ t_2^{-}) & = (\mathit{tm}^{-}\mathit{ToTm}\ t_1^{-})\ @\ (\mathit{tm}^{-}\mathit{ToTm}\ t_2^{-})\ ::^{\equiv}_{\vdash}\ \ldots \\
\mathit{tm}^{-}\mathit{ToTm}\ (t^{-}\ ::^{\equiv}_{\vdash^{-}}\ \mathit{eq}) & = \mathit{tm}^{-}\mathit{ToTm}\ t^{-}\ ::^{\equiv}_{\vdash}\ \mathit{eq}_{\vdash}\mathit{eq}_{\star}\ \mathit{eq}
\end{array}
\end{array}
$$

(The ellipsis stands for uninteresting code that has been omitted.)

It would be possible to index implicit terms on types instead. However, by indexing on explicit terms soundness results are easily expressed in the types of functions constructing implicit terms. For instance, the function $\mathit{tmToTm}^{-}$ which converts explicit terms to implicit terms has the type $(t : \Gamma \vdash \tau) \rightarrow \mathit{Tm}^{-}\ t$, which guarantees that the result is $\beta\eta$-equal to $t$. The key to making this work is the cast constructor $(::^{\equiv}_{\vdash^{-}})$, which makes it possible to include equality proofs in an implicit term; without $(::^{\equiv}_{\vdash^-})$ no implicit term could be indexed on $t \mathbin{/\!\!\vdash} \rho$, for instance.

Explicit terms are converted to implicit terms using techniques similar to those in [McBb]. This conversion is not discussed further here.

## 5 Normalisation Proof

This section proves that every explicit term has a normal form. The proof uses normalisation by evaluation (NBE). Type-based NBE proceeds as follows:

- First terms (in this case implicit terms) are evaluated by a function $[\![\cdot]\!]$ (Sect. 5.4), resulting in "values" (Sect. 5.2). Termination issues are avoided by representing function types using the function space of the meta-language.

- Then these values are converted to normal forms by using two functions, often called *reify* and *reflect*, defined by recursion on the (spines of the) types of their arguments (Sect. 5.5).

### 5.1 Normal Forms

Let us begin by defining what a normal form is. Normal forms (actually long $\beta\eta$-normal forms) and atomic forms are defined simultaneously. Both type families are indexed on a $\beta\eta$-equivalent term, just like $\mathit{Tm}^{-}$ (see Sect. 4):

$$
\begin{array}{l}
\textbf{data}\ \mathit{Atom} : \Gamma \vdash \tau \to \mathit{Set}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{var}_{\mathit{At}} & : & (v : \Gamma \ni \tau) \to \mathit{Atom}\ (\mathit{var}\ v) \\
(@_{\mathit{At}}) & : & \mathit{Atom}\ t_1 \to \mathit{NF}\ t_2 \to \mathit{Atom}\ (t_1 @ t_2) \\
(::^{\equiv}_{\mathit{At}}) & : & \mathit{Atom}\ t_1 \to t_1 =_{\vdash} t_2 \to \mathit{Atom}\ t_2
\end{array} \\
\textbf{data}\ \mathit{NF} : \Gamma \vdash \tau \to \mathit{Set}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{atom}^{\star}_{\mathit{NF}} & : & \{t : \Gamma \vdash \star\} \quad\quad\ \to \mathit{Atom}\ t \to \mathit{NF}\ t \\
\mathit{atom}^{\mathit{El}}_{\mathit{NF}} & : & \{t : \Gamma \vdash \mathit{El}\ t'\} \to \mathit{Atom}\ t \to \mathit{NF}\ t \\
\lambda_{\mathit{NF}} & : & \mathit{NF}\ t \to \mathit{NF}\ (\lambda\ t) \\
(::^{\equiv}_{\mathit{NF}}) & : & \mathit{NF}\ t_1 \to t_1 =_{\vdash} t_2 \to \mathit{NF}\ t_2
\end{array}
\end{array}
$$

Note how long $\eta$-normality is ensured by only allowing atoms to be normal forms when they are not of function type; the $t$ argument to the two $\mathit{atom}_{\mathit{NF}}$ constructors has to be of type $\star$ or $\mathit{El}\ t'$ for some $t'$.

