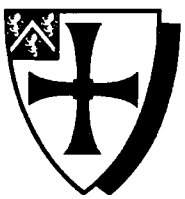
Practical implementation of a dependently typed functional programming language

Brady, Edwin .

Practical Implementation of a Dependently Typed Functional Programming Language

by

Edwin C. Brady



Submitted in conformity with the requirements
for the degree of PhD
Department of Computer Science
University of Durham

Abstract

Practical Implementation of a Dependently Typed Functional Programming Language

Edwin C. Brady

Types express a program's meaning, and checking types ensures that a program has the intended meaning. In a dependently typed programming language types are predicated on values, leading to the possibility of expressing invariants of a program's behaviour in its type. Dependent types allow us to give more detailed meanings to programs, and hence be more confident of their correctness.

This thesis considers the practical implementation of a dependently typed programming language, using the EPIGRAM notation defined by McBride and McKinna. EPIGRAM is a high level notation for dependently typed functional programming elaborating to a core type theory based on Luo's UTT, using Dybjer's inductive families and elimination rules to implement pattern matching. This gives us a rich framework for reasoning about programs. However, a naïve implementation introduces several run-time overheads since the type system blurs the distinction between types and values; these overheads include the duplication of values, and the storage of redundant information and explicit proofs.

A practical implementation of any programming language should be as efficient as possible; in this thesis we see how the apparent efficiency problems of dependently typed programming can be overcome and that in many cases the richer type information allows us to apply optimisations which are not directly available in traditional languages. I introduce three storage optimisations on inductive families; forcing, detagging and collapsing. I further introduce a compilation scheme from the core type theory to G-machine code, including a pattern matching compiler for elimination rules and a compilation scheme for efficient run-time implementation of Peano's natural numbers. We also see some low level optimisations for removal of identity functions, unused arguments and impossible case branches. As a result, we see that a dependent type theory is an effective base on which to build a feasible programming language.

Acknowledgements

My thanks to my supervisors, James McKinna and Zhaohui Luo. James made the original suggestion for a thesis topic and has constantly provided advice and feedback. Zhaohui's enthusiasm and knowledge has been an inspiration, and I would like to thank both for their feedback on previous drafts of this thesis.

I would also like to thank the other members of the Computer Assisted Reasoning Group, in particular Conor McBride. Conor taught me a lot of what I know about type theory and introduced several implementation techniques to me. I also owe him thanks for his useful comments on an earlier draft of this thesis, and a collection of LaTeX macros which made writing it marginally less painful!

My office mates, successively Yong Luo, Paul Townend, Simon Pears, David Johnstone, Robert Kiessling, and Chris Lindop helped to provide a friendly environment in which to work, and I thank them. Thank you also to the Computing Society, the Go Club and Ustinov College Cricket Club for providing distractions when I needed them.

Finally, I would like to thank my family for their love and encouragement, and Jenny for her support and limitless patience over the last few months.

Contents

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Durham or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

Large parts of the work presented in the first part of Chapter 4 have previously appeared in [BMM04], co-authored with Conor McBride and James McKinna.

Copyright Notice

Chapter 1

Introduction

Computer programs are ubiquitous. As we rely on computers more and more in all aspects of daily life, it becomes more important to minimise errors in computer software; it is particularly important where privacy or safety is concerned. An error-free computer program is, however, rare — a programmer attempts to minimise the number of errors by using a combination of techniques including formal specification, careful design, correctness proofs and extensive testing.

Part of the difficulty in writing a correct computer program lies in the problem of converting the design in the programmer's head (which one would hope is well understood) to a program which a computer can execute. Over the last fifty years increasingly powerful programming languages have been developed to allow the programmer to express a design as a program in more familiar terms; more modern languages feature more powerful type systems, which allow the programmer to specify more precisely the intended semantics of the program and provide more powerful paradigms for modelling.

