A modern back-end for a dependently typed language

MSc Thesis (*Afstudeerscriptie*)

written by

Remi Turk
(born January 11, 1983 in Rotterdam)

under the supervision of **Dr Andres Löh** and **Dr Piet Rodenburg**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
October 19, 2010	Dr Andres Löh
	Dr Piet Rodenburg
	Prof. Dr Frank Veltman
	Prof. Dr. Jan van Eijck

Institute for Logic, Language and Computation

Contents

Abstract

This thesis describes the connection between Agda, a modern dependently typed functional language and GRIN, a modern functional back end language, and compares the resulting system with the existing alternatives.

Acknowledgements

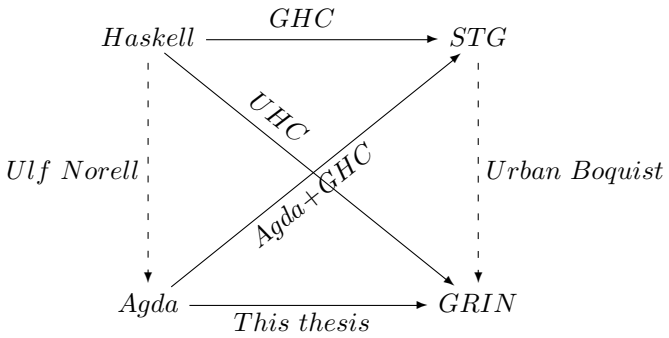
First I would like to thank both of my advisors, Andres Löh and Piet Rodenburg, for supervising me for over four times the originally allotted time. I would also like to thank Wouter Swierstra for passing on the original idea for this thesis to Andres. Atze Dijkstra and Jeroen Fokker were able to help me on whenever I got stuck with their EHC/UHC compiler, for which I thank them. I am grateful to my fellow students Chris Regenboog and Arvid Halma for keeping me company during many hours of staring at LaTeX and, to a small extent, beer and tea glasses. Finally, I would like to thank my father for relentlessly bothering me to finally finish this thesis.

1 Introduction

The field of functional programming has undergone two major evolutions in the last few decades. The first change is that whereas two decades ago a functional program usually ran multiple orders of magnitude slower than its imperative counterpart, compiler technology for functional languages has now reached the point of being capable of generating code that is in some cases competitive with C and FORTRAN. Second, the type systems of new functional programming languages are becoming more and more complicated, moving from being based on the simply typed λ-calculus to being based on λ-calculi like the calculus of inductive constructions. These systems are blurring the lines between programming languages and type theories, thereby situating themselves between editors and compilers on the one hand, and proof assistants on the other hand.

Relatively little work has been done however on connecting these two parts of the functional programming world. In this thesis, a newly built back-end for Agda, a new functional language based on dependent type theory, is introduced. This new back-end is based on GRIN, which is a modern back-end language designed for optimization and efficient compilation to recent processor architectures. In addition, it is shown how existing optimizations for dependently typed languages can be introduced into and generalized using the GRIN optimization framework.

The following diagram gives an overview of the current situation:



In it, a solid arrow ($\longrightarrow$) from a to b with label c means that language a is being compiled to language b by compiler c. A dashed arrow ($- \ - \ \rightarrow$) from a to b with label c implies that b was designed with a as a basis or inspiration by c.

In the left top of the diagram is Haskell, the canonical example of a lazy functional language. The top arrow is GHC, the most commonly used compiler for Haskell, which uses STG as its back-end intermediate language.[12] Then in 1999, Urban Boquist presented a new back-end language in his PhD thesis.[8] This new language, called GRIN, was intended for (whole-program) optimization and efficient compilation to modern CPU architectures. His prototype back-end using GRIN was, however, never published. The Utrecht Haskell Compiler[6] is a recently released compiler that compiles Haskell into GRIN (and ultimately into machine code).

However, the only currently available Agda compiler compiles Agda to Haskell (with some GHC specific extensions), and then uses GHC to generate an executable program. This is suboptimal due to the imperfect match between (mostly the type systems of) Agda and Haskell and because Haskell is intended as a programming language for human beings, not as a target for compilers. Consequently, much information that could be useful in later stages of compilation is lost in the translation from Agda to Haskell. In addition, GHC uses the rather old STG as its back end language.

One thing that is not shown in the diagram is the work of Edwin Brady. In his PhD thesis[10] he describes a compiler including a number of optimizations for Epigram, which is a predecessor to Agda. However, the back end language used in his thesis is the G-machine, which is a predecessor of STG that is even less suited for compilation to modern micro architectures.

This thesis introduces the first compiler that uses the modern back end language GRIN to compile the modern dependently typed language Agda. Furthermore, it is shown how three optimizations that were developed for Agda's close cousin Epigram can be generalized within the GRIN optimization framework, and may then even be used to subsume an important part of the current GRIN optimizations.

1.1 Overview of the following chapters

In chapter 2, the programming language Agda is introduced. This introduction is not meant as a programming tutorial, but should enable the reader to comprehend simple Agda programs and to understand some of the challenges and

advantages of a dependently typed programming language. Some parts of the language are not explained at all, as they are beyond the scope of this thesis. These parts are mostly in the module system and in the syntactic sugar intended to make working with Agda more convenient.

The GRIN back-end for this thesis does not work directly on Agda code, however: The front-end first parses and type checks it, after which the back-end is passed a set of data structures called *Agda internal syntax*. The second part of this chapter introduces the data types used for this internal syntax.

Chapter 3 introduces the Graph Reduction Intermediate Notation, or GRIN. GRIN is immediately introduced as a set of data types and not as a language with a concrete syntax, because neither the back-end nor the programmer needs to work with GRIN in concrete syntax form.

Chapter 4 is the first chapter dealing with the newly built back-end itself. It goes into detail on all aspects of the implementation of the translation from Agda's internal syntax to GRIN.

Chapter 5 is about optimization. This chapter explains how a number of optimizations designed for a close cousin of Agda can be adapted and generalized for the combination of Agda and GRIN.

Finally, the thesis ends with a discussion of related work, contributions and future work in chapter 6.

2 Introduction to Agda

In this section we will introduce Agda[15], one of the two main technologies this thesis is based on. Both Agda the language and some of the Agda compiler internals will be explained here. After this section the reader should be able to understand the example Agda programs and the intermediate language that Agda compiles into.

2.1 Agda – the language

Agda is a dependently typed programming language based on Martin-Löf's type theory.[14] Martin-Löf's type theory is essentially a typed λ-calculus, although certainly not a *simply* typed λ-calculus.[2] Several things are remarkable about Agda:

- Agda is a total language. That is, all functions in Agda are total, i.e. defined for all inputs. By implication, Agda is not a Turing complete language.
- Agda is a pure language, which means that functions in Agda have no side-effects: The only result of calling a function is its return value, and calling a function twice with the same arguments yields the same result.

Taken together, this means that functions in Agda are mathematical functions from sets to sets. In addition, the language is extended with data types and lots of syntactic sugar to make the language more pleasant to actually program in.

As this thesis is about an Agda back end, however, we will not discuss much of the syntactic sugar, nor will we spend much time on the intricacies of the type system. We will especially not give an introduction to dependently typed programming. Instead, this chapter will focus on the syntactic structure of Agda programs.

Before continuing to explain the high level structure of an Agda program, we will first explain a bit about the syntax of expressions in Agda.

2.1.1 Expressions

Expressions can be found almost everywhere in an Agda program, and will be used in most of the examples in the coming sections. Therefore, they will be explained at this point, before going on to types. Because expressions are so common in Agda, it is convenient that their syntax is rather lightweight: An expression consists of either just a variable or of an expression applied to another expression. Application is written by juxtaposition, which means that application of e_1 to e_2 is written as $e_1\ e_2$.

As in many other functional languages, functions in Agda are automatically curried so, for example, the function *plus* on natural numbers is really a function of one natural number that, when applied, yields a function of a natural number to a natural number. Application is also left-associative. That is, $e_1\ e_2\ e_3$ means $(e_1\ e_2)\ e_3$, which is a natural fit with curried functions.

Identifiers may consist of almost any sequence of unicode characters, including symbols that in most programming languages are reserved for operators.

Identifiers may also contain underscores (_), in which case they can be used as (infix or even mixfix) operators. This way, after defining a function $\_+\_$, one can write $suc\ (m + n)$, which means the same as $suc\ ((\_+\_\ m)\ n)$.

Finally, functions may have both explicit and implicit parameters. Implicit arguments are intended to be inferred and inserted by the compiler. In cases where the compiler is not able to infer an implicit argument, it can be made explicit by enclosing the argument in $\{...\}$. An explicit argument may also be given as _, in which case the compiler will try to infer it as if it were an implicit argument. For example, if f is a function of two implicit and two explicit parameters (in that order) then $f\ \{x\}\ y\ \_$ passes x for the first implicit parameter, leaves the second implicit parameter for the compiler to infer, passes y for the first explicit parameter and leaves the second explicit parameter for the compiler to infer.

2.1.2 The structure of an Agda program

An Agda program consists of one or more modules, each of which may in turn contain other modules. Furthermore, each module consists of a number of declarations. The two most important kinds of declarations are data type declarations and function (or variable) declarations. In addition, a program may also contain declarations for records, postulates and primitives. These will all be explained in the next few sections.

To give the reader a first taste of what an Agda program looks like, a simple but complete example module follows now. The declarations in this example need not be completely understood yet as they will be explained in detail in the following sections.

```
module RunningExample where

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

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

```
data Vec (α : Set) : ℕ → Set where
  []    : Vec α zero
  _::_ : {n : ℕ} → α → Vec α n → Vec α (suc n)
_++_          : {α : Set} {m n : ℕ} → Vec α m → Vec α n → Vec α (m + n)
[] ++ ys      = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

This module, named *RunningExample*, first declares a data type representing Peano numbers and a corresponding addition function, then a data type of vectors and a function for appending two such vectors, and finally declares a variable containing a vector of two numbers.

2.1.3 Types

Agda is a statically typed programming language. Like in most such languages, one may have types like *Bool* and $\mathbb{N}$ in Agda. Types may also be parameterized by other types, so there may be a type *List Bool*, or *List* $\mathbb{N}$. These types are the application of *List* to *Bool* and $\mathbb{N}$, respectively: The syntax of types is the same as the syntax of expressions.

A major difference between Agda and most other statically typed languages (including simply typed λ-calculus) however, is that in Agda terms and types do not live in separate worlds: Types are also terms. This leads to a number of important consequences.

- Terms have types, so if types are also terms, then should types not also have types? They do. For example, the types mentioned before all have type *Set*, which can be written as *Bool* : *Set*, $\mathbb{N}$: *Set* etcetera. And we also have *Set* : Set_1, and more in general, Set_n : Set_{n+1}.
- If a type is also a term, then it should be possible to have functions accepting types as arguments, or returning types. In fact, we have already shown such a function: *List* : *Set* → *Set*, so *List Bool* is not only the type of lists of booleans, but also the application of the *List* function to the *Bool* type.
- Types can be parameterized by other types, but in fact, they can be parameterized by arbitrary terms. In Agda, one may have a type *Vec Bool 2* : *Set*, the type of vectors of booleans of length 2. Of course, *Vec* : *Set* → $\mathbb{N}$ → *Set*.

Agda is also a *dependently* typed language: Types may depend not only on constant values, but on arbitrary terms, including the values of other function parameters.

For example, although it is possible to define functions $\mathbb{N}$*Append* : *List* $\mathbb{N}$ → *List* $\mathbb{N}$ → *List* $\mathbb{N}$, *boolAppend* : *List Bool* → *List Bool* → *List Bool* and so on, this is actually not necessary in Agda, as it is possible to define one function that appends lists of arbitrary type: *append* : (α : *Set*) → *List* α → *List* α → *List* α, which means that append takes one parameter of type *Set* called α, and then two parameters whose type is *List* applied to this α, finally yielding a value of again *List* α. In other words, in a dependently typed language, parametric polymorphism need not be implemented as a separate type system feature: It has become a special case of dependent types.

Things are more complicated for the type of the append function for the vector datatype mentioned above. If we have xs : $Vec\ Bool\ 2$ and ys : $Vec\ Bool\ 3$, then $vecAppend\ xs\ ys$: $Vec\ Bool\ 5$, but what type should $vecAppend$ itself have? The answer is $vecAppend$: $(\alpha$: $Set)\ (m\ n$: $\mathbb{N}) \rightarrow Vec\ \alpha\ m \rightarrow Vec\ \alpha\ n \rightarrow Vec\ \alpha\ (m + n)$, so $vecAppend$ first takes the type of the vector elements and two natural numbers as parameters[1] , followed by two vectors of the lengths corresponding to the two natural number parameters, and then returns a vector whose length is equal to the sum of the first two vectors. This assumes, of course, that there is also a plus function of type $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$.

Finally, when types are terms, and types may contain other expressions, the question arises when two types are considered equal by the compiler. The answer is that in order to be considered equal, the normal forms of the types must be equal. This has already been used by the previous example, where we saw that $vecAppend\ xs\ ys$ has type $Vec\ Bool\ 5$, which is the normal form of $Vec\ Bool\ (2 + 3)$, the type deduced from $vecAppend$s type signature.

2.1.4 Data declarations

In Agda, one can declare new datatypes. Each data declaration introduces a new type to the program[2]. It also declares zero or more *data constructors* which are used to build values of said datatype.

A data declaration first specifies the name and type of the new datatype, and then states for each of the constructors what its name and type is.

More formally, a data declaration looks like

$$
\begin{array}{l}
\textbf{data}\ D\ \Delta\ :\ \Gamma\ \rightarrow\ Set_i\ \textbf{where} \\
\quad c_1\ :\ \Theta_1\ \rightarrow\ D\ \Delta\ t_1 \\
\quad \vdots \\
\quad c_n\ :\ \Theta_n\ \rightarrow\ D\ \Delta\ t_n
\end{array}
$$

The first line declares D to be a new datatype with parameter telescope Δ and index telescope Γ, and specifies what universe level D is a member of. Then zero or more constructors are defined. Each constructor has a name, c_i, a telescope of constructor arguments, Θ_i and, when applied, yields a value parameterized by Δ and t_i.

Telescopes are sequences of types of arguments, where the types of later arguments may depend on earlier types. Each argument may be either explicit or implicit. The formal syntax of telescopes is this:

$$
\begin{array}{lcll}
\Gamma, \Delta, \Theta & ::= & \epsilon & \\
 & | & (name : type)\Gamma & \text{telescope} \\
 & | & \{name : type\}\Gamma & \text{telescope (implicit)}
\end{array}
$$

Data declarations will now be further explained using two examples. The first example defines a datatype for Peano natural numbers:

$$
\begin{array}{l}
\textbf{data}\ \mathbb{N}\ :\ Set\ \textbf{where} \\
\quad zero\ :\ \mathbb{N} \\
\quad suc\ :\ \mathbb{N}\ \rightarrow\ \mathbb{N}
\end{array}
$$

[1] Note that $(m : \mathbb{N}) \rightarrow (n : \mathbb{N})$ has been abbreviated to $(m\ n : \mathbb{N})$.

[2] Or in many cases actually a whole family of new types

This should be read as follows: The type $\mathbb{N}$ is a simple datatype without parameters or indices that lives in *Set*.[3] It has two constructors, one of which is a simple value: *zero*. The other constructor takes an element of $\mathbb{N}$ as its argument and constructs its successor, which is of course again an element of $\mathbb{N}$. In practice, this declaration introduces three new identifiers ($\mathbb{N}$, *zero* and *suc*) into the current scope. All can be used in other (type) declarations and expressions, and the constructors may also be used when pattern matching. An example of such an expression would be *suc* (*suc zero*), which is of type $\mathbb{N}$ and represents the number 2.

As we have just seen, Agda datatypes can be used for what in other languages are called *tagged unions and records*, or in yet other languages *sums of products*. In fact, however, Agda's data declarations are more powerful, as the following example demonstrates.

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

This declaration defines the vector type parameterized both by its element type and by its length.

A number of things should be clarified about this definition:

- An identifier in Agda may contain underscores (_). Such an identifier can be used as an operator by filling in arguments for the underscores.
- The Δ telescope declares the *type parameters* of the datatype, which must occur in the same way in all constructor types. In this case, there is only one type parameter, α : *Set*, which is the element type parameter of the defined vector type. Each constructor gets all type parameters as implicit parameters, so the full type of _::_ is $\{\alpha : \mathit{Set}\}\ \{n : \mathbb{N}\} \to \alpha \to \mathit{Vec}\ \alpha\ n \to \mathit{Vec}\ \alpha\ (\mathit{suc}\ n)$
- The Γ telescope declares the *type indices* of the datatype. Where the type parameters scope over the entire data declaration, the indices are local and differ between constructors. *Vec* has only one type index, which is of type $\mathbb{N}$ and represents the length of the vector.
- [] is a perfectly normal identifier in Agda. In this case it is the name used for the empty vector, which contains *zero* elements.
- _::_ (pronounced "cons") is a constructor that appends an element to the front of a vector, taking the new element and a vector of length n to a vector of length *suc* n. The first argument ($\{n : \mathbb{N}\}$, the length of the vector) is an implicit parameter, which means that it will not be written down by the programmer but instead inferred by the compiler.

An example vector expression is *false* :: (*true* :: []) which has type *Vec Bool 2* (assuming a suitable Bool datatype). This example can also be written without using either infix operators or implicit arguments, in which case it is the rather unreadable _::_ {ℕ} { *1* } *false* (_::_ {ℕ} { *0* } *true* ([] {ℕ})).

[3] Set_0 is usually abbreviated as just *Set*.