A consequence of the inclusion of the cast constructors $(::^{\equiv}_{\mathit{At}})$ and $(::^{\equiv}_{\mathit{NF}})$ is that normal forms are not unique. However, the equality on normal and atomic forms (congruence plus postulates stating that casts can be removed freely) ensures that equality can be decided by erasing all casts and annotations and then checking syntactic equality.

A normal form can be converted to a term in the obvious way, and the resulting term is $\beta\eta$-equal to the index (cf. $\mathit{tm}^{-}\mathit{ToTm}$ in Sect. 4):

$$\begin{aligned}
&\mathit{nfToTm} && : \{ t : \Gamma \vdash \tau \} \to \mathit{NF}\ t \to \Gamma \vdash \tau \\
&\mathit{nfToTmEq} && : (\mathit{nf} : \mathit{NF}\ t) \to \mathit{nfToTm}\ \mathit{nf} =_{\vdash} t
\end{aligned}$$

Similar functions are defined for atomic forms.

We also need to weaken normal and atomic forms. In fact, multiple weakenings will be performed at once. In order to express these multi-weakenings context extensions are introduced. The type $\mathit{Ctxt}^{+}\ \Gamma$ stands for context extensions which can be put "to the right of" the context $\Gamma$ by using $(+\!\!+)$:

$$\begin{aligned}
&\textbf{data}\ \mathit{Ctxt}^{+}\ (\Gamma : \mathit{Ctxt}) : \mathit{Set}\ \textbf{where} \\
&\quad \varepsilon^{+} \quad : \mathit{Ctxt}^{+}\ \Gamma \\
&\quad (\rhd^{+}) : (\Gamma' : \mathit{Ctxt}^{+}\ \Gamma) \to \mathit{Ty}\ (\Gamma +\!\!+ \Gamma') \to \mathit{Ctxt}^{+}\ \Gamma \\
&(+\!\!+) : (\Gamma : \mathit{Ctxt}) \to \mathit{Ctxt}^{+}\ \Gamma \to \mathit{Ctxt} \\
&\Gamma +\!\!+ \varepsilon^{+} \qquad\quad = \Gamma \\
&\Gamma +\!\!+ (\Gamma' \rhd^{+} \tau) = (\Gamma +\!\!+ \Gamma') \rhd \tau
\end{aligned}$$

Now the following type signatures can be understood:

$$\begin{aligned}
&\mathit{wk}^{*} && : (\Gamma' : \mathit{Ctxt}^{+}\ \Gamma) \to \Gamma \Rightarrow \Gamma +\!\!+ \Gamma' \\
&\mathit{wk}^{*}_{\mathit{At}} && : \mathit{Atom}\ t \to (\Gamma' : \mathit{Ctxt}^{+}\ \Gamma) \to \mathit{Atom}\ (t \mathbin{/_{\vdash}} \mathit{wk}^{*}\ \Gamma')
\end{aligned}$$

## 5.2 Values

Values are represented using one constructor for each type constructor, plus a case for casts (along the lines of previously introduced types indexed on terms). Values of function type are represented using meta-language functions:

$$\begin{aligned}
&\textbf{data}\ \mathit{Val} : \Gamma \vdash \tau \to \mathit{Set}\ \textbf{where} \\
&\quad (::_{\mathit{Val}}) : \mathit{Val}\ t_1 \to t_1 =_{\vdash} t_2 \to \mathit{Val}\ t_2 \\
&\quad \star_{\mathit{Val}} \quad : \{ t : \Gamma \vdash \star \} \qquad \to \mathit{Atom}\ t \to \mathit{Val}\ t \\
&\quad \mathit{El}_{\mathit{Val}} \ \ : \{ t : \Gamma \vdash \mathit{El}\ t' \} \to \mathit{Atom}\ t \to \mathit{Val}\ t
\end{aligned}$$