Dependent type systems [ML71, Luo94] allow types to be predicated on values and have traditionally been applied to reasoning and program verification. More recent research, however [Aug98, Xi98, McB00a, MM04b], has led to the use of dependent types in programming itself. The principle is that the richer type system allows a more precise type to be given to a program so that more errors can be detected at compile-time which would previously have remained undetected until run-time, and even then perhaps only in unusual circumstances. Dependent types also allow us to give types to more programs than traditional simple type systems.

The use of dependent types in programming leads to several implementation difficulties on the one hand, and optimisation opportunities on the other hand. One difficulty is that the distinction between types and values is blurred so it is less clear how to erase types at run-time. Types can also express relationships between values — such relationships may mean one value can be computed from another, so we need not store both. With rich type information, we know more about the possible inputs and outputs of a program and ought

to be able to use this information to optimise a program. In this thesis, I begin to explore techniques for removing the run-time overheads of dependent types and gaining run-time benefits from our richer type information.

1.1 Types in Programming

Following Mitchell [Mit03], we identify three main purposes which types serve in modern programming languages. These are:

1. **Naming and organising concepts.** The type of a function or data structure reflects the way that structure is used in a program. In this way, types provide documentation to programmers and aid maintainability.

2. **Ensuring that the machine interprets data consistently.** Types ensure that operations are applied to objects of the correct form. For example, typechecking prevents an operation which expects an integer being given a floating point number, which would then be interpreted incorrectly. An object will always be treated in a way consistent with its representation.

3. **Providing information to the compiler about data.** A compiler uses the type of an object to decide how to lay out that object in memory. Two objects of the same type will always be represented in the same way.

These purposes assist the programmer, the machine and the compiler respectively. The importance of data types in programming languages has been acknowledged throughout the history of programming. Originally, languages attached types to values out of necessity — different types are laid out in memory in different ways, so the programmer was required to declare the purpose of a variable. As such, the first of the major computer languages, FORTRAN [IBM54], included primitive types for describing integers and real numbers and basic support for data structures with arrays.

Modern functional programming languages such as Haskell [P+02] and the ML family [MTHM97, Ler02] take this idea much further, allowing user defined data structures and function types. Primitive types, which effectively give an interpretation to bit patterns (for the benefit of the machine), are combined into compound types which give a higher level understanding of data (for the benefit of the programmer).

The development of more advanced type systems has led to two further purposes for types; in modern languages types are not only present because they are a necessity for the compiler, but because they provide documentation for the programmer and consistency checking for programs — giving a type to a function effectively gives a specification to that function, which serves as documentation for the programmer, and which the compiler verifies by typechecking.

1.2 Dependent Types in Programming

The characteristic feature of a dependent type system, as opposed to the "simple" type systems of Haskell and ML, is that types can be predicated on values. This allows the programmer to give a more precise type to a value, with the effect that more errors can be caught at compile-time, rather than manifesting themselves only when the right circumstances arise at run-time. As an introductory example, we shall consider the following simple Haskell function which appends two lists:

```
append :: [a] -> [a] -> [a]
append [] xs = xs
append (x:xs) ys = x:(append xs ys)
```

We can compute the length of a list as follows:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1+(length xs)
```

The append function, if written correctly, satisfies the property that the length of the output is the sum of the lengths of the inputs:

$$\texttt{length}\,(\texttt{append}\,xs\;ys) = \texttt{length}\,xs + \texttt{length}\,ys$$

This is not checkable directly in Haskell, although we could use a tool such as QuickCheck [CH00] to generate random test cases, or write a correctness proof externally. With a dependent type system, we can give this function a more precise type which reflects the property directly in the type. This helps avoid a common class of error (using the wrong list) by giving each input list a distinct type. Although such an error is unlikely in a small function such as this, in a large system it may not be so difficult to confuse one list for another.

There have been various approaches to implementing dependent types in programming languages, so the type of the function, and how the list is represented, varies from system to system. Let us consider now some different implementations of dependent types and discuss how they might be used to implement list append such that it provably preserves the length property above.

1.2.1 Cayenne

Cayenne [Aug98] is a dependently typed functional language loosely based on Haskell, and similar to the language of the Agda theorem prover [Hal01]. Cayenne allows functions to compute types, which allows more functions to be typeable; examples given are `printf`,

the type of which is computed by examining the format string, and a well-typed interpreter [AC99], the return type of which depends on the object level expression to be evaluated.

Without going into too much detail on the syntax, let us consider how to implement append. We can express lists of a given length (known as vectors) in Cayenne by writing a function `vect` to compute an appropriate type via a recursive definition (rather than by declaring a data structure):

