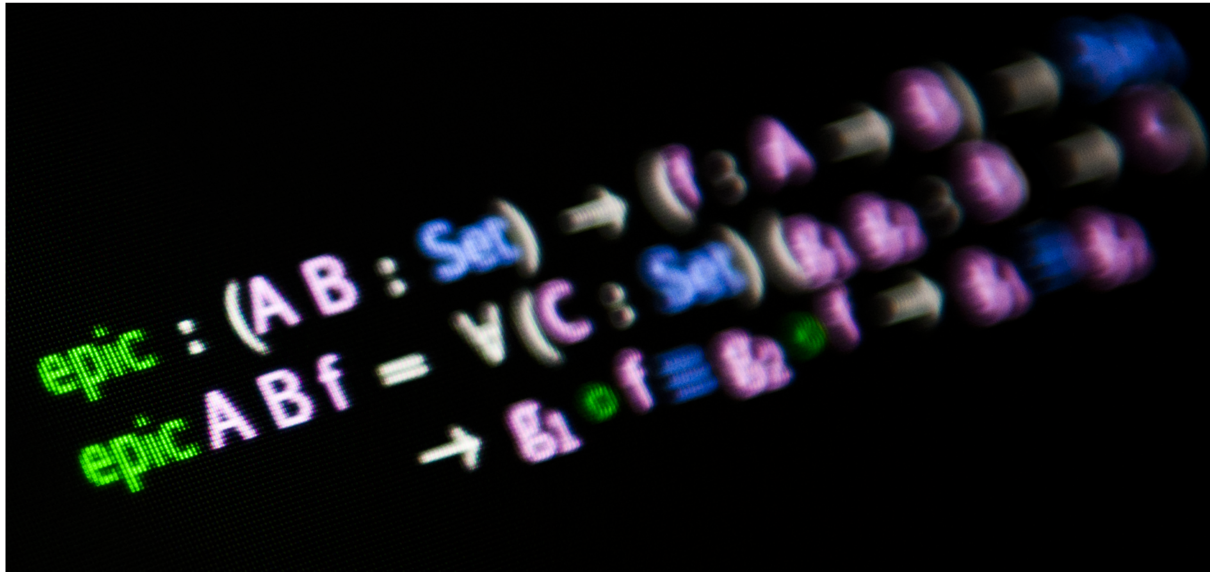
CHALMERS



A totally Epic backend for Agda

Master of Science Thesis in the Programme Computer Science: Algorithms, Languages and Logic

OLLE FREDRIKSSON

DANIEL GUSTAFSSON

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Department of Computer Science and Engineering
Göteborg, Sweden, May 2011

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

A totally Epic backend for Agda

OLLE FREDRIKSSON,
DANIEL GUSTAFSSON,

© OLLE FREDRIKSSON, May 2011.
© DANIEL GUSTAFSSON, May 2011.

Examiner: PETER DYBJER

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, May 2011

Abstract

Agda is a programming language that utilises dependent types which add the power to express properties about terms very precisely, but this also introduces a runtime performance overhead. This thesis presents a new compiler backend for Agda targeting Epic with the aim to use the types for optimising programs and for removing the overhead.

One source of inefficiencies is in the data representation. Two ways to remove these inefficiencies are presented: Forcing, which deletes fields in constructors that may be inferred, and representing datatypes as primitive data, which is done by using machine integers for types on a certain form.

Many terms in dependently typed languages have no computational content (observable at runtime). Three optimisations for getting rid of such terms are presented: Erasure, which erases types and unused arguments to functions, smashing, which replaces inefficient computations with a result that is the only observable one, and injection detection, which detects inefficient identity functions so that they can be replaced by faster versions.

The results of doing the optimisations are benchmarked, and in some examples the runtime of an optimised program compared to an unoptimised one is almost one third and the produced executable size is halved. The code that is produced is also closer to efficient code written without using sexy types, and it does not take as long to compile it.

This work makes it more feasible to write programs usable in the "real world" using Agda, and shows that it is perfectly possible to use dependently typed languages for programming.

Acknowledgements

We would like to thank everyone who has helped make this thesis possible. Our supervisor Ulf Norell for the thesis idea and for all his help during its creation, Stevan Andjelkovic for valuable lunch discussions around the project, and Edwin Brady for creating Epic and being helpful in correspondences around its details.

Lastly, we would like to thank our families, without whom the project would certainly have been impossible.

Olle Fredriksson & Daniel Gustafsson, Göteborg, May 2011

Contents

1 Introduction **1**
1.1 Overview 2

2 Background **3**
2.1 Agda by example 3
2.2 Examples 7
2.3 Strong normalisation 8
2.4 Target language 9
2.5 Epic 10
2.6 Related work 12

3 Optimisations **15**
3.1 Compiler overview 15
3.2 Forcing 16
3.3 Erasure 20
3.4 Injection detection 23
3.5 Smashing 26
3.6 Primitive data 27
3.7 Partial evaluation 29

4 Results **31**
4.1 Forcing and primitive data 31
4.2 Erasure 32
4.3 Injection detection 35
4.4 Smashing 37
4.5 Case study: A lambda calculus 38
4.6 Case study: A file access DSL 41
4.7 Benchmarks 45

5 Discussion **50**
5.1 Forcing 50
5.2 Erasure 51
5.3 Injection detection 52
5.4 Smashing 54
5.5 Primitive data 55
5.6 Partial evaluation 56
5.7 Laziness 57
5.8 Target language 58

6 Conclusion **59**

A Foreign function interface **61**

B Haskell lambda calculus **63**

1

Introduction

Ensuring that a computer program is correct is increasingly important in our society, as large parts of our infrastructures are being computerised and errors in programs can be catastrophic. A computer program is here said to be correct if it works according to a given specification. There are several approaches to making sure that a program does so. One possibility is testing; a program that outputs the correct value for a large number of inputs can be trusted to be correct to a certain degree. In some cases it is even feasible to try *all* possible inputs (but this is not true in general).

Another approach to correctness is using programming languages where the set of valid programs is limited. This limitation can be acquired by providing a way to formally express specifications by giving the *type* of computations. These types dictate what forms of input are valid to a program, and what the form of the output will be given inputs on the right form. The programs can then be checked by a type checker to ensure that they do have the correct type. For example, a program that expects text as input but is given an integer number may be rejected. Type systems thus relate values (a string of text, a number) to their type of value (the type of strings of text, the type of numbers).

Type systems differ in what programs they allow and in how precisely the type specifications can be given. This can be a double-edged sword: on the one hand more bugs may be caught if fewer programs are allowed, but on the other hand the programmer's job becomes harder.

One particularly expressive kind of type system is one that allows *dependent* types, which essentially means that a type may depend on a value and blurs the distinction between types and values. There has been an increasing interest in dependently typed programming recently. For instance, a large number of the accepted papers at the International Conference on Functional Programming 2010 deal with it in some way. Dependent types allow expressing extremely precise invariants and can even be used to prove properties about programs, making it possible to write programs that cannot fail and can be fully trusted to be correct[1].

One dependently typed language is Agda [Nor07], which is the focus of this thesis. Agda is a functional language with a syntax that is similar to the programming language Haskell's [P+03] syntax. The current Agda implementation is geared towards making sure that programs and proofs are

[1] Assuming the compiler, operating system and hardware are correct as well.

correct by type checking them, and it also has good tool support making it a practical proof assistant. However, it appears that most users do not go further to actually produce stand-alone executable programs. This is unfortunate, in our opinion, as Agda is not just a proof assistant but also a very powerful programming language.

In this thesis a new compiler backend for Agda is presented. The aim is to make Agda more usable as a programming language by producing efficient executable programs. In the context of dependent types, programs may be more complicated than their counterparts written in languages with less sophisticated type systems, as using the type system to its full capacity also means writing out details of programs more precisely to convince the type checker that the program does in fact follow its specification.

The work presented is aimed to make these programs run faster (to *optimise* them) even though they are written with more precise detail. This is done by exploiting knowledge about programs, their data representations and which computations actually have an effect on the outcome of a program at runtime.

A program that computes its result according to the specification but does it faster and using fewer resources than another is something that is always desirable. A faster program of course makes it possible to get the results in less time, but as it may use fewer resources it also makes it possible to take on more complex problems, and additionally has the potential to reduce power consumption.

The main contribution of this thesis is showing how optimisations specific to programming languages with dependent types can be carried out in a practical implementation of a compiler backend for the language.

1.1 Overview

The rest of the thesis starts out by giving the necessary background knowledge by introducing the compiler's source language, Agda, its target language, Epic, and some related work. This is done in chapter 2.

The background leads up to chapter 3 where the different optimisation techniques are presented. In this chapter it is shown how to go from naïvely compiled code to more efficient code, by exploiting knowledge about both the source and the target language. Optimisations to the data representation as well as terms are presented.

After the optimisations have been introduced, the results of applying them are showcased. This is done in chapter 4 by exemplifying how code is compiled, performing case studies and giving benchmark results. The results are discussed and suggestions for further improvements and future work are presented in chapter 5. Lastly the work is concluded in chapter 6.

2

Background

THIS chapter will give the background necessary to understand the rest of the thesis. It first introduces the Agda programming language. In section 2.4 a small overview of target language alternatives is given, leading up to section 2.5 which introduces Epic, the target language of the compiler that is presented. Readers that are already familiar with these languages may skip parts of this chapter.

2.1 Agda by example

Data definitions in Agda are written in a style similar to Haskell GADTs[1] [PWW04]. One difference is that Agda uses a single colon instead of double colons (: instead of ::) for type annotations. The following is the inductive definition of Peano style natural numbers $\mathbb{N}$. It is constructed using two constructors: zero denotes the number 0, and suc which can be used to construct $n + 1$ given a number n.

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

In this representation the natural number 2 is suc (suc zero).

Set is used as the type of small types in Agda, corresponding roughly to the kind * in Haskell.

Agda supports inductive families [Dyb91] which is the inspiration for Haskell GADTs. In these families each constructor may have a different result type. The language is **dependently typed**, which means that types may depend on values (of another type). Instead of function types on the form A → B, this allows the type (x : A) → B (where x may appear in the type B) meaning that the return type depends on the value of the argument x. This allows expressing more precisely what the valid inputs to a function are. For example, the vector type defined below depends on a natural number which states its length.

```
data Vec (A : Set) : ℕ → Set where
  []    : Vec A zero
  _::_  : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

[1] Generalised Algebraic Datatypes.

Note that constructors and identifiers in general do not have any lexical restriction (unlike Haskell). It is perfectly valid to mix symbols and alphanumeric characters in an identifier; whitespace is used to separate identifiers. As such the programmer has to be more generous with whitespace than in most other languages.

In the above declaration (A : Set) is a **parameter** to Vec which means that the identifier A is brought into scope and corresponds to something of type Set in all constructors (and the rest of the type signature of Vec). Another difference to the definition of natural numbers is that the type of Vec A is not Set but ℕ → Set, meaning that Vec A is a family of Set indexed by a natural number ℕ.

In the definition of the constructor for empty vectors, [] : Vec A zero, the index is zero : ℕ. This is because it is indexed by length and the empty vector has length zero. In the definition of the _::_ constructor the return type is Vec A (suc n). Here n is bound in the implicit function space {n : ℕ} → .... The difference between functions of type f : (x : A) → B x (parentheses) and g : {x : A} → B x (curly braces) is that Agda will try to infer what argument to pass to g while the programmer has to supply the argument to f. Thus g takes **implicit** arguments[2] while f takes **explicit** arguments. The length of a vector constructed with _::_ which adds an element in front of of another vector is one more than the length of that vector.

Agda allows the definition of mixfix operators by using the underscore character. The underscores are used as holes for arguments during parsing. x :: xs is equivalent to _::_ x xs but gives more flexibility in the notation. The programmer is not restricted to just two underscores but can define functions like if_then_else_ and use it as if p then x else y.

To write the lookup function for vectors, it is convenient to first define a type of bounded numbers Fin n, representing a natural number smaller than n : ℕ:

```
data Fin : ℕ → Set where
  fz : {n : ℕ} →          Fin (suc n)
  fs : {n : ℕ} → Fin n → Fin (suc n)
```

The intended meaning of a term of this type is similar to that of ℕ. For example, fs (fs fz) denotes the natural number 2 (compare suc (suc zero)). The difference is that the type is also indexed by an argument of type ℕ that is always bigger than the number that is being represented. The fz constructor states that the number 0 is less than $n + 1$ for any natural number n. The fs constructor states that the number $x + 1$ is less than $n + 1$ if x is less than n.

[2]If necessary g can be applied to an explicit argument using the g {e} syntax.

With this type it is possible to write the lookup function _!_, by performing pattern matching in a style similar to Haskell's:

```
_!_ : ∀ {A n} → Vec A n → Fin n → A
[]        ! ()
(x :: xs) ! fz     = x
(x :: xs) ! (fs i) = xs ! i
```

First some explanation of the function's type. The forall sign ∀ is used to make Agda infer the type of the arguments. The ∀ {A n} is equivalent to {A : _} {n : _} where a single underscore _ is used to tell Agda to infer this term. Here it will be inferred to be {A : Set} {n : ℕ}. This latter syntax is also a convenient shorthand allowing the programmer to give a list of types and bind their terms to names. It is called a telescope, and is syntactic sugar for {A : Set} → {n : ℕ} → ...

In the first clause of _!_ the first argument is matched against [] which means that n is zero. Therefore the second argument is of type Fin zero. But no constructor of Fin has zero as the index, which means that the clause is impossible. This is called an impossible pattern and is marked with (). How did Agda detect that no constructor of Fin have zero as index? The answer is unification. In this case both constructors of Fin have Fin (suc n) as the return type. For the types to unify an equation on the form Fin zero = Fin (suc n) has to hold. This equation can however never hold since constructors of inductive types are injective[3], meaning that zero and suc n are different.

In order to unify (or as in this case refute) these types the type checker may have to reduce an open term, i.e a term containing unknown variables, into weak head normal form (in this case into a constructor application). If the type was Vec A n → Fin (f n) → A for some function f : ℕ → ℕ for example, the type checker would have to normalise f zero when matching against [] and f (suc m) for some m in the case of _::_. This is a consequence of having terms in types (dependent types) – the type checker has to perform normalisation on open terms.

Dot patterns When working with dependent types it is expected that sometimes the types actually do depend on each other. Take vectors for example; when matching on them something is learnt about their length. This is illustrated by the map function that is explicit about the natural number used as the vector's length index:

```
map : {A B : Set} → (n : ℕ) → (A → B) → Vec A n → Vec B n
map zero    f []                = []
map (suc n) f (_::_ {n} x xs) = f x :: map n f xs
```

[3]This applies to both type and data constructors.

This pattern matching is problematic since the pattern is no longer linear (as they must be in Haskell) due to multiple occurrences of the variable n on the left-hand side. But the natural number, which states the length of the vector, has to be zero when the vector is empty, and suc n when the vector is not. So the matching on the natural number is not necessary as only one possibility exists. This is expressed using a dot pattern:

```
map : {A B : Set} → (n : ℕ) → (A → B) → Vec A n → Vec B n
map .zero     f []                  = []
map .(suc n) f (_::_ {n} x xs)  = f x :: map n f xs
```

Here it is clear that the natural number has that particular form, not by pattern matching but due to types. Inside the dot pattern is an expression, not a pattern, which states what the value of that argument is known to be. The representation of the inverse of a function is an example where the type checker will find an expression which would not be a valid pattern:

```
data Inv {A B : Set} (f : A → B) : B → Set where
  inv : (x : A) → Inv f (f x)
invert : {A B : Set} → (f : A → B) → (y : B) → Inv f y → A
invert f .(f x) (inv x) = x
```

The type Inv f y is a type that represents the inverse of f at y; the value in its constructor is an x such that y = f x. The function invert picks out this x and returns it. The interesting bit is the y argument, which is known to be equal to f x which is expressed by the dot pattern, as f x is certainly not a valid pattern by itself.

with clauses Sometimes it is useful to perform pattern matching on an expression involving a function's bound arguments. In these cases a **with** clause may be used. This construct is akin to pattern guards in Haskell [EJ00] since new variables may be bound. For example here is a way to define a minimum function:

```
minimum : {n : ℕ} → Vec ℕ (suc n) → ℕ
minimum xs with sort xs
minimum xs | y :: ys = y
```

The **with** is written after the patterns to add a new expression that the following lines can pattern match on, and a | is used to mark that the next pattern comes from a **with**. In this case no further pattern matching is done on the previous pattern xs, which is such a common case that there is special syntax for when only adding pattern matching on the new terms introduced. Before the vertical bar delimiter three dots ... are written to indicate that it is the same pattern as above:

```
minimum : {n : ℕ} → Vec ℕ (suc n) → ℕ
minimum xs with sort xs
... | y :: ys = y
```

2.2 Examples

In this section two examples of dependent types are given. The first example shows a problem in GUI[4] application frameworks that can be solved nicely with dependent types, and the second shows that a dependently typed programming language can be used to perform formal proofs.

GUI A real world example of dependent types is GUI applications with events. In these systems it is common to have the possibility of registering callback functions for when an event is raised (e.g when a button is clicked or a window is redrawn). A problem with this approach is that the callback functions need different data depending on what kind of event they are connected to. For example, the callback for the button click may want to know which button was pressed, whereas a window redraw event may contain a window element and coordinates of the area to be redrawn. The callback functions do not have the same type in general, which would imply that a different register function is needed for every type of callback. It is not necessarily so, as this can be solved nicely with dependent types:

```
data Event : Set where
  ButtonPressed  : Event
  WindowRedraw   : Event
  ⋮
CallBack : Event → Set
CallBack ButtonPressed  = Button                          → IO ⊤
CallBack WindowRedraw   = (w : Window) → Coordinate w → IO ⊤
⋮
```

CallBack is a function that takes an event and returns the type of a callback function for that event. With these it is possible to give the type of a register function that will work for all the different events:

```
register : (e : Event) → CallBack e → IO ⊤
register = ...
```

The type of register WindowRedraw is ((w : Window) → Coordinate w → IO ⊤) → IO ⊤ so the function register will only get callback functions of the correct type and it is guaranteed by the type checker.

[4]Graphical User Interface.

Proofs One of the interesting aspects of dependent types is that through the lens of the Curry-Howard isomorphism (which relates type theory to logic), programs are proofs. For example dependent functions correspond to intuitionistic universal quantification. The other quantifier in intuitionistic logic, $\exists x \in A.P(x)$, is interpreted as a pair of an actual witness x and a proof that $P(x)$ holds. It is possible to define Σ, which will act as existential quantification in Agda, as follows:

```
data Σ (A : Set) (P : A → Set) : Set where
  _,_ : (x : A) → P x → Σ A P
```

A language with dependent types does not only have to be used as a programming language, but can also be used as an assistant for performing formal proofs.

An important datatype for proofs is the equality datatype, defined as follows:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

The datatype has both a type parameter of type A, called x, and a type index of type A (which comes after the colon). The refl (for reflexive) constructor fixes the type index to be the same as the parameter. In order for a type a ≡ b to unify with the type of the refl constructor (making it valid to pattern match on the term of type a ≡ b) a has to be equal to b according to Agda's internal definition of equality.

Using this datatype the proof that the addition operation _+_ on natural numbers ℕ is commutative gets the following type[5]:

```
+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm m n = ...
```

2.3 Strong normalisation

Since arbitrary Agda terms can occur in a type, and the type checker evaluates terms in types, termination of the evaluation of open terms is of interest. There are several choices to make regarding checking termination:

- Allow arbitrary terms and limit the number of evaluation steps the type checker may perform. This is the approach taken by the Cayenne language [Aug98].

[5] The implementation is omitted here.

- Program only with eliminators for inductive types, and prove that evaluation with eliminators always terminates. This is how Epigram [MM04] works. By introducing views it is still possible to get something similar to ordinary pattern matching.
- Run an external checker that checks that the terms are terminating. This is usually done by checking if recursive calls are always performed on something structurally smaller than their current values. Agda uses this approach, and a similar syntactic check is also used in Coq [Tea09].
- Using sized types [Abe10] in which all types also contain an upper bound on the size of data. When making recursive calls this size has to decrease. This is similar to checking that the argument is structurally smaller, but instead of making a syntactic check a more semantic comparison is used.
- Allowing arbitrary terms, but recording termination effects in the types. These effects can either be *total* for terminating terms, or *general* for terms which have not yet been proven to terminate. The type checker only evaluates terms that are marked as *total*. When the programmer wants a *total* term but only has a *general* one the term has to be cast using a termination cast [SSW10] provided that the programmer can prove that the term actually terminates.

Strong normalisation is a property of programs that helps when writing a compiler. As there are no computations that loop endlessly, evaluation order does not affect the outcome of running a program, and the meaning of a program is not changed depending on if it is lazily or strictly evaluated. This also allows the compiler to remove computations of results that it is not interested in. Optimisations like this are not always possible in ordinary languages since the computation may be non-terminating, and it is not semantically valid for an optimiser to make an non-terminating program terminating.

2.4 Target language

An important decision to make when creating a compiler is what language to compile to – picking a target language. Compiling to machine code is not a viable solution in most cases as a new backend would have to be created for each target architecture. Thus, there are two families of languages that are good candidates:

Low-level languages One alternative is to pick a systems programming language such as C or LLVM as the target language. As C and LLVM compile to most architectures a cross-platform compiler is gotten for free.

The compilers for these languages are also mature, producing optimised machine code. The only problem is that these imperative languages are far from functional languages in terms of semantics, and it is quite a task just to get partial function applications (closures) working, which is something that is taken for granted in functional languages.

High(er)-level languages Using a target language which is close to the source language makes the implementation simpler as the focus does not have to lie on low-level details. Since Agda is a functional language a natural choice would be another functional language for which there already is a compiler.

There exists an Agda backend targeting Haskell, presented by Marcin Benke [Ben07]. A problem with compiling to Haskell is that Haskell is strongly typed, but not dependently typed. For the generated programs to pass Haskell's type checker, type coercions[6] are used, which unfortunately prevents Haskell from performing some optimisations.

The target language used in this thesis is Epic, presented in the next section, which avoids the typing problem altogether as it is not type checked, while still being relatively high-level.

2.5 Epic

Epic[7] is a strict functional language that is intended as the backend for dependently typed functional languages. It was made by Edwin Brady for Epigram but has received wider use in both Idris and now Agda. Epic has type annotations which are not checked, and they are also normally not used when compiling a program. This allows languages with different type disciplines to use Epic as a backend. The following is the definition of the map function in Epic:

```
map (f : Any, xs : Data) → Any = case xs of
  { Con 0 ()                    → Con 0 ()
  ; Con 1 (y : Any, ys : Any)   → Con 1 (f (y), map (f, ys))
  }
```

Any is a type that can stand for anything, Data is the type of (applied) constructors, and Con stands for a constructor and has a tag and a list of fields. The constructor tag is just an integer, and as such all constructors of a certain type have to be associated with a unique integer. In the example the tag 0 is used for the [] constructor and tag 1 for the _::_ constructor of Lists.

[6]A (sometimes unsafe) function that casts a term to a different type.

[7]**Epi**gram **C**ompiler.

If the tag is known before the case analysis is performed it is unnecessary to check it before getting at the constructor's fields. For these cases Epic provides field projections. This is common for Agda records, but can also occur due to dependent types only allowing one constructor in certain contexts. Field projections are written e ! i where e is the expression and i is the field position. The following example of uncurry illustrates how this can be used. Here p is a pair, and it must be of the right tag (0 in this case), since pairs only have one constructor.

```
uncurry (f : Any, p : Data) → Any
  = case p of
    {Con 0 (x : Any, y : Any) → f (x, y)
    }
```

This can be rewritten to the following more efficient version, which does not need to check the tags, but simply projects the fields of the pair:

```
uncurry (f : Any, p : Any) → Any =
  let x : Any = p ! 0
      y : Any = p ! 1
    in f (x, y)
```

Epic has a type called BigInt, which allows for an efficient representation of unbounded integers and simple operations on them.

The following describes the syntax of the Epic language in BNF form[8] (although some type annotations are omitted):

	p	$::= \overline{def}$	Program
	def	$::= x(\overline{x}) = t$	Top level definition
Terms	t	$::= x$	Variable
		$\mid t(\bar{t})$	Application
		$\mid$ **Con** i $(\bar{t})$	Constructor application
		$\mid$ **if** t **then** t **else** t	If term
		$\mid$ **case** t **of** $\overline{alt}$	Case term
		$\mid$ **let** $x = t$ **in** t	Let term
		$\mid$ **lazy** t	Suspended term
		$\mid t \,!\, i$	Field projection
		$\mid i$	Constant
	alt	$::=$ **Con** i $(\overline{x}) \to t$	Constructors
		$\mid i \to t$	Integer constants
		$\mid$ **default** $\to t$	Match anything

[8]Here an overline, such as the one in $\overline{x}$, denotes a list of things.

The operational semantics (the precise meaning of the programs) of Epic will not be presented here, but follow what is expected of a strict functional language.

Epic has a foreign function interface which makes it possible to call functions written in C from an Epic definition. The following example defines the function openFile which uses the C function fopen to get a file handle from a filename and a mode string. This file handle can then be passed around and used in calls to other C functions, and is represented by the Epic type Ptr (which is the type of C pointers).

```
openFile (filename : String, mode : String) → Ptr
   = foreign Ptr "fopen" (filename : String, mode : String)
```

Using the foreign function interface, the type annotations are important, because Epic uses them to marshal values from its own representation to a C representation. For example, a String may be represented by a struct with a tag (an integer) stating that the value is a String and a pointer to a null-terminated C-string in Epic. To marshal this into a C-string of type char* is to just use the pointer instead of the whole struct. In this way the C function fopen can be used normally even though Epic's internal representation of Strings differs from that of C.

In appendix A examples of how to call Epic (and by transitivity C) functions from Agda are given.

2.6 Related work

The act of compiling dependent types is treated rather sparsely in the literature, but a large part of what applies to compiling functional languages also applies here. This section will give a brief overview over the approaches to compiling dependently typed languages that do exist.

Epigram Edwin Brady's excellent PhD thesis [Bra05] explains how the compiler for a language called Epigram works. Most of what is presented there is also relevant to this thesis, although there are differences between Epigram and Agda. The most noticeable difference is that Epigram's case analysis is not written using pattern matching but is instead performed by using eliminators. These eliminators are generated by the data structure, or created by the programmer (in terms of the generated eliminators). The following example is the eliminator for Vec:

$$
\begin{array}{lll}
\mathsf{Vect}\text{-}\mathbf{Elim} : & \forall \mathsf{A} & : \star \,.\, \forall \mathsf{n} : \mathbb{N} \,.\, \forall \mathsf{v} : \mathsf{Vect}\ \mathsf{A}\ \mathsf{n}\ . \\
& \forall \mathsf{P} & : (\forall \mathsf{n} : \mathbb{N} \,.\, \forall \mathsf{v} : \mathsf{Vect}\ \mathsf{A}\ \mathsf{n}\ .\ \star)\ . \\
& \forall \mathsf{m}_{\epsilon} & : \mathsf{P}\ 0\ (\epsilon\ \mathsf{A})\ . \\
& \forall \mathsf{m}_{::} & : (\forall \mathsf{k} : \mathbb{N} \,.\, \forall \mathsf{x} : \mathsf{A} \,.\, \forall \mathsf{xs} : \mathsf{Vect}\ \mathsf{A}\ \mathsf{k}\ . \\
& & \quad \forall \mathsf{ih} : \mathsf{P}\ \mathsf{k}\ \mathsf{xs}\ .\ \mathsf{P}\ (\mathsf{s}\ \mathsf{k})\ (::\ \mathsf{A}\ \mathsf{k}\ \mathsf{x}\ \mathsf{xs}))\ . \\
& \mathsf{P}\ \mathsf{n}\ \mathsf{v} & \\
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Vect}\text{-}\mathbf{Elim}\ \mathsf{A}\ 0\ \ \ (\epsilon\ \mathsf{A})\ \ \ \ \mathsf{P}\ \mathsf{m}_{\epsilon}\ \mathsf{m}_{::} \rightsquigarrow \mathsf{m}_{\epsilon} \\
\mathsf{Vect}\text{-}\mathbf{Elim}\ \mathsf{A}\ (\mathsf{S}\ \mathsf{k})\ (::\ \mathsf{A}\ \mathsf{k}\ \mathsf{x}\ \mathsf{xs})\ \mathsf{P}\ \mathsf{m}_{\epsilon}\ \mathsf{m}_{::} \\
\qquad \rightsquigarrow \mathsf{m}_{::}\ \mathsf{k}\ \mathsf{x}\ \mathsf{xs}\ (\mathsf{Vect}\text{-}\mathbf{Elim}\ \mathsf{A}\ \mathsf{k}\ \mathsf{xs}\ \mathsf{P}\ \mathsf{m}_{\epsilon}\ \mathsf{m}_{::})
\end{array}
$$

The generated eliminators are given by pattern matching, but this is hidden from the programmer. Vect-**Elim** is the only function that performs pattern matching, which is exploited in some optimisations, and allows for local reasoning about all functions that inspect a Vec for example. In Agda all functions can perform pattern matching on a Vec argument and hence all functions need to be inspected when performing certain transformations that change the way pattern matching is done.

In the later stages of the Epigram compiler the eliminators are inlined and all functions can perform pattern matching, which is done because pattern matching can be compiled into more efficient code.

Cayenne The Cayenne language [Aug98] is a dependently typed language which compiles to Lennart Augustsson's own compiler for LML[9]. Cayenne performs type erasure which removes the programs' type information and then compiles to LML, which is run without type checking. This works since the LML compiler itself does not rely on the fact that the terms are type correct in the LML language, only that they "make sense" (just like Epic, although LML code is more high-level). One drawback of this approach is that since LML does not keep the types, type-based optimisations are not applicable at this level.

MAlonzo - Agda's Haskell backend As mentioned in section 2.4 a backend targeting Haskell may not be the perfect fit for Agda, as Agda's type system is more expressible than Haskell's. The backend has to insert coercions around the result of all function calls and all function arguments to make the program pass the type checker.

Agda has pragmas for expressing how types should be represented in the compiled Haskell code and there is also a mechanism for adding bindings to Haskell functions. The translation of Agda's internal code representation to Haskell is rather easy and amounts to nothing more than pretty printing. No optimisations are applied at this level in the hope that the GHC compiler will compile good code.

[9] A lazy variant of ML.

Some of the optimisations presented in this paper would also apply to the MAlonzo backend.

Program Extraction The Coq language employs program extraction targeting both OCaml [OCa] and Haskell. The possibility to do program extraction [Let03] is a consequence of the Curry-Howard isomorphism, stating that it is possible to translate formal proofs in Coq to programs in respective language. This can be used to get certified programs in these languages, and has for example been used to verify the window manager XMonad's functionality [Swi11].

3

Optimisations

THIS chapter presents the details and theory of the different optimisations that are performed in the Epic backend for Agda. Some of the optimisations also apply to different implementations of dependently typed theories.

Chapter 2 gave the background needed to understand the optimisations, but no details on the specifics of the compiler. Before the optimisations are presented a brief overview of the compiler and the different representations that Agda code goes through before it becomes executable will thus be presented (section 3.1).

3.1 Compiler overview

Figure 3.1 provides an overview over the different representations that an Agda program goes through, that are relevant to the compiler presented in this thesis. Parsing, which produces terms in Agda's internal syntax is left out. Type checking is also left out, but works on Agda's internal syntax.

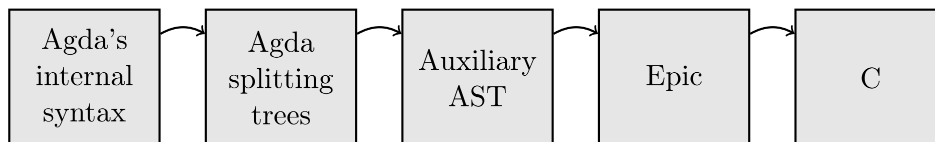


Figure 3.1: Compiler overview

Agda's internal syntax has pattern matching, which is compiled into case splitting trees using a well-known algorithm [Aug84]. It is on these splitting trees that the Epic backend starts its work, by first translating it into an auxiliary abstract syntax tree (AuxAST), which is syntactically very similar to Epic. The AuxAST is then pretty printed into an Epic source file, which Epic compiles to C. The last step is to run a C compiler on the C code, producing an executable program, which is performed by the Epic compiler.

Most optimisations work on AuxAST terms, but in some cases it is convenient to use Agda's internal syntax for doing analyses, as the Agda implementation already provides many helpful functions for working with that structure.

Pattern matching and case splitting trees The difference between pattern matching and case splitting trees is that pattern matching allows

simultaneous and nested pattern matching on several arguments, while a case splitting tree matches an expression to a list of constructors and variables (with no patterns in patterns). To exemplify this, consider the zipWith function on vectors which combines two vectors of equal length by running a function down the vectors element wise, first written using pattern matching:

```
zipWith : {A B C : Set} {n : ℕ}
        → (A → B → C) → Vec A n → Vec B n → Vec C n
zipWith f []        []        = []
zipWith f (x :: xs) (y :: ys) = f x y :: zipWith f xs ys
```

An equivalent case splitting tree is the following:

```
zipWith f as bs = case as of
  []        → case bs of
    [] → []
  _::_ x xs → case bs of
    _::_ y ys → f x y :: zipWith f xs ys
```

Note that in the second version of the function, only one variable is considered at a time and the execution order is made more explicit.

As mentioned above, Agda's internal syntax uses pattern matching and subsequent representations in the compiler use case splitting trees. In the rest of the thesis, both pattern matching and case splitting will be used depending on what is appropriate for examples given and what representation is being considered.

3.2 Forcing

The type of finite natural numbers that was introduced in section 2.1 stores the type index in every constructor. This may have a big impact on how much space is required. For example, consider how the representation of the number 2 is in the type Fin 3 (with explicit arguments):

```
data Fin : ℕ → Set where
  fz : (n : ℕ)          → Fin (suc n)
  fs : (n : ℕ) → Fin n → Fin (suc n)

ex : Fin 3
ex = fs (suc (suc zero)) (fs (suc zero) (fz zero))
```

The above representation is particularly wasteful since the type index is always in scope when pattern matching on a term of a Fin type. This is true because it is needed to be able to provide the type index to the Fin argument. Observe for example the function inject which converts a Fin n to a Fin (suc n) (but still represents the same number):

```
inject : (n : ℕ) → Fin n → Fin (suc n)
inject .(suc n) (fz n)   = fz (suc n)
inject .(suc n) (fs n i) = fs (suc n) (inject n i)
```

Since n occurs in the first argument, inject can be rewritten as inject' given below, which does not use the n from the Fin argument:

```
inject' : (n : ℕ) → Fin n → Fin (suc n)
inject' zero    ()
inject' (suc n) (fz .n)   = fz (suc n)
inject' (suc n) (fs .n i) = fs (suc n) (inject' n i)
```

For Agda to see that this function is total a new equation has been introduced, with an impossible pattern.

It is always possible to infer the n : ℕ argument to the constructors of Fin when it is used, which means that it is not necessary to store it. Fin can thus be changed into the following, where constructor fields that are red and underlined are not actually stored in the compiled representation:

```
data Fin : ℕ → Set where
  fz : (n : ℕ)          → Fin (suc n)
  fs : (n : ℕ) → Fin n → Fin (suc n)
```

The optimisation of identifying and rewriting the arguments of a constructor that can be inferred from their use as type indices has been explored in Epigram [BMM04], and is there called **forcing**. These arguments do not need to be stored in the constructor and so less space is used to represent the data. It is noteworthy that the Fin type without the indices has the same form as the type of ordinary natural numbers, ℕ. This correspondence allows the Fin type to be represented as a primitive integer, which will be further explained in section 3.6.

Detecting indices that do not need be stored To find the fields of a constructor that can always be inferred from the outside scope, the constructor's return type is first examined. The zero constructor from the Fin type illustrates the point:

```
fz : (n : ℕ) → Fin (suc n)
```

The return type of the fz constructor is Fin (suc n). To find variables that can potentially be forced, each variable that can be obtained by pattern matching on the index term (here suc n) is collected. This is true for variables in constructor applications, which means that n above is a candidate since it is applied to the constructor suc. Now n can be gotten by pattern matching on the type index guaranteed to be in scope in a function using the datatype.

If the term, however, looks like Fin (f n) for some *function* f, it is not certain that the function has a computable inverse, so then n can not (generally) be considered.

After the collection of variables obtainable from the outside, the field in the constructor where each collected variable is bound is found and marked as **forced**. In fz : (n : ℕ) → Fin (suc n), n is bound in the first field of the constructor, which is then marked.

Rewriting patterns Internally Agda uses splitting trees to describe pattern matching in functions. The inject function is represented internally as:

```
inject n i  =  case i of
  fz n'   → fz (suc n')
  fs n' i → fs (suc n') (inject n' i)
```

To make it valid to remove type indices from constructors, the case tree has to be rewritten so that the code does not use n', as it was marked as forced. The case should instead be performed on n to get n'. In general, for every case that introduces variables that can be forced, a new case on the argument used in the type index is created to "dig out" the forced variables. If the forced variable is equal to an already existing variable a simple substitution in the body will suffice. For inject the new case-tree looks like this:

```
inject n i  =  case i of
  fz n'  → case n of
    suc n'  → fz (suc n')
  fs n' i  → case n of
    suc n'  → fs (suc n') (inject n' i)
```

The constructor fields marked with red and underline can now be removed, as they are not used anymore. Since n must be a suc constructor there is also need to check the constructor tag, so n' can be gotten using a projection in Epic.

The equalities between previous variables and the current pattern can be found by using unification on their types. A telescope of the types of in-scope variables is introduced and updated as the case tree is traversed. At first the telescope is (n : ℕ) (i : Fin n). When the fz n' branch is inspected the type of the fz constructor is unified with the current telescope.

Since fz : (n' : ℕ) → Fin (suc n') the unification with Fin n yields n = suc n', and hence it is possible to get the value of n' by pattern matching on n.

A more general algorithm will now be presented, for transforming case expressions in a function with a type environment represented by the telescope Γ.

The algorithm is run on each case expression on the following form:

... **case** i **of**
  C_1 x_1 ... x_k → ...
    ⋮
  C_n x_1 ... x_m → ...

Here i represents the variable that is being cased on, and has a type in the telescope Γ. Now suppose that i has type $\mathtt{T}\ \Delta\ \overline{\alpha}$ (overlines are used here, e.g $\overline{\alpha}$, denoting lists of things) where Δ are the parameters of the type and $\overline{\alpha}$ are the indices. For each case $\mathtt{C}_j$ where the constructor is of type $\tau_j = \Delta \to \Theta \to \mathtt{T}\ \Delta\ \overline{\beta}$, the following is done:

1. τ_j is applied to Δ yielding $\tau_j\ \Delta = \Theta \to \mathtt{T}\ \Delta\ \overline{\beta}$, the type of the constructor with the correct parameters applied.
2. The i:th position in the telescope Γ is replaced with Θ, so that the telescope is updated with the variables that come into scope in the branch.
3. All occurrences of i in Γ are replaced with $\mathtt{C}_j\ \Theta$, since more is known about i as we enter the branch.
4. $\overline{\alpha}$, the type indices from the type of i, is unified with $\overline{\beta}$, the type indices from the type of the current branch's constructor, in the (updated) telescope Γ.

   The equality constraints from this unification are used to rewrite the body of the case such that no forced variable is used. Assume that x is a forced variable. Then the following two types of constraints may be found:

   - y = ...x...: An unforced variable y is equal to an expression containing x in a position that can be reached through pattern matching. Here x is reachable in e if:
     - e = x, i.e they are equal
     - e = C $\overline{\text{es}}$, where x is reachable in at least one e' $\in$ $\overline{\text{es}}$ such that e' is not forced in C.

     In this case pattern matching on y can be performed to reach an x' that does not stem from something that is forced such that x' = x. Then x' can be substituted for x in the body and the telescope.
   - x = e: The forced variable is equal to an expression e. In this case all occurrences of x in the body of the current case and the telescope are replaced by e.

3.3 Erasure

Agda has no way of performing pattern matching on types, which means that there is no information that can be gained from a term of type Set at runtime. Therefore the runtime behaviour of functions that receive an argument of type Set may not depend on which Set it is. The following example illustrates the point:

```
f : Set → ℕ
f x = c
```

It is not allowed to perform any pattern matching on the input x of the function, which means that the function must return some constant value c (of type ℕ). This means that at runtime, types have no computational content and do not even need to be stored, as they have no way of affecting the result of the program.

A correctness proof of this in a similar setting is done in [Aug98], for the dependently typed functional language Cayenne, which also has no way of performing pattern matching on types.

Instead of using full type erasure, the worker/wrapper [GH09] method is here used to remove unused arguments. And since it is not possible to pattern match on types they will never be used in a function other than as an argument to another function. The map function is an example of this:

```
map : (A B : Set) (n : ℕ) (f : A → B) → Vec A n → Vec B n
map A B .0       f []                 = []
map A B .(suc n) f (_::_ {n} x xs) = _::_ {n} (f x) (map A B n f xs)
```

The arguments A and B are only used in the recursive call, and are hence not needed. Using the worker/wrapper transform map is compiled into two functions:

The worker which only takes the needed arguments.

The wrapper which takes the same arguments as the original map function and calls the worker.

This method is also used in Brady's PhD thesis [Bra05]. The following is the result for the map function:

```
map_wrap A B n f xs = map_work n f xs

map_work .0       f []                 = []
map_work .(suc n) f (_::_ {n} x xs) = _::_ {n} (f x) (map_work n f xs)
```

The wrapper is what other functions call from the outside, and the worker function does all the actual work. One added benefit of this approach is that the wrapper function is non-recursive and rather small. A

perfect candidate for inlining! If a function application of map is fully saturated it will be inlined and becomes a call to the worker without the unused arguments.

If forcing is also applied then n also becomes unused[1] and an even better $\mathsf{map}_{\mathsf{work}}$ can be made. This corresponds to the usual map function for Lists:

$$
\begin{aligned}
\mathsf{map}_{\mathsf{work}}\ \mathsf{f}\ [\,] &= [\,] \\
\mathsf{map}_{\mathsf{work}}\ \mathsf{f}\ (\mathsf{x} :: \mathsf{xs}) &= \mathsf{f}\ \mathsf{x} :: \mathsf{map}_{\mathsf{work}}\ \mathsf{f}\ \mathsf{xs}
\end{aligned}
$$

Identification of unused arguments To find the arguments that are unused, so that the worker/wrapper transform can be performed, an abstract interpretation of the function definitions in the AuxAST is used. In the interpretation the domain consists of two values:

0 The argument is definitely not used.

1 The argument may be used.

The return value of a function call in this interpretation answers the question whether any of the arguments that are **1** may have been used in the function. The idea is that any argument that does not appear on the right-hand side of a definition or only appears as an argument to a function in which it is unused should be marked as unused.

Since there may be circular dependencies, there is no linear order of evaluation that will yield the correct result. One way to calculate it is by using the fix-point of a series of approximations, where the result becomes increasingly refined in each step.

Consider the following example functions:

$$
\begin{aligned}
\mathsf{f}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} &= \mathsf{z} + \mathsf{g}\ \mathsf{x}\ \mathsf{y}\ \mathsf{x} \\
\mathsf{g}\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} &= \mathsf{y} + \mathsf{f}\ \mathsf{x}\ \mathsf{y}\ \mathsf{y}
\end{aligned}
$$

Addition uses both of its arguments, and is thus the boolean *or* function in this interpretation. This means that if any of its arguments are used, the result of the function call will also be used.

$$
\mathsf{x} + \mathsf{y} = \mathsf{x} \vee \mathsf{y}
$$

The first approximation, approximation zero, is that each argument is unused. This approximation will then be used in successive approximations, yielding better and better results.

$$
\begin{aligned}
\mathsf{f}_0\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} &= 0 \\
\mathsf{g}_0\ \mathsf{x}\ \mathsf{y}\ \mathsf{z} &= 0
\end{aligned}
$$

[1] Actually n could be removed in the above example as well, but that is because n is bound in the _::_ and not from the ℕ parameter of the function.

In general, approximation n will use approximation $n-1$ instead of recursive calls:

$$\mathsf{f_n\ x\ y\ z} = \ldots\ \mathsf{f_{(n\ -\ 1)}}\ \cdots\ \mathsf{g_{(n\ -\ 1)}}\ \cdots$$
$$\mathsf{g_n\ x\ y\ z} = \ldots\ \mathsf{g_{(n\ -\ 1)}}\ \cdots\ \mathsf{f_{(n\ -\ 1)}}\ \cdots$$

In the example, the following approximations are gotten:

$$\begin{aligned}&\mathsf{f_1\ x\ y\ z}\\ &\quad= \mathsf{z} + \mathsf{g_0\ x\ y\ x}\\ &\quad= \mathsf{z} \vee \mathsf{0}\\ &\quad= \mathsf{z}\end{aligned}$$

$$\begin{aligned}&\mathsf{g_1\ x\ y\ z}\\ &\quad= \mathsf{y} + \mathsf{f_0\ x\ y\ y}\\ &\quad= \mathsf{y} \vee \mathsf{0}\\ &\quad= \mathsf{y}\end{aligned}$$

$$\begin{aligned}&\mathsf{f_2\ x\ y\ z}\\ &\quad= \mathsf{z} + \mathsf{g_1\ x\ y\ x}\\ &\quad= \mathsf{z} \vee \mathsf{y}\end{aligned}$$

$$\begin{aligned}&\mathsf{g_2\ x\ y\ z}\\ &\quad= \mathsf{y} + \mathsf{f_1\ x\ y\ y}\\ &\quad= \mathsf{y} \vee \mathsf{y}\\ &\quad= \mathsf{y}\end{aligned}$$

$$\begin{aligned}&\mathsf{f_3\ x\ y\ z}\\ &\quad= \mathsf{z} + \mathsf{g_2\ x\ y\ x}\\ &\quad= \mathsf{z} \vee \mathsf{y}\end{aligned}$$

$$\begin{aligned}&\mathsf{g_3\ x\ y\ z}\\ &\quad= \mathsf{y} + \mathsf{f_2\ x\ y\ y}\\ &\quad= \mathsf{y} \vee \mathsf{y} \vee \mathsf{y}\\ &\quad= \mathsf{y}\end{aligned}$$

A fix-point is reached here, as the fourth approximation is the same as the third.

To determine whether an argument may be used in a function, the abstractly interpreted version of the function gotten from the approximation is called with that argument set to **1** while the others are set to **0**. This means that f uses y and z and g uses only y in the example.

The details of going from an ordinary definition to the abstract interpretation will now follow. Here $[\![\mathsf{e}]\!]$ stands for the abstract interpretation of e.

A variable is simply translated to itself (but will now be in the domain of the abstract interpretation), and constants are interpreted as **0**, as they do not affect any of the variables. This means that a call to the + function with constant arguments yields the result **0**.

$[\![\mathsf{v}]\!] = \mathsf{v}$ v is a variable

$[\![\mathsf{c}]\!] = \mathsf{0}$ c is a constant

In a constructor application, the constructor itself does not affect any variables so it can be left out in the abstract interpretation. The arguments may however contain uses of variables, so the interpretations of the arguments are combined together.

$$[\![\mathsf{C}\ \mathsf{e}_1\ \ldots\ \mathsf{e}_n]\!] = [\![\mathsf{e}_1]\!] \vee \ldots \vee [\![\mathsf{e}_n]\!]$$

A function call is transformed into a function call to the abstract interpretation of f. When using successive approximations as was done above, this will be the previous approximation.

$$[\![\mathsf{f}\ \mathsf{e}_1\ \ldots\ \mathsf{e}_n]\!] = \mathsf{f}_{\#}\ [\![\mathsf{e}_1]\!] \ldots [\![\mathsf{e}_n]\!]$$

$\mathsf{f}_{\#}$ is the abstract interpretation of f

Let-bound expressions, which are sometimes used in the AuxAST, are substituted into body of the let expression, since the effect of using the bound variable is that of using the variables from the expression.

$$[\![\ \textbf{let}\ \mathsf{v} = \mathsf{e}\ \textbf{in}\ \mathsf{t}]\!] = [\![\mathsf{t}\ [\mathsf{e}\ /\ \mathsf{v}]]\!]$$

Since the bound variables in case branches stem from the value that was cased on, they are replaced with that in the abstract interpretation.

$$\begin{array}{l} [\![\textbf{case}\ \mathsf{e}\ \textbf{of} \\ \quad \mathsf{p}_1 \to \mathsf{e}_1 \\ \quad \ldots \\ \quad \mathsf{p}_n \to \mathsf{e}_n]\!] \\ \quad = \\ [\![\mathsf{e}]\!] \vee [\![\mathsf{e'}_1]\!] \vee \ldots \vee [\![\mathsf{e'}_k]\!] \end{array}$$

each $\mathsf{e'}_i$ is e_i with e substituted for all variables in p_i

3.4 Injection detection

The following function takes a length indexed vector and converts it into a list[2]. It forgets the length indexing:

```
forget : ∀ {A n} → Vec A n → List A
forget []        = []
forget (x :: xs) = x :: forget xs
```

When forcing has been done, the index of the vector is removed, which makes the representation of vectors essentially the same as lists. If the

[2]Note that the constructors for lists and vectors have the same name here, which is allowed in Agda.

constructor tags for the two constructors are chosen to be the same for both lists and vectors, the compiled function looks like this:

$$\begin{aligned} &\mathsf{forget\ A\ n\ (Con\ 0)} &&= \mathsf{Con\ 0} \\ &\mathsf{forget\ A\ n\ (Con\ 1\ (x, xs))} &&= \mathsf{Con\ 1\ (x, forget\ xs)} \end{aligned}$$

In both clauses, the function's return value is the same as its third argument, so it is really an inefficient identity function, and could be simplified to the following:

$$\mathsf{forget\ A\ n\ xs} = \mathsf{xs}$$

The new function runs in constant time, whereas the original ran in linear time in the length of the vector. A similar optimisation is applicable to the inject function introduced in section 3.2:

$$\begin{aligned} &\mathsf{inject} : \{\mathsf{n} : \mathbb{N}\} \to \mathsf{Fin\ n} \to \mathsf{Fin\ (suc\ n)} \\ &\mathsf{inject\ fz} = \mathsf{fz} \\ &\mathsf{inject\ (fs\ i)} = \mathsf{fs\ (inject\ i)} \end{aligned}$$

To do this in general, the idea is to not choose tags for constructors until functions for which the above transformation would work have been identified. When that has been done, the tags can be chosen using the information from the identification.

Identification of injective functions Consider a function on the following form in Agda's internal syntax:

$$\begin{aligned} &\mathsf{f}\ \mathsf{p}_{11} \ldots \mathsf{p}_{n1} = \mathsf{b}_1 \\ &\qquad \vdots \\ &\mathsf{f}\ \mathsf{p}_{1k} \ldots \mathsf{p}_{nk} = \mathsf{b}_k \end{aligned}$$

The first step of the process is to pick an argument index i from $\{1, \ldots, n\}$. For each branch body with index j, the branch body b_j is reduced and compared to argument i's pattern in that branch, p_{ij}, to find what preconditions there are to this function being an injection in that argument.

A precondition can be one of three things:

Never The precondition can never hold. Comparing two different constructors from the same datatype will for example never hold, as they cannot have the same constructor tag, meaning that there cannot be an identity function between them.

Always The precondition will always hold. A function which does not pattern match on one of the arguments but just returns it, or one that maps it to the same constructor as the pattern adds no precondition.

$\mathbf{c_1 = c_2}$ The precondition holds as long as the constructor tags for c_1 and c_2 are equal. An example is the forget function mentioned above, which can only be an injection if the constructor tags for vectors and lists are the same.

The comparison is done by case analysis in the following way:

- If the branch body b_j reduces to a recursive call to f on the form `f` $a_1 \ldots a_n$, the induction hypothesis that the function is in fact an injection for pattern i is assumed to hold true. The body is substituted for a_i, and the comparison is done again with this value.
- If the branch body b_j reduces to a constructor application on the form $c_1\ a_1\ \ldots a_m$ and the pattern p_{ij} is a constructor pattern on the form $c_2\ b_1\ \ldots b_m$, where c_1 and c_2 come from different datatypes, then the injection holds under the precondition that $\mathbf{c_1 = c_2}$. If they come from the same datatype and are the same constructor, it **always** holds, and it **never** holds if they are not the same constructor. The preconditions gotten from comparing $a_1\ \ldots a_m$ and $b_1\ \ldots b_m$ are also added as preconditions. When doing this comparison, forced arguments should not be compared, since they will not appear as constructor arguments in the generated code at all. Failure to do so would not let the forget function be an injection.
- If the branch body b_j reduces to a variable that is equal to p_{ij}, the precondition **always** holds.
- If the pattern p_{ij} is a dot pattern on the form `.(f` $e_1\ \ldots e_n$`)` and the branch body b_j is `f` $e'_1\ \ldots e'_n$ the precondition **always** holds if the preconditions gotten from comparing $e_1\ \ldots e_n$ and $e'_1\ \ldots e'_n$ are also added as preconditions.
- In all other cases, the precondition **never** holds.

When this comparison has been run for all argument indices on all functions, the result can be used to select the tags for all constructors in a way that makes many functions injections.

Choosing constructor tags A straight-forward way to choose constructor tags that we have found to work well for us in practice is described here:

1. The preconditions for functions that have been chosen to hold are kept track of in equivalence classes over the constructors that must have the same tag. Initially, all constructors are in their own equivalence classes.

2. Two equivalence classes can be unified if there are no two (different) constructors from the same datatype in the resulting union. If that does not hold there is a conflict between preconditions.

3. For each function that has not previously been tested, the preconditions for that function are checked to see if they conflict with the preconditions that have been chosen to hold. If that is not the case, the function's preconditions are unified with the equivalence classes, and the function is marked as an injection.

4. At the end of the algorithm the tag for each of the constructors in every equivalence class can simply be chosen to be a tag which is not yet used in any of the datatypes that the constructors in the class appear in.

All the functions marked as being injections in argument i can now be transformed into the following efficient definition:

```
f v₁ ... vᵢ ... vₙ = vᵢ
```

3.5 Smashing

The following code is proof that the addition operation on ℕ is commutative taken from Agda's standard library. The details are not important, but do note that it does a recursive call and thus takes time (at least[3]) linear in its first argument.

```
+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero    n = sym $ proj₂ +-identity n
+-comm (suc m) n = begin
    suc m + n
  ≡⟨ refl ⟩
    suc (m + n)
  ≡⟨ cong suc (+-comm m n) ⟩
    suc (n + m)
  ≡⟨ sym (m+1+n≡1+m+n n m) ⟩
    n + suc m ∎
```

Now consider the definition of the equality datatype that +-comm returns, which was explained in section 2.2. It has only one constructor with zero arguments:

[3]If _+_ is a primitive function running in constant time, it takes time in $\mathcal{O}(m \cdot n)$, and if not it is even worse.

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

The representation of such a constructor at runtime is just a constructor tag. Its type and the type indices cannot affect the outcome (observed at runtime) of a function and thus have no computational content. This means that the runtime version of +-comm might as well be rewritten to simply become the following:

```
+-comm m n = refl
```

We call this optimisation **smashing**. In general, a function can be smashed if there is only one way to construct a value of its return type, which means that the value can be inferred without looking at any arguments. If this is the case the function does not need to do anything other than return that value. A simple criterion which can be used to find some types which are inferable is the following:

- A datatype can be **inferred** if it has only one constructor which
  - has no fields, or
  - has fields which are all **inferable**

In the example above, m + n ≡ n + m is inferable because refl has no fields, and thus +-comm can be smashed. All other functions which return something of the equality type can be smashed, since their return types are inferable.

3.6 Primitive data

Consider the datatype of natural numbers:

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

The representation of a number in this datatype is a series of applications of the successor constructor followed by a zero. For example, the number three is represented by the following:

```
3 = suc (suc (suc zero))
```

This would be compiled to the following Epic expression:

```
Con 1 (Con 1 (Con 1 (Con 0 ())))
```

In this encoding, natural numbers take up space that is proportional to the number they correspond to. A more space-efficient way to represent numbers is to use BigInts, which exploit the target machine's native integers. For most (non-huge) natural numbers used in a program the space cost of this representation is constant.

All datatypes that have two constructors similar to the ones for natural numbers – one constructor with zero arguments and the other constructor with a recursive argument – can use BigInts as their compiled representation.

As was shown in section 3.2, the Fin datatype does not need to store all the fields of its constructors. Notice how similar the following definition is to that of ℕ if the forced fields (red and underlined) are not taken into consideration:

```
data Fin : ℕ → Set where
  fz : (n : ℕ)          → Fin (suc n)
  fs : (n : ℕ) → Fin n → Fin (suc n)
```

The Fin datatype can thus also be represented more efficiently using this scheme.

Translation of a datatype where one constructor has no arguments, call it zero, and one constructor with one argument, call it suc, will now be presented:

- The zero constructor is translated to the BigInt 0.
- The expression suc n is translated to 1 + n', where n' is the translation of n.
- Case expressions on following form (left) are translated into if expressions (right):

```
case n of                  if n == 0
  zero   → e_z               then e_z
  suc n' → e_s               else let n' = n - 1 in e_s
```

Here == is the equality comparison function on BigInts. The translation is also done recursively in subexpressions.

Primitive operations Consider the following operator on natural numbers:

```
_+_ : ℕ → ℕ → ℕ
zero  + m = m
suc n + m = suc (n + m)
```

This function is recursive in its first argument, so takes time proportional to the number that its first argument represents. But since the numbers are now compiled as BigInts the same result would be gotten if the function was compiled as addition on BigInts.

It is possible to mark datatypes and functions as built-in in Agda. Functions marked as addition and multiplication on natural numbers will be changed to use the corresponding BigInt operations. Some extensions to this scheme are proposed in section 5.5.

3.7 Partial evaluation

As languages with dependent types do normalisation when type checking, there already exists functionality to evaluate open terms. This is called partial evaluation, and a classical example of its usage is the power function on natural numbers:

```
power : ℕ → ℕ → ℕ
power n zero    = 1
power n (suc k) = power n k * n
```

When the second argument is known at compile time, the function can be specialised for that number statically. If the user wants the fourth power of some unknown number n, the function can be normalised to the following:

```
power n 4 = n * n * n * n
```

Since the second argument is the one being cased on, the branches are known in advance and the recursive calls have been inlined.

The only addition to Agda to make this work is to expose the functionality to the user. This is done by making it possible to mark functions for which usages should be normalised at compile time. In the current implementation, a pragma is used to do so:

```
{-# STATIC power #-}
```

This example marks the power function as STATIC, which means that the optimisation should be applied to it before the code is compiled.

To get this optimisation to work, all terms in a program are gone through, looking for function calls to functions marked with STATIC. When such a call is found, it is normalised. This is done on Agda's internal syntax, so that the already existing normalisation functionality can be used.

The following function is an example using power:

```
somePowers : ℕ → ℕ
somePowers n = 1 + power n 2 + power n 3
```

When power is marked as STATIC, somePowers will be transformed into the following efficient definition at compile time:

```
somePowers n = 1 + n * n + n * n * n
```

Partial evaluation has previously been used to improve the performance of domain-specific languages [BH10]. A larger example of using partial evaluation will be developed in section 4.6.

4

Results

THIS chapter presents the effectiveness of the compiler in different aspects and scenarios. It starts out by showing how the code quality is affected by the different optimisations. The applicability of smashing in some modules of Agda's standard library is presented in section 4.4. Section 4.5 shows a case study over how the different optimisations apply to an interpreter for simply typed lambda calculus, and section 4.6 shows a case study using partial evaluation which displays how interpretative overhead can be removed.

Finally, in section 4.7, the results of benchmarks comparing the runtime, executable size and compilation time of four different programs using different compiler backends are presented.

4.1 Forcing and primitive data

Forcing is an important optimisation as it applies to many datatypes and allows other optimisations to kick in. It is especially important for representing data as primitive BigInts. As has previously been shown, the Fin datatype can be represented as a BigInt after forcing.

The following are examples of datatypes where forcing applies, some of which can also be represented as BigInts. To show which constructor fields have been forced, red colour and underlines are used.

The _⩽_ datatype describes the less than or equal relation between two natural numbers. It is inductively defined; the z ⩽ n constructor relates zero to any natural number, since zero is less than or equal to any other number in ℕ, and the s ⩽ s constructor relates $m + 1$ and $n + 1$ given that m is less than or equal to n:

```
data _⩽_ : ℕ → ℕ → Set where
  z⩽n : ∀ {n}                  → zero  ⩽ n
  s⩽s : ∀ {m n} (lt : m ⩽ n) → suc m ⩽ suc n
```

After removing forced constructor fields this datatype has one constructor with zero arguments and one constructor with a recursive argument – it can be represented as a primitive BigInt!

The Ordering relation is what is called a comparison view [MM04], and is used to compare two natural numbers.