$$\begin{array}{lll}\Pi_{\mathit{Val}} & : & \{\, t_1 : \Gamma \vdash \Pi\ \tau_1\ \tau_2 \,\} \\ & \rightarrow & (f \ : \ (\Gamma' : \mathit{Ctxt}^{+}\ \Gamma) \\ & & \quad \rightarrow \{\, t_2 : \Gamma +\!\!\!+\ \Gamma' \vdash \tau_1 \ / \ \mathit{wk}^{*}\ \Gamma' \,\} \\ & & \quad \rightarrow (v : \mathit{Val}\ t_2) \\ & & \quad \rightarrow \mathit{Val}\ ((t_1 \ /\!\!\!\vdash \ \mathit{wk}^{*}\ \Gamma') @ t_2)) \\ & \rightarrow & \mathit{Val}\ t_1\end{array}$$

The function $f$ given to $\Pi_{\mathit{Val}}\,\{\, t_1 \,\}$ essentially takes an argument value and evaluates $t_1$ applied to this argument. For technical reasons, however, we need to be able to weaken $t_1$ (see *reify* in Sect. 5.5). This makes *Val* look suspiciously like a Kripke model [MM91] (suitably generalised to a dependently typed setting); this has not been verified in detail, though. The application operation of this supposed model is defined as follows. Notice that the function component of $\Pi_{\mathit{Val}}$ is applied to an empty $\mathit{Ctxt}^{+}$ here:

$$\begin{array}{l}(@_{\mathit{Val}}) : \mathit{Val}\ t_1 \rightarrow \mathit{Val}\ t_2 \rightarrow \mathit{Val}\ (t_1 @ t_2) \\ \Pi_{\mathit{Val}}\ f \qquad\quad @_{\mathit{Val}}\ v_2 = f\ \varepsilon^{+}\ (v_2\ ::_{\mathit{Val}}\ \ldots)\ ::_{\mathit{Val}}\ \ldots \\ (v_1\ ::_{\mathit{Val}}\ \mathit{eq})\ @_{\mathit{Val}}\ v_2 = (v_1\ @_{\mathit{Val}}\ (v_2\ ::_{\mathit{Val}} \ldots))\ ::_{\mathit{Val}}\ \ldots\end{array}$$

The transition function of the model weakens values:

$$\mathit{wk}^{\star}_{\mathit{Val}} : \mathit{Val}\ t \rightarrow (\Gamma' : \mathit{Ctxt}^{+}\ \Gamma) \rightarrow \mathit{Val}\ (t \ /\!\!\!\vdash \ \mathit{wk}^{*}\ \Gamma')$$

Note that *Val* is not a positive data type, due to the negative occurrence of *Val* inside of $\Pi_{\mathit{Val}}$, so this data type is not part of the treatment in [DS06]. In practise this should not be problematic, since the type index of that occurrence, $\tau_1 \ / \ \mathit{wk}^{*}\ \Gamma'$, is smaller than the type index of $\Pi_{\mathit{Val}}\ f$, which is $\Pi\ \tau_1\ \tau_2$. Here we count just the *spine* of the type, ignoring the contents of *El*, so that $\tau$ and $\tau \ / \ \rho$ have the same size, and two equal types also have the same size. In fact, by supplying a spine argument explicitly it should not be difficult to define *Val* as a structurally recursive function instead of as a data type.

## 5.3 Environments

The function $[\![\cdot]\!]$, defined in Sect. 5.4, makes use of environments, which are basically substitutions containing values instead of terms:

$$\begin{array}{l}\mathbf{data}\ \mathit{Env} : \Gamma \Rightarrow \Delta \rightarrow \mathit{Set}\ \mathbf{where} \\ \quad \emptyset_{\mathit{Env}} \quad\ \ : \mathit{Env}\ \emptyset \\ \quad (\blacktriangleright_{\mathit{Env}}) : \mathit{Env}\ \rho \rightarrow \mathit{Val}\ t \rightarrow \mathit{Env}\ (\rho \blacktriangleright t) \\ \quad (::^{\equiv}_{\mathit{Env}}) : \mathit{Env}\ \rho_1 \rightarrow \rho_1 =_{\Rightarrow} \rho_2 \rightarrow \mathit{Env}\ \rho_2\end{array}$$

Note that the substitutions $\emptyset$ and $(\blacktriangleright)$ from Sect. 3.7 are used as indices here.