```
vect :: (a :: #) -> Nat -> #;
vect a (Zero) = Unit;
vect a (Succ x) = (Pair a (vect a x));
```

An empty vector is represented by the unit type, and a non-empty vector by a pair of the head and tail. Peano style natural numbers are used here to represent the length of the vector, `Zero` and `Succ` being the constructors of a data type of natural numbers. The type of the append function now expresses the property that the length of the resulting list is the sum of the length of the inputs:

```
append :: (a :: #) |-> (n,m :: Nat) |->
          (vect a n) -> (vect a m) -> (vect a (n + m));
append xs ys = ys;
append xs ys =  case xs of {
          (z,zs) -> (z,append zs ys)
     };
```

Note here that pattern matching is on the length, rather than the vector itself. Pattern matching on the vector is not allowed, since empty and non-empty vectors are represented by different concrete types. There is a small notational overhead here (i.e., the additional arguments `a`, `n` and `m`, which are required as the type of the function depends on them), but the advantage is that we know from the type that append satisfies the property we want.

The drawback to Cayenne's powerful type system is that typechecking becomes undecidable. This is because typechecking in this type system requires the evaluation of type level programs at compile-time — if a type level program does not terminate, typechecking will not terminate. Cayenne deals with the problem by inserting a configurable upper bound on the number of reduction steps allowed in the typechecker; reaching this upper bound is treated as a type error. Hence the result of typechecking is "Correct", "Incorrect" or "Don't know".

1.2.2 DML

DML [Xi98] is an extension to ML allowing a form of dependent types. It is really a family of languages DML(C) where C is a constraint domain from which we draw the values on

which types can be predicated. In DML, we do not write functions which compute types — instead, we give constraints on the types which are verified by a constraint checker. In his thesis, Xi implements the domain of natural numbers, and adds a syntax for annotating ML types with indices. Lists can be annotated as follows:

```
datatype 'a list = nil | cons of 'a * 'a list
with nil <| 'a list(0)
   | cons <| {n:nat} 'a list(n) -> 'a list(n+1)
```

Using this annotated list type, we can also declare the type of append in terms of annotated lists. The definition is the same as the non-dependently typed version, but the type expresses the length property which the definition must satisfy:

```
fun ('a)
    append(nil,ys) = ys
  | append(cons(x,xs),ys) = Cons x (append xs ys)
where append <| {m:nat,n:nat} List(m) * List(n) -> List(m+n)
```

Here we have used a standard list type, but added annotations which describe the length. The advantage is that we can pattern match on the list as usual, however there is not the full dependency of Cayenne in that only types for which a constraint checker has been implemented can be used as indices.

The original motivation for this was to catch more errors at compile-time; however, Xi has also used dependent types to direct optimisations including array bounds check elimination [XP98] and dead code elimination [Xi99a].

1.2.3 Inductive Families and Epigram

EPIGRAM is a platform for dependently typed functional programming based on **inductive families** [Dyb94]. Inductive families are a form of simultaneously defined collections of algebraic data types (such as Haskell `data` declarations) which can be parametrised over values as well as types. For our list append example, we can declare an inductive family for vectors, parametrised over the element type and indexed over the length. To do this, first we declare a type of natural numbers, using the natural deduction style notation proposed in [MM04b]:

$$\underline{\text{data}} \quad \frac{}{\mathbb{N} : \star} \quad \underline{\text{where}} \quad \frac{}{0 : \mathbb{N}} \quad \frac{n : \mathbb{N}}{\mathsf{s}\, n : \mathbb{N}}$$

The reason for using the natural deduction style notation, rather than the more standard Haskell style `data` declaration is that a constructor of a family is allowed to target a *subset* of the family if desired, where the subset is given by a parametrised function which itself may be a constructor (of another family). In the following declaration of vectors, for example,

note that Vnil only targets vectors of length zero, and Vcons only targets vectors of length greater than zero:

$$\underline{\text{data}} \quad \frac{A : \star \quad n : \mathbb{N}}{\mathsf{Vect}\ A\ n : \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{Vnil} : \mathsf{Vect}\ A\ 0}$$
$$\frac{x : A \quad xs : \mathsf{Vect}\ A\ k}{\mathsf{Vcons}\ x\ xs : \mathsf{Vect}\ A\ \mathsf{s}k}$$

To write **append**, since lists are indexed over their lengths, we first need "append on lengths", namely **plus**. The type of a function is introduced with a let declaration, also written in a natural deduction style. The function itself is written in a pattern matching style, with elim n indicating that the function is primitive recursive over n. We will discuss this notation in detail in Chapter 2 — elim n in particular gives access to an **elimination rule** for $\mathbb{N}$ which implements primitive recursion over $\mathbb{N}$. Elimination rules, implemented by pattern matching, are an important feature of EPIGRAM which we will introduce in section 2.1.3. We write **plus** as follows:

$$\underline{\text{let}} \quad \frac{n, m : \mathbb{N}}{\mathbf{plus}\ n\ m : \mathbb{N}}$$

$$\begin{array}{lll} \mathbf{plus} & n & m \Leftarrow \underline{\text{elim}}\ n \\ \mathbf{plus} & 0 & m \mapsto m \\ \mathbf{plus}\ (\mathsf{s}\ k) & m & \mapsto \mathsf{s}\ (\mathbf{plus}\ k\ m) \end{array}$$

We are now in a position to write the **append** function. The type signature of this function is similar to the equivalent function in Cayenne, but written using the natural deduction style notation. The two arguments n and m are implicit – since they are used in the types of xs and ys, and we know the type of Vect, the typechecker infers that they must also be arguments to **append** and so there is no need to write them down:

$$\underline{\text{let}} \quad \frac{xs : \mathsf{Vect}\ A\ n \quad ys : \mathsf{Vect}\ A\ m}{\mathbf{append}\ xs\ ys : \mathsf{Vect}\ A\ (\mathbf{plus}\ n\ m)}$$

$$\begin{array}{lll} \mathbf{append} & xs & ys \Leftarrow \underline{\text{elim}}\ xs \\ \mathbf{append} & \mathsf{Vnil} & ys \mapsto ys \\ \mathbf{append}\ (\mathsf{Vcons}\ x\ xs) & ys & \mapsto \mathsf{Vcons}\ x\ (\mathbf{append}\ xs\ ys) \end{array}$$

This is similar to the DML definition, and the program itself (ignoring the type) is similar to the Haskell definition. However we are not limited to indexing only over natural numbers, as in DML. The disadvantage is that the checking of more complex constraints is not automated — for example we may have to write extra functions to prove commutativity or associativity of **plus**.

The **length** function is straightforward to write in this setting, as the length is passed implicitly as an argument with any Vect. Since it is implicit, we subscript it in the definition (as n):

$$\underline{\text{let}} \quad \frac{xs : \mathsf{Vect}\ A\ n}{\mathbf{length}\ xs : \mathbb{N}}$$

$$\mathbf{length}_n\ xs \mapsto n$$

Even this function is redundant; we know the length is n from the type before we evaluate this function. We effectively carry the length around with every list, trading time (computing the length) for space, as with vectors in the C++ STL [MSD01].