The final example declares a data type for representing proofs of propositional equality:

$$\begin{array}{l}\textbf{data}\ \_{\equiv}\_\ \{\alpha\ :\ \mathit{Set}\}\ (x\ :\ \alpha)\ :\ \alpha\ \to\ \mathit{Set}\ \textbf{where}\\ \quad \mathit{refl}\ :\ x\ \equiv\ x\end{array}$$

The $\_\equiv\_$ type has two type parameters (an implicit parameter $\alpha\ :\ \mathit{Set}$ and an explicit $x\ :\ \alpha$) and one type index (α). Its only constructor, *refl*, has no explicit arguments at all. It does, however, get both type parameters of $\_\equiv\_$ as implicit parameters, so we actually have $\mathit{refl}\ :\ \{\alpha\ :\ \mathit{Set}\}\ \{x\ :\ \alpha\}\ \to\ x\ \equiv\ x$. Equivalently, this can also be written without using implicit parameters and infix operators as $\mathit{refl}\ :\ \{\alpha\ :\ \mathit{Set}\}\ \{x\ :\ \alpha\}\ \to\ \_\equiv\_\ \{\alpha\}\ x\ x$.

The natural number example was rather obvious, but what is $\_\equiv\_$ good for? This is best explained by an example: If p is a value of type $n\ \equiv\ \mathit{suc}\ k$ then p is a *proof* that $n\ \equiv\ \mathit{suc}\ k$.

2.1.5 Function definitions

A function definition has the following form:

$$\begin{array}{l}f\ :\ \Delta\ \to\ t\\ f\ p_{1,1}\ \ldots\ p_{1,m}\ =\ e_1\\ \vdots\\ f\ p_{n,1}\ \ldots\ p_{n,m}\ =\ e_n\end{array}$$

A function definition first declares the name and type of the function in the *type signature*. Then all the function clauses are defined. The type signature is comparable to the first line of a datatype declaration: The function name is stated, followed by its type. A function type consists of a telescope for the function parameters, followed by the result type of the function.

What follows are the *function clauses*. The left hand side of a function clause consists of the function name followed by a *pattern* for each of the parameters in the telescope Δ. A pattern is either a variable or a constructor followed by one pattern for each of its (zero or more) fields. The right hand side of the function clause is an arbitrary term, and may use the variables bound in the left hand side.

When a function is called, its arguments are *pattern matched* against the patterns of each function clause from top to bottom. The first clause whose patterns match the arguments is executed. Functions in Agda are total, so the type checker will ensure that every possible combination of arguments that can be passed to the function matches at least one of the function clauses.

Again, we will clarify by giving a few examples. The first uses the first data declaration of the data type subsection, and defines a plus function over natural numbers:

$$\begin{array}{lcl}\_+\_ & : & \mathbb{N}\ \to\ \mathbb{N}\ \to\ \mathbb{N}\\ \mathit{zero}\ +\ n & = & n\\ \mathit{suc}\ m\ +\ n & = & \mathit{suc}\ (m\ +\ n)\end{array}$$

First, $\_+\_$ is declared to be a function taking two natural numbers to another natural number.[4]

Just like with data type definitions, functions with names containing underscores ($\_$) can be called as operators, so $\_+\_$ can also be used as a binary operator. The first clause, $zero + n = n$, means that if the first argument to $\_+\_$ matches $zero$ and the second argument matches n, the result will be n. The compiler can distinguish between constructor patterns like $zero$ and variable patterns like n using its knowledge about the parameter types.

If the first clause does not match, that is if the first argument is not $zero$, then the first argument must be of the form $suc\ m$, where m is some arbitrary element of $\mathbb{N}$, and the second clause will be selected.

A more complicated example is the *vecAppend* function from the section about types, although it has a slightly different name and type here.

$$
\begin{array}{l}
\_{+\!\!+}\_ \quad : \{\alpha : Set\}\,\{m\ n : \mathbb{N}\} \to Vec\ \alpha\ m \to Vec\ \alpha\ n \to Vec\ \alpha\ (m + n) \\
[\,] \quad {+\!\!+}\ ys = ys \\
(x :: xs)\ {+\!\!+}\ ys = x :: (xs\ {+\!\!+}\ ys)
\end{array}
$$

$\_{+\!\!+}\_$ is a function of five parameters, of which the first three are implicit parameters: In most circumstances the programmer need not specify those parameters, as the type checker will be able to infer them. The rest of its type is as explained before: It takes a vector of length m and one of length n and returns a vector of length $m + n$. Its definition is very similar to the definition of $\_+\_$: Appending two vectors is defined by structural recursion on the first vector argument. Note that the implicit parameters are not specified at all in the definition itself.

The final function definition example uses the propositional equality datatype again.

$$
\begin{array}{l}
\equiv\!-trans \quad : \{\alpha : Set\}\,\{x\ y\ z : \alpha\} \to x \equiv y \to y \equiv z \to x \equiv z \\
\equiv\!-trans\ refl\ refl = refl
\end{array}
$$

$\equiv\!-trans$ is a function of six parameters, of which only the last two are non-trivial. The other arguments are all implicit as their value can always be inferred from the last two arguments. Although the type signature is rather complicated, the function has only one rather simple clause: $\_\equiv\_$ has only one constructor, which has no arguments. Therefore, this one clause covers all the possible combinations of arguments that can be passed to $\equiv\!-trans$: The real work here, is of course done by the Agda type checker when it checks that this simple definition actually has the stated type. An example of its use would be that assuming

$$
\begin{array}{l}
n : \mathbb{N} \\
p : suc\ (n + n) \equiv suc\ n + n \\
q : suc\ n + n \equiv n + suc\ n
\end{array}
$$

we have that

$$
\equiv\!-trans\ p\ q : suc\ (n + n) \equiv n + suc\ n
$$

[4] Strictly speaking, it is defined to be a function taking *one* natural number to another function taking *one* natural number to a natural number: Functions in Agda are curried.

2.1.6 Records

A record declaration in Agda is used to define *record types* (known as "structs" in e.g. C). In Agda, a record declaration is purely syntactic sugar for a datatype declaration with exactly one constructor and a new module containing functions that can be used to extract the values a record.

```
record List (α : Set) : Set where
  field
    length : ℕ
    vector : Vec α length
```

This example defines a new parameterized record type *List* with two fields. The first field, *length*, is a simple natural number, but the other field, *vector* has a type dependent on the value of the first field: Records in Agda can actually be *dependent* record types.

```
xs : List ℕ
xs = record {length = suc (suc zero); vector = zero :: suc zero :: []}
```

Then, *xs* is defined as a value of this new dependent record type, where the compiler is left to infer the value of the first value for us. As can be seen, records are not constructed using an explicit constructor function but using special record syntactic sugar.

```
n : ℕ
n = List.length xs
```

Finally, the *length* function from the automatically generated *List* module is used to retrieve the length of the *Vec* inside the *List*.

2.1.7 Postulates and primitives

The last two forms of declarations are postulates and primitives. Syntactically, the two are very similar:

```
postulate
  Int : Set
primitive
  primIntegerPlus : Int → Int → Int
```

Both declarations begin by specifying what kind of declaration they are. Then one or more type signatures follow. The meaning of both declarations differs quite a bit, however:

A postulate or axiom tells the type checker to assume the existence of the given identifiers with the specified types, but those new identifiers do not have any computational meaning. They cannot be further evaluated by either the interactive Agda prompt or the compiler while type checking, and trying to compile a program with postulates will usually result in the compiler failing with an error message.

In contrast, although a primitive also brings an identifier with the given type in scope, it definitely does have computational content here. A primitive

declaration is a request to the compiler to add code for this function, usually because the function could not (efficiently) be implemented with Agda code. There is a fixed list of primitive functions that can be added and used in this way.

2.2 Agda – the internal syntax

So far, we have described the Agda surface syntax. That is, we have looked at the *input* to the Agda compiler. As this thesis is actually about connecting the Agda front end to the EHC back end, we will now describe what the *output* of the Agda front end looks like. This output is what we call *Agda internal syntax*. Internal syntax is one of the three major kinds of syntax trees used in the Agda compiler, the other two being *concrete syntax* and *abstract syntax*.

Agda's concrete syntax is a set of datatypes meant to exactly represent what the compiler found in the files it parsed. All syntactic sugar is still present, as is all information about comments and line and column numbers. In Agda's abstract syntax, part of the syntactic sugar has been removed and scope analysis has been performed. Some of the range information has also been removed and type information is only available in so far as the programmer has provided type signatures. Agda's type checker works on the abstract syntax and transforms it into internal syntax. In internal syntax, all types are explicit. It is this internal syntax that we will now further discuss, as this is what the Agda back ends use: The interpreter, the Haskell back end and the EHC back end. The internal syntax will be presented as a set of Haskell datatypes, in a slightly simplified form from the actual datatypes used in Agda.

An Agda program in Internal Syntax is essentially a list of definitions.

```
type InternalSyntaxAgdaProgram = [Definition]
```

This list contains all definitions of all compiled Agda modules together, so all the identifiers in it must be fully qualified to prevent namespace clashes, that is, in the internal syntax an identifier *bar* from the module *Foo* will always be represented as `"Foo.bar"`. All definitions have a qualified name and a type, but the other fields are different depending on what kind of declaration it was.

```
data Definition = Defn QName Type Defn
data Defn
  = Datatype ℕ ℕ [QName]
  | Constructor ℕ QName
  | Function [Clause]
  | Record [QName]
  | Axiom
  | Primitive String
```

We will again use examples from our *RunningExample* module to explain what the different kinds of Agda declarations are compiled to.

2.2.1 Datatypes and constructors

Again, we begin with datatype declarations. In Agda's internal syntax, datatypes and their constructors have been separated. The list of the names of its constructors is (almost) all that is left of a datatype itself.

```
data Defn
  = Datatype ℕ ℕ [QName]
  | Constructor ℕ QName
```

The *Datatype* constructor has three fields: The number of type parameters of the datatype, the number of type indices and the list of constructor names. The *Constructor* constructor has fields for its number of parameters and for the name of the datatype it is a constructor of.

There are two more things that need to be explained before we can go on to the first example, and they are the internal syntaxes for types and terms.

```
data Type = El Sort Term
data Sort = Type ℕ
```

Recall that the internal syntax is the *output* of the type checker, so all types and parameters are known, whether explicit or implicit. Together with the fact that types are just terms, this means that a type in the internal syntax is built from a term (recall that in Agda, types are terms, too) and the *sort* of which the type is an element. A sort is just the type of a type. A sort itself stores only its universe level, that is, what the n in Set_n is. The definition of *Term*, the internal syntax representation of an Agda term, will be explained now.

```
data Term
  = Var ℕ [Term]
```

A *Term* may be a variable, represented by a de Bruijn index. Variables may be applied to other *Term*s.

```
  | Def QName [Term]
```

A *Term* may be a global identifier, again applied to zero or more other *Term*s.

```
  | Con QName [Term]
```

A *Term* may also be a constructor applied to zero or more other *Term*s.

```
  | Lam Term
```

A *Term* may be a λ, containing the body of the function. This is the first *Term* that binds de Bruijn indices that can then be referenced by *Var* constructors. For example, the term $\backslash x \rightarrow x$ would be encoded as *Lam* (*Var 0*).

```
  | Pi Type Type
```

The other constructor introducing new de Bruijn indices is *Pi*, representing the dependent function types. The arguments to *Pi* are the domain and the codomain of the function type. A de Bruijn index is then bound referring to the value of the domain type that is actually passed. For example, the term $(\alpha : Set) \rightarrow List\ \alpha$ could be encoded as
Pi (*El* $Type_1$ $Type_0$) (*El* $Type_0$ (*Def* `"Prelude.List"` [*Var 0* []]))

```
  | Fun Type Type
```

Fun is a term for representing the non-dependent function types. Its use is almost the same as *Pi*, except that no new de Bruijn indices are introduced. Note that both *Pi* and *Fun* represent *types*, which in Agda are also *Term*s.

```
  | Sort Sort
```

The other kinds of types that a *Term* can represent, besides (dependent) function types, are sorts: Set_0, Set_1 etc. These are represented by the *Sort* constructor.

```
  | Lit Literal
```

Finally, a *Term* in Agda can be a literal. Although it not been used so far in the examples, Agda supports integer, double, string and character literals. Their internal syntax representation uses this auxiliary definition.

```
data Literal
  = LitInt Integer
  | LitFloat Double
  | LitString String
  | LitChar Char
```

We will now give an example datatype definition in Agda internal syntax. Recall the definition of our vector datatype:

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

This is then compiled into three different *Definition*s, one for the type constructor and one for each data constructor. The first is a *Datatype* and tells us about the *Vec* type constructor:

```
let arg₁ = El Type₁ (Sort Type₀)
    arg₂ = El Type₀ (Def "RunningExample.ℕ" [])
    res  = El Type₁ (Sort Type₀)
in  Defn "RunningExample.Vec"
      (El Type₁ (Pi arg₁ (El Type₁ (Fun arg₂ res))))
      (Datatype 1 1 ["RunningExample.[]","RunningExample._::_"])
```

First, it should be noted that in the name of readability, a few liberties have been taken with the exact syntax: (*Type* n) has been written as $Type_n$ to save a bit on the number of parentheses. Also, we use Haskell's **let–in** syntax to divide the *Definition* into more intelligible parts.

That said, we will now explain the actual *Definition* for *Vec*.

- The first field is the fully qualified name of the thing defined, which is *RunningExample.Vec* in this case.
- The second field of the *Defn* constructor is the *Type* of *Vec*. Again, the internal syntax has already been type checked, and on almost every level of the internal syntax terms are annotated with their type. In this case, the *Type* field represents the type $(\alpha : Set) \to \mathbb{N} \to Set$, which is

indeed the type of the type constructor *Vec*. The outermost $Type_1$ tells us that the type itself has type Set_1.

Inside, we find a *Pi* constructor, which is the constructor specifying a dependent function type. *Pi* itself has two fields: The domain and the codomain of the function type. The domain type is here an element of Set_0 (represented by $Type_0$) which is itself an element of Set_1 (represented by $Type_1$).

The codomain of the dependent function type is again a function type, although this time non-dependent one, which is why a *Fun* instead of a *Pi* constructor is used. The domain of this function type ($\texttt{arg}_2$) is $\mathbb{N} : Set_0$. *Def identifier arguments* is the representation of an identifier applied to a number of arguments; In this case, the identifier `"RunningExample.ℕ"` applied to no arguments at all.

This is also the first indication of another fact about Agda's internal syntax: There is no constructor for application. Instead, every constructor representing something that might be applied to something has a list of arguments (often empty) representing the things that it is applied to.

Finally, the codomain of the innermost *Fun* is the same as the first argument: $Set_0 : Set_1$, represented by $El\ Type_1\ (Sort\ Type_0)$.

- The last field of this *Definition*, of type *Defn*, is a *Datatype* constructor: *Vec* is a datatype (constructor). As *Vec* has one type parameter ($\alpha : Set$) and one type index ($\mathbb{N}$), the corresponding fields both have 1 as their value. The third field is the list of the names of its constructors, of which *Vec* has two.

The second *Definition* is a *Constructor* and tells us about the data constructor [].

```
let arg₁ = El Type₁ (Sort Type₀)
    res  = El Type₀ (Def "RunningExample.Vec"
                         [Var 0 [], Con "RunningExample.zero" []])
in  Defn "RunningExample.[]"
         (El Type₁ (Pi arg₁ res))
         (Constructor 1 "RunningExample.Vec")
```

Again, we will describe the fields of the *Defn* constructor, although the first field has nothing new and will therefore not be explained again

- The second field is the internal syntax representation of the type of the [] constructor, which happens to be $\{\alpha : Set\} \rightarrow Vec\ \alpha\ zero$. Again, the type of this type is Set_1, represented by $Type_1$. The type itself is once more a dependent function type, and thus represented by a *Pi* constructor. Its domain ($\texttt{arg}_1$) is again a $Set_0 : Set_1$. [] is a constructor of the *Vec* datatype, so the codomain must ultimately be *Vec* applied to something, which is itself an element of Set_0.

  The arguments of *Vec* are what is new here: The first *Var 0* [], is a local variable applied to no arguments. The 0 is the variable name, or rather, the de Bruijn index of the variable. In this case, *Var 0* [] thus stands for α, the only argument of []. This also explains why both *Pi* and *Fun* have

two *Type* fields: Because of the use of de Bruijn indices as variables, *Pi* can bind an argument without having to explicitly name it.

The second argument is *Con* `"RunningExample.zero"` [] and represents the data constructor *zero*. Note that just like *Def* and *Var*, the *Con* constructor has a list of arguments, too.

- The third field of the *Defn* is a *Constructor* constructor. It tells us that the [] constructor has 1 parameter, and that the corresponding datatype is *Vec*.

The third *Definition* that compiling the declaration of the *Vec* datatype results in represents the $\_::\_$ constructor.

$$
\begin{array}{l}
\textbf{let}\ \mathtt{arg}_1 \ = \ \mathit{El}\ \mathit{Type}_1\ (\mathit{Sort}\ \mathit{Type}_0) \\
\quad \mathtt{arg}_2 \ = \ \mathit{El}\ \mathit{Type}_0\ (\mathit{Def}\ \texttt{"RunningExample.}\mathbb{N}\texttt{"}\ [\,]) \\
\quad \mathtt{arg}_3 \ = \ \mathit{El}\ \mathit{Type}_0\ (\mathit{Var}\ 1\ [\,]) \\
\quad \mathtt{arg}_4 \ = \ \mathit{El}\ \mathit{Type}_0\ (\mathit{Def}\ \texttt{"RunningExample.Vec"}\ [\mathit{Var}\ 1\ [\,], \mathit{Var}\ 0\,]) \\
\quad \mathtt{res} \ \ = \ \mathit{El}\ \mathit{Type}_0\ (\mathit{Def}\ \texttt{"RunningExample.Vec"} \\
\qquad\qquad [\mathit{Var}\ 1\ [\,], \mathit{Con}\ \texttt{"RunningExample.suc"}\ [\mathit{Var}\ 0\ [\,]]]) \\
\textbf{in}\ \mathit{Defn}\ \texttt{"RunningExample.\_::\_"} \\
\quad (\mathit{El}\ \mathit{Type}_1\ (\mathit{Pi}\ \mathtt{arg}_1\ (\mathit{El}\ \mathit{Type}_0\ (\mathit{Pi}\ \mathtt{arg}_2 \\
\qquad (\mathit{El}\ \mathit{Type}_0\ (\mathit{Fun}\ \mathtt{arg}_3\ (\mathit{El}\ \mathit{Type}_0\ (\mathit{Fun}\ \mathtt{arg}_4\ \mathtt{res}))))))))) \\
\quad (\mathit{Constructor}\ 1\ \texttt{"RunningExample.Vec"})
\end{array}
$$

- The second field describes the type of $\_::\_$, which is

$$\{\alpha\ :\ \mathit{Set}\}\ \rightarrow\ \{n\ :\ \mathbb{N}\}\ \rightarrow\ \alpha\ \rightarrow\ \mathit{Vec}\ \alpha\ n\ \rightarrow\ \mathit{Vec}\ \alpha\ (\mathit{suc}\ n)$$

Thus, $\_::\_$ is a constructor of 4 parameters (although its type is curried). Except for the first parameter, which has a type of sort Set_1, all parameters have types of sort Set_0. The first two parameters are dependent, i.e. their function type is a dependent function space. Therefore, they are represented by a *Pi* constructor, whereas the other two parameters use a *Fun* constructor. Then there are the types of the arguments themselves

$\mathtt{arg}_1$ The first argument to $\_::\_$ is of type Set_0.