It is straight-forward to define functions for looking up a variable in an environment and weakening an environment:

$$
\begin{aligned}
&\mathit{lookup} : (v : \Gamma \ni \tau) \rightarrow \mathit{Env}\ \rho \rightarrow \mathit{Val}\ (\mathit{var}\ v \mathbin{/\!\!\vdash} \rho)\\
&\mathit{wk}^{\star}_{\mathit{Env}} : \mathit{Env}\ \rho \rightarrow (\Delta' : \mathit{Ctxt}^{+}\ \Delta) \rightarrow \mathit{Env}\ (\rho \odot \mathit{wk}^{*}\ \Delta')
\end{aligned}
$$

## 5.4 Evaluating Terms

Now we can evaluate an implicit term, i.e. convert it to a value. The most interesting case is $\lambda^{-}\ t_1^{-}$, where $t_1^{-}$ is evaluated in an extended, weakened environment:

$$
\begin{aligned}
&[\![\cdot]\!] : \mathit{Tm}^{-}\ t \rightarrow \mathit{Env}\ \rho \rightarrow \mathit{Val}\ (t \mathbin{/\!\!\vdash} \rho)\\
&[\![\mathit{var}^{-}\ v]\!]\,\gamma \quad = \mathit{lookup}\ v\ \gamma\\
&[\![t_1^{-}\ @^{-}\ t_2^{-}]\!]\,\gamma \ = ([\![t_1^{-}]\!]\,\gamma\ @_{\mathit{Val}}\ [\![t_2^{-}]\!]\,\gamma)\ ::_{\mathit{Val}}\ \ldots\\
&[\![t^{-}\ ::^{\equiv}_{\vdash^{-}}\ eq]\!]\,\gamma = [\![t^{-}]\!]\,\gamma\ ::_{\mathit{Val}}\ \ldots\\
&[\![\lambda^{-}\ t_1^{-}]\!]\,\gamma \quad =\\
&\quad \Pi_{\mathit{Val}}\,(\backslash\Delta'\ v_2 \rightarrow [\![t_1^{-}]\!]\,(\mathit{wk}^{\star}_{\mathit{Env}}\ \gamma\ \Delta' \blacktriangleright_{\mathit{Env}}\ (v_2\ ::_{\mathit{Val}}\ldots))\ ::_{\mathit{Val}}\ \ldots\beta\ldots)
\end{aligned}
$$

(The notation $\backslash x \rightarrow \ldots$ is lambda abstraction in the meta-language.) It would probably be straightforward to evaluate explicit terms directly, without going through implicit terms (cf. [Coq02]). Here I have chosen to separate these two steps, though.

## 5.5 *reify* and *reflect*

Let us now define *reify* and *reflect*. These functions are implemented by recursion over spines (see Sect. 5.2), in order to make them structurally recursive, but to avoid clutter the spine arguments are not written out below.

The interesting cases correspond to function types, for which *reify* and *reflect* use each other recursively. Notice how *reify* applies the function component of $\Pi_{\mathit{Val}}$ to a singleton $\mathit{Ctxt}^{+}$, to enable using the reflection of variable zero, which has a weakened type; this is the reason for including weakening in the definition of $\Pi_{\mathit{Val}}$:

$$
\begin{aligned}
&\mathit{reify} : (\tau : \mathit{Ty}\ \Gamma) \rightarrow \{t : \Gamma \vdash \tau\} \rightarrow \mathit{Val}\ t \rightarrow \mathit{NF}\ t\\
&\mathit{reify}\ \_\ (v\ ::_{\mathit{Val}}\ eq) = \mathit{reify}\ \_\ v\ ::_{\mathit{NF}}\ eq\\
&\mathit{reify}\ \_\ (\star_{\mathit{Val}}\ \ at) \quad\ = \mathit{atom}^{\star}_{\mathit{NF}}\ at\\
&\mathit{reify}\ \_\ (\mathit{El}_{\mathit{Val}}\ at) \quad = \mathit{atom}^{\mathit{El}}_{\mathit{NF}}\ at
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{reify}\ (\Pi\ \tau_1\ \tau_2)\ (\Pi_{\mathit{Val}}\ f) = \\
\quad \lambda_{\mathit{NF}}\ (\mathit{reify}\ (\tau_2\ /\ \_\ /\ \_) \\
\qquad\qquad (f\ (\varepsilon^+\ \triangleright^+\ \tau_1)\ (\mathit{reflect}\ (\tau_1\ /\ \_)\ (\mathit{var}_{\mathit{At}}\ \mathit{vz})\ ::_{\mathit{Val}}\ \ldots))) \\
\quad ::_{\mathit{NF}} \ldots \eta \ldots
\end{array}
$$