Inductive families have been used extensively by theorem provers including COQ [Coq01], LEGO [LP92], ALF [Mag94] and Plastic [CL01]. In this kind of setting dependent types can be used to prove properties of simply typed programs, for example by declaring an inductive family to represent the desired property. A trivial example, the less than or equal relation, can be represented as an inductive family:

$$\underline{\text{data}} \quad \frac{x, y \ : \ \mathbb{N}}{x \leq y \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{leO} \ : \ 0 \leq y} \quad \frac{p \ : \ x \leq y}{\mathsf{leS}\, p \ : \ \mathsf{s}x \leq \mathsf{s}y}$$

To construct an instance of $a \leq b$ is effectively to prove the proposition that a is less than or equal to b; hence we can prove that a program has this property by constructing instances of $a \leq b$ for appropriate a and b. We will see an example in section 2.3.2, where the **minus** function is defined to take a third argument which ensures that the smaller number is subtracted from the larger number.

The EPIGRAM notation is defined by McBride and McKinna in [MM04b]. This notation elaborates to a dependent type theory based on Luo's UTT [Luo94]. The research documented in this thesis has been carried out in the context of a prototype back-end for EPIGRAM and so I will discuss the notation briefly introduced here in greater detail in Chapter 2. The main innovative feature of EPIGRAM is to take inductive families seriously as data structures, rather than as a basis for describing properties of programs.

1.2.4 Benefits of Dependent Types

Types for Specification

We have seen an example, with list append, of how dependent types allow us to give a more precise type to functions. Functions over the Vect family specify *invariant* properties, namely the lengths of the vectors involved. Such invariants allow the typechecker to check properties which would otherwise need to be verified by the programmer by hand. Another example, red-black trees [Oka99], must maintain the invariants that a red node does not have a red child, and all paths from the root to an empty node pass through the same number of black nodes. Xi shows an implementation of this with dependent types [Xi99b], so that the invariants are checked by the typechecker.

With dependent types, the programmer and compiler have more information about what the program is intended to do prior to writing the program. This helps the programmer, in that it aids understanding of the problem and helps them write a correct program, and helps the compiler, in that it has more information with which to identify potential errors and optimisations. By giving more precise types, we are giving a more precise specification.

Therefore, implementation errors are more likely to be identified at compile-time rather than run-time.

We prefer, therefore, to take types as the prior notion to programs, treating them as specifications of programs. Rather than writing a program without type annotations then allowing the compiler to infer the type afterwards (if indeed the program is well-typed) we prefer to write the type first, restricting the number of programs we can write. In this way, types can drive the process of program development, encouraging the programmer to understand the problem in advance and guiding the programmer to a correct program by refinement. With type inference, *any* well-typed program will do, whatever its type — with the type as the prior notion, however, only a well-typed program of the given type will do. Dependent types enable a programmer to say more precisely which programs are acceptable.

Proofs as Programs

Another benefit of using a dependent type system is that proofs of correctness can be written in the language itself, such is the richness of the type system. Rather than showing some property of a function externally (an error prone process since it depends on correctly transcribing the program from one setting to another) a property can be shown in the language itself. This has the advantage that the proof of a property of a function is based on the actual implementation, rather than some external model. In this way, dependent types can also be used to prove properties of simply typed programs. The Curry Howard isomorphism [CF58, How80] describes the correspondence between proofs and programs.

There are two approaches to showing properties of a program within the language. The apparently simpler approach is to represent the property as a datatype (for example the less than or equal type in section 1.2.3). Then we can write functions which build instances of that type to prove properties of the program. However, it is often preferable to represent the property as an *index* of a datatype. For example, Vect is indexed over its length, which means that any well-typed function which manipulates a Vect is implicitly also a proof of the length invariant of that function. So by using inductive families with appropriate indices, we do not need to write proofs after writing the program — the proof is implicit in the fact that the program is well-typed.

Dependent types are also used to extract simply typed programs from proofs of their specifications. Program extraction in COQ [PM89, Let02] extracts the computational parts from the proof of a specification and generates an ML or Haskell program. We can also consider the use of dependent types for hardware verification. In Chapter 5 we will see a development of binary arithmetic, representing numbers as an inductive family in order to ensure consistency of some aspects of the implementation.