$\mathtt{arg}_2$ The second must be a natural number: Its type is $\mathbb{N}$.

$\mathtt{arg}_3$ The type of the third argument is the value of the most-recently-but-one bound argument, that is, the value of the first argument.

$\mathtt{arg}_4$ The fourth argument must be of type $\mathit{Vec}\ \alpha\ n$, which is represented by applying the identifier *Def* `"RunningExample.Vec"` to the most-recently-but-one (*Var 1* []) and the most recently (*Var 0* []) bound arguments.

The resulting value is then also a vector: n, being bound by the most-recent (syntactically speaking) dependent function type, is represented by *Var 0* []. Then *Con* `"RunningExample.suc"` [*Var 0* []] is the representation of the application of the constructor *suc* to this n. Also, α, being bound by the most-recently-but-one function type, is represented by *Var 1* []. Finally, *Def* `"RunningExample.Vec"` is applied to both the representation of α and of *suc* n.

- The third field is of course again a *Constructor* constructor. It has one point of interest: It says "1", even though the Agda _::_ data constructor has four arguments. The reason for this is that this "1" represents the number of *type parameters*, of which _::_ has only 1; Its other parameters are either *type indices* (in the case of $\{n : \mathbb{N}\}$) or regular *constructor field parameters* (in the cases of α and $Vec\ \alpha\ n$).

2.2.2 Function definitions

An internal syntax *Definition* representing an Agda function will contain a *Defn* with the *Function* constructor.

data *Defn*
  ...
  | *Function* [*Clause*]

For example, take the definition of the vector append function (_++_)

$$\begin{array}{lll} \_{+\!\!+}\_ & : & \{\alpha : Set\}\,\{m\ n : \mathbb{N}\} \to Vec\ \alpha\ m \to Vec\ \alpha\ n \to Vec\ \alpha\ (m + n) \\ [\,] +\!\!+ ys & = & ys \\ (x :: xs) +\!\!+ ys & = & x :: (xs +\!\!+ ys) \end{array}$$

This will then be compiled into the following *Definition*:

Defn "RunningExample._++_"
  type
  (*Function* [clause$_1$, clause$_2$])

The type field in this *Definition* will be shown for completeness. However, as it contains no new features of the internal syntax representation, it will not be further explained.

$$\begin{array}{lcl} \texttt{arg}_1 & = & El\ Type_1\ (Sort\ Type_0) \\ \texttt{arg}_2 & = & El\ Type_0\ (Def\ \texttt{"RunningExample.}\mathbb{N}\texttt{"}\ [\,]) \\ \texttt{arg}_3 & = & El\ Type_0\ (Def\ \texttt{"RunningExample.}\mathbb{N}\texttt{"}\ [\,]) \\ \texttt{arg}_4 & = & El\ Type_0\ (Def\ RunningExample.Vec\ [\,Var\ 2\ [\,],\ Var\ 1\ [\,]]) \\ \texttt{arg}_5 & = & El\ Type_0\ (Def\ \texttt{"RunningExample.Vec"}\ [\,Var\ 2\ [\,],\ Var\ 0\ [\,]]) \\ \texttt{res} & = & El\ Type_0\ (Def\ \texttt{"RunningExample.Vec"}\ [\,Var\ 2\ [\,],\ Def\ \texttt{"RunningExample._+_"}\ [\,Var\ 1\ [\,],\ Var \\ \texttt{type} & = & El\ Type_1\ (Pi\ \texttt{arg}_1\ (El\ Type_0\ (Pi\ \texttt{arg}_2\ (El\ Type_0\ (Pi\ \texttt{arg}_3 \\ & & \quad (El\ Type_0\ (Fun\ \texttt{arg}_4\ (El\ Type_0\ (Fun\ \texttt{arg}_5\ \texttt{res}))))))))) \end{array}$$

The representation of an Agda function clause is as follows

$$\begin{array}{lcl} \textbf{data}\ Clause & = & Clause\ [Pattern]\ ClauseBody \\ \textbf{data}\ Pattern & = & VarP\ String \\ & | & ConP\ QName\ [Pattern] \\ & | & DotP\ Term \\ & | & LitP\ Literal \\ \textbf{data}\ ClauseBody & = & Body\ Term \\ & | & Bind\ ClauseBody \\ & | & NoBind\ ClauseBody \\ & | & NoBody \end{array}$$

A *Clause* consists of a list of *Pattern*s, representing the pattern constructors and variables on the left hand side of the clause, and of a *ClauseBody* representing the right hand side of the clause. A *Pattern* can be a pattern variable with a certain variable name, or it can be a constructor pattern with a certain qualified name and a list of *Pattern*s. It can also be a *DotP* or a *LitP*, which will be explained when necessary. A *ClauseBody* describes which of the pattern variables of the *Pattern* list actually bind de Bruijn indices and which do not, and finally describes the actual expression on the right hand side of the clause if there is one.

The *NoBody* constructor requires some further explanation, as it is only used for an Agda feature which has not yet been explained, namely, functions without a right hand side. Suppose a datatype *False* is defined having no constructors at all:

```
data False : Set where
```

Under the Curry-Howard isomorphism, this datatype then represents the false proposition, having no proof. Given this datatype we can then define the following function:

```
elim-False : {α : Set} → False → α
elim-False ()
```

This function then represents the *principle of explosion*, or *ex falso quodlibet*: Given a value of type *False*, we can produce a value of any arbitrary type. The () here tells the Agda compiler that the function pattern matches on an argument which cannot take any value at all, and thus it makes no sense to give a right hand side for the function either as it cannot possibly be called.

Before showing the actual internal syntax that the first clause of our example function _++_ is compiled into, it may help to show this clause without much of its syntactic sugar.

```
_++_ {α} {.zero} {n} [] ys = ys
```

There are two differences between this version and the original ([] ++ *ys* = *ys*): First, its implicit parameters have been made explicit by placing them between curly brackets ("{}"). Second, the second implicit parameter is a constructor pattern prefixed with a dot. This tells Agda it should not actually pattern match on this argument because its value can be inferred from the other patterns. In this case, because the vector is empty ([]), its length *must* be *zero*. The representation of this clause in internal syntax is this:

```
clause₁ = Clause
  [ VarP "α"
  , DotP (Con "RunningExample.zero" [])
  , VarP "n"
  , ConP "RunningExample.[]" []
  , VarP "ys"
  ]
  (Bind (Bind (Bind (Bind (Body (Var 0 [])))))
```

Note that in internal syntax, all parameters have become explicit parameters. Apart from that, the *Pattern* list should be self explanatory by now. However,

the *ClauseBody* deserves further explanation. It contains nested *Bind* constructors for each of the *VarP* constructors in the *Pattern*s, meaning that all variable patterns actually do bind de Bruijn indices. The body of this function is then *Var 0* []: The most recently bound variable, which is *VarP* `"ys"`. Note that as de Bruijn indices are used, the variable names (`"`α`"`, `"n"` and `"ys"`) exist mostly for debugging reasons.

We can now turn to the second clause. Before giving its internal syntax, we will again first give a mostly-desugared version of this clause.

$$\_{+\!+}\_\ \{\alpha\}\ \{.(suc\ m)\}\ \{n\}\ (\_::\_\ \{m\}\ x\ xs)\ ys\ =\ \_::\_\ x\ (\_{+\!+}\_\ xs\ ys)$$

Note in particular that the pattern variable m is bound by its occurrence as an implicit field in the $\_::\_$ constructor pattern. The value of the second (implicit) pattern is then deduced by the type checker to be $suc\ m$, which is why the second pattern is a dot-pattern.

We now show the internal syntax of the second clause of $\_{+\!+}\_$.

```
clause₂ = Clause
  [VarP "α"
  , DotP (Con "RunningExample.suc" [Var 3 []])
  , VarP "n"
  , ConP "RunningExample._::_" [VarP "n", VarP "x", VarP "xs"]
  , VarP "ys"
  ]
  (Bind (Bind (Bind (Bind (Bind (Bind (Bind (Body
    (Con "RunningExample._::_"
      [Def "RunningExample._+_" [Var 3 [], Var 4 []], Var 2 []
      , Def "RunningExample._++_" [Var 6 [], Var 3 [], Var 4 [], Var 1 [], Var 0 []]
      ]))))))))
```

As can be seen from the *Pattern* list, $\_{+\!+}\_$ has five parameters. The second parameter uses a *DotP* constructor, which means for our purposes means that it can be regarded as a variable: There will be no pattern matching on it, and it binds a single de Bruijn index. The fourth parameter demonstrates that variables can be bound from within constructor patterns. In fact, in the internal syntax, patterns can still be nested arbitrarily deep. The order in which patterns bind de Bruijn indices is textually from left to right: *ys* is referred to by *Var 0*, *xs* by *Var 1*, *x* by *Var 2* and so on.

2.2.3 Records

An internal syntax *Definition* representing an Agda record declaration will contain a *Defn* with the *Record* constructor.

```
data Defn
  ...
  | Record [QName]
```

Consider the declaration of our earlier *List* record:

```
record List (α : Set) : Set where
  field
```

$length\ :\ \mathbb{N}$
$vector\ :\ Vec\ \alpha\ length$

In internal syntax, this has been turned into the following *Record* definition

Defn "RunningExample.List"
  (*El* $Type_1$ (*Pi* (*El* $Type_1$ (*Sort* $Type_0$)) (*El* $Type_1$ (*Sort* $Type_0$))))
  (*Record* ["RunningExample.List.length", "RunningExample.List.vector"])

As can be seen from this definition, the type represents $Set_0 \rightarrow Set_0$, the type of the *List* type constructor. Note also that the fields of the record are qualified not only by the module *RunningExample*, but also by the newly introduced record/module *List*.

In addition to this *Record* definition, two *Function* definitions are also generated. The first of these definitions represents the selector function for the *length* field

Defn "RunningExample.List.length"
  type
  (*Function* [$\texttt{clause}_1$])

type is actually the following

let $\texttt{arg}_1$ = *El* $Type_1$ (*Sort* $Type_0$)
  $\texttt{arg}_2$ = *Def* "RunningExample.List" [*Var 0* []]
  res = *El* $Type_0$ (*Def* "RunningExample.ℕ" [])
in type = *El* $Type_1$ (*Pi* $\texttt{arg}_1$ (*El* $Type_0$ (*Fun* (*El* $Type_0$ $\texttt{arg}_2$) res)))

and represents the type of the *RunningExample.List.length* selector function: $\{\alpha\ :\ Set\} \rightarrow List\ \alpha \rightarrow \mathbb{N}$

$\texttt{clause}_1$ = *Clause* [*VarP* "α", *ConP* "RunningExample.List" [*VarP* "x", *VarP* "x"]]
  (*NoBind* (*Bind* (*NoBind* (*Body* (*Var 0* [])))))

The *Clause* is more interesting: First, it can be seen that in Agda's internal syntax, one can pattern match on records just like on types defined using a data declarations. The name of the record type is then used as the constructor name. It should also be noted then when, in internal syntax, a record is constructed in an expression, this is also done just as with normal datatypes. Here too, the record name is used as the constructor. Second, in this pattern match, both variables are called "x". Although this seems to lead to an ambiguity, it does not actually do so as the variable names are ignored in favour of using de Bruijn indices. The body of the clause shows that, from left to right in the text, the first and the third pattern variables are not to be bound to de Bruijn indices. For this reason, the second pattern variable can be referenced using *Var 0* [].

The internal syntax definition of *RunningExample.List.vector* is similar and will be omitted for brevity.

2.2.4 Axioms and primitives

data *Defn*
  ...

| *Axiom*
| *Primitive String*

What is called a "postulate" in Agda itself is called an "axiom" in the internal syntax. Once one knows an axiom's name and type, there is nothing more to say about the axiom, so this constructor has no fields at all. Primitives are only slightly more interesting from an internal syntax point of view. As the compiler has to generate code for a primitive, it should know the exact name of the primitive, without having to know in what module it was declared. The primitives name is what the *String* field is for.

3 Introduction to GRIN

GRIN, or Graph Reduction Intermediate Notation, is a language designed for compiler back ends for (lazy) functional languages. As its name suggests, it is based on graph reduction, the common evaluation model for non-strict languages. GRIN is essentially a very simple functional language with explicit evaluation order and support for expressing lazy evaluation.

Its most important features when compared to e.g. the more well known G-machine, are that many things have been shifted from being primitive notions to being expressible within the language. For example, closures are not a primitive concept in GRIN, but can instead be represented as normal GRIN values. In addition, GRIN exists in a number of "dialects", differing mostly in how low-level they are. For example, in the highest level there is an instruction implementing *call-by-need* evaluation, whereas in the lower GRIN levels this has been replaced by explicit *update* and *fetch* operations. Due to these properties, GRIN is very suitable for all kinds of program transformations and optimizations.

We will now give a small example of what GRIN code looks like. Recall our Agda vector append function ($\_{+\!\!+}\_$) which, without most of its syntactic sugar, looks like this

$$
\begin{array}{l}
\_{+\!\!+}\_ : (\alpha : \mathit{Set})\ (m\ n : \mathbb{N}) \rightarrow \mathit{Vec}\ \alpha\ m \rightarrow \mathit{Vec}\ \alpha\ n \rightarrow \mathit{Vec}\ \alpha\ (\_+\_\ m\ n) \\
\_{+\!\!+}\_\ \alpha\ .\mathit{zero}\ n\ []\ \mathit{ys} = \mathit{ys} \\
\_{+\!\!+}\_\ \alpha\ .(\mathit{suc}\ m)\ n\ (\_::\_\ m\ x\ \mathit{xs})\ \mathit{ys} = \_::\_\ (\_+\_\ m\ n)\ x\ (\_{+\!\!+}\_\ \alpha\ m\ n\ \mathit{xs}\ \mathit{ys})
\end{array}
$$

This will be compiled into the following GRIN function:

$$
\begin{array}{l}
\_{+\!\!+}\_\ \alpha\ m\ n\ \mathit{vec1}\ \mathit{vec2} \\
\quad = \mathit{eval}\ \mathit{vec1}; \backslash \mathit{vec1}' \rightarrow \\
\qquad \mathit{case}\ \mathit{vec1}'\ \mathit{of} \\
\qquad\quad (C\ []) \rightarrow \mathit{eval}\ \mathit{vec2} \\
\qquad\quad (C\_::\_\ m'\ x\ \mathit{xs}) \rightarrow \\
\qquad\qquad \mathit{store}\ (F\_+\_\ m'\ n); \backslash i \rightarrow
\end{array}
$$

```
store (F_++_ α m' n xs vec2); \zs →
unit (C_::_ i x zs)
```

The first thing to note is that no longer is there any kind of type signature. Indeed, GRIN is an untyped language. However, types in Agda are terms, and the Agda-to-GRIN translator treats all terms as equals, so types do still exist in the GRIN code as normal GRIN values. Second, it still looks quite a bit like a functional program, which was to be suspected, as GRIN is still a mostly functional (albeit very low level) language. However, as opposed to truly functional programs, although like monadic programs, the ordering of execution is explicit in this program: the $x; \backslash y \rightarrow z$ construct first executes x and then executes z, with y bound to the result of x. The other thing that is clearly non-functional about this example is the use of *eval* and *store*. These two constructs support the implementation of laziness in GRIN, and will be explained in the next sections.

This example is the last GRIN program that will be shown as such in this thesis; as GRIN is used as the back-end of our compiler, we are actually much more interested in viewing the GRIN language as a set of datatypes than in what its concrete syntax is. So it is the GRIN datatypes that the next sections will focus on.

3.1 Bindings

Just like an Agda program, a GRIN program consists of a bunch of definitions, or bindings.

```
data GrBind
  = Bind HsName [HsName] GrExpr
  | Rec [GrBind]
```

As GRIN is an untyped language there are only function (or variable) definitions and no datatype declarations. There may be, however, mutually recursive definitions. A GRIN function binding consists of the name of the function, the list of its parameters (which are just simple names, no patterns like in Agda) and the expression, or body, of the function.

3.2 Values

Before delving into the details of what expressions (*GrExpr*) in GRIN look like, we will first explain what *values* in GRIN are. Values in GRIN come in two kinds, *words* and *nodes*.

- Words are most often pointers to nodes. A word may also be a primitive integer: An integer value implemented directly on the level of machine integers. Words are the only kind of values that can be passed to functions.
- Nodes are compound values. They correspond mostly to Agda or Haskell (algebraic) datatypes. A node consists of a *tag* and zero or more fields. These fields themselves always contain words, not nodes. Nodes are the only values that can be returned from functions in GRIN. The tag is used to distinguish different constructors and determines the number of fields.