(Above underscores (_) have been used instead of giving non-hidden arguments which can be inferred automatically by the AgdaLight type checker.) The companion function *reflect* uses weakening and $\Pi_{\mathit{Val}}$ in the function type case:

$$
\begin{array}{l}
\mathit{reflect} : (\tau : \mathit{Ty}\ \Gamma)\ \rightarrow\ \{t : \Gamma \vdash \tau\}\ \rightarrow\ \mathit{Atom}\ t\ \rightarrow\ \mathit{Val}\ t \\
\mathit{reflect}\ \star \qquad\qquad \mathit{at} = \star_{\mathit{Val}}\ \ \mathit{at} \\
\mathit{reflect}\ (\mathit{El}\ t) \qquad \mathit{at} = \mathit{El}_{\mathit{Val}}\ \mathit{at} \\
\mathit{reflect}\ (\Pi\ \tau_1\ \tau_2)\ \mathit{at} = \Pi_{\mathit{Val}}\ (\backslash \Gamma'\ v\ \rightarrow \\
\quad \mathit{reflect}\ (\tau_2\ /\ \_\ /\ \_)\ (\mathit{wk}^*_{\mathit{At}}\ \mathit{at}\ \Gamma'\ @_{\mathit{At}}\ \mathit{reify}\ (\tau_1\ /\ \_)\ v))
\end{array}
$$

## 5.6 Normalisation

After having defined $[\![\cdot]\!]$ and *reify* it is very easy to normalise a term. First we build an identity environment by applying *reflect* to all the variables in the context:

$$\mathit{id}_{\mathit{Env}} : (\Gamma : \mathit{Ctxt})\ \rightarrow\ \mathit{Env}\ (\mathit{id}\ \Gamma)$$

Then an explicit term can be normalised by converting it to an implicit term, evaluating the result in the identity environment, and then reifying:

$$
\begin{array}{l}
\mathit{normalise} : (t : \Gamma \vdash \tau)\ \rightarrow\ \mathit{NF}\ t \\
\mathit{normalise}\ t = \mathit{reify}\ \_\ ([\![\mathit{tmToTm}^-\ t]\!]\ (\mathit{id}_{\mathit{Env}}\ \_)\ ::_{\mathit{Val}} \ldots)
\end{array}
$$

Since a normal form is indexed on an equivalent term it is easy to show that *normalise* is sound:

$$
\begin{array}{l}
\mathit{normaliseEq} : (t : \Gamma \vdash \tau)\ \rightarrow\ \mathit{nfToTm}\ (\mathit{normalise}\ t) =_{\vdash} t \\
\mathit{normaliseEq}\ t = \mathit{nfToTmEq}\ (\mathit{normalise}\ t)
\end{array}
$$

If this normalising function is to be really useful (as part of a type checker, for instance) it should also be proved, for the normal form equality ($=_{\mathit{NF}}$), that $t_1 =_{\vdash} t_2$ implies that $\mathit{normalise}\ t_1\ =_{\mathit{NF}}\ \mathit{normalise}\ t_2$. This is left for future work, though.

## 6 Related Work

As stated in the introduction Dybjer's formalisation of categories with families [Dyb96] seems to be the only prior example of a formalisation of a dependently typed language done using the internal approach to handle the type system invariants. Other formalisations of dependently typed languages, such as McKinna/Pollack [MP99] and Barras/Werner [BW97], have used the external approach. There is also an example, due to Adams [Ada04], of a hybrid approach which handles some invariants internally, but not the type system.