```
data Ordering : ℕ → ℕ → Set where
  less    : ∀ m k → Ordering m (suc (m + k))
  equal   : ∀ m   → Ordering m m
  greater : ∀ m k → Ordering (suc (m + k)) m
```

It is usually coupled with a function compare : (m k : ℕ) → Ordering m k which provides information on how two numbers relate to each other.

In all of the constructors the m field can be forced and does not need to be stored. The less constructor can only be constructed when the first number is less than the second, and it also has information about the difference between the first and second number in the relation (k). The equal constructor relates a number to itself, and finally greater is the flipped version of less. Note that any two natural numbers can be related in exactly one way. With this datatype and the compare function it is for example possible to define a subtraction function:

```
_-_ : ℕ → ℕ → ℕ
a - b with compare a b
.m              - .(suc (m + k)) | less    m k = 0
.m              - .m             | equal   m   = 0
.(suc (m + k)) - .m              | greater m k = suc k
```

The _∈_ predicate is the predicate that a certain element is in a Vec. Elements of x ∈ xs can be constructed only if x is somewhere inside xs. This datatype can also be used to find out where.

```
data _∈_ {A : Set} (x : A) : {n : ℕ} → Vec A n → Set where
  here  : ∀ {n}   {xs : Vec A n} → x ∈ x :: xs
  there : ∀ {n y} {xs : Vec A n} (x∈xs : x ∈ xs) → x ∈ y :: xs
```

The constructor here can be constructed if x is the head of the Vec since the return type is x ∈ x :: xs. Here x occurs twice to denote that both are the same. The second constructor, there, can be constructed if x occurs in the tail of the Vec no matter what the head is.

Perhaps surprisingly, this datatype can be represented as a primitive integer, as almost all of its constructor fields do not have to be stored at runtime, leaving just a constructor with zero arguments and a constructor with a recursive argument. The value of the integer that it can be represented as corresponds to the index into the Vec that the element is at.

4.2 Erasure

Sometimes when working with dependent types one has to deal with unnecessarily complicated functions. This may be because the function does not

follow the structure of the data type. The example below illustrates this problem. At first sight it seems to be the ordinary append function but the result type is Vec A (n + m) instead of Vec A (m + n), hence the name append'. This innocent looking difference forces the programmer to write some proofs that the types are indeed the same. In doing so the following two lemmas are needed:

```
+-right-identity : (n   : ℕ) → n + zero  ≡ n
+-suc-right      : (n m : ℕ) → n + suc m ≡ suc (n + m)
```

Only the types are stated as the implementations are of no importance. Smashing will make these functions just return refl. Now append' can be implemented:

```
append' : (A : Set) (m n : ℕ) → Vec A m → Vec A n
   → Vec A (n + m)
append' A .zero n [] ys with n + zero | +-right-identity n
... | .n | refl = ys
append' A (suc m) n (x :: xs) ys with n + suc m | +-suc-right n m
... | .(suc (n + m)) | refl = x :: append' A m n xs ys
```

This function uses the **with** construct for doing the pattern matching necessary to convince Agda that the types are indeed correct. The **with** construct will be desugared into mutual function definitions by the type checker. These functions receive all variables in scope and the extra arguments that were requested for pattern matching. As such the append function gets desugared into the following **mutual**[1] block[2].

```
mutual
  append' : (A : Set) (m n : ℕ) → Vec A m → Vec A n
     → Vec A (n + m)
  append' A .zero n [] ys
     = append'_a A n ys (n + zero) (+-right-identity n)
  append' A (suc m) n (x :: xs) ys
     = append'_b A m n x xs ys (n + suc m) (+-suc-right n m)
  append'_a : (A : Set) (n : ℕ) → Vec A n → (n+0 : ℕ)
     → n+0 ≡ n → Vec A n+0
  append'_a A n ys .n refl = ys
  append'_b : (A : Set) (m n : ℕ) → A → Vec A m → Vec A n
     → (n+sm : ℕ) → n+sm ≡ suc (n + m) → Vec A n+sm
  append'_b A m n x xs ys .(suc (n + m)) refl
     = x :: append' A m n xs ys
```

[1]Normally Agda functions can only depend on functions written above them, but **mutual** allows them to use all function inside the block as well.

[2]Notice that n+0 and n+sm are identifiers and not usages of the _+_ operator.

In the AuxAST the pattern matching on the left hand side is given by case expressions. The above definitions correspond to the following code:

```
mutual
  append' : (A : Set) (m n : ℕ) → Vec A m → Vec A n
    → Vec A (n + m)
  append' A m n xs ys = case xs of
    {[]      → append'a A n ys (n + zero) (+-right-identity n)
    ;x :: xs → case m of
      {suc m' → append'b A m' n x xs ys (n + suc m) (+-suc-right n m)}
    }
  append'a : (A : Set) (n : ℕ) → Vec A n → (n+0 : ℕ)
    → n+0 ≡ n → Vec A n+0
  append'a A n ys n+0 prf = case prf of
    {refl → ys}
  append'b : (A : Set) (m n : ℕ) → A → Vec A m → Vec A n
    → (n+sm : ℕ) → n+sm ≡ suc (n + m) → Vec A n+sm
  append'b A m n x xs ys n+sm prf = case prf of
    {refl → x :: append' A m n xs ys}
```

Notice that in both append'$_a$ and append'$_b$ the pattern matching on prf is unnecessary since there is no other possible value than refl for _≡_. For ease of presentation both of these functions will be inlined back into append' (but this is not necessary for the removal of unused arguments in general).

```
append' : (A : Set) (m n : ℕ) → Vec A m → Vec A n
  → Vec A (n + m)
append' A m n xs ys = case xs of
  {[]       → ys
  ;x :: xs' → case m of
    {suc m' → x :: append' A m' n xs' ys}
  }
```

The proofs have disappeared and the structure is very similar to the append' function on Lists save for two extra length arguments. By following the steps outlined in section 3.3 for removing unused arguments the following equations are gotten in the abstract interpretation:

```
append'0 A m n xs ys = 0         -- Initial assumption
append'1 A m n xs ys = xs ∨ ys   -- m is not used since suc m'
    -- is the only branch and m' not used
append'2 A m n xs ys = xs ∨ ys   -- fixed point reached
```

By the abstract interpretation only the Vecs are needed and the rest of the arguments can be removed in the worker/wrapper stage. This gives the final function that is identical to append' for Lists:

```
append'_wrap A m n xs ys = append'_work xs ys
append'_work xs ys = case xs of
  { []       → ys
  ; x :: xs' → x :: append'_work xs' ys
  }
```

4.3 Injection detection

In this section some functions that get detected as injections are presented, which shows that injection detection is applicable in ordinary (not *always* contrived) programs.

List to tree conversion The examples that have been shown so far have been on functions between structures that are isomorphic (after forcing) with a bijection between them, yet the optimisation has been called injection detection all along. Thus, it might not come as a surprise that it is also possible to detect functions between non-isomorphic datatypes that are injections. The following example illustrates it using the type of trees with either one or two branches at each node, to which lists can be converted:

```
data List (A : Set) : Set where
  []    : List A
  _::_  : A → List A → List A
```

```
data Tree (A : Set) : Set where
  ◊       : Tree A
  _◁_▷_   : Tree A → A → Tree A → Tree A
  _▷_     : A → Tree A → Tree A
```

```
listToTree : {A : Set} → List A → Tree A
listToTree []        = ◊
listToTree (x :: xs) = x ▷ listToTree xs
```

The listToTree function becomes an injection and is simply replaced by listToTree xs = xs, as the constructor tags for [] and ◊ as well as _::_ and _ ▷ _ are chosen to be equal. Since the tags are chosen to make as many functions injections as possible, it does not matter in which order the constructors are written in the definitions of List and Tree.

An example from Agda's standard library The module Data.BoundedVec.Inefficient from the standard library is a good example of the effectiveness of injection detection. This module consists of three functions which all become injections. It is not so inefficient after all![3]

```
data BoundedVec {a} (A : Set a) : ℕ → Set a where
  []   : ∀ {n} → BoundedVec A n
  _::_ : ∀ {n} (x : A) (xs : BoundedVec A n) → BoundedVec A (suc n)
```

A BoundedVec differs from a Vec in that its ℕ index does not state the actual length of the vector, but its maximum possible length. Something of type BoundedVec ℕ 3 is thus a vector of length between 0 and 3.

A BoundedVec of maximum length n can easily be converted to a BoundedVec of maximum length suc n, which is what the function for increasing the bound does:

```
↑ : ∀ {a n} {A : Set a} → BoundedVec A n → BoundedVec A (suc n)
↑ []        = []
↑ (x :: xs) = x :: ↑ xs
```

The difference in the resulting vector is not visible in the code, as it uses implicit arguments, but the constructor field named n actually changes.

This function becomes an identity function as the index to the vector is forced (so the field that is changed is removed). It can simply be replaced with ↑ xs = xs at runtime and thus goes from linear (in the length of the vector) time complexity to a constant.

The following conversion functions from and to lists are also identity functions after forcing has been done:

```
fromList : ∀ {a} {A : Set a} → (xs : List A) → BoundedVec A (length xs)
fromList []        = []
fromList (x :: xs) = x :: fromList xs

toList : ∀ {a n} {A : Set a} → BoundedVec A n → List A
toList []        = []
toList (x :: xs) = x :: toList xs
```

Optimisations helping each other In section 4.1, the _⩽_ datatype was introduced, and it was noted that after forcing it could be represented as a primitive integer:

```
data _⩽_ : ℕ → ℕ → Set where
  z⩽n : ∀ {n}                  → zero  ⩽ n
  s⩽s : ∀ {m n} (lt : m ⩽ n) → suc m ⩽ suc n
```

[3] This code uses set level (universe) polymorphism, which is not relevant to our work, but explains why Set is given a parameter.

Now consider the following function, which is proof that a natural number m is always less than or equal to m + n (for all n):

```
lemma : (m n : ℕ) → m ⩽ m + n
lemma zero    n = z ⩽ n
lemma (suc m) n = s ⩽ s (lemma m n)
```

Notice that in the clause where the first parameter is zero, it returns the constructor of the _⩽_ datatype that corresponds to zero in its primitive representation. In the same way, the function returns the constructor that corresponds to suc in the clause where the first parameter is a suc. This means that the function is actually an identity function, and shows how forcing helps with finding more potential injective functions, and that some datatypes and functions on them can really get a lot more efficient.

4.4 Smashing

This section examines to what extent the smashing algorithm that was described in section 3.5 is applicable. In the following table some modules from Agda's standard library are listed. These modules have been picked to give an overview of how many functions return inferable data and can thus be smashed.

Module	Functions	Smashed	%
Algebra	905	71	8%
Data.Fin	23	3	13%
Data.Fin.Props	74	21	28%
Data.Nat.Properties	143	78	55%
Data.Nat	36	8	22%
Data.Vec	42	0	0%
Data.Vec.Properties	94	58	62%
Relation.Binary.PropositionalEquality	25	11	44%
Relation.Binary.Vec.Pointwise	46	5	11%

Table 4.1: The percentage of function smashed in a selection of modules

The module Data.Nat.Properties is filled with proofs returning equalities, so smashing is expected to be applicable to most functions there. So why are "only" 55% of the functions smashed? There are 26 functions which can not be smashed because they are either decidability predicates (i.e. they are functions returning fancy forms of Bool) or they build other structures which have multiple inhabitants. Most of these structures are models of algebraic structures which are records that have fields for operations which can not be inferred.

A majority of the functions, 34 to be exact, return something of the _⩽_ type. This type is by the current simple criteria not inferable, but a solution to this is discussed in section 5.4.

4.5 Case study: A lambda calculus

This section will give a bigger example of an Agda program and how various optimisations apply to it. The program is a normaliser for simply typed lambda calculus where natural numbers have been added, making the language slightly more interesting. This example will be using some of the techniques often used when working with dependent types in general.

The syntax of the terms will guarantee that the terms are well typed, so it has been fused with the corresponding typing rules. This makes it impossible to construct terms that are not well typed. This particular calculus contains two type formers, one for the type of natural numbers and one for the function space:

```
data Type : Set where
  ℕ#    : Type
  _⇒_  : Type → Type → Type
Ctx : Set
Ctx = List Type
```

Here $\mathbb{N}^{\#}$ is the syntactic representation of the type $\mathbb{N}$ in the calculus and similarly _⇒_ denotes the function space. The type Ctx represents a typing context. Variables will be de Bruijn indices[4], so Ctx is just a simple list of Types. Some more type safety than simple $\mathbb{N}$s for variables is desirable, so a type that guarantees that the variable points to a specific type inside the context is used instead. Forced arguments are marked with red colour and underlined.

```
data Var (τ : Type) : Ctx → Set where
  here  : ∀ {Γ}   → Var τ (τ :: Γ)
  there : ∀ {Γ σ} → Var τ Γ → Var τ (σ :: Γ)
```

This type Var τ Γ should be read as a variable of type τ inside the context Γ. The constructor here represents a variable that points to the top of context. Notice that the type τ is really at the top in context $\tau :: \Gamma$.

The second constructor there is used to point further away inside the context, so it requires a variable for Var τ Γ as an argument to get the variable of type Var τ $(\sigma :: \Gamma)$ for some σ that is on the top of the context.

After forcing the here constructor takes zero arguments, and the there constructor takes one recursive argument. It can therefore be implemented

[4]The variables refer to types in contexts by position instead of using names.

as a BigInt, so no extra price is paid for the extra type information used here over using $\mathbb{N}$ as indices. Here follows the definition of the terms in the language. The term e : Expr Γ τ denotes a term e such that $\Gamma \vdash$ e : τ.

```
data Expr (Γ : Ctx) : Type → Set where
  var    : ∀ {τ}   → Var τ Γ           → Expr Γ τ
  lam    : ∀ {τ σ} → Expr (τ :: Γ) σ   → Expr Γ (τ ⇒ σ)
  _$_    : ∀ {τ σ} → Expr Γ (τ ⇒ σ) → Expr Γ τ → Expr Γ σ
  zero   : Expr Γ ℕ#
  suc    : Expr Γ ℕ# → Expr Γ ℕ#
  natrec : ∀ {τ}   → Expr Γ ℕ# → Expr Γ τ
         → Expr (ℕ# :: τ :: Γ) τ → Expr Γ τ
```

Here the var simply wraps up a Var for the correct type τ in the context. The lam constructor produces a term of type $\tau \Rightarrow \sigma$ by taking a term in the context Γ extended with the type σ. This added type correspond to the variable that gets abstracted by the λ. The constructor _$_ represents function application. The function that gets applied has the type $\tau \Rightarrow \sigma$ so the argument must have the same type τ. Notice that the type τ is not forced since it does not occur in the result type Expr Γ σ.

The last three constructors are used for working with natural numbers. zero and suc are the basic constructors. The constructor natrec corresponds to the primitive recursion function, and is the eliminator for natural numbers. This function is implemented in Agda as follows:

```
Natrec : {A : ℕ → Set} → (n : ℕ) → A 0
       → ((m : ℕ) → A m → A (suc m)) → A n
Natrec zero    base rec = base
Natrec (suc n) base rec = rec n (Natrec n base rec)
```

Here it has been given a more dependent type than necessary, to illustrate that it implements the induction hypothesis of natural numbers.

The normalisation function will give an operational semantics to Expr Γ τ by returning the corresponding Agda term. The type of this Agda term depends on the value of τ and is given by the following function:

```
normal-T : Type → Set
normal-T ℕ#      = ℕ
normal-T (τ ⇒ σ) = normal-T τ → normal-T σ
```

The syntactic type $\mathbb{N}^{\#}$ becomes the Agda type $\mathbb{N}$ and the syntactic function space _⇒_ becomes functions in Agda. This explains how to translate τ to Agda, but before the normalisation function can be tackled some way of dealing with Γ needs to devised. For this purpose yet another data type

will be constructed: Env Γ which is a datatype for the variable environment for a typing context Γ.