As functions always take words as arguments and return nodes, some converting of values will often be necessary. GRIN constructs exist for storing a node value, which yields a (pointer) word value, and for evaluating a pointer value, which will return a node value. These constructs and their use will be explained in more detail later.

The GRIN datatype corresponding to GRIN values, *GrVal*, contains one important extra case: A *GrVal* may also be a *variable*. *GrVal* is defined as follows:

```
data GrVal
  = Var HsName
```

A variable can be one of a few things: A global identifier, a function parameter (bound by a *GrBind*) or a local variable (bound by *GrExpr_Seq*). A variable may be either a word-value or a node-value.

```
  | LitInt Int
```

The *LitInt* constructor is used to denote word-values that are machine integers. Note that GRIN knows no literal pointers, making *LitInt* the only way to write a word-value directly in GRIN code.

```
  | Node GrTag [GrVal]
```

The *Node* constructor can be used to build a node-value. As can be seen in the definition, a *Node* consists of a tag and zero or more values. Although not enforced by the type, these fields may not be node-values themselves: They must be word values. The definition of *GrTag* will be explained in a later section.

```
  | BasicNode GrTag HsName
  | EnumNode HsName
```

Finally, there are the constructors *BasicNode* and *EnumNode*. A *BasicNode* is conceptually the same as a normal *Node*, but only works for primitive types like machine integers. *EnumNode* is intended for simple enumeration types such as *Bool*. Both are used to interface between normal GRIN functions and functions built-in to the run-time system.

3.3 Expressions

```
data GrExpr
  = App HsName [GrVal]
  | Call HsName [GrVal]
```

The first kinds of expressions in GRIN are two different ways of applying a function In both cases, the *HsName* (GRIN's equivalent of Agda's *QName*) is the identifier of the function to call, and the list of *GrVal* are the arguments to the function. The difference is what identifiers can be passed: *Call* is intended for applying *known functions* only, that is, applications where the function applied is known statically. In practice, this means that the *HsName* in that case will

directly be the name of a global function. *App* on the other hand, expects the *HsName* to be a function parameter or local variable. In other words, *App* is for higher-order application.

```
  | Unit GrVal
```

Unit can be used to "lift" a value to the expression level. It can be used to bind a node to a variable name, but is most often used at the end of a function definition to return a node.

```
  | Eval HsName
  | Store GrVal
```

These are the constructs used to convert between nodes and words or, alternatively, to help implementing laziness. *Eval* takes a variable name that must be a pointer value, and evaluates whatever the value points to. The result will be a node value. *Store*, on the other hand, takes a node value and stores it, returning a word value pointing to the stored node.

```
  | FFI String [GrVal]
```

FFI stands for "foreign function interface". GRIN has no primitives for interacting with the world outside GRIN. Instead, it provides the *FFI* instruction as a way of calling arbitrary C-functions. The *String* argument tells the compiler what function to call. Also, only pointers and some primitive types may be passed to C.

```
  | Seq GrExpr GrPatLam GrExpr
```

Seq stands for "sequence". *Seq* e_1 p e_2 is what has been written as $e_1; \backslash p \rightarrow e_2$ above, and first executes e_1, binds the results to the pattern p and then executes e_2. The different forms of what *GrPatLam* can be will be explained later.

```
  | Case GrVal [GrAlt]
```

In GRIN there is only one way of conditional control flow: Pattern matching using *Case* expressions. The first argument to a *Case* constructor is the expression to pattern match on, and the second argument is a list of the different cases, whose exact meaning will be explained later.

3.4 Tags

Every *Node* has a *GrTag*, telling the compiler what kind of node it is. Ultimately, when GRIN is compiled into even lower level intermediate languages (and finally machine code), all tags will simply be machine integers, but while in GRIN tags still have some structure.

```
data GrTag
  = Con GrTagAnn Int HsName
```

The simplest kind of tag is a *Con* tag, which means that the *Node* represents a constructor in Agda-country. A *Con* has three fields:

- A *GrTagAnn* is about constructor-arities. A *GrTagAnn* contains two *Int* fields, the first of which is the arity of the constructor we are working with, and the second is the maximum of the arities of all the constructors of the datatype this constructor belongs to:

  data *GrTagAnn* = *TagAnn Int Int*

- The *Int* field specifies the actual tag, that is, the number of the constructor we are dealing with.

- Finally, the *HsName* is the name of the constructor, which is mostly for debugging purposes.

Recall for example our earlier $\mathbb{N}$ datatype. Its *zero* constructor is be compiled to the following GRIN *GrVal*

Node (*Con* (*TagAnn 0 1*) *0* `"zero"`) []

and the Agda expression *suc zero* to

Node (*Con* (*TagAnn 1 1*) *1* `"suc"`) [*Var* `"x"`]

where `"x"` would be a pointer to above *zero* node.

As indicated before, GRIN has no built-in closures, yet is intended as an intermediate language for lazy languages. Instead of directly supporting closures, GRIN allows closures to be represented by ordinary *Node*s with special tags. The following *GrTag* constructors are intended for this goal:

```
| Fun HsName
| App HsName
```

*Node*s with an *App* or *Fun* tag represent the application of a function. For example, *Node* (*Fun* `"f"`) [*Var* `"x"`, *Var* `"y"`] is a node representing the application of a function f to x and y. GRIN knows nothing about currying, so this implies that f is actually a function with arity 2. The difference between *Fun* and *App* is the same as with their *GrExpr* equivalents: *Fun* is for known and *App* for unknown functions. So in the above *Fun* `"f"`, the name "f" should be bound with a *Bind*. If it would have been a name bound by as a function parameter, case pattern or seq pattern, it would have been *App* `"f"` instead.

```
| PApp Int HsName
```

Finally, a *PApp* stands for *partial application*, applying a function to less arguments than it expects. The *Int* arguments indicates the number of arguments that is still missing.

3.5 Case alternatives

Recall the *Case* constructor of the *GrExpr* datatype:

```
| Case GrVal [GrAlt]
```

Each branch of a *Case* construct contains a pattern and an expression:

```
data GrAlt
  = Alt GrPatAlt GrExpr
data GrPatAlt
  = PatAltLitInt Int
  | PatAltNode GrTag [HsName]
  | PatAltOtherwise
```

It can be seen that there are three possible patterns to match on: The first is for pattern matching on literal integers, which map to the built-in integer type. A *PatAltNode* matches on a certain tag and binds the remaining fields of the node to the given *HsName*s. Finally, a *PatAltOtherwise* is a default case that always matches and does not bind any variables.

3.6 *Seq*-patterns

The last GRIN datatype left to explain is *GrPatLam*, the type of the patterns bound by *Seq* expressions.

```
data GrPatLam
  = PatLamVar HsName
  | PatLamBasicAnnot HsName
  | PatLamBasicNode HsName
  | PatLamEnumAnnot [GrTag] HsName
  | PatLamEnumNode HsName
```

For now, the only important pattern is the *PatLamVar*: *Seq* e_1 (*PatLamVar* `"x"`) e_2 simply means that in e_2, `"x"` will be bound to the result of e_1. The other patterns all for marshalling primitive types to and from C, and will be explained when necessary.

3.7 An example

As the end of this GRIN introduction, this is the previous _++_ GRIN function again, but this time shown directly as a data structure instead of a program.

```
Rec [Bind "_++_" ["α","m","n","vec1","vec2"]
  (Seq (Eval "vec1") (PatLamVar "vec1'")
  (Case "vec1'"
    [Alt (PatAltNode (Con (TagAnn 0 3) 0 "[]") [])
      (Eval "vec2")
    ,Alt (PatAltNode (Con (TagAnn 3 3) 1 "_::_") ["m'","x","xs"])
      (Seq (Store (Node (Fun "_+_") [Var "m'", Var "n"])) (PatLamVar "i")
      (Seq (Store (Node (Fun "_++_") [Var "α", Var "m'", Var "n", Var "xs", Var "vec2"]))
        (PatLamVar "zs")
      (Unit (Node (Con (TagAnn 3 3) 1 "_::_") [Var "i", Var "x", Var "zs"])))))
    ]))]
```

Compare the verbosity of the abstract syntax to the more simple version of the GRIN concrete syntax shown on page 23.

4

Compiling Agda to GRIN

This chapter will start by comparing the structure of an archetypical Agda internal syntax program to the GRIN code it is compiled into. A high altitude overview will then be given of the structure of the implemented compiler. The infrastructure created to aid this compilation will also be introduced. Then, each part of the compilation will be discussed in detail.

4.1 Comparing an Agda and a GRIN module

This section will show the outline of a very simple but representative Agda program and compare it with the Agda internal syntax program it is transformed into. Then, the Agda internal syntax will be compared with the GRIN code it is finally compiled into.

Agda	Agda internal syntax
module A **where**	**module** A **where**
data D_1 : *Set* **where**	**data** $A.D_1$: *Set*
C_1 : D_1	*con* $A.C_1$ *of* $A.D1$: $A.D_1$
C_2 : $D_1 \rightarrow D_1$	*con* $A.C_2$ *of* $A.D1$: $A.D_1 \rightarrow A.D_1$
f : ...	*fun* $A.f$
$f\ x$ = ...	*type* : ...
g : ...	*def* : $\backslash 0 \rightarrow \ldots$
$g\ C_1\ y$ = ...	*fun* $A.g$
$g\ C_2\ y$ = ...	*type* : ...
module B **where**	*def* : $\backslash A.C\ 0 \rightarrow \ldots$
open import A	$\backslash A.C\ 0 \rightarrow \ldots$
data D_2 : *Set* $\rightarrow$ *Set* **where**	**module** B **where**
C_3 : ... D_1 ...	**data** $B.D_2$: *Set* $\rightarrow$ *Set*
h : ...	*con* $B.C_3$ *of* $B.D_2$: ... $A.D_2$ ...
$h\ x\ y$ = ... f ...	*fun* $B.h$
{-# BUILTIN ... t #-}	*type* : ...
postulate t : *Set*	*def* : $\backslash 1\ 0 \rightarrow \ldots\ A.f\ \ldots$
primitive *prim* : ...	**postulate** $B.t$: *Set*
	primitive $B.prim$: ...
	BUILTIN ... $B.t$

The left column shows the outline of a simple Agda program consisting of two modules with a few simple definitions. The right column shows the corresponding Agda internal syntax in pseudo-syntax. The most important differences between the structures of the Agda source and the Agda internal syntax are:

- Datatype definitions have been split into separate definitions for the type constructor and the value constructors.
- The separate parts of function definitions have been merged: The type signature and each clause of a function is now part of one definition.
- Every non-local identifier is fully qualified in Agda internal syntax
- Parameter names have been replaced by De Bruijn indices.

The Agda internal syntax program will now be compared to the GRIN code it is compiled into.

Agda internal syntax	GRIN
module A **where**	$impossible\ =\ \ldots$
data $A.D_1\ :\ Set$	$SetX\ =\ \ldots$
$con\ A.C_1\ of\ A.D1\ :\ A.D_1$	$A\_D_1\ =\ SetX$
$con\ A.C_2\ of\ A.D1\ :\ A.D_1\ \rightarrow\ A.D_1$	$A\_C_1\ =\ \ldots$
$fun\ A.f$	$A\_C_2\ 0\ =\ \ldots$
$type\ :\ \ldots$	Rec
$def\ :\ \backslash 0\ \rightarrow\ \ldots$	$A\_f\ 0\ =\ \ldots$
$fun\ A.g$	$A\_g\ 1\ 0\ =\ case\ 1\ of$
$type\ :\ \ldots$	$A\_C_1\ \rightarrow\ \ldots$
$def\ :\ \backslash A.C\ 0\ \rightarrow\ \ldots$	$A\_C_2\ \rightarrow\ \ldots$
$\backslash A.C\ 0\ \rightarrow\ \ldots$	$B\_D_2\ 0\ =\ SetX$
module B **where**	$B\_C_3\ \ldots\ =\ \ldots\ A\_D_1\ \ldots$
data $B.D_2\ :\ Set\ \rightarrow\ Set$	Rec
$con\ B.C_3\ of\ B.D_2\ :\ \ldots\ A.D_2\ \ldots$	$B\_h\ 1\ 0\ =\ \ldots\ A\_f\ \ldots$
$fun\ B.h$	$B\_t\ =\ SetX$
$type\ :\ \ldots$	$B\_prim\ \ldots\ =\ \ldots$
$def\ :\ \backslash 1\ 0\ \rightarrow\ \ldots\ A.f\ \ldots$	
postulate $B.t\ :\ Set$	
primitive $B.prim\ :\ \ldots$	
$BUILTIN\ \ldots\ B.t$	

The most important differences between the structures of the Agda internal syntax and the GRIN code are:

- The module structure has disappeared: All modules are collapsed into one set of GRIN bindings.
- Some definitions are combined into *Rec*ursive groups.
- All different types of Agda definitions are compiled into just one (the only) kind of GRIN binding.
- Even type constructors have become function bindings.
- A few extra bindings are added: *impossible* and *SetX*. These might be considered to form part of the run-time system.
- No type signatures exist anymore
- Primitive declarations are compiled into fixed pieces of GRIN code. Primitive declarations can be considered as optional pieces of the run-time system.
- No vestiges remain of the existence of built-in pragmas (although they are still stored in a separate map of all built-in pragmas).

4.2 Compilation overview

This section will give an overview of compiling from Agda internal syntax to GRIN. The top-level view of compiling Agda's internal syntax to GRIN begins by collapsing the module structure of the Agda internal syntax. As every name in Agda internal syntax is already fully qualified, this is a rather simple task.

Then, a number of lookup tables are constructed:

- A table is constructed from all BUILTIN pragmas. Creating this mapping serves two purposes.
  - It links the information from BUILTIN pragmas for constructors to the information from the BUILTIN pragmas for the corresponding types. This allows the compiler to make requests such as "give me all the constructors of whatever type I should use as BOOL".
  - It checks whether all the required BUILTIN pragmas have been given, and whether they have been defined in a correct way.
- An arity lookup table is constructed.

  This is a mapping from top-level identifiers to their arities. Note that, although in Agda all functions are curried and thus have arity one, functions in GRIN can have any (fixed) number of parameters, and this is used for efficiency reasons.

  This table is not used to find the arity of data, and in particular record, value constructors though. The reason for this is that in Agda internal syntax, record type and value constructors have the same name. However, as in general they do not have the same arity, this mapping can only be used for retrieving the arity of the record type constructor, and not for the arity of the record value constructor. Then, in order to be able to treat record and data types in the same way, we also do not use the mapping for the arities of datatype constructors.
- The special name lookup table is created.

  Agda's identifiers cannot be used as such in GRIN: Agda identifiers can contain almost any unicode character, while GRIN identifiers are limited to a subset of ASCII. In addition, there is no one-one mapping from Agda name spaces to GRIN name spaces. For these reasons, there is a somewhat complicated renaming scheme from Agda names to GRIN names. Certain names however (e.g. *main* and *True*) must be treated specially in order to allow the run-time system to work correctly.

  Thus, a mapping is created containing the translation for these special names.
- Two datatype information lookup tables are constructed.

  Two interlinked mappings are created containing information about defined datatypes. One of these maps each datatype name to all information about that datatype, and the other maps each constructor name to all information about that constructor. Note that during the creation of these mappings, records are treated as (syntactic sugar for) datatypes with a single constructor, and from that point on need no special treatment.

The next part then is the actual generation of GRIN bindings, which take place in the following order.

1. Compile postulates (described in more detail in section 4.3)

   This part walks over all declared postulates, checks them for validity and either emits the corresponding GRIN code or fails with an error message.

2. Compile primitives (described in more detail in section 4.4)

   This part walks over all declared primitives, checks them for validity and either emits the corresponding GRIN code or fails with an error message.

3. Generate run-time system bindings

   A few fixed GRIN bindings are always generated for later use by other generated code.

4. Generate bindings for data/record type/value constructors (described in more detail in section 4.7)

   For each datatype and each data constructor, a GRIN binding is generated. This is done based on the datatype information lookup table constructed before, so records are treated as single constructor datatypes here too.

5. Check whether main has the right type.

6. Compile all functions, grouping them by mutual recursion group. (described in more detail in section 4.8)

This ultimately results in a list of GRIN bindings. Then, EHC is used to compile these GRIN bindings into a *C* program, and GCC is used to compile and link the C program.

In the following sections, all this will be explained in more detail.

4.3 Compiling built-in datatypes

The compiler supports two different classes of "built-in datatypes": Types that are defined as perfectly normal Agda datatypes, but have special support in the back end for primitive functions, and types that are defined in Agda as **postulate**s, and whose actual definition is created only in the back end. The first class of built-in datatypes currently consists of the following types:

- $\mathbb{N}$ is the datatype of (peano) natural numbers.

  In Agda, as in many other programming languages, it is possible to write a special kind of comment called a *pragma*, which is used to tell the compiler about various things that the compiler has to know about, but that do not really fit into the language. For example, Agda allows the programmer to use literal numbers, like 0 or 42. However, Agda does not have a primitive type for numbers, so the programmer will have to use one of a fixed set of compiler pragma to tell the compiler what to translate literal numbers into. For example, the programmer could write

  ```
  data ℕ : Set where
    zero : ℕ
  ```

```
  suc  : ℕ → ℕ
{-# BUILTIN NATURAL ℕ #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC suc #-}
```

Now, using these "BUILTIN" pragmas, the Agda compiler knows that the literal *2* must be interpreted as *suc* (*suc zero*).

- *Bool* is also a perfectly normal Agda datatype

```
data Bool : Set where
  false : Bool
  true  : Bool
```

However, the compiler supports special pragmas for it

```
{-# BUILTIN BOOL Bool #-}
{-# BUILTIN FALSE false #-}
{-# BUILTIN TRUE true #-}
```

When these built-in pragmas are used, the Agda front end checks that *Bool* is indeed a valid type for use as a boolean. (e.g. that it has exactly two constructors, both of which are nullary, etcetera. Note that Agda does not care what names are used.) The back end then uses the information from the built-in pragmas to be able to construct boolean values in primitive functions like *primIntegerEquality* : *Integer* → *Integer* → *Bool*.