Normalisation by evaluation (NBE) seems to have been discovered independently by Martin-Löf (for a version of his type theory) [ML75] and Berger and Schwichtenberg (for simply typed lambda calculus) [BS91]. Martin-Löf has also defined an NBE algorithm for his logical framework [ML04], and recently Dybjer, Abel and Aehlig have done the same for Martin-Löf type theory with one universe [AAD07].

NBE has been formalised, using the internal approach, by T. Coquand and Dybjer, who treated a combinatory version of Gödel's System T [CD97]. C. Coquand has formalised normalisation for a simply typed lambda calculus with explicit substitutions, also using the internal approach [Coq02]. Her normalisation proof uses NBE and Kripke models, and in that respect it bears much resemblance to this one. McBride has implemented NBE for the untyped lambda calculus [McBa]. His implementation uses an internal approach (nested types in Haskell) to ensure that terms are well-scoped, and that aspect of his code is similar to mine.

My work seems to be the first formalised NBE algorithm for a dependently typed language.

## 7 Discussion

I have presented a formalisation of a dependently typed lambda calculus, including a proof of normalisation, using the internal approach to handle typing rules. This formalisation demonstrates that, at least in this case, it is feasible to use the internal approach when programming with invariants strong enough to encode the typing rules of a dependently typed language. How this method compares to other approaches is a more difficult question, which I do not attempt to answer here.

## Acknowledgements

I am grateful to Ulf Norell, who has taught me a lot about dependently typed programming, discussed many aspects of this work, and fixed many of the bugs in AgdaLight that I have reported. I would also like to thank Thierry Coquand and Peter Dybjer for interesting discussions and useful pointers, and Patrik Jansson, Anton Setzer and the anonymous referees for helping me with the presentation.

## References

[AA02] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, 2002.

[AAD07] Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with one universe. In *MFPS XXIII*, volume 173 of *ENTCS*, pages 17–39, 2007.

[AC06] Thorsten Altenkirch and James Chapman. Tait in one big step. In *MSFP 2006*, 2006.

[Ada04] Robin Adams. Formalized metatheory with terms represented by an indexed family of types. In *TYPES 2004*, volume 3839 of *LNCS*, pages 1–16, 2004.

[AR99] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL '99*, volume 1683 of *LNCS*, pages 453–468, 1999.

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed $\lambda$-calculus. In *LICS '91*, pages 203–211, 1991.

[BW97] Bruno Barras and Benjamin Werner. Coq in Coq. Unpublished manuscript, 1997.

[CD97] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1):75–94, 1997.

[Coq02] Catarina Coquand. A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. *Higher-Order and Symbolic Computation*, 15:57–90, 2002.

[Dan07] Nils Anders Danielsson. Personal web page. Available at `http://www.cs.chalmers.se/~nad/`, 2007.

[DS06] Peter Dybjer and Anton Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 66(1):1–49, 2006.

[Dyb96] Peter Dybjer. Internal type theory. In *TYPES '95*, volume 1158 of *LNCS*, pages 120–134, 1996.

[McBa] Conor McBride. Beta-normalization for untyped lambda-calculus. Unpublished program.

[McBb] Conor McBride. Type-preserving renaming and substitution. Unpublished manuscript.

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

[ML04] Per Martin-Löf. Normalization by evaluation and by the method of computability. Lecture series given at Logikseminariet Stockholm–Uppsala, 2004.

[MM91] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51:99–124, 1991.

[MM04] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[MP99] James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3):373–409, 1999.

[MW06] James McKinna and Joel Wright. A type-correct, stack-safe, provably correct expression compiler in Epigram. Accepted for publication in the Journal of Functional Programming, 2006.

[Nor07] Ulf Norell. AgdaLight home page. Available at `http://www.cs.chalmers.se/~ulfn/agdaLight/`, 2007.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory, An Introduction.* Oxford University Press, 1990.

[PL04] Emir Pašalić and Nathan Linger. Meta-programming with typed object-language representations. In *GPCE 2004*, volume 3286 of *LNCS*, pages 136–167, 2004.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL '03*, pages 224–235, 2003.