```
data Env : Ctx → Set where
  []   : Env []
  _::_ : ∀ {Γ τ} → normal-T τ → Env Γ → Env (τ :: Γ)
lookup : ∀ {Γ τ} → Var τ Γ → Env Γ → normal-T τ
lookup here      (v :: _)    = v
lookup (there x) (_ :: rest) = lookup x rest
```

An Env matches a typing context, but stores actual values of the types in the context. The _::_ constructor contains a value of type normal-T τ and a recursive argument. It is thus similar to a list, but is not homogeneous as each element's type is specified by the context.

The lookup function takes a variable of type τ in context Γ and an environment following the context Γ and finds the value of that variable in the environment. Since the representation is typed a rather simple implementation can be given. This function works similar to how _!_ works for Vec. Because this function does not use the context Γ or the type τ both of these can be erased in their compiled versions, and since the environment is known to be _::_, the values needed can just be projected out. Here is the Epic version of the lookup function:

```
lookup_work idx env = if idx ≡ 0
  then env ! 0   -- ! is epic for projecting out field 0 of the constructor
  else lookup_work (idx - 1) (env ! 1)
```

Finally it is time for the normalisation function. This function will take a term of the Agda type Expr Γ τ and an environment Env Γ and produce a value of type normal-T τ.

```
normal : ∀ {Γ τ} → Expr Γ τ → Env Γ → normal-T τ
normal (var v) env = lookup v env
normal (lam e) env = λ x → normal e (x :: env)
normal (f $ x) env = normal f env (normal x env)
normal zero _      = 0
normal (suc n) env = suc (normal n env)
normal (natrec n g h) env = Natrec (normal n env) (normal g env)
                                   (λ a b → normal h (a :: b :: env))
eval : ∀ {τ} → Expr [] τ → normal-T τ
eval e = normal e []
```

The eval function does closed term normalisation, i.e starting in the empty environment. The real deal is the normal function which does the actual normalisation and is defined by cases on the Expr Γ τ:

case var v: Lookup the variable v in the environment.

case lam e: In this case the result should be of type normal-T $\tau \rightarrow$ normal-T σ where e : Expr $(\tau :: \Gamma)\ \sigma$. Since the result should be a function a lambda is returned that normalises e in the environment extended with x : normal-T τ.

case f \$ x: f will normalise to a function normal-T $\tau \rightarrow$ normal-T σ and x will normalise to normal-T τ. The result should be of type normal-T σ so the normalised value of f is applied to the normalised value of x.

case zero: zero is simply 0.

case suc n: suc maps to the suc constructor of type $\mathbb{N}$. Its argument is normalised to a $\mathbb{N}$ before being applied.

case natrec n g h: This corresponds to the Natrec function. When performing normalisation for h the environment is extended accordingly.

To round off this section an example of using the language will be shown. It is a simple addition function. The function plus is the Agda function that is gotten from evaluating add. What it reduces to is shown in the comment:

```
add : ∀ {Γ} → Expr Γ (ℕ# ⇒ ℕ# ⇒ ℕ#)
add = lam (lam (natrec (var (there here))
                       (var here)
                       (suc (var (there here)))))
plus : ℕ → ℕ → ℕ
plus = eval add  -- normalises to λ x y → Natrec x y (λ a b → suc b)
```

The plus function normalises to ordinary Agda functions, which indicates that eval is a good candidate for partial evaluation. The effectiveness of this is measured in section 4.7.

4.6 Case study: A file access DSL

One benefit of dependent types is that invariants about programs can be encoded in the types. In this section an embedded domain-specific language (DSL) for handling files in the same vein as presented by Edwin Brady and Kevin Hammond [BH10] is developed in Agda, and is then optimised using partial evaluation. The DSL will make sure that all files that are opened will eventually be closed, and that it is only possible to read from open files.

The basic approach here is to create a new language inside Agda, which makes use of the type system to guarantee the safety properties that are interesting. Later an interpreter is created that interprets this language

into the low-level primitive functions. For example, the primitive readFile, readLine and close functions have the following types:

```
data Purpose : Set where
  Reading : Purpose
  Writing : Purpose
openFile  : FilePath   → Purpose → IO FileHandle
readLine  : FileHandle → IO String
closeFile : FileHandle → IO Unit
```

The openFile function creates a FileHandle which can be passed to readLine, but there is no static check that the closeFile function has not been called in between. To remedy this the DSL introduces a new IO type called FIO which keeps track of the state of the FileHandles before and after running an action. The functions from above get the following types:

```
openFile : {n : ℕ} {ts : Files n} (fp : FilePath) (p : Purpose)
         → FIO ts (add (Open p) ts) (Handle (1 + n))
readLine : {n : ℕ} {ts : Files n} (i : Handle n)
         {p : openH i Reading ts} → FIO ts ts String
closeFile : {n : ℕ} {ts : Files n} (i : Handle n)
         → FIO ts (set i Close ts) Unit
```

Here the types are more complex, as they are carrying more information about the files. The openFile function uses add to add a new file to the state, which is Open for the particular Purpose (reading or writing). In a similar way closeFile will change the state by changing the file's particular Handle to Closed.

A new burden is put on the programmer, as she must now prove that the file she is trying to read is actually Open for Reading when calling readLine. This could be cumbersome but Agda can infer it when working on known File states.

Overhead of interpreting A price is paid for the extra static guarantees. The primitive IO functions are not called directly, but a datatype is inspected and depending on this, different function calls are dispatched. All file handles are also grouped inside an environment so that the file state may be updated correctly, which also adds an overhead.

The good news is that the program written using the DSL is mostly known at compile time, and by running the interpreter at compile time it is possible to get a program that does not mention FIO or use the environment at all. However, to get the functions to compute well statically, the FIO type will be written in continuation passing style, meaning that each constructor has a field representing what to do next. This field is a function

that takes the return value of the action and continues the execution. The important point is that the statically known data should not flow through functions that can only be evaluated during runtime (e.g the flow should not go through _>>=_).

```
data FIO (A : Set) {n : ℕ} (fs : Files n)
          : {n' : ℕ} (fs' : Files n') → Set where
  DONE  : (x : A) → FIO A fs fs
  OPEN  : ∀ {m} {fs' : Files m} (p : Purpose) → FilePath
          → (Handle (1 + n) → FIO A (add fs (Open p)) fs')
          → FIO A fs fs'
  CLOSE : ∀ {m} {fs' : Files m} (h : Handle n)
          → FIO A (set h Closed fs) fs' → FIO A fs fs'
  READ  : ∀ {m} {fs' : Files m} (h : Handle n)
         {prf : OpenH h Reading fs} → (String → FIO A fs fs')
          → FIO A fs fs'
  WRITE : ∀ {m} {fs' : Files m} (h : Handle n)
         {prf : OpenH h Writing fs} → String → FIO A fs fs'
          → FIO A fs fs'
```

All the constructors above (except DONE) have a continuation field of return type FIO. For example OPEN has a function Handle (1 + n) → FIO A (add fs (Open p)) fs'. The handle is for the newly opened file and the new FIO action has a state where a new file has been opened, and produces a new state fs'. A function that does not return a new result (like a new file handle), such as CLOSE, only has FIO A (set h Closed ts) ts' as its continuation.

The constructors READ and WRITE contain proofs that the handle that they either read from or write to is indeed open for the appropriate action. This proof can often be inferred when working with a closed term so it is marked as implicit with curly braces.

It is possible to write a new bind function for the FIO type – otherwise it would not be much of a monad. This function will be called _>>=′_ and is defined by pattern matching on the first argument:

```
_>>=′_ : ∀ {A B n m l} {fs₁ : Files n} {fs₂ : Files m} {fs₃ : Files l}
        → FIO A fs₁ fs₂ → (A → FIO B fs₂ fs₃) → FIO B fs₁ fs₃
DONE x              >>=′ f = f x
OPEN p fp k         >>=′ f = OPEN  p fp        (λ h → k h >>=′ f)
CLOSE h k           >>=′ f = CLOSE h           (      k   >>=′ f)
READ h {prf} k      >>=′ f = READ  h {prf}     (λ s → k s >>=′ f)
WRITE h {prf} s k   >>=′ f = WRITE h {prf} s   (      k   >>=′ f)
```

In the interpretation of FIO an environment of Files is needed to store the “real” file handles. This environment will be updated when interpreting

the OPEN and CLOSE operations, and will be sent to the continuation.[5]

```
runFIO : ∀ {A n m} {fs : Files n} {fs' : Files m} → Env n
       → FIO A fs fs' → IO A
runFIO env (DONE x)       = return x
runFIO env (OPEN p fp k)  =
  openFile fp (getMode p) ≫= λ h →
  runFIO (add env h) (k bound)
runFIO env (CLOSE h k)    =
  closeFile (env ! h)     ≫= λ _ →
  runFIO env k
runFIO env (READ h k)     =
  readFile (env ! h)      ≫= λ s →
  runFIO env (k s)
runFIO env (WRITE h s k)  =
  writeFile (env ! h) s   ≫= λ _ →
  runFIO env k
```

Now, what happens when (partially) evaluating runFIO empty (Open Reading "fp" (λ h → Close h (Done tt)))? Do the FIO and Env types in fact disappear?[6]

```
  runFIO empty (OPEN Reading "fp" (λ h → CLOSE h (DONE tt)))
⇝β
  openFile "fp" (getMode Reading) ≫= λ h →
  runFIO (add empty h) ((λ h → CLOSE h (DONE tt)) bound)
⇝β
  openFile "fp" (getMode Reading) ≫= λ h →
  runFIO (add empty h) (CLOSE bound (DONE tt))
⇝β
  openFile "fp" (getMode Reading) ≫= λ h →
  closeFile ((add empty h) ! bound) ≫= λ _ →
  runFIO (add empty h) (DONE tt)
⇝β {-(add empty h) ! bound ≡ h -}
  openFile "fp" (getMode Reading) ≫= λ h →
  closeFile h                     ≫= λ _ →
  return tt
```

Indeed they do! What is left is just the primitive IO operations, as if they were written without the extra type safety that the DSL provides.

An example where the computation gets stuck If the FIO type was not written in continuation passing style, but instead used an explicit BIND,

[5] bound : {n : ℕ} → Fin (suc n) will create a number as large as possible, denoting the end of environment.

[6] a $\rightsquigarrow_\beta$ b means that a evaluates to b.

the computation may not go through. This will demonstrate that known data that goes through abstract functions becomes abstract:

```
data FIO (A : Set) {n : ℕ} (fs : Files n) : {n' : ℕ} (fs' : Files n')
                                              → Set where
  ...
  BIND : ∀ {B n m l} {fs' : Files m} {fs'' : Files l}
       → FIO B fs fs' → (B → FIO A fs' fs'') → FIO A fs fs''
```

Here the environment corresponding to fs' will be needed and hence runFIO will have to return the new Env as well. If the return type is changed to IO (A × Env m) a problem occurs. What follows is the example from above with these changes made to it, which demonstrates the problem:

```
  runFIO empty (BIND (OPEN Reading "fp") (λ h → CLOSE h)
⇝β
  runFIO empty (OPEN Reading "fp") >>= λ v →
  runFIO (snd v) (CLOSE (fst v))
⇝β
  (openFile "fp" (getMode Reading)    >>= λ h →
    return (bound, add empty h)        >>= λ v →
  runFIO (snd v) (CLOSE (fst v))
```

Here the computation gets stuck since >>= is an abstract function and Agda does not know that the monad laws apply. In particular it does not know that >>= is associative[7] and that return is a unit[8].

There is another way to make this compute with the BIND constructor, and that is to instead use the return type IO A × Env m. The Env is passed around independently of the abstract IO type, and therefore the type is seen by Agda and can be gotten rid of using partial evaluation.

However, another problem arises when adding the BIND constructor like this: It is not logically sound. The B in the type of BIND is itself a type, and this is not allowed in Agda (due to paradoxes similar to Russel's paradox).

4.7 Benchmarks

This section presents four different benchmarks, which compare the runtime, amount of memory allocated, executable size and compilation time of the Epic backend with and without optimisations and the MAlonzo backend. In the last benchmark, an Agda program compiled using the Epic backend is compared to an equivalent Haskell program compiled with GHC.

All benchmarks were run on a computer with an Intel Core i7 920 processor and 6GB of memory. The memory allocation measurements were

[7]((m >>= λ h → f h) >>= λ v → k v) ≡ (m >>= \h → f h >>= \v → k v)

[8](return x >>= λ v → k v) ≡ k x

obtained from the garbage collector using the Epic backend and from the runtime system of GHC using both the MAlonzo backend and Haskell directly. Note that it is the total memory allocated during the whole run that is presented, and not the peak memory usage.

Markov chains This benchmark consists of running a Markov chain with 100 states for different numbers of steps. The implementation is doing n matrix/vector multiplications where the matrix' sides and the vector's length are 100. The matrix is implemented as a Vec (Vec Double 100) 100. The results are shown in table 4.2.

Backend		Epic	Epic unopt.	MAlonzo
Compilation time		2m 37s	2m 35s	1m 55s
Executable size		4.5MB	5.1MB	8.3MB
$n = 1$	Time	0.006s	0.006s	0.004s
	Alloc.	1.38MB	1.41MB	1.55MB
$n = 10$	Time	0.017s	0.018s	0.007s
	Alloc.	13.1MB	13.3MB	14.7MB
$n = 100$	Time	0.13s	0.14s	0.041s
	Alloc.	130MB	132MB	146MB
$n = 1000$	Time	1.31s	1.38s	0.42s
	Alloc.	1.30GB	1.32GB	1.46GB
$n = 5000$	Time	7.47s	8.13s	3.1s
	Alloc.	6.49GB	6.58GB	7.30GB

Table 4.2: Markov chain benchmark results

The example uses Agda's standard library, which is the reason for the rather long compilation times and large files. The speed improvements of using the optimised backend are not great in this program, since there are not that many opportunities for optimisations. The main improvement is that the representation of Vec does not store its length in each _::_ constructor as it is forced. This is visible in the rows labelled "Alloc.", which show the total memory allocated during the whole run (not to be confused with maximum memory usage). The Haskell compiler, GHC, produces such good code that it is hard to beat the MAlonzo backend in this benchmark, even though the MAlonzo backend is not performing optimisations itself.

Total parser combinators This benchmark is an example from a total parser combinator library developed by Nils Anders Danielsson [Dan10], which parses arithmetic expressions. It runs quite slow, which explains the short input strings. Table 4.3 below shows how the different Agda backends perform, where x is the program's input string.

	Backend	Epic	Epic, unopt.	MAlonzo
	Compilation time	4m 48s	6m 47s	1h 20m 25s
	Executable size	7.5MB	15MB	13MB
$x = 1$	Time	0.006s	0.006s	0.004s
	Alloc.	837kB	808kB	156kB
$x = 1 + 2$	Time	0.039s	0.039s	0.004s
	Alloc.	24.0MB	25.8MB	381kB
$x = 1 + 2 * 3$	Time	0.75s	0.86s	0.005s
	Alloc.	435MB	483MB	1.65MB
$x = 1 + 2 * 3 + 4$	Time	25.1s	27.0s	0.009s
	Alloc.	10.4GB	11.6GB	7.75MB
$x = 1 + 2 * 3 + 4 * 5$	Time	17m 48s	18m 58s	0.028s
	Alloc.	297GB	333GB	27.6MB

Table 4.3: Total parser combinator benchmark results

The executables produced by the Epic backend perform slowly in this benchmark, and their running time grows much faster as the input size grows compared to those produced by the MAlonzo backend. The probable cause is that combinator libraries like this one benefit from laziness, as control structures unnecessarily evaluate their arguments otherwise. This is discussed more in section 5.7.

Even though the Epic backend cannot compete with the MAlonzo backend in this benchmark, it shows that the optimisations that are performed in Epic really do help compared to not using them. Also note that the compilation time with optimisations on is actually lower than without optimisations. Even though the compiler does more work, it is faster, because the code becomes smaller.

Most of the compilation time using the MAlonzo backend is spent in the Haskell compiler GHC (probably doing type checking) and using the Epic backend in the Epic compiler (which in turn calls the C compiler GCC). The size of the modules in Agda's standard library, which is not tailored for compiling programs, contributes to the slowness. The compilation time and code size may be helped further by removing unused functions, which is discussed in section 5.2.

Simply typed λ-calculus This benchmark uses the lambda calculus that was defined in section 4.5. The multiplication and power function have been defined using `natrec` in this language (in a similar style to the `add` function which was presented in that section).

The benchmarks, shown in table 4.4, are simple:

- The first one calculates $500 \cdot 500$, i.e `eval mul 500 500` where `mul` is the term for multiplication.

- The second one calculates 10^5, i.e eval pow 10 5 where pow is the power function.

		eval mul 500 500		eval pow 10 4		
Backend	Optimisation	Time	Alloc.	Time	Alloc.	Size
Epic	opt + static	0.15s	96.2MB	0.84s	492MB	4.5MB
Epic	opt	0.23s	161MB	1.37s	850MB	4.5MB
Epic	opt + no-forcing	0.25s	185MB	1.35s	984MB	4.5MB
Epic	no-opt	0.25s	185MB	1.69s	1.03GB	4.5MB
Epic	no-opt + no-forcing	0.25s	186MB	1.48s	984MB	4.5MB
MAlonzo		0.15s	49.1MB	2.91s	283MB	8.1MB
GHC		0.40s	115MB	16.4s	668MB	1.0MB
GHC	-O2	0.18s	57.0MB	2.62s	302MB	0.9MB

Table 4.4: Simply typed lambda calculus benchmark results

The benchmarks were run using some different optimisation configurations, showing what yields the most gains:

opt means that all optimisations are on

static means that the eval function is marked as STATIC

no-forcing means that forcing is not performed

no-opt means that no optimisations other than forcing and finding primitive data are performed

An equivalent lambda calculus was also written in Haskell (presented in appendix B) and compiled using GHC with different optimisation flags.

The table shows that the Epic backend beats both the MAlonzo backend and GHC when evaluating the power function (comparing execution time), and it also gets close in the multiplication case. It can be seen that all optimisations do their part of the job to yield faster programs; the programs run a bit faster for each optimisation that is turned on.

The power function actually runs slower when using forcing without the other optimisations (the **no-opt** case). This is because a function on forced data may sometimes do more work to "dig out" the forced variables, while the unforced version can get it directly from a constructor's fields (though it uses more memory for the representation).

The factorial function This benchmark shows the infamous factorial function, written equivalently in Haskell (left) and in Agda (right):

```
fac :: Integer → Integer                       fac : ℕ → ℕ
fac = fac' 1                                   fac = fac' 1

fac' :: Integer → Integer → Integer            fac' : ℕ → ℕ → ℕ
fac' a 0 = a                                   fac' a zero    = a
fac' a n = fac' (a * n) (n - 1)                fac' a (suc n) = fac' (a * suc n) n
```

In this benchmark, the MAlonzo backend did not perform well and was only able to produce results for inputs below 9. For fac 9 it is more than 65 times slower than Epic. This is because MAlonzo converts natural numbers between Integers and a Peano style inductive datatype. When performing addition and multiplication an Integer is used, but otherwise the inductive datatype is used. Due to MAlonzo not being able to compete, only Epic and Haskell are shown in the results, in table 4.5.

	Epic	GHC	GHC -O2
Executable size	78kB	1.2MB	1.2MB
$n = 9$	0.008s	0.003s	0.003s
$n = 100$	0.008s	0.004s	0.004s
$n = 1000$	0.010s	0.004s	0.004s
$n = 10000$	0.065s	0.061s	0.034s
$n = 100000$	5.71s	18.42s	3.82s

Table 4.5: Factorial benchmark results

This shows that the Epic backend is almost as performant as Haskell compiled with GHC. The reason for GHC without optmisations being so slow for n=100000 is probably due to laziness. The arguments to fac' become thunks so all the multiplications are delayed.

5

Discussion

THIS chapter hosts a discussion about the way we solved some problems in our backend and presents some future work that can be done in areas related to compiling and optimising Agda. It includes a general comparison of forcing and eliminators in section 5.1. Different ways of doing type erasure, and a possible extension to our approach to also remove unused definitions is discussed in section 5.2. The way that the tags are chosen to benefit injection detection is discussed in section 5.3.

An optimisation that we have not implemented, but that could prove to work well is collapsing, which would for example help the smashing optimisation. It is discussed in section 5.4. Currently we have a fast BigInt representation for some data types, but not for all operations on them. How to do this and how more datatypes can get a better representation is discussed in section 5.5.

Section 5.6 presents some problems with doing partial evaluation and how they can be solved. Some points on the evaluation strategy are made in section 5.7, and section 5.8 forms an opinion of whether the choice of compiling to Epic is the right one.

5.1 Forcing

Forcing is an important optimisation as it enables other optimisations, most notably using primitive data representations, and makes dependently typed programs more space-efficient.

In Epigram [BMM04], forcing becomes different than what has been presented here. This is because the Epigram is a language that uses elimination operators, and these are the only way to deconstruct data. If that operator is changed to not use any forced variable, all functions over a datatype to which forcing applies automatically gain the benefits of forcing.

The current Agda implementation uses case splitting trees, meaning that all functions can pattern match on data. All functions over a datatype thus have to be rewritten when forcing has been done. The drawback of this is that a lot of extra work has to be performed, whereas only one function has to be changed when using eliminators. The algorithm for removing forced variables in the context of case splitting trees is also complicated to get right.

5.2 Erasure

In the Cayenne language [Aug98], type erasure is done by removing types that are used as arguments to functions, record components or the result type of a function. Cayenne is similar to Agda in that it has no way to perform pattern matching on a type. This means that it may be possible to do this form of type erasure also in Agda.

The route that we took to solve the problem is detecting unused arguments (which will detect most types) and creating wrapper functions calling worker functions without the unused arguments. Our solution is simple and also has the benefit of removing not just types, but anything that is not used in a computation. It may, however, not be as efficient as Cayenne's solution, depending on how many wrapper function calls can be inlined (which is where the real removal of arguments happens). Also, this approach does not address erasure for higher order functions, which is something Cayenne does.

If Agda's metatheory allows it, the optimal solution would be to first remove types. After that has been done we can also perform the worker/wrapper transform to remove unused arguments (which will not be as many in this case).

Removal of unreachable definitions Another optimisation that would be interesting to investigate is the removal of definitions in a module that are actually unreachable; dead code elimination but on the function level. This is important since Agda allows modules inside modules and also parameterised modules. A parameterised module has arguments that can be used in its definitions.

```
module Map (Key : Set)
           (A : Set)
           (_<_ : Key → Key → Bool) where
  data Map : Set where
    ..
  empty  : Map
  insert : Key → A → Map → Map
  lookup : Key → Map → Maybe A
module UseOfMap
  open Map ℕ String Nat._<_
  test : Map
  test = insert 0 "hello" empty
```

The module UseOfMap instantiates the arguments of Map. Internally these modules will be translated to modules without parameters, by turning them into parameters to the definitions instead. The places where the

module gets opened and applied turns into a new module with new definitions where the parameter arguments have already been applied. This new module introduces a lot of new definitions, some of which may never be used.

The example above will be compiled to something similar to the following:

```
module Map where
  data Map (Key : Set)
           (A : Set)
           (_≤_ : Key → Key → Bool) : Set where
    ..
  empty  : (Key : Set) (A : Set) (_≤_ : Key → Key → Bool)
         → Map Key A _≤_
  insert : (Key : Set) (A : Set) (_≤_ : Key → Key → Bool)
         → Key → A → Map Key A _≤_ → Map Key A _≤_
  lookup : (Key : Set) (A : Set) (_≤_ : Key → Key → Bool)
         → Key → Map Key A _≤_ → Maybe A
module UseOfMap where
  module Map' where
    Map    = Map.Map    ℕ String Nat._<_
    empty  = Map.empty  ℕ String Nat._<_
    insert = Map.insert ℕ String Nat._<_
    lookup = Map.lookup ℕ String Nat._<_
  open Map'
  test : Map
  test = insert 0 "hello" empty
```

These definitions take up a lot of space in the code, and makes compilation take more time than necessary. There are also functions that may not be used anymore due to optimisations. Smashing can for example make helper lemmas for equality proofs unused and unreachable if they are private to a module.

Because of the issues with having unnecessary definitions it would be beneficial to add functionality to remove unused definitions as well as unused arguments.

5.3 Injection detection

The same optimisation as the injection detection presented in this thesis has been described in Edwin Brady's PhD thesis [Bra05]. Our work extends it by considering injective functions that operate on constructors of different datatypes whose representations will (partly) be the same, and makes the

selection of tags only after potentially injective functions have been identified.

Just like Brady's work, our implementation may still suffer the problem with mutually recursive definitions, which may not be detected if reduction of the body of one of a function f's clauses calls a mutually recursive function g with unknown arguments, so that it is not possible to reduce the call to g further, even though it may be an injective function at runtime. The problem is that g may be an identity function in one argument as well, and this fact depends on the fact that f is an identity function in one argument. But to try all argument combinations that f and g may be identities in would be too costly, so a smarter algorithm has to be be devised.

Choosing constructor tags The way we choose constructor tags has been tested on Agda's standard library and gives the expected results. It can, however, yield results that are not perfect, as the method does not take into account what potentially injective functions are blocked by marking another function as an injection. This means that depending on the order of functions that the algorithm is run on, it may choose constructor tags to make different functions injections.

A way to solve this problem would be to use a heuristic when unifying the preconditions for injective functions to maximise the number of functions that are solved, or to solve functions which are the most important before other functions.

We have not yet run into any problems with using the current simplified approach, as most functions' preconditions can be solved, but there may be cases where it performs badly. A point to remember is that all candidate injection functions can not always be chosen to be injections at the same time, so a perfect solution is impossible.

Another problem is that the constructor tags are chosen per module, which is because modules are compiled separately in the backend to not have to recompile if only one file is changed. For each module that is compiled, an Agda/Epic-interface file is created, which among other things contains information about constructor tags.

This means that if lists are defined in one module, vectors are defined in another module and a third module implements the forget : ∀ {A n} → Vec A n → List A function there is a possibility that lists and vectors do *not* get the same constructor tags, as they were chosen before the forget function was known. If, however, the forget function is in either of the list and vector modules, it will work as expected.

5.4 Smashing

The presented criterion for inferability does not work for functions for which a constructor can be determined by its indices. For example, a value of type Vec A 0, vectors of length zero, only has one runtime representation but it is not caught by the criterion since Vec has two constructors.

A way to solve this problem would be to try to unify the return type of the function with the type of the different constructors in the datatype. If the return type can only be unified with one of the constructors, it means that the function has to return that constructor.

Another possible extension is to walk through the term to find smashable subterms. Since the arguments to a function can have known values in an application, this may allow the return type to reduce to something smashable, meaning that the application can be replaced with the smashed value.

Collapsing The type _⩽_ represents the less than or equal relation between two numbers, so it is only possible to construct m ⩽ n if m is really less than or equal to n. This is can be defined in Agda as follows (forced arguments are red and underlined):

```
data _⩽_ : ℕ → ℕ → Set where
  z⩽n : {n : ℕ}                 → zero  ⩽ n
  s⩽s : {m n : ℕ} → m ⩽ n → suc m ⩽ suc n
```

It is possible to perform yet another optimisation called **detagging** [BMM04] on this type. It depends on the following observation: The first index completely determines which constructor it can be. Given a term le : zero ⩽ n it is known that le = z⩽n {n} since no other type is possible. Similarly it must be s⩽s if the first index is suc m for some m.

If detagging is implemented there would be no need to store anything in the _⩽_ datatype: No tag since it is detaggable and can therefore be inferred, and no argument since they are removed by forcing. The only thing left is the recursive argument m ⩽ n in s⩽s, but by an inductive reasoning this does not contain any information either. By this we can replace _⩽_ by a simple constant[1]. When a combination of detagging and forcing produces a type in which it is possible to decide which constructor it is with only information from indices and where all non-recursive arguments are forced, it can be replaced with a constant. This is called **collapsing** and is beneficial for space, since it reduces terms of the type to just a constant. Because it is a constant, it is inferable what value it is during runtime, and hence functions returning _⩽_, or any other collapsible types, could also be smashed.

[1] This relies on type checking since it is important to not have a constant like 5 ⩽ 2.

5.5 Primitive data

The translation of the Fin type to BigInt makes it more space-efficient. But it also makes new optimisations possible, namely making use of the time-efficient operations that BigInt provides. Currently we only use these operations when they are marked as built-in, but there are no such built-in operations for the Fin type.

It would be interesting to try to detect functions that can be represented by primitive operations. Below is an addition function _+F_ for Fin which could be implemented as addition on BigInts.

```
toN : {m : ℕ} → Fin m → ℕ
toN fz     = zero
toN (fs i) = suc (toN i)
_+F_ : {m n : ℕ} (i : Fin m) → Fin n → Fin (toN i + n)
fz   +F j = j
fs i +F j = fs (i +F j)
```

Another way of defining an addition on the Fin datatype is the following:

```
raiseIdx : {n : ℕ} (m : ℕ) → Fin n → Fin (m + n)
raiseIdx zero    j = j
raiseIdx (suc n) j = raiseIdx n j
_+F'_ : {m n : ℕ} → Fin m → Fin n → Fin (m + n)
(zero {n}) +F' j = raiseIdx (suc n) j
suc i      +F' j = suc (i +F' j)
```

For this to be detected as an operation on natural numbers, it would have to use the information that raiseIdx is an identity function in its second argument. This once again shows that the optimisations sometimes rely on eachother.

In the same way that injection detection is not limited to constructors of the same datatype, there is no need to restrict the arguments and results to be of the same type[2]. As long as the used arguments and the result are all represented by BigInts at runtime it will work. An example is the raise function which increases the value of a Fin by a ℕ. This can also be represented as BigInt addition at runtime.