The other class of built-in datatypes consists of two types

- *Integer* is a datatype that can be defined and used as follows

```
postulate
  Int : Set
 {-# BUILTIN INTEGER Int #-}
primitive
  primIntegerPlus : Int → Int → Int
```

An *Int* value is then implemented as a machine integer

- *World* is a datatype used for the implementation of monadic IO. It can be defined as follows:

```
postulate
  World : Set
 {-# BUILTIN WORLD World #-}
primitive
  primPutInt : Int → World → Int
```

Its further use will be explained in section 4.9.

4.4 Compiling primitives

When the compiler encounters a primitive, it proceeds as follows: First, it checks whether the GRIN back end actually supports this particular primitive, and terminates with an error message if it does not. Otherwise, it checks what names to use for the primitive function itself and, if necessary, for the built-in types it uses, and generates the hard coded definition for the primitive using these names.

4.5 Renaming identifiers

Renaming identifiers from Agda to GRIN follows the following main scheme: Qualified names in Agda (*QName*) contain a magic number that is unique for each name. We take the last part of the qualified name, and add the magic number as a suffix, and add an underscore (_) as a prefix. For example, *RunningExample.Vec* might become $\_Vec\_42$. The underscore is added to all user defined Agda names to ensure that no collisions can occur with internally generated names, which never start with an underscore. The last part of the qualified name is added merely to make the intermediate GRIN code somewhat more readable for compiler debugging purposes. The magic number added at the end is what actually guarantees each name to be unique, a property which is required in GRIN. In addition, we may have to escape part of the name, as Agda identifiers can use almost any valid Unicode character, while GRIN only supports a more limited subset.[1]

There are, however, three classes of names that do not follow the previous renaming scheme:

- The EHC back end expects a few constructs to have certain names in order to be able to interface with some primitive functions. For example, the compiler must be able to generate code for the primitive function $primIntegerEquality :: Int \rightarrow Int \rightarrow Bool$. In order to generate the *Bool* values to return, they are required to be called *FALSE* and *TRUE* in GRIN. Another example is that the compiler must be able to find the *main* function. Therefore, care is taken to ensure that the boolean constructors are renamed to *FALSE* and *TRUE*, and that the Agda function *main* is called *main* in GRIN.

- The second class is the class of value constructors. When defining a datatype one declares, among others, the name of the datatype constructor, and the names of the value constructors. However, when declaring a record, one does not explicitly declare the name of its one value constructor: After all, records values are always constructed using **record** $\{\ldots\}$ in Agda. Unfortunately, within Agda internal syntax however, both record types and record values simply use the name of the record type constructor. To prevent nasty clashes between them, we use a separate namespace

[1] EHC does in fact do its own escaping of GRIN identifiers when compiling further down to C, but is tailored to escaping the characters which can occur in Haskell but not in C. Agda's set of valid characters for use in identifiers is larger than Haskell's, so EHC's escaping is unfortunately inadequate for our purposes.

for value constructors: For both datatype and record types, all value constructors are prefixed not with _ but with *con_*. In this manner, we do not run into conflicts between type and value constructors while still being able to otherwise treat datatypes and record types in the same way in our back end.

- Finally, some GRIN bindings are added for which we generate our own names without any renaming at all. For example, bindings are always generated for the GRIN value *SetX*, which is the value that all types reduce to in our GRIN implementation, and *impossible*, which is a special value that aborts the problem with an error message and is used in cases where GRIN expects some code, but that we know are actually unreachable. In addition to that, when compiling Agda lambda's and pattern matching code, we often generate internal helper functions whose names we always prefix with, respectively, *lam_* and *alt_*.

4.6 Modules

The first interesting piece of code of the compilation overview is

$$\begin{array}{rcl} \textbf{let}\ \mathit{defs} & = & \mathit{mainModuleDefinitions} \cup \mathit{otherModulesDefinitions} \\ \mathit{builtins} & = & \mathit{mainModuleBuiltins} \cup \mathit{otherModulesBuiltins} \end{array}$$

This is (apart from a few minor renaming issues) the only part where special care has to be taken to support Agda modules. The implementation of support for modules works as follows: The compiler first retrieves from the Agda front end the definitions and information about "BUILTIN" pragmas of the "main" module. Then, it asks an Agda utility function to construct a list of all modules used by the "main" module (directly or indirectly through other modules). This list is then used to retrieve the definitions and information about the "BUILTIN" pragmas of these modules. For brevity, the part so far has not been shown in the overview.

In Agda's internal syntax, all identifiers are already globally unique (due to the fact that they are already fully qualified, and there are no local names), so it is then a simple matter to union all definitions of both the "main" module and all the visited modules together. For built-ins the situation is only slightly more complicated: Multiple modules may contain "BUILTIN" pragmas for the same types, but they will then all be exactly the same. Therefore, the union-function used for built-ins does not require the built-in mappings to be entirely disjoint: There may be, for example, two built-ins for *NATURAL*, but if one of them says $\mathbb{N}$, the other has to say exactly the same. This merging of modules happens in the code shown above.

From then on, we can go on compiling as if there was only one big module from the start. As GRIN is a language meant for whole program compilation anyway, this does not impede separate compilation either.

4.7 Datatypes and records

Compilation of datatypes consists of two separate parts. During the first part, all information from the internal syntax program about datatypes and records

is collected into one single data structure. The second part consists of iterating over this data structure and generating various GRIN bindings based on what is found in this structure.

The reason behind this split is that during compilation of other parts of the program (e.g. while compiling functions, or primitives) the compiler often needs to know all kinds of information about constructors used there. Therefore, constructing the big datatype information data structure is one of the first things the GRIN back-end does.

In both parts, records are treated as a special case of datatypes.

4.7.1 Preprocessing datatype and record definitions

The first step in compiling datatypes consists of iterating over them all and creating a data structure containing all the information about type and value constructors. In particular, two mappings are created:

- from datatype (or record) name to a *DataInfo* value holding information about that datatype (or record type) (*Map QName DataInfo*)
- from datatype (or record) value constructor to a *ConstructorInfo* value holding information about that constructor (*Map QName ConstructorInfo*)

DataInfo is defined as follows:

```
data DataInfo = DataInfo
  { diName        :: HsName
  , diArity       :: Int
  , diMaxConArity :: Int
  , diConCount    :: Int
  , diCons        :: [ConstructorInfo]
  }
```

It consists of the name of the datatype, the arity of the type constructor, the maximum arity of all its value constructors, the number of constructors it has and a list of the corresponding *ConstructorInfo* values. *ConstructorInfo* has the following definition:

```
data ConstructorInfo = ConstructorInfo
  { ciName  :: HsName
  , ciTagNr :: Int
  , ciArity :: Int
  , ciData  :: DataInfo
  }
```

It contains the name of the constructor, the tag number that will represent this constructor when the program is finally compiled from GRIN to C (or some other low-level back-end language), the constructor arity and the corresponding *DataInfo*.

4.7.2 Creating GRIN bindings

Once all needed information about datatypes defined and used in the program has been collected, the compiler can proceed to the second step in compiling

datatypes: GRIN bindings need to be generated for all these types and constructors.

However, there are also types used in each Agda program that have no definition in the internal syntax program: Set_0, Set_1 etcetera. In GRIN, these types are all collapsed into one type called *SetX*. Note that this has little to do with the possibility of having $Set : Set$, because GRIN is an untyped language anyway. A binding *SetX* is generated as if *SetX* where a nullary constructor of a singleton type. As we will shortly see, *SetX* is also slightly abused for another purpose.

Then, the compiler iterates over all items in the *DataInfo* mapping and for each datatype performs the following actions:

- A GRIN binding is generated of the same name and arity as the type constructor, that returns the value *SetX*. Note that this means that any Agda type application such as e.g. *Vec Bool* $\mathbb{N}$ will, when compiled to GRIN, ultimately evaluate to *SetX*. This is not the right answer, as *Vec* would ideally be a true constructor and *Vec Bool* $\mathbb{N}$ a normal form. However, as we have no type case, there is no way to distinguish different types and we can get away with this shortcut.

- In addition, one binding is generated for each value-level constructor. This GRIN binding has the same name and arity as the corresponding constructor.[2] When called, it constructs a GRIN *Node* corresponding to the desired constructor. Now, if a constructor is encountered while compiling Agda internal syntax expressions, the compiler can simply emit a call to the function of the same name, without needing to worry whether the constructor call is fully saturated or not.

4.8 Compiling functions

The compilation of Agda internal syntax functions starts with the following piece of code:

```
mapM_ (genRecBindGroup ∘ mapM_ compileFunction)
  $ groupMutuals functionDefinitions
```

First, a list of all Agda internal syntax functions is retrieved and grouped into sets of mutually recursive functions using *groupMutuals*. This is quite easily done, as each function contains a "mutually recursive set ID". Then, each function in each list of mutually recursive functions is independently compiled into one or more GRIN bindings using *compileFunction*. Finally, *genRecBindGroup* wraps each set of mutually recursive GRIN bindings into a *Rec* construct.

The rest of this section will be about *compileFunction*, which compiles a single Agda internal syntax function into one or more GRIN bindings. The work performed by this function consists of two major parts:

- Compiling the left-hand side side (that is, the patterns) into GRIN pattern matching code.

[2] Note that in case of record types a constructor has been invented to make records fit in the general datatype scheme.

- Compiling the right-hand side (an Agda internal syntax expression) into GRIN code.

These will be explained in more detail in the following sections.

The following is an overview of how compilation of a function proceeds:

```
compileFunction :: Definition → GrinT TCM ()
compileFunction definition = do
  — Disconnect LHSs from RHSs
  let funName = defName definition
      clauses    = funClauses (theDef definition)
      (pmcClauseNums, numToBruijnTerm)
                 = disconnectCaseRHSs (genPMClauses clauses)
  — Compile LHSs (pattern match compilation) into a CaseTree
  args ← genArgs ∘ length ∘ clauseArgs $ head clauses
  caseTree ← compPatMatchClauses freshGrinName freshDummyName
    (LZ.fromList args) pmcClauseNums
  — Compile RHSs into GRIN code,
  — top-level-lifting each duplicated RHS into a function first
  let rhsInfoMap = M.intersectionWith (,) numToBruijnTerm
    $ M.fromListWith (const $ first succ)
    [(num, (1, transMap))
      | (transMap, Just num) ← F.toList caseTree]
  rhsMap ← traverse (compCaseRHS (unQName funName)) rhsInfoMap
  — Reconnect RHSs to the LHSs that have become a CaseTree.
  — (Add error-code for any remaining "impossible" cases.)
  let caseTree' = maybe
      (GrExpr_Eval $ HNm "impossible")
      (mapIndex "compileFunction" rhsMap)
    ∘ snd
    <$> caseTree
  — Compile CaseTree into a GRIN binding
  hsFunName ← getHsName funName
  execExprT [] (compileCaseTree caseTree')
    ⋙= genBind hsFunName args
```

4.8.1 Compiling a functions' left-hand side: Patterns

In Agda's internal syntax, the left-hand side of each function clause consists of a number of potentially nested patterns. At run-time, the first clause whose patterns match the actual arguments passed to a function is actually executed. In GRIN however, a function has no patterns at all. Instead, a function has a number of arguments that are simple variables. When called, a function may perform case analysis on these variables, one at a time and not nested. Thus, an Agda function consisting of one or more clauses that match on one or more patterns must be compiled into a GRIN function that has only variables as its arguments, and then chooses which right-hand side to execute by performing case analysis (effectively in the form of a decision tree) on these arguments.

Unfortunately, during this compiling of multiple function clauses into one function definition with an explicit case tree, right-hand sides could get duplicated. As an example, consider the boolean AND operator:

```
_∧_          : Bool → Bool → Bool
true ∧ true  = true
_    ∧ _     = false
```

When compiling this into GRIN, it will ultimately become

```
Bind "_&&_" ["a1","a2"]
  (Seq (Eval "a1") (PatLamVar "v1")
  (Case "v1"
    [Alt (PatAltNode (Con (TagAnn 0 0) 1 "true") [])
      (Seq (Eval "a2") (PatLamVar "v2")
      (Case "v2"
        [Alt (PatAltNode (Con (TagAnn 0 0) 1 "true") [])
          (Unit (Node (Con (TagAnn 0 0) 1 "true") []))
        , Alt PatAltOtherwise
          (Unit (Node (Con (TagAnn 0 0) 0 "false") []))
        ]))
    , Alt PatAltOtherwise
      (Unit (Node (Con (TagAnn 0 0) 0 "false") []))
    ]))
```

Note that the right hand side *false*, which occurs only once in the Agda code, occurs twice in the GRIN code. In this case that is not a problem, as *false* is a very small expression. In general however, compiling pattern matching can cause arbitrarily large expressions to be duplicated a number of times.

Therefore, extra care is taken to prevent such duplication by lifting each right-hand side expression that will be duplicated into its own function. In this way, only the call to such a new function will be duplicated, thereby bounding the amount of duplicated work. It is for this reason that the above overview of function compilation is more complicated than a simple "first compile the left-hand sides of each function, and then compile the right-hand sides".

*PMClause*s

When compiling the left-hand side of a function, the first step consists of transforming the function definition from Agda's internal syntax into our own types for *pattern matching clauses*:

```
data Pat con var
  = PatVar var
  | PatCon con [Pat con var]
data PMClause con var a
  = PMClause (ListZipper (Pat con var)) a
```

A *ListZipper a* is just a list with a *cursor*, or "current location", enabling some operations to be performed much more efficiently than with a normal list. We use a function *genPMClauses* to generate these *PMClause* values based on the Agda internal syntax.

```
genPMClauses :: [Clause] → [PMClause QName Nat (Maybe Term)]
```

We use *QName* for constructors and *Nat* (de Bruijn indices) for variables here, just like in the original Agda internal syntax. Note that the right-hand side is still in Agda internal syntax (that is, a *Term*). However, as this may be an impossible case, there may not be a right-hand side: Hence the *Maybe Term*.

The next step is necessary to prevent duplicating the right-hand sides. Given a list of (pattern matching) function clauses, it returns these function clauses with each *Term* replaced by a separate natural number, together with a mapping that identifies each such natural number with the corresponding *Term*.

```
disconnectCaseRHSs
  :: [PMClause con var (Maybe a)]
  → ([PMClause con var (Maybe Nat)], Map Nat a)
```

Thus, instead of duplicating arbitrary terms, we will duplicate natural numbers, which can be easily tested for. When such duplication is then found, we will not substitute the original expression again, but only a call to the top-level-lifted version of such an expression.

These functions are then combined to translate an Agda internal syntax function definition into a list of pattern match clauses with numbers for right-hand sides, called *pmcClauseNums*, and a mapping from these numbers to the real right-hand sides, *numToBruijnTerm*.

```
compileFunction definition = do
  — Disconnect LHSs from RHSs
  let funName = defName definition
      clauses = funClauses (theDef definition)
      (pmcClauseNums, numToBruijnTerm)
              = disconnectCaseRHSs (genPMClauses clauses)
```

Compiling into *CaseTree*s

The next step is the actual pattern match compiling, which turns a list of *PMClause*s into a *CaseTree*:

```
data CaseTree con var a
  = Leaf a
  | Impossible
  | Branch var [(Maybe (con, [var]), CaseTree con var a)]
```

A *CaseTree* has three constructors:

- A *Leaf* is a trivial *CaseTree*. It is used for a single term without any pattern matching.
- *Impossible* is used for functions with no clauses at all. (e.g. False-elimination.)
- *Branch var alts* pattern matches on *var*, and *alts* is a list of alternatives. Each alternative consists of a pair, of which the first element is a

Maybe $(con, [var])$, which can be either a *Nothing*, implying a default-case, or a *Just*, in which case the *con* is the constructor it matches and the $[var]$ the list of variables that the fields of *con* are bound to. The second part of an alternative is again a *CaseTree*.

The function that actually turns the list of *PMClause*s into a *CaseTree* is *compPatMatchClauses*:

```
compPatMatchClauses
  :: (Monad m, Functor m, Ord con, Ord var)
  ⇒ (String → m var′)
  → (String → m var)
  → ListZipper var′
  → [PMClause con var a]
  → m (CaseTree con var′ (Map var var′, a))
```

When called as *compPatMatchClauses freshName1 freshName2 funArgs pmcs*, the first two arguments are functions generating fresh variable names for the two (potentially) different types of variables used. *funArgs* is the list(zipper) of variables that the patterns in the *PMClause*s are actually matching on, and *pmcs* is are the clauses themselves. Finally, a *CaseTree* is returned. Note that the leaves of the returned *CaseTree* contain not only the right hand sides of the original clauses, but also a substitution for the variables in the *PMClause*s: The variables bound by the *CaseTree* are not (and cannot be, in general) the same variables as those bound by the original clauses.

This function is used as follows:

```
— Compile LHSs (pattern match compilation) into a CaseTree
args ← genArgs ∘ length ∘ clauseArgs $ head clauses
caseTree ← compPatMatchClauses freshGrinName freshDummyName
  (LZ.fromList args) pmcClauseNums
```

First, the correct number of GRIN function arguments is created, after which the *PMClause*s (called *pmcClauseNums*) are compiled into a *CaseTree* (called *caseTree*). This value *caseTree* is of type *CaseTree QName HsName* (*Map Nat HsName*, *Maybe Nat*).