Articulacy

Aside from improving the safety of programs, dependent type systems give us more articulacy and subsume many other sophisticated programming techniques and language extensions. Phantom Types [Hin03] and Generic Haskell [CL02] for example provide extensions which are also handled in a dependently typed setting. Furthermore, there are programs we can write in a dependently typed language which would not be typeable in a simply typed language.

The C function `printf` takes a format string which determines the form of the rest of the arguments. This is an obvious example where dependent types would be useful, and a straightforward implementation is given in Cayenne [Aug98]. Functional unparsing [Dan98] presents a technique for producing formatted output in a simply typed language, but this relies on using sophisticated implementation techniques to get around the less sophisticated type system.

The Haskell standard prelude includes a family of functions for applying a function of n arguments to corresponding items in n lists. There are 8 functions defined separately for this, `zipWith1`...`zipWith8`. Again, techniques have been proposed to allow the implementation of this more generically [FI00, McB02], but again these rely on sophisticated implementation tricks (and often clumsy notation) to get around the type system. Dependent types give a more elegant approach to solving such problems — the hard work is done by the type system, not the programmer.

With dependent types, we can implement lists with varying element type in a type safe fashion. The interpreter example in Chapter 4 includes an example of this, where values in the environment may be any one of several types. This interpreter, based on [AC99], uses dependent types to avoid the need to "tag" each value with its type — instead types are determined by the expression being interpreted.

A recent extensions to the Glasgow Haskell Compiler, Generalised Algebraic Data Types [PWW04], adds some of the power of dependent types to Haskell. For example, well-typed terms can be given a more precise type as in [AC99]. However, they still do not allow types to be predicated on values, as with a full implementation of dependent types.

Interactive Development

A potential further benefit of dependent types is that it gives more information to an interactive type-directed programming system. The kind of type-directed editing used in theorem provers, such as COQ and LEGO, is not often seen for programming languages (*CYNTHIA* [WBBL99], for ML, is an exception). A possible reason for this is that the type system does not give enough information for type-directed editing to be worthwhile; with dependent types, there is both more possibility of the system being able to direct the programmer to a program, and more need of such a system since the more precise types can make it harder to find a well-typed program without machine assistance.

Efficiency

Dependent types give us more static information about what a program is intended to do. Altenkirch [Alt93] mentions that this information could potentially be used to make programs more efficient. However, this potential has been exploited very little until recently. Xi has used dependent types to aid with array bounds check elimination [XP98] and dead code elimination [Xi99a] in DML, and Augustsson and Carlsson's tagless interpreter [AC99] is an example of how dependent types allow more efficient code. However, there has been little work on optimisation of programs built on inductive families, largely because inductive families have not, until now, been taken seriously as an approach to programming.

Unfortunately, in a naïve implementation of a dependently typed language with inductive families there are several overheads. The separation between types and values is blurred; types can be computed from values, and values can hold information about types. In particular, inductive families can store information about their invariants. There seem to be several sources of overhead here; there are space overheads in storing the indices and time overheads in the complex manipulations required on types. In a naïve implementation, this can lead to quite an overhead. However, the opposite ought to be true — the type system tells us more about what a program is supposed to do, therefore we ought to be able to produce *more* efficient code. This thesis investigates techniques for doing so.

1.2.5 Strong Normalisation

A distinctive feature of EPIGRAM is that all (well-typed) terms are **strongly normalising**. A term is strongly normalising if all reduction sequences starting from that term terminate without error at the same normal form; Goguen shows that the strong normalisation property holds for UTT [Gog94], and as a result EPIGRAM programs (being based on a type theory similar to UTT) are strongly normalising. This has several implications and advantages. Firstly, we have a much stronger notion of type safety. In a type safe, but not strongly normalising language such as Haskell, running a type correct program can have one of three results:

- The program terminates, giving a result of the appropriate type.
- The program terminates with an error due to an expression not being defined for all possible inputs. This kind of error means that reduction can not progress.
- The program does not terminate. This kind of error means that reduction will progress infinitely.

In EPIGRAM, strong normalisation ensures that only the first possibility can apply. To put it another way, the error value (denoted $\bot$) is implicitly an element of all types if non-termination and partial definitions are allowed, but it is not an element of any type in EPIGRAM. There is a clear advantage here, in that running a program is guaranteed to yield a

result. Strong normalisation also ensures the decidability of typechecking; we no longer have the difficulty that type level programs may not terminate, as in Cayenne. The undecidability of the Halting Problem for Turing complete languages means that we cannot tell for *any* program whether or not it terminates, and so we write programs for which the machine can establish termination by checking that recursive calls are on syntactically structurally smaller values. Turner discusses this in [Tur96]; he observes that in practice most programs are structurally recursive, and many of those which aren't (such as quicksort) can be made so (we will discuss the quicksort example in particular in section 2.3.4). Nevertheless, there are some programs which it will always be impossible to write, since a strongly normalising language can not be Turing complete.

We could imagine a hypothetical dependently typed language being on one of three levels:

- All programs terminate (Strongly normalising).
- Type level programs (those run at compile-time) terminate.
- No termination restriction.

Dependent type theory and EPIGRAM sit on the first level, Cayenne on the last. DML, by having less sophisticated type level programs, sits on the second level. In practice, we might consider relaxing the strong normalisation restriction in EPIGRAM if given an appropriate compiler flag, to move to the second and third levels; however in this thesis I will consider strongly normalising programs only, because we can use the strong normalisation property to our advantage in optimisation.

1.3 Contributions

Types give us static information about a program; they tell us what a program is supposed to do. Dependent types allow *more accurate typing* and hence give us *more static information.* We ought to be able to make use of this not only to have more confidence about whether a program works as planned, but also to optimise more aggressively. This thesis explores the optimisation of dependently typed programs, the primary contributions being:

- A technique for removing redundant and duplicated information from data structures. This technique examines type dependencies and removes terms whose values are forced by other values. Also, it identifies and removes constructor tags which are made redundant by case analysis on other values. The values which are removed are introduced by the use of dependent types; it is therefore important that such values are identified and removed in order for dependently typed programs to have comparable run-time to simply typed programs.
- A compilation scheme for a dependently typed lazy functional language. I extend well-understood technology for efficient evaluation of lazy functional languages (specifically,

Johnsson's G-machine [Joh84] and Augustsson's pattern matching compiler [Aug85]) to take advantage of our detailed type information. It is not essential that we use the G-machine; other methods for executing functional languages (whether lazy or eager) can be adapted in a similar way.

- An optimised representation of natural numbers using an external implementation. I consider what is required to provide external implementations of type theoretic data structures, taking natural numbers as an example.

- Specific techniques for transforming decorated terms in a dependent type theory into efficiently executable code which leads in particular to the removal of unreachable code branches, identified by typing.

While this work presents several optimisations for dependently typed programs, it is important to understand that since dependently typed programs are initially decorated with much more static information *in the program* as well as in the type, we are starting at what seems like a big disadvantage. Perhaps, then, the most significant contribution is the removal of redundant static information from the program and its data, without affecting the operational behaviour of the program and the meaning of its data. Having reached this point (which merely catches up with where we *start* optimising simply typed programs) we can begin to apply further optimisations based on our rich type information.

1.4 Related Work

Martin-Löf's constructive type theory [ML71] has been the basis for much research in theorem proving and programming via the Curry Howard isomorphism [CF58, How80]; other dependent type theories are Luo's Extended Calculus of Constructions [Luo94] and the Calculus of Inductive Constructions [Coq01]. Interactive theorem provers are often based on some form of dependent type theory — the logical language of NuPrl [C+86] is similar to Martin-Löf's type theory with universes [ML75]; LEGO [LP92] is based on Luo's ECC and COQ is based on the Calculus of Inductive Constructions. OLEG [McB00a] builds on the LEGO [LP92] theorem prover, introducing tactics geared towards *programming* with inductive families as well as theorem proving. Further (unpublished) work on OLEG resulted in tactics for interactive development of pattern matching programs — these tactics led to the design of EPIGRAM [MM04b].

Recent extensions to the Haskell type system can be used to support some aspects of programming with dependent types. McBride's "Faking It" style of programming [McB02] shows how Haskell type classes with functional dependencies can be used to implement some vector operations (and some inductive families). Each constructor of the vector, `Vnil` and `Vcons`, is a separate type, and is an instance of a type class `Vect`. To define a function over vectors then involves implementing a method as part of the type class. While this

gives some of the advantages of dependent types, such as the more precise types of vector operations, there are some big problems with this approach. Firstly, it does not generalise to all inductive families. Secondly, the notation required to program in this way is rather inconvenient — function definitions are distributed among several `instance` declarations. Thirdly, there is a potential run-time overhead in that the implementation of type classes necessitates the passing around of a dictionary of functions representing the methods of a class (although this can often be inlined).

The majority of this work is concerned with the efficient execution of terms in a dependent type theory. For this, we consider interpretation and compilation. Interpretation is based on the normalisation by evaluation technique of Berger and Schwichtenberg [BS91], and compilation is based on Johnsson's G-machine [Joh84] and Augustsson's pattern matching compilation [Aug85]. We are therefore considering the execution of the type theory itself, rather than translating to some other setting as is the approach of program extraction [PM89, Let02] (which translates type theory terms to ML or Haskell) and Cayenne (which translates Cayenne programs to Lazy ML, and compiles the resulting program with the typechecker switched off). By compiling directly, rather than via another high level language, we have the opportunity to take advantage of features of the type theory in implementing compilation efficiently.

An important aspect of efficient execution is the optimisation of programs. There is potentially a large amount of redundant information in types, and many of the optimisations of dependently typed programs we will see involve the removal of computationally irrelevant or unused parts of code, in a similar manner to Berardi's pruning of simply typed λ-terms [Ber96]. We will see methods for removing redundant information from dependently typed data structures in Chapter 4. Some of the techniques we shall see here, in particular the removal of content-free data structures, have a similar effect to aspects of program extraction in COQ which aims to remove the purely logical parts of a proof to retrieve a program. The advantage to the techniques we use in Chapter 4 over program extraction is that it is not only the logical parts which are removed, but *all* parts which can be shown not to be used at run-time. Nevertheless, the techniques we shall see are equally applicable to program extraction.

Another optimisation, which we shall see in Chapter 5, involves the transformation of a high level representation of natural numbers into a low level primitive type. A similar approach is taken by [MB01, Mag03] for implementing numbers more efficiently in COQ. The Isabelle theorem prover [NPW02] also implements natural numbers natively, although the techniques for doing so are not documented[1]. The low level implementation of natural numbers leads to the possibility of a further optimisation, unboxing the representation [PL91a], in which numbers are represented directly rather than as pointers to their binary representation.

Many techniques which apply to simply typed languages can also be adapted towards

[1] Larry Paulson, personal communication

optimising dependently typed programs; for example, the Glasgow Haskell Compiler's compilation by transformation approach [San95, PS98] applies correctness preserving transformation rules to an intermediate representation. Inlining in particular [PM02] is an important optimisation for two reasons; firstly, functional programmers use functions in much the same way as C programmers use macros, and hence a good inliner is vital, and secondly inlining often exposes further optimisation opportunities. We will examine some program transformations in Chapter 6.

1.5 Overview

The research documented in this thesis has been carried out in the context of an experimental implementation of a back end for EPIGRAM. In this section I will give an overview of the implementation and an outline of the rest of the thesis.

1.5.1 System Overview