```
raise : {m : ℕ} (n : ℕ) → Fin m → Fin (n + m)
raise zero    i = i
raise (suc n) i = fs (raise n i)
```

[2]Not even _+F_ conforms to this.

More types of data There may be ways to represent many other datatypes than ones that look like natural numbers more efficiently. The general requirement is that there is a bijection between the new representation and the constructor representation. For example, another datatype that could be represented as a BigInt is the following definition of positive binary numbers:

```
data Binary : Set where
  _I : Binary → Binary
  _O : Binary → Binary
  bI : Binary

five : Binary
five = bI O I
```

To make an alternative encoding of a datatype, a bijection between the old and the new representation is required.

The problematic bit about detecting this bijection is which of the constructors _I and _O should be 1 and 0 respectively. Of course one could make an arbitrary choice, but it would be better to make the choice based on what allows us to detect the largest number of operations (e.g _+_ and _*_).

Another datatype that could be represented differently is Vec, which could use an array (a continuous memory region). Pattern matching and returning sub vectors could be done efficiently by using pointers into the array, and indexing functions could then be detected and changed to use array indexing. A problem is functions that build new vectors, which may be more expensive using this representation.

5.6 Partial evaluation

The use of partial evaluation can often remove the overhead of certain abstractions. When using domain-specific languages the overhead of interpreting can for example be removed in some cases. Here it is important to note that for the example to work the abstraction needs to be able to compute well during open-term evaluation.

If the function that we want evaluated at compile time depends on a value that is unknown at compile time (from for example pattern matching), the computation will block. One technique to solve this is to move the pattern matching to a separate function that receives continuation arguments for the different outcomes.

Consider this rather contrived example: An expression language with one global boolean variable. The language has an IfThen_Else_ construct to perform a choice depending on this variable.

```
data Expr : Set where
  zero   : Expr
  suc    : Expr → Expr
  IfThen_Else_ : Expr → Expr → Expr
  _+_ : Expr → Expr → Expr
  -- Assume the Bool is only known during runtime
eval : Expr → Bool → ℕ
eval zero     b = 0
eval (suc n) b = suc (eval n)
eval (x + y) b = eval x b ℕ.+ eval
eval (IfThen a Else c) true = eval a true
eval (IfThen a Else c) false = eval c false
```

If the Bool is not known at compile time a partial evaluation of eval applied to a IfThen_Else_ constructor will get stuck, and no further computation will occur. This can be remedied as explained above, by changing the last two lines to:

```
eval (IfThen a Else c) b = if b then eval a true else eval c false
```

This will continue to compute, even if the b is unknown, and leaves the if_then_else_ to be evaluated at runtime. Notice that in the branches the value of the Bool is known so IfThen_Else_ constructors inside these can be evaluated completely.

Edwin Brady and Kevin Hammond have a more general partial evaluator in their paper [BH10]. In their paper, recursive functions which can not be reduced in all cases can still be specialised for the known input. In this scheme a new function is created for each of the static applications of known arguments. The partial evaluator then uses these functions instead of the general one when the evaluation has failed to give a result. This is something that we have not yet implemented, due to the need to then change the normalisation function in Agda to support it, but it would still be an interesting direction to take in the future.

5.7 Laziness

Epic uses strict evaluation, which means that the arguments to a function are evaluated before the function is entered. It does, however, have a lazy keyword, which can be used to create a lazy expression which is only evaluated when its value is needed. This is currently only used for co-inductive definitions (not dealt with in this thesis) in our compiler backend, and for defining common default branches from case splitting trees.

When defining your own control structures, for example the if_then_else_ function, it is sometimes desirable to have laziness in specific arguments of

a function. The if_then_else_ function would benefit from being lazy in the second and third arguments, as only one of them is returned (depending on the value of the first argument, the Bool). Without laziness, the function performs some unnecessary work each time it is called. The good thing about Agda is that we know that all arguments to if_then_else_ terminate, which means that the result of the program (save for the running time) is not affected by evaluating the arguments strictly.

One way to solve this problem would be to let the programmer mark arguments as lazy in functions that need them. Another way would be to perform a laziness analysis, which would detect functions where an argument is not used in all pattern matching clauses and thus might benefit from laziness.

A drawback of both approaches is that Epic currently has no way to mark arguments to functions as lazy – it only has a keyword to mark expressions as lazy. Laziness then has to be marked at the call site. In some cases this leads to not being able to provide laziness where it might be desired. For example, consider the following function:

```
test : {A : Set} ( Bool → A → A → A)
                 → Bool → A → A → A
test f b x y = f b x y
```

The test function may be called with if_then_else_ as the first argument, but it is not possible for the compiler to know that it would be a good idea to mark the arguments as lazy in the call to the f function, as it may also be some other function with the same type signature.

To solve this problem, Epic could be changed to allow marking function arguments as lazy or to be lazy by default.

5.8 Target language

The results of the benchmarks show promising results for our new backend, but it is still hard to beat the Haskell compiler GHC, which is very mature and produces fast code. In comparison, Epic is a new and experimental research language, which does not always produce code of the same quality.

Many of the optimisations shown here would also apply to other target languages than Epic, so it would be interesting to see them applied to a backend targeting another language. To get around the problem of type checking and having to use type coercions when using Haskell as the target language, it may be possible to plug into the GHC API at a later stage, to still be able to reap the benefits of that compiler's optimisations and get lazy evaluation which may help performance when making your own combinator libraries.

6

Conclusion

THE goal of this thesis has been to create a compiler for Agda that produces efficient programs. To do so, a big part of the work has gone into optimisations. As programs written in a dependently typed language often differ from those of conventional programming languages, it also means that the optimisations that help the most are different ones. There are also optimisations that simply do not apply to conventional languages. The focus of our work has been on optimisations that are specific to dependently typed languages, as that has previously been explored to a lesser extent.

We have looked at a number of different optimisations (chapter 3), showcased the results by examples and measurements (chapter 4) and lastly discussed what could have been done differently and in what direction future work can be taken (chapter 5).

The optimisations fall into two broad categories:

Data representation Agda has a minimal core and even natural numbers are defined as ordinary inductive datatypes. In a compiled program, that representation is not efficient enough, which is why we look at the shape of datatypes to determine if they can be represented more efficiently on the target platform by using native integer representations, and if so, rewrite the code to do so. This has been explored previously in Edwin Brady's thesis [Bra05].

Dependent types allow for a new kind of expressiveness in defining datatypes, but it also means that naïve data representations would store some values in constructor fields which are used as type indices. It has previously been shown that these can be removed, as they can always be inferred from the context [BMM04]. This optimisation is called forcing and it also means that more datatypes can be represented natively, as some datatypes are isomorphic to natural numbers once their forced arguments have been removed.

In the context of Agda, the forcing algorithm takes a different form than what has been dealt with previously, since Agda has pattern matching while Epigram uses an underlying type theory with elimination operators.

Term optimisations Parts of terms written in a dependently typed language may not have any computational meaning and can be removed

at runtime. An example is types, which only have a meaning statically and can be removed at runtime. This sort of type erasure has been done before in the Cayenne language [Aug98]. We present an implementation that uses an abstract interpretation and simply looks for unused arguments, which means that it can remove more than just types.

Another source of inefficiencies is computations that do something complicated but can only have one outcome at runtime. Since Agda is pure and does not allow non-terminating or partial functions it means that these computations can simply be replaced by their values. We call this optimisation smashing.

Some functions may be (recursive) identity functions in disguise in their compiled representation. By exploiting our knowledge of the low-level representation of data we can choose constructor tags in a way that even allows functions between different datatypes to be replaced with non-recursive identity functions.

The work that we have presented in this thesis has been carried out in a real-world implementation of a dependently typed programming language. Our work shows that there are a lot of opportunities for optimisations in languages of this kind, and it is very likely that there are many more optimisations that can be carried out (some have even been mentioned in chapter 5). While there is still room for improvement, our compiler certainly makes Agda more viable to use as a programming language by producing smaller and faster programs faster than before.

A

Foreign function interface

There is an Agda pragma, COMPILED_EPIC, which makes it possible to run arbitrary Epic functions in an Agda program. The following example shows how it is used:

```
postulate natToString : ℕ → String
{-# COMPILED_EPIC natToString (n : BigInt) → String
   = bigIntToString (n) #-}
```

The pragma looks like an ordinary Epic definition, but the first argument is parsed as an Agda identifier (natToString in the example above). Each reference to the postulate in the Agda code will, in the generated Epic code, be to a definition where the code from the COMPILED_EPIC pragma has been pasted verbatim.

As Epic also has a foreign function interface, i.e making it possible to call C functions, this functionality is automatically exposed in Agda. The following example defines a function numArgs which calls the C function numArgsBig:

```
postulate numArgs : ℕ
{-# COMPILED_EPIC numArgs () → BigInt
   = foreign BigInt "numArgsBig" () #-}
```

In this way it is possible to create bindings to C libraries.

IO Similar to the way it is done in Haskell, IO is implemented as a monad. The interface is built up using postulated functions whose definitions are given in COMPILED_EPIC pragmas.

```
postulate
  IO       : Set      → Set
  return   : ∀ {A}    → A    → IO A
  _≫=_     : ∀ {A B}  → IO A → (A → IO B) → IO B
  putStrLn : String   → IO Unit
  readStr  : IO String
```

To make the order of IO actions explicit, the type IO A is modelled in Epic as the type Unit → A. Since Epic does not make a distinction between pure and impure functions this function can also have a side effect.

The point is that the side effect will not happen until the function is applied to a unit.

In Agda it is not possible to make a call to an IO function from a pure (non-IO) function using this interface, since the type IO A cannot be used in place of A.

The return function just returns its argument when applied to a unit. Its first argument is the type A which is seen in the postulated signature of the function. The type argument will not be used, so it is safe to assume that it is a unit.

```
{-# COMPILED_EPIC return (a : Unit, x : Any, u : Unit) → Any
   = x #-}
```

The bind function _>>=_ runs two computations sequentially. In its Epic definition the first two arguments are the types A and B from the signature, and its last argument is the unit it gets from being an IO computation. The Epic pragma %effect is used to mark effectful computations as such, so that Epic does not optimise away the side effect.

```
{-# COMPILED_EPIC _>>=_
  (a : Unit, b : Unit, ioa : Any, f : Any, u : Unit) → Any
   = %effect (f (%effect (ioa (u)), u)) #-}
```

The functions putStrLn and readStr simply call the functions that do the job that is wanted. The only difference between these and an ordinary definition of these functions in Epic is that they take the above-mentioned unit.

```
{-# COMPILED_EPIC putStrLn
  (a : String, u : Unit) → Unit
   = epic_putStrLn (a) #-}
{-# COMPILED_EPIC readStr
  (u : Unit) → String
   = epic_readStr () #-}
```

The main function in an Agda program acts as its entry point, and has type IO A for some type A. When the Epic code is generated, the main function is applied to a unit, so that it is run.

The following code is an example of a simple IO program:

```
main : IO Unit
main = putStrLn "What is your name?" >>= λ _    →
       readStr                       >>= λ name →
       putStrLn ("Hello, " ++ name ++ "!")
```

$\mathcal{B}$

Haskell lambda calculus

```
data Var t g where
  Here  :: Var t (t, g)
  There :: Var t g → Var t (s, g)

data Expr g t where
  Var    :: Var t g → Expr g t
  Lam    :: Expr (t, g) s → Expr g (t → s)
  App    :: Expr g (t → s) → Expr g t → Expr g s
  Zero   :: Expr g ℕ
  Suc    :: Expr g ℕ → Expr g ℕ
  NatRec :: Expr g ℕ → Expr g t → Expr (ℕ, (t, g)) t → Expr g t

data Env g where
  Nil  :: Env ()
  Cons :: t → Env g → Env (t, g)

lookupEnv :: Var t g → Env g → t
lookupEnv Here      (Cons v _)    = v
lookupEnv (There i) (Cons _ rest) = lookupEnv i rest

natrec :: ℕ → t → (ℕ → t → t) → t
natrec 0 base rec = base
natrec n base rec = rec n' (natrec n' base rec)
  where n' = n - 1

normal :: Expr g t → Env g → t
normal (Var v)          env = lookupEnv v env
normal (Lam e)          env = λ x → normal e (Cons x env)
normal (App f x)        env = normal f env (normal x env)
normal Zero             _   = 0
normal (Suc n)          env = 1 + normal n env
normal (NatRec n g h)   env = natrec (normal n env) (normal g env)
                                (\a b → normal h (Cons a (Cons b env)))
```

```
eval :: Expr () t → t
eval e = normal e Nil


add :: Expr g (ℕ → ℕ → ℕ)
add = Lam (Lam (NatRec (Var (There Here))
                       (Var Here)
                       (Suc (Var (There Here)))))


mul :: Expr g (ℕ → ℕ → ℕ)
mul = Lam (Lam (NatRec (Var (There Here))
                       Zero
                       ((add `App` Var (There (There Here)))
                             `App` (Var (There Here)))))


power :: Expr g (ℕ → ℕ → ℕ)
power = Lam (Lam (NatRec (Var (There Here))
                         (Suc Zero)
                         ((mul `App` Var (There (There Here)))
                               `App` (Var (There Here)))))
```

Bibliography

[Abe10] Andreas Abel. MiniAgda: Integrating sized and dependent types. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Workshop on Partiality And Recursion in Interative Theorem Provers (PAR 2010), Satellite Workshop of ITP'10 at FLoC 2010*, 2010.

[Aug84] Lennart Augustsson. A compiler for lazy ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 218–227, New York, NY, USA, 1984. ACM.

[Aug98] Lennart Augustsson. Cayenne - a language with dependent types. In *ICFP*, pages 239–250, 1998.

[Ben07] Marcin Benke. Alonzo - a compiler for agda. 2007.

[BH10] Edwin Brady and Kevin Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 297–308, New York, NY, USA, 2010. ACM.

[BMM04] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs, Torino, 2003, volume 3085 of LNCS*, pages 115–129. Springer-Verlag, 2004.

[Bra05] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, 2005.

[Dan10] Nils Anders Danielsson. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 285–296, New York, NY, USA, 2010. ACM.

[Dyb91] Peter Dybjer. *Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics*, pages 280–306. Cambridge University Press, New York, NY, USA, 1991.

[EJ00] Martin Erwig and Simon Peyton Jones. Pattern guards and transformational patterns. In *In Haskell Workshop*, pages 12–1, 2000.

[GH09] Andy Gill and Graham Hutton. The Worker/Wrapper Transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.

[Let03] Pierre Letouzey. A new extraction for Coq. In *Proceedings of the 2002 international conference on Types for proofs and programs*, TYPES'02, pages 200–219, Berlin, Heidelberg, 2003. Springer-Verlag.

[MM04] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[OCa] Objective Caml (OCaml) programming language website. http://caml.inria.fr/.

[P+03] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

[PWW04] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2004.

[SSW10] Aaron Stump, Vilhelm Sjoeberg, and Stephanie Weirich. Termination casts: a flexible approach to termination with general recursion. 2010.

[Swi11] Wouter Swierstra. Adventures in extraction, 2011.

[Tea09] The Coq Development Team. *The Coq Proof Assistant Reference Manual - Version 8.2-bugfix*, 2009.