Compiling each right-hand side exactly once

At this point, we can find out what right-hand sides are in danger of being duplicated, simply by checking which numbers occur more than once in the *CaseTree*. Depending on this, we will now do one of two things:

- If a number occurs exactly once, we can simply compile the corresponding Agda internal syntax *Term* into a GRIN *GrExpr*, of course also using the substitution returned by *compPatMatchClauses*.
- If it occurs more than once, we lift the corresponding *Term* into the top-level and then compile it into GRIN. That is, the right-hand side is parameterized by all the variables in scope, and becomes a *GrExpr* inside of a completely new top-level GRIN *GrBind*. Then a GRIN expression calling this new GRIN function with the correct arguments is substituted for the original right-hand side.

The code in the function definition compiler doing this is:

```
— Compile RHSs into GRIN code,
— top-level-lifting each duplicated RHS into a function first
let rhsInfoMap = M.intersectionWith (,) numToBruijnTerm
  $ M.fromListWith (const $ first succ)
  [(num, (1, transMap))
    | (transMap, Just num) ← F.toList caseTree]
rhsMap ← traverse (compCaseRHS (unQName funName)) rhsInfoMap
```

Based on the previously constructed values *caseTree* and *numToBruijnTerm*, this creates a value *rhsInfoMap* :: *Map Nat* (*Term*, (*Int*, *Map Nat HsName*)), which maps each number representing a right-hand sides to all information that is needed to compile such a right-hand side:

- An Agda internal syntax *Term*.
- An *Int* that is the occurrence count of the right-hand side in the *CaseTree*.
- A *Map Nat HsName* that translates de Bruijn numbers as used in Agda internal syntax to GRIN *HsName*s.

This mapping is then traversed using the function *compCaseRHS*. That is, *compCaseRHS* is applied to each value in the mapping, and a new mapping is created where each key maps to the result of *compCaseRHS* applied to the original value.

```
compCaseRHS
  :: String
  → (Term, (Int, Map Nat HsName))
  → GrinT TCM GrExpr
```

compCaseRHS then performs the check explained at the start of this section and either

- compiles the expression directly into GRIN code using *compileTermToNode*. This function is described in more detail in section 4.8.2.
- or lifts it to its own top-level function using *liftCaseRHS*, which in turn also calls *compileTermToNode*.

The *liftCaseRHS* function has the following type

```
liftCaseRHS
  :: String
  → [HsName]
  → Term
  → GrinT TCM GrExpr
```

and works as follows: *liftCaseRHS name listOfVariablesInScope term*, it generates a function with a name based on *name* (in practice, *name* will be the name of the Agda internal syntax function that is being compiled) and with one argument for each variable in scope, that when called evaluates whatever the Agda internal syntax *term* is compiled to. It also returns a GRIN expression that performs such a call.

Reconnecting left-hand and right-hand sides

So far, we have compiled the left-hand sides of all function clauses into one big *CaseTree* pointing to numbers representing right-hand sides, and we have compiled each right-hand side into GRIN expressions. The next step, therefore, is to replace these numbers by the GRIN expressions they represent. This is done by the following code, with a few minor complications: The *CaseTree* doesn't actually point directly to *Nat*, but to (*Map Nat HsName*, *Maybe Nat*). The first part, the variable substitution, is no longer relevant and can be trivially thrown away. The second part is a *Maybe Nat*, because some function clauses may not actually have any right-hand side due the clause matching on impossible cases. This is easily solved by substituting some GRIN code that aborts the program with an error message in such cases. Note that ideally, such cases do not occur at all: As long as no "unsafe" Agda features are used, "impossible" really means "impossible", in which case such impossible clauses can simply be optimized away.

```
— Reconnect RHSs to the LHSs that have become a CaseTree.
— (Add error-code for any remaining "impossible" cases.)
let caseTree' = maybe
  (GrExpr_Eval $ HNm "impossible")
  (mapIndex "compileFunction" rhsMap)
  ∘ snd
  <$> caseTree
```

Compiling *CaseTree*s

The only thing left to do is to compile the resulting *CaseTree* into GRIN case analysis code, and generate the actual GRIN binding:

```
— Compile CaseTree into a GRIN binding
hsFunName ← getHsName funName
execExprT [] (compileCaseTree caseTree')
  >>= genBind hsFunName args
```

This uses the function *compileCaseTree*, which turns a *CaseTree* of GRIN expressions into a GRIN expression itself:

```
compileCaseTree
  :: CaseTree QName HsName GrExpr
  → ExprT (GrinT TCM) GrExpr
```

This function simply recurses trough the *CaseTree*, turning it into a number of nested GRIN *GrExpr_Case* expressions. The resulting *GrExpr* is finally turned into a GRIN binding using the aforementioned *genBind* and *execExprT*.

4.8.2 Compiling a functions' right-hand side: Expressions

We have now seen an overview of how a function is compiled into GRIN, and we have seen the details of compiling the left-hand side. A more in-depth explanation of compiling the right-hand side of a function — that is, expressions — will

now follow. Compiling the right-hand side of a function always begins with the function *compCaseRHS* (see section 4.8.1), which either directly or indirectly via *liftCaseRHS* calls the function *compileTermToNode*. This function is one of two functions that recursively call each other to perform the actual compilation of an Agda internal syntax *Term* into a GRIN *GrExpr*:

```
compileTermToNode :: Term → ExprT (GrinT TCM) GrExpr
compileTermToWord :: Term → ExprT (GrinT TCM) GrVal
```

compileTermToNode compiles the right-hand side of a function — an Agda internal syntax term — into a GRIN expression that returns a node value. In the process of doing so, it may use *genExpr* to emit "statements" calculating intermediate results (or more properly, to emit GRIN expressions that will be later combined into the function body using *Seq*). *compileTermToWord* is a close cousin to *compileTermToNode* that compiles a *Term* into a word value. The difference is that where the final return value of *compileTermToNode* is an expression returning a node value, *compileTermToWord* returns the name of a variable to which the wanted word value has been assigned. Another way of looking at this is that the result of *compileTermToNode* is a GRIN expression that, when executed computes the *weak head normal form* (WHNF) of the compiled *Term*, whereas *compileTermToWord* returns a pointer to a *thunk* representing the compiled *Term*.

We will now describe *compileTermToNode* in more detail. This function performs case analysis on the *Term* and uses the result to decide what kind of GRIN code to generate. Recall the definition of *Term*:

```
data Term
  = Con QName [Term]
  | Def QName [Term]
  | Var ℕ [Term]
  | Lit Literal
  | Lam Term
  | Fun Type Type
  | Pi Type Type
  | Sort Sort
```

An explanation will follow of what GRIN code it generates for each of these constructors.

- *Con QName* [*Term*]

  When *compileTermToNode* encounters an Agda internal syntax value constructor, it first looks up the arity of the constructor and the name it will have in the generated GRIN code, and then calls a utility function *compCallToNode* with this name, arity and the arguments passed to the constructor. This function will then construct the correct GRIN code to call the corresponding GRIN constructor function.

- *Def QName* [*Term*]

  This is treated almost the same as the previous constructor. The arity of the global binding (which may be any non-value-constructor global

identifier) and its GRIN name are looked up, after which *compCallToNode* takes care of constructing the actual call.

There is one exception though: If the *QName* has the value *primNatToInteger*, which is a primitive function intended to efficiently convert user-defined Peano numbers to efficient machine integers, and if it is applied to an integer literal, then we do not generate the code to construct the Peano number together with a call to *primNatToInteger* to convert it to a machine integer. Instead, we immediately insert wanted machine integer. Using this hack, the programmer may write code such at $primNatToInteger\ (1000000 : \mathbb{N}) : Int$ without being afraid that a million *suc*-constructors will be created and processed.

- $Var\ \mathbb{N}\ [Term]$

  For locally bound variables, there are two possibilities: If the *Var* has no arguments, simply emit an *Eval* GRIN construct. If it does have arguments, we perform the following steps:

  - Emit an *Eval* constructo to evaluate the variable name.
  - Call *compileTermToWord* once for each of the arguments. This effectively generates GRIN code that creates thunks for the arguments.
  - Emit an *App* construct with the evaluated variable as function and the thunks as arguments.

- *Lit Literal*

  At the moment, the only literal values supported are integers. These are currently compiled into Peano naturals using the built-ins *NATURAL*, *SUC* and *ZERO* as explained in section 4.3.

- *Lam Term*

  GRIN does not support lambdas, so we build a lambda-lifted top-level function for the lambda using the function *lambdaLiftTerm*: GRIN does not support lambdas, so whenever we find a (nested) λ-abstraction, we create a new function with as its arguments all the variables in scope at the point where the λ occurred *and* the parameters of the (nested) λ itself. This is done using the following function:

  $$\begin{aligned} &lambdaLiftTerm \\ &\quad :: \ Term \\ &\quad \rightarrow \ ExprT\ (GrinT\ TCM)\ (HsName, Nat, [Term]) \end{aligned}$$

  Then, *compCallToNode* is used to construct GRIN code to partially apply this new top-level function to the variables in scope.

- *Fun Type Type*, *Pi Type Type*, *Sort Sort*

  As explained in section 4.7.2, the Set_n hierarchy has been collapsed in GRIN, and even normal types are compiled down to *SetX*. Therefore, all these constructors are simply compiled to an *Eval* of *SetX*.

compileTermToWord is very similar to *compileTermToNode*, although it of course uses *compCallToWord* as its most important utility function. The important helper functions *compCallToNode* and *compCallToWord* will now be explained.

Compiling function calls

The *compCallToNode* and *compCallToWord* functions have the following type signature:

```
compCallToNode :: HsName → Nat → [Term] → ExprT (GrinT TCM) GrExpr
compCallToWord :: HsName → Nat → [Term] → ExprT (GrinT TCM) GrVal
```

That is, given the name of the function that should be called, its arity, and its not yet compiled arguments, it constructs GRIN code to execute this call. The first action performed by both *compCallToNode* and *compCallToWord* is to compile the not yet compiled arguments into GRIN words, using *compileTermToWord*. What happens then depends on the arity of the function and on the actual number of arguments it is being passed.

```
compCallToNode name arity args = do
  args' ← mapM compileTermToWord args
  let nArgs = L.genericLength args
  case (arity, arity `compare` nArgs) of
    (0, GT) → error "impossible: length args < 0"
    (0, EQ) → return $ GrExpr_Eval name
    (0, LT) → do
      f ← genFreshExpr $ GrExpr_Eval name
      return $ GrExpr_App f args'
    (_, GT) → return $ GrExpr_Unit
                     $ GrVal_Node (GrTag_PApp (fromIntegral $ arity − nArgs) name) args'
    (_, EQ) → return $ GrExpr_Call name args'
    (_, LT) → do
      let (args1, args2) = splitAt (fromIntegral arity) args'
      f ← genFreshExpr $ GrExpr_Call name args1
      return $ GrExpr_App f args2
```

Let us first consider the case where the arity of the function is zero. This means that the GRIN bindings really doesn't represent a function but a constant applicative form. Then there are three further possibilities:

- The arity is greater than the number of arguments passed. As we already know the arity is zero, this means that a negative number of arguments is being passed. This is obviously nonsense, so we had better terminate the compiler with an error message.
- The arity is equal to the number of arguments passed. That is, no arguments are being passed at all. In this case we can simply evaluate the global binding.
- The arity is smaller than the number of arguments passed. This means that although the global GRIN binding has arity zero, it will evaluate to a function. Therefore, we generate code to evaluate the binding, and call the resulting function with the passed arguments.

The other alternative is that the arity of the function being called is greater than zero. In that case, there are again three possibilities:

- The arity is greater than the number of arguments actually passed. In other words, the function is being partially applied. Therefore, we have to create a "P(artial) Application" *Node*.
- The arity is equal to the number of arguments passed. This is the one case where we can actually create code to do a perfectly normal function call, which is what we do.
- The arity is smaller than the number of arguments passed. We first generate code to call the function with as many arguments as it wants, like in the previous case. This function call will then evaluate to another function. We generate additional code to apply the resulting function to the remaining arguments.

compCallToWord is a variation on the same theme. When *compCallToNode* has a simple name it must evaluate it to create a node, while *compCallToWord* can simply return the name. On the other hand, function calls cannot be executed directly. Instead, nodes are created and stored representing such calls, and the pointer to such a node is returned. To help keep things manageable, we add a few helper functions to create code for storing such nodes.

```
storeApp :: HsName → [GrVal] → GrExpr
storeApp fun args
    = GrExpr_Store $ GrVal_Node (GrTag_App (HNm "_"))
      (GrVal_Var fun : args)
storeFun :: HsName → [GrVal] → GrExpr
storeFun fun args
    = GrExpr_Store (GrVal_Node (GrTag_Fun fun) args)
storePApp :: Int → HsName → [GrVal] → GrExpr
storePApp n fun args
    = GrExpr_Store (GrVal_Node (GrTag_PApp n fun) args)
```

One more thing to note about this is that storing an application of an unknown function is actually using an internal GRIN function called _ to do the actual call. The function to call is passed as its first argument. The code for *compCallToWord* follows for completeness.

```
compCallToWord name arity args = do
  args' ← mapM compileTermToWord args
  let nArgs = L.genericLength args
  GrVal_Var `liftM`
    case (arity, arity `compare` nArgs) of
      (0, GT) → error "impossible: length args < 0"
      (0, EQ) → return name
      (0, LT) → do
        f ← genFreshExpr $ storeFun name []
        genFreshExpr $ storeApp f args'
      (_, GT) → genFreshExpr $ storePApp (fromIntegral $ arity − nArgs) name args'
      (_, EQ) → genFreshExpr $ storeFun name args'
      (_, LT) → do
        let (args1, args2) = splitAt (fromIntegral arity) args'
```

```
f ← genFreshExpr $ storeFun name args1
genFreshExpr $ storeApp f args2
```

4.9 IO

IO in GRIN works by defining *main* to be of type *World* → *Int*. The *Int* is a GRIN built-in machine integer, and *World* is a magic token type that may or not be optimized away. In Agda, both these types are brought into scope by a **postulate** declaration and marked with a *BUILTIN* pragma.

Note that, as far as the compiler is concerned, *main* need not be referentially transparent. E.g., the use of the primitive function *primPutInt* : *Int* → *World* → *Int* easily breaks referential transparency. It is up to the designer of the prelude to encapsulate this functionality in a referentially transparent way, most likely by defining *World* to be an abstract and private type, and by only exposing a set of combinators that cannot be used in such a way to break referential transparency. For example, in the following example, *World* is defined to be **private** and therefore invisible in other modules, and *IO* is declared abstract, so the fact that *IO a* is really *World* → *a* is also invisible outside of this module. Then, two combinators *return* and _≫=_ are defined, making *IO* into a monad. This monad is essentially a reader monad over *World*, but makes sure of proper sequencing of side effects using the primitive *primSeq* function, which returns its second argument after evaluating its first argument to weak head normal form. Finally, *primPutInt* is defined to be **private**, and thus invisible outside of the module. The only way to use it is through the *putInt* function, which doesn't use the type *World* but *IO* and thus cannot be abused to break referential transparency. Finally, because built-ins can be declared only once and because *World* is private, there is no way for code outside of this module to mess with *World* itself.

```
postulate
  Int : Set
private
  primitive
    primSeq : {a b : Set} → a → b → b
infixr 0 _$!_
_$!_  : {a b : Set} → (a → b) → a → b
f $! x  = primSeq x (f x)
 {-# BUILTIN INTEGER Int #-}
abstract
  private
    postulate
      World : Set
  IO  : Set → Set
  IO a = World → a
   {-# BUILTIN WORLD World #-}
  return    : {a : Set} → a → IO a
  return a w = a
```

```
infixl 1 _>>=_
_>>=_       : {a b : Set} → IO a → (a → IO b) → IO b
f >>= g     = \w → (g $! f w) w
private
  primitive
    primPutInt : Int → World → Int
putInt : Int → IO Int
putInt = primPutInt
```

5 Optimizations

In this chapter we will look at a few optimizations originally introduced by Brady[10] for Epigram and the G-machine, and will investigate how they can be adapted and generalized to work in the framework of GRIN transformations.

5.1 Dead data

Take the following few Agda definitions, which are again the familiar vector and addition.[1]

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


_++_ : (α : Set) (m n : ℕ) → Vec α m → Vec α n → Vec α (m + n)
_++_ α .zero n [] ys                  = ys
_++_ α .(suc m) n (_::_ m x xs) ys = _::_ (m + n) x (_++_ α m n xs ys)
```

Note that there is an important redundancy in this piece of code: Every $\_::\_$ value contains as one of its fields the length of its tail, even though, whenever a function actually uses the length of a vector, it could already know the length: The type checker requires us to pass it as an argument. In this case, the actual length of the list is "pattern matched" on using a dot-pattern, which gets compiled to nothing at all.

Instead, we could also have written the definition of $\_+\!\!+\_$ as follows:

```
_++_ α zero n [] ys                  = ys
_++_ α (suc m) n (_::_ .m x xs) ys = _::_ (m + n) x (_++_ α m n xs ys)
```

[1] They are shown with all parameters explicit because, after all, optimizations are usually performed *after* desugaring

Now, the length field of the vector-constructor is no longer used — it has become a dot-pattern instead. As Brady proves in his thesis[10], this translation can in fact be mechanically performed for *every* function and thus, the length field is truly redundant.

However, once this translation has been performed, we can actually go further: By pattern matching on the natural number, it has already been uniquely determined what constructor the vector will be. In other words, the generated code need not test what tag the vector constructor has but can directly retrieve the correct values from memory. In addition, when all code using vectors is being translated in this way, no code will ever look at the tag, and we need never store it either.

This is based on the fact that in some cases, the value of a type index, which will always be passed as a parameter to satisfy the type checker, completely determines the value of a constructor tag or field. These optimizations are respectively called *forcing* and *detagging* by Brady. Brady also introduces an optimization called *collapsing* that, when both tag and all non-recursive fields of a constructor are removed, then removes the entire constructor. This optimization too can be implemented based on the analysis introduced in this chapter.

In the next sections, we will adapt and expand this concept for use in the whole-program optimizing GRIN compiler.

5.2 Dead data elimination

Brady's optimizations could rather directly be translated from working on a level intermediate between Epigram and G-code to working on Agda intermediate syntax. However, it is also possible to actually put the our new intermediate language GRIN to good use and split Brady's optimizations into two largely orthogonal parts, one working on Agda internal syntax (or even on the Agda source language) and one working on low level GRIN code:

- First, datatype definitions can be analyzed to find out how much of each constructor can be determined based on the value of type indices. Pattern matching code on these types can then be expanded to *also* pattern match on these indices, and use variables bound by indices instead of constructors as much as possible.

  This analysis and transformation works on a relatively high level, on Agda internal syntax. In fact, it can even be performed by the programmer: The second version of the $\_+\!\!+\_$ function seen above is exactly the result of this transformation on the first version.

  As this part of the optimization is already explained by Brady in detail, we will not treat it further here.

- The second part is a low level GRIN whole program optimization, and might be called a form of "dead data elimination": It analyses the entire program and removes constructor tags and fields (and in the full version also function parameters and local variables) wherever it can prove they will not be used. As a side effect, it will also find constructors being applied without ever being used and case alternatives that cannot ever

match. After the first transformation, and together with other GRIN transformations (notably *trivial case optimization*[8]), this optimization subsumes Brady's optimization: Not only does it do everything that *forcing*, *detagging* and *collapsing* do, it is also able to detect cases where a program does not in practice use certain fields or tags, even though this cannot be determined based on type information alone. Being a whole program optimization is crucial here.

In fact, together with aggressively inlining GRIN *eval* and *apply* calls, this dead data elimination also subsumes the eval/apply inlining transformation that is crucial in the design of the entire GRIN framework, although it is conceivable that it would not in practice be used as such due to efficiency considerations. This is not entirely surprising because *points-to analysis*, a static analysis that forms the basis for eval/apply inlining, has been generalized into a new analysis called *created-by analysis* for the new optimizations.

In this chapter we will focus on the analysis stage, as the actual removal of tags and fields based on the analysis is rather trivial.

There are a few reasons for choosing to split and generalize Brady's optimization in this way. One reason is that by splitting the optimization, the first part becomes simpler: The first part need not concern itself with actually removing any tags or fields, only with *adding* additional patterns. This may not seem like an important difference, but the removal of tags and fields at a stage in the compiler pipeline where the program representation is still typed is rather problematic: The same tags and fields may be removed in some uses of a datatype, but not in others, causing a single type to have multiple representations. Only in the later and more low-level stages of GRIN transformations does it become practical to extend GRIN to support nodes without tags. In addition, by splitting the optimization into two parts, they can be implemented and tested independently.

A more important reason is that the second part of our new optimization is a useful optimization in itself. Take for example the following program:

```
data List (α : Set) : Set where
  nil  : List α
  cons : α → List α → List α
length : {α : Set} → List α → ℕ
length nil          = zero
length (cons _ xs)  = suc (length xs)
filter : {α : Set} → (α → Bool) → List α → List α
filter p nil        = nil
filter p (cons x xs) = if p x
                       then cons x (filter p xs)
                       else filter p xs
count : {α : Set} → (α → Bool) → List α → ℕ
count p xs = length (filter p xs)
```

The program first defines *List* datatype, which is just like the previous *Vec*, but the length of a *List* is not reflected in its type. It also defines a function *length*

which obviously calculates the length of a *List*, a function *filter* which takes a predicate and a list and returns a new list containing only those elements of the input satisfying the predicate and, finally, a function *count* that combines *length* and *filter* to count the number of elements satisfying a predicate.

Of course this is only a fragment of a complete program, but let us assume that *count* is the only caller of *filter*.[2]

Now, when *count* is called, *filter* will recurse over the passed list and construct a new list. The only user of this new list is then *length*, which does not do anything at all with the elements of the newly created list.

The new optimization is capable of detecting that the elements of the list created by *filter* are in fact guaranteed not to be used by the code being compiled. It may then decide not to calculate and store these elements, and may shrink the size of each list cell.

Note that this may happen even if the elements of the list parameter to *filter* cannot be removed, which means that this optimization may introduce a non-uniform representation of datatypes.

A third reason for implementing the second part as a general dead tag/field elimination stage is that it makes dead parameter elimination much easier to implement: In a first-order language, dead parameter elimination is quite straightforward. One checks for unused function parameters and removes them from both function definition and callers. Unfortunately, dead parameter elimination is much more complicated in a language with lazy and higher-order functions and partial applications. However, because GRIN implements all of these using GRIN nodes, we need merely implement the traditional first-order dead parameter elimination, and dead field elimination will take care of the rest.

5.2.1 Overview

A short overview will now be given of the different parts of *dead data elimination*. In the next few sections we will explain each step in more detail.

- Created-by analysis

  The first step in performing dead data elimination is to construct a conservative approximation of which code may have created the value of each constructor field in existence during execution of the program, called created-by analysis. This is a straightforward generalization of the existing points-to analysis that forms an important part of the GRIN optimization framework.

- Dead variable detection

  We are interested in eliminating unused constructor fields, so for each case alternative, we must find out which constructor fields are bound to variables that subsequently remain unused.

- Connecting producers to consumers

  Based on the results of the created-by-analysis, we construct a bipartite graph consisting of all producers (expressions that create GRIN constructor nodes) and all consumers (case alternatives).

[2] In real world programs, where a function like *filter* is probably used in many places, this may still very well be true due to inlining (e.g. after performing the static argument transformation) of *filter* into its callers.

- Dead field analysis

  Then, we inspect the connected components of the created graph to determine which fields, tags and complete nodes are unused, and whether we may remove them or not.

- Removing the found dead data

  The last step, which has not yet been implemented, is to actually remove those elements from the code.

Except for the last step, a prototype of this optimization has been implemented for a simplified version of the GRIN abstract syntax that is used by EHC.

5.2.2 Created-by analysis

The first part of the analysis is a slightly modified form of points-to analysis. Points-to analysis is a static code analysis technique in which a set of possible values is determined for each variable, heap reference and function result in the program. This analysis is performed by EHC on a relatively high-level form of GRIN, before eval/apply inlining, and the results of the analysis are kept up to date during the following transformations into optimized and low-level GRIN. Due to the fact that for our purposes we only need the points-to analysis information in a stage of the compiler pipeline after eval/apply inlining, the points-to analysis in the prototype has been implemented directly on a simplified form of the post eval/apply inlining low-level GRIN. In the prototype, this analysis has been extended by remembering, for each of the sets of possible node values, which expressions may have created the node. We call this extended form of points-to analysis a *created-by analysis*. Although the EHC points-to analysis is significantly more complicated than the prototype (due to having to work with the full, high-level form of GRIN), this extension is expected to be straightforward to implement in EHC too.

5.2.3 Dead variable detection

When the heap points-to analysis has been completed, the next step is dead variable detection or, more specifically, dead case alternative bound variable detection. Currently, this part consists of walking through the program and for each case alternative calculating the difference between the set of variables bound by the alternative and the set of variable occurring in the body of the alternative.

In addition, we represent the tag of the case alternative as the zeroth field of its constructor. We consider a tag to be dead when the case expression contains only a single alternative.

In a production implementation this analysis should be more intelligent as there may be interactions between, for example, dead fields and dead parameters: The current system does not detect dead parameters and will fail to detect a variable that is only used as a dead argument as such. To make matters more complicated, parameters may be dead or alive depending on the liveness of other constructor fields, and there may even be cycles in such deadness chains.

To deal with such cycles, the current system would not merely need a smarter dead variable detection, but would have to be set up to generate equations of

the form "this field is dead when a certain set of variables is dead". A separate equation solver could then calculate the final set of dead variables, parameters, fields and tags.

5.2.4 Connecting producers to consumers

The next step, which in the current implementation is intertwined with the previous step, consists again of walking through the entire program and, for each *Case* expression, looking up in the environment what expressions may have created the node contained in the variable that is being case analysed. Based on this information, a bipartite graph is created, which consists of the following:

- The first set of vertices represents the producers: all expressions that create nodes. Each vertex is a pair of (*ExprID*, *Tag*).
- The second set of vertices represent the consumers: all case alternatives. Each vertex is a pair of (*ExprID*, *Tag*).
- Each edge connects a node-producing expression to a consuming case-alternative that may get a node created by the producer as its node.

5.2.5 Dead field analysis

We calculate the connected components of the created bipartite graph. Each resulting vertex set falls into one of three classes:

- The set consists of a single producer. This producer expression thus has no consumers. At this point, the entire node-producing expression may be removed, and its result replaced by a dummy value.

  Thus, a *DummyNode* constructor has to be added to the *GrVal* type. The original produced *GrVal* value can be replaced by the new *GrVal_DummyNode*.

  This value is guaranteed not to ever be analyzed by a case expression. However, it may still be returned using *unit* (from functions, for example), stored and re-fetched, so it may not be entirely eliminated from the program in all cases. In later GRIN transformations, and when the final GRIN code is transformed into e.g. C, *DummyNode* should be treated as a singleton constructor.

  The set requires no further analysis or action.

- The set consists of a single consumer. This consuming case alternatives has no corresponding node-creating expression, and will thus never be executed: It is a dead case alternative that can safely be removed. When the case alternative has been removed, the set requires no further analysis or action.

  This removal overlaps with the normal *eval/apply inlining* step.

- The set consists of at least one producer and at least one consumer. In this case, we group each producer with the set of all its consumers. (Consumers may thus occur in more than one set.) For each pair of producer and consumer-set, we now calculate which fields (including the tag, its

zeroth field) are dead in *all* consumers. In the current implementation, we consider a field to be *dead* if and only if its binder does not occur in the body of the case alternative. A more advanced implementation would be based on a system of constraints about dead fields, variables and parameters combined with a constraint solver.

Fields that are found to be dead in all consumers are marked as *dummy*. Fields that are marked as dummy in all pairs are marked as *removable*. If a field is neither *dummy* nor *removable*, it is marked *active*.

5.2.6 Removing the found dead data

Finally, the result of the previous analysis can be used to actually remove dead data. Implementing this rather simple step seemed not very useful in the current prototype implementation. We will, however, discuss what needs to be done and what further extensions to GRIN have to be made in order to support this optimization.

As we have already dealt with both kinds of singleton vertex sets, we will only treat the sets consisting of at least one producer and one consumer here.

Each field (including the tag, the "zeroeth field") in the set of consumers and producers will now be labeled as either *active*, *dummy* or *removable*. We will now, both for node producing expressions and for node consuming case alternatives, determine what needs to be done for each of these three cases.

Active fields

When a field is *active*, the field will actually be used, and thus nothing will be done.

Dummy fields

When a field has been marked as *dummy*, this means its value will never be used but cannot be entirely omitted because its consumers expect the field to be there. This is because at least one of these consumers will share a producer with another consumer that does actually use the corresponding field.

After extending the *GrVal* type with a *DummyWord* constructor, we may now use *DummyWord* as the value of the field that has been marked as *dummy* in the node producing code. Note that in many cases, the original word value stored in the field will no longer be used, thus causing new dead fields, variables etcetera. When finally compiling low-level GRIN into e.g. C, a *DummyWord* constructor can either be compiled into the value 0 (whether it be a null-pointer or the integer 0) or simply not be written at all: Fields that are guaranteed not to be read may as well be left uninitialized.

In the dummy field consuming code, nothing need be done: The "split fetch" transformation will later turn the pattern match into individual instructions for retrieving the value of each field. As the dummy field is obviously unused, its fetch instruction will be removed by a subsequent dead expression removal pass.

For cases where a tag instead of a normal field has been marked as *dummy*, we also have to extend *GrTag* with a *DummyTag* constructor. In node producing code, this can be used to replace the old value analogously to the way *DummyWord* is used for fields. A dummy tag in node consuming code can

only occur when the consuming code consists of a case expression with a single alternative. The current "split fetch" transformation already omits the fetch instruction for such a known tag value, so one may either leave the *GrTag* in the consuming code as it is, or replace it too with a *DummyTag* constructor, which should then be treated by "split fetch" as just another known tag, depending on what is more convenient for later analysis stages.

Removable fields

When a field (or tag) has been marked as *removable*, not only is the value of the field irrelevant, even the space it takes can be reclaimed. This means that in both all producers and all consumers of the vertex set, we can completely remove the field, thereby "shifting" all later fields to the left to take its place. If a tag is marked as *removable* things are somewhat more complicated: The most straightforward approach is to add a *NoTag* constructor to the *GrTag* datatype. Then, the "split fetch" stage should be taught that when the tag is a *NoTag*, the fields start at position 0 instead of 1, and the final transformation from low-level GRIN to e.g. C should do the same to the node producing code.

In addition, because GRIN nodes contain some meta-information about their arity and the maximum arity of all constructors, it may be necessary to "clone" a datatype and pretend that e.g. a pair type from which the first element has been removed and the normal pair were actually defined as two separate datatypes in Agda.

6 Discussion

We will begin this chapter with a discussion of how the presented new Agda back end compares to other relevant work. This thesis' contributions will then be discussed, and several possible directions for future work will be listed.

6.1 Related work

There are a number of other projects that are related to this thesis. The most comparable one is Edwin Brady's PhD thesis, which we will therefore discuss first.

6.1.1 Epigram to G-machine

In his PhD thesis, Edwin Brady develops a translation from the dependently typed language Epigram to the G-machine [10, 13, 11]. Epigram is a close cousin to Agda, and the G-machine is an older relative of GRIN.

Before explaining more about the differences between Brady's work and this thesis, a small summary will be given of the differences between Agda and Epigram, and between GRIN and the G-machine. Finally, the differences between both compilers will be discussed.

Agda versus Epigram

Epigram[4] is a dependently typed programming language designed by Conor McBride and James McKinna. The design of Agda is greatly inspired by Epigram, so there are many similarities between both languages.

The main non-syntactic language difference between Agda and Epigram is that Epigram generates an eliminator for every inductive family, while Agda is based on dependent pattern matching. Epigram provides support for using eliminators in such a way that it looks like pattern matching — essentially a

kind of generalized views[16]. Agda, on the other hand, generalizes the pattern matching in function definitions in Haskell to dependent pattern matching. These different approaches to discriminating constructors of datatypes lead to a few important differences:

- An Agda function definition may pattern match on nested patterns or, a variation on the same theme, on multiple function arguments. The order in which these different patterns are discriminated is left unspecified (although there are obviously some dependencies). This means that an Agda compiler has to compile the pattern matching function definitions into a function with an explicit case tree, something that in otherwise equivalent Epigram code has already been done by the programmer.

- When compiling Epigram code, the programmer-written code does not pattern match, it only contains calls to compiler generated eliminator functions. Of course, these eliminator functions will still be using pattern matching when compiled down to GRIN (or the G-machine), but when compiling Epigram code, all the pattern matching will be localized to one function for each datatype.[1]

  This means that changing the representation of a datatype or the implementation of pattern matching, most likely for efficiency reasons, needs only to generate different eliminator functions. This is likely to be easier than replacing many pattern matches scattered over the program.

G-machine versus GRIN

Brady uses the G-machine as his back-end language instead of GRIN. The most important differences are:

- The G-machine uses a (multiple) stack-based model. All modern mainstream CPU architectures use a register based model, as does GRIN. A stack-based model makes both optimization transformations and the final translation into efficient machine code unnecessarily difficult on modern CPUs.

- GRIN is explicitly designed for whole-program analysis. As a consequence, GRIN compilers can make use of a number of optimization techniques that are not available when using a system intended to support separate compilation.

- The G-machine has the concept of *closures* and *higher order functions* built-in, whereas in GRIN these are built on top of lower level features. This in turn means that GRIN optimizations working on normal data values will also work on closures.

- The G-machine is designed and described as a *virtual machine*, while GRIN is an *intermediate notation*. Although this is no fundamental technical difference, it does represent a difference in attitude: Whereas the focus with the G-machine is on executing code, the design of GRIN is

[1] Of course, the Epigram programmer may write new "eliminators" which can be used just like the compiler generated eliminators, which is one of its great attractions. However, these new eliminators are ultimately defined in terms of the original compiler generated eliminators.

focused on transforming GRIN code into other GRIN code and, in particular, optimization.

Structure of the compilers

Now that we have discussed the most important differences between both the source languages and the intermediate languages, we will discuss the main differences in the structure of both compilers.

Brady's (not actually implemented) compiler is, more than the new GRIN back end for Agda, established on a pipeline structure. This pipeline is based on transformations between a number of intermediate languages. Note that a number of earlier pipeline stages such as parsing and type checking are not relevant here, and therefore omitted. The first intermediate language used by Brady is called *TT*. TT is a dependently typed λ-calculus with definitions, inductive families and equality. In the next stage, TT is transformed into *ExTT'*, a variation on TT permitting annotations on terms that can safely be deleted. An ExTT' to ExTT' optimization stage follows. Then, the resulting ExTT' code is transformed into *RunTT'*, which is a simplified version of ExTT' consisting of only super combinator definitions. Another optimization stage follows (this time from RunTT' to RunTT') and finally G-code (code for the G-machine) is generated. Now, the differences between the structure of the two compilers will be discussed in detail.

Brady starts by type checking Epigram programs and elaborating them into the core language *TT*. TT is somewhat comparable to Agda's internal syntax, but there are important differences: Agda's internal syntax is purely an implementation language; its denotational and operational semantics have not formally been defined. This is of course not as nice a starting point for a compiler as a formally defined λ-calculus. Just as with Epigram itself, TT supports discriminating the constructors of inductive families by providing eliminator rules. Instead, Agda's internal syntax supports multiple pattern matching function clauses for the same purpose. However, TT's elimination rules (also called ι-schemes) are generated by the compiler in the form of pattern matching function clauses too.

The next step in the compilation of an Epigram program is that the TT code is translated into ExTT'[2] Patterns in a ι-scheme definition in ExTT' may be annotated to specify that they are not necessary to compute its result. In a later stage, these patterns may then be deleted. There are two classes of patterns that can be annotated in this way: Repeated variables and arguments that are not in constructor form. In Agda, these can only be written as dot-patterns in the source code. Having no computational content, dot-patterns are then turned into fresh variables during the translation from Agda's internal syntax to GRIN.

Then, ExTT' is compiled into RunTT'. This is done using Johnsson's supercombinator lifting algorithm[11]. The elimination rules are translated using a variation on Augustsson's pattern matching compiler.[7] Note that the full generality of Augustsson's pattern matching compiler is not needed when compiling Epigram: It is only used to translate elimination rules, and thus only for simple non-nested pattern matches. In addition, arguments that are annotated

[2]We are only discussing the optimizing approach here. Brady also describes a naïve approach that is beyond the scope of this thesis.

for deletion in ExTT' are removed from the pattern matching code. In the Agda case, these arguments would be fresh (unused) GRIN variables, generated from dot-patterns in Agda. These fresh variables may then be analyzed by the newly introduced dead data elimination and possibly removed.

Then, Brady describes two optimization stages. The first transforms ExTT' using elimination rules into ExTT' making direct recursive calls. As Agda is based on pattern matching all the way from the beginning down to GRIN, no corresponding stage exists in the Agda compiler. However, even if Agda was changed to use elimination rules in the same way as Epigram, such a stage would still not be necessary, as the standard GRIN inlining that is performed anyway can be relied upon to inline functions that do nothing but pattern match.

The other optimization stage works on RunTT', and performs some standard optimizations like inlining and dead parameter elimination that the Agda compiler can also leave to GRIN transformations. This stage also drops impossible cases, something that the Agda GRIN back-end does as part of its pattern match compilation.

Neither ExTT' nor RunTT' have a real counterpart in the translation from Agda internal syntax to GRIN. This is partly due to the fact that repeated arguments in Agda are already marked as dot-patterns by the programmer (or by the implicit-parameter resolution machinery), and to some extent due to the fact that the Agda-to-GRIN compilation is not setup as a pipeline. The reason it does not have a pipeline-architecture is partly an historical accident, and partly a consequence of the fact that we choose to perform all optimizations after compilation to GRIN.

Finally, RunTT' is translated into G-machine code. Brady introduces a few minor modifications to the G-machine to deal with compiling a dependently typed language and to take advantage of the earlier optimizations in ExTT'.

Differences in optimization techniques

Brady invents a number of optimizations and describes them in terms of his own intermediate languages, ExTT' and RunTT'. These optimizations are performed before the generation of G-machine code. Thanks to the fact that we use GRIN as our back-end language, we can achieve the same results by one transformation within Agda's internal syntax. This is accomplished by implementing a number of new GRIN-to-GRIN transformations, in addition to one source to source (or in practice, Agda internal syntax to Agda internal syntax) transformation. These transformations are then also somewhat more general than Brady's optimizations, and may also allow other — possibly non-dependently typed — languages to reap the benefits of these optimizations.

It should be noted that Brady is interested both in evaluation at compile-time (recall that type checking in a dependently typed language may cause evaluation of terms) and when running the compiled program. However, when evaluating while type checking, the program may still turn out to be type incorrect. As some of Brady's optimizations crucially depend on properties such as well-typedness, this means that these optimizations may not be used when evaluating at compile-time.[3] As Agda's GRIN back-end starts working with internal syntax, which has already been type-checked, it obviously only concerns

[3]Well-typedness is not the only such property. The absence of free variables is another.

itself with run-time evaluation. Therefore, it need not worry about when which optimizations are valid.

6.1.2 Agda to Haskell to STG

The first back end to the Agda compiler, MAlonzo[4], generates Haskell (albeit with some GHC extensions) — the language on which Agda is based, and the language in which the Agda compiler itself is written.

Using Haskell as a target language for an Agda compiler has one major advantage: Simplicity. As Agda is based on Haskell, much of Agda's syntactic structures can be trivially translated to Haskell. For example, whereas in GRIN every application needs to be given a name explicitly, Haskell supports the same kind of nested expressions as Agda. Pattern matching too is supported in almost the same way in Haskell and Agda (internal syntax). In fact, when translating Agda functions and datatypes to Haskell, the main complication is often the different name spaces and rules of what constitutes an identifier in both languages.

Such a direct translation also immediately leads to a large disadvantage: Haskell is not a dependently typed language, and therefore unable to express many of the types used in Agda programs. This is solved with a liberal sprinkling of the GHC primitive function $unsafeCoerce :: a \rightarrow b$. Although this function is only used in a technically safe way due to the type correctness of dependently typed original program, the use of this function is discouraged, highly implementation specific, and its use causes GHC to be rather conservative in applying some optimizations to the program.

This is actually exemplary for the use of Haskell as a target language: It is a programming language designed to be used by programmers, not by compilers. As a result, whenever an Agda compiler wants to make decisions about low level behaviour of the compiled program (such as decisions about sharing or strictness), the compiler has to be extremely careful to generate Haskell code in such a way that GHC will not interfere and decide otherwise.

In addition, Haskell is not a total language: Functions may not terminate, and pattern matching may fail. This in turn means that GHC cannot make many of the assumptions that a native Agda compiler could in aggressively optimizing the program, leading to larger and slower code.

Using Haskell as a target does have other advantages: The run-time system becomes very easy to implement, as one may use Haskell to code all primitive and supporting functions, whereas they all had to be implemented directly in GRIN for this thesis. In addition, using existing Haskell libraries from within Agda is obviously much easier when your target language is Haskell.

The other major difference between the GHC/Haskell back end and the new GRIN back end is that GHC uses the older STG as its back end language, which has not been designed as a language suitable for effective optimizations for modern micro architectures, and which is not designed for whole program optimization. As the newly introduced optimizations rely on the whole program optimization capabilities of GRIN, this generalization of Brady's optimizations would be impossible when using STG (by way of GHC) as the back end language.

[4] We ignore the two earlier compiler back ends for Agda, Agate and Alonzo, because they also emitted Haskell, and because they have been completely replaced by MAlonzo.

6.1.3 Idris to Epic

Edwin Brady, after completing his PhD thesis on compiling Epigram to the G-machine, has gone on to develop the dependently typed programming language Idris[9, 5], and its back end language Epic[3]. Although not much information is available on this ongoing work, Brady's earlier *forcing*, *detagging* and *collapsing* optimizations have been implemented for the system and do indeed yield large performance gains.[1] Idris is quite similar to both Epigram and Agda, while Epic seems to be much closer to STG than to GRIN. In particular, Epic is not designed to be used for whole program optimizing transformations, and laziness is built into the language, and not built on top of other primitives.

6.2 Contributions

This thesis provides the following contributions:

- The feasability of developing a modern (GRIN-based) back end for Agda has been (constructively) proven. Almost the full language is supported, including dependent datatypes, records and functions, compilation of pattern matching, modules, primitive types and functions and input/output.

  No serious dependent type specific difficulties in compilation were encountered, although several possibilities for new optimizations compared to the current uses of GRIN did present themselves.

  The only fundamental difference compared to compiling a non-dependently typed language is the handling of types: Whereas Haskell compilers such as GHC are able to erase all types in a special *type erasure* transformation at compile-time, we collapse all types (and types of types, etc) into the value level, and rely on optimization techniques such as dead parameter elimination, extended with our new dead data elimination design to remove all unnecessary type information from the program.

- Three optimizations developed for a predecessor of Agda have been translated to the combination of Agda and GRIN. Instead of directly modifying the existing optimization to work on Agda instead of Epigram, it has been split into two independent parts: An Agda source to source translation and a generic GRIN optimization. In addition, a few minor extensions had to be proposed to the GRIN language for use by the new optimization.

  The resulting new optimization strictly subsumes the original optimization. Whereas the original optimization removes fields and tags if it can prove that they need never be used, the new optimization uses whole program information to check whether fields happen to be used in the current program. The set of fields (and tags) that are not used in the current program of course fully includes the set of fields that are not used in any possible program.

  Together with an aggressive application of the standard inlining transformation, the new optimization additionally subsumes the existing *eval/apply inlining*. This optimization, based upon points-to analysis that has been extended to created-by analysis, lies at the core of GRIN's claim for being well suited for compilation to modern micro architectures.

6.3 Future work

There are a number of interesting topics for further research based upon the work in this thesis:

- The optimizations introduced in chapter 5 currently exist in part as a prototype (the different analyses stages), while other parts have merely been sketched (the actual removal, and the equation solver for dead variable, parameter, tag and field detection). Implementing this optimization in a concrete GRIN back end such as EHC should be relatively straightforward, and would yield interesting data on the advantages in performance that this optimization might yield. A relevant complication might be the existence of non-termination, incomplete pattern matching and side effects when using the new optimization for EHC's currently only front end language — Haskell.

- The current connection between Agda and GRIN supports most but not all of Agda the language: Some parts that are not or partially supported are sized datatypes, primitive types such as strings and floating point numbers and many built-in functions. In addition, both Agda and EHC are research projects under active development, so bit rot will unfortunately remain a persistent problem, unless the Agda-GRIN connection is integrated into Agda and, additionally, EHC's back end library stabilizes.

- Haskell is a non-strict language, and EHC uses the usual evaluation strategy for compiled programs: call-by-need. Explicit strictness annotations and strictness analysis can then be used to introduce occasional call-by-value applications for performance reasons. Due to the fact that every function in Agda is total, the used evaluation strategy matters *only* when considering performance. An interesting avenue of research would be combining both call-by-need and call-by-value within the same Agda program, using either explicit annotations or some sort of strictness utility analysis to decide when to use which kind of evaluation.

- This thesis has focused entirely on generating and optimizing code for execution at run-time. However, in a dependently typed language such as Agda, substantial amounts of code may be executed during type checking. Generating GRIN code for execution during type checking would be an interesting topic of research. Considering the fact that GRIN is a whole program compiler, while only part of the program is available at each stage of type checking, this would be a rather ambitious endeavour. In addition, during type checking, some of the properties (such as type correctness) that make advanced optimization possible are not guaranteed yet, complicating efforts at optimization.

A

Infrastructure for generating GRIN bindings

The main components of the infrastructure to facilitate compiling Agda internal syntax definitions to GRIN bindings are two monad transformers: *GrinT* and *ExprT*. The former is primarily intended to aid in the generation of GRIN bindings, whereas the latter is designed for generating GRIN expressions within bindings. Most of the actual code generation uses both of these monad transformers.

In this appendix, *GrinT* will be introduced. This monad transformer facilitates the generation of GRIN bindings by supporting three different kinds of operations: Asking for pre-calculated information about the GRIN internal syntax code and command line flags, generating fresh names and emitting GRIN bindings. All actual generation of GRIN code is done from within this monad.

```
newtype GrinT m a
  = GrinT (WriterT (Endo [GrBind])
           (ReaderT GrinTInfo (StateT Integer m)) a)
```

The *GrinT* monad transformer is the composition of three standard monad transformers:

- *StateT Integer* The state monad part contains an *Integer*, which is used exclusively for the generation of unique names for GRIN identifiers.

  Based on this part of the *GrinT* monad transformer is the function

  $$freshGrinName :: Monad\ m \Rightarrow String \rightarrow GrinT\ m\ HsName$$

  Note that this function only takes care of generating *unique names*, not of the mapping from Agda to GRIN names.

- *ReaderT GrinTInfo* The reader monad part contains a *GrinTInfo*, which is a collection of the look-up tables generated in the first phase of compiling the Agda internal syntax.

Functions based on this part of the definition are, among others

$$\begin{array}{l} \mathit{getFunction} :: \mathit{Monad}\ m \Rightarrow \mathit{QName} \rightarrow \mathit{GrinT}\ m\ \mathit{Definition} \\ \mathit{getArity} :: \mathit{Monad}\ m \Rightarrow \mathit{QName} \rightarrow \mathit{GrinT}\ m\ \mathit{Nat} \\ \mathit{hasGrinFlag} :: \mathit{Monad}\ m \Rightarrow \mathit{String} \rightarrow \mathit{GrinT}\ m\ \mathit{Bool} \end{array}$$

- *WriterT* [*GrBind*] The writer monad part is meant for actually generating code or, to be more precise, for generating GRIN bindings.

  The function exposing the functionality offered by this part of the *GrinT* monad transformer is

  $$\begin{array}{l} \mathit{genBind} :: \mathit{Monad}\ m \\ \qquad \Rightarrow \mathit{HsName} \rightarrow [\mathit{HsName}] \rightarrow \mathit{GrExpr} \rightarrow \mathit{GrinT}\ m\ () \end{array}$$

  Given a name, arguments and a GRIN expression, this function generates a GRIN binding.

Finally, there is a function that, when given a *GrinTInfo* and a *GrinT* computation, runs the computation and returns the list of all generated *GRIN* bindings.

$$\mathit{execGrinT} :: \mathit{Monad}\ m \Rightarrow \mathit{GrinTInfo} \rightarrow \mathit{GrinT}\ m\ () \rightarrow m\ [\mathit{GrBind}]$$

B

Infrastructure for generating GRIN expressions

The other monad transformer is *ExprT*. This transformer is built to support in the generation of GRIN expressions, as opposed to *GrinT* which is for generating entire GRIN bindings. The definition of the *ExprT* monad transformer is as follows

```
newtype ExprT m a
  = ExprT (WriterT (Endo GrExpr) (ReaderT [HsName] m) a)
```

As can be seen from the definition, it is composed of two standard monad transformers

- *ReaderT* [*HsName*] Agda's internal syntax is based on de Bruijn indices, whereas GRIN uses explicit variables. This reader monad transformer provides the mapping between those. In particular, it provides access to a list of GRIN names, of which the GRIN name corresponding to de Bruin index n can be found at the index n in the list:

  ```
  getVarName :: Monad m ⇒ Nat → ExprT m HsName
  ```

- *WriterT* (*Endo GrExpr*) Just like in *GrinT*, a *WriterT* is used to allow for the actual generation of code. However, where *GrinT* could use a simple list of bindings, with one binding generated after another, the situation for *ExprT* is more complex. Recall that the definition of *GrExpr* looks as follows

  ```
  data GrExpr
    = App HsName [GrVal]
    | Case GrVal [GrAlt]
    — lots of other constructors...
    | Seq GrExpr GrPatLam GrExpr
  ```

In particular, note that *GrExpr* itself is a recursive datatype: A GRIN function binding contains only one expression as its body, but this expression will in all but the most simple cases have been nested using the *Seq* constructor. However, all a writer monad requires is something that is a *Monoid*. Now, *Seq expr pat* has type *GrExpr* → *GrExpr*, and thus *Endo* (*Seq expr pat*) (of type *Endo GrExpr*) is a *Monoid*, and so we can have a *WriterT* (*Endo GrExpr*) which builds a composition of *GrExpr* → *GrExpr* functions that will prepend nested expressions using *Seq* to their final argument. The functions using this writer monad are

```
genExpr      :: Monad m ⇒ GrPatLam → GrExpr → ExprT m ()
genFreshExpr :: Monad m ⇒ GrExpr → ExprT (GrinT m) HsName
```

The first directly uses the underlying *WriterT* monad, while the second first uses *GrinT*s *freshGrinName* to generate a fresh variable to create a *GrPatLam* from, and then calls *genExpr* on the result.

The last *ExprT* function explained here is analogous to *execGrinT*:

```
execExprT :: Monad m ⇒ [HsName] → ExprT m GrExpr → m GrExpr
```

execExprT, when given a "de Bruijn to GRIN name" mapping and an *ExprT* computation returning an expression, will run the computation and return the nested expression consisting of the expression returned by the computation preceded by all the expressions generated during this computation through calls to one of the *genExpr* variants.

To clarify the use of *genFreshExpr* and *execExprT* we will now give an example

```
foo :: Monad m ⇒ ExprT (GrinT m) (HsName, HsName)
foo = do
  v1 ← genFreshExpr (Call "-" [Var "x", Var "y"])
  v2 ← genFreshExpr (Call "abs" [Var "z"])
  return (v1, v2)
```

The *ExprT* computation *foo* generates GRIN code to calculate the difference between the (integer) variables and x and y and the absolute value of z, and returns the names of the GRIN nodes in which the results have been stored. Now, the Haskell expression *execExprT* [] *foo*[1] has type *Monad m* ⇒ *GrinT m GrExpr* and, when executed, will yield the following GRIN expression

```
Seq  (Call "-"      [Var "x", Var "y"]) (PatLamVar "v_0")
(Seq (Call "abs"    [Var "v_0"        ]) (PatLamVar "v_1")
     (Call "square" [Var "v_1"        ]))
```

[1] Note that it has been given the empty mapping from de Bruijn indices to GRIN variables, which is not a problem as *foo* does not use any Agda internal syntax variables.

Bibliography

[1] Blogpost about idris forcing, detagging and collapsing. http://www-fp.cs.st-and.ac.uk/wordpress/?p=140.

[2] Some simply typed λ-calculus introductions. http://en.wikipedia.org/wiki/Simply_typed_lambda_calculus http://www.lfcs.inf.ed.ac.uk/reports/98/ECS-LFCS-98-381/ Barendregt?

[3] Website of epic - a supercombinator compiler. http://www.cs.st-andrews.ac.uk/ eb/epic.php.

[4] Website of epigram. http://www.e-pig.org.

[5] Website of idris. http://www.cs.st-andrews.ac.uk/ eb/Idris/.

[6] Website of the utrecht haskell compiler. http://www.cs.uu.nl/wiki/bin/view/UHC/WebHome.

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

[8] U. Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, April 1999.

[9] E. Brady. Idris, a language with dependent types . extended abstract.

[10] E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

[11] T. Johnsson. Efficient compilation of lazy evaluation. In *SIGPLAN Notices*, pages 58–69, 1984.

[12] S. L. P. Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine - version 2.5. *Journal of Functional Programming*, 2:127–202, 1992.

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

[14] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory — An Introduction*. Oxford University Press, 1990. http://www.cs.chalmers.se/Cs/Research/Logic/book/.

[15] U. 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.

[16] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction, 1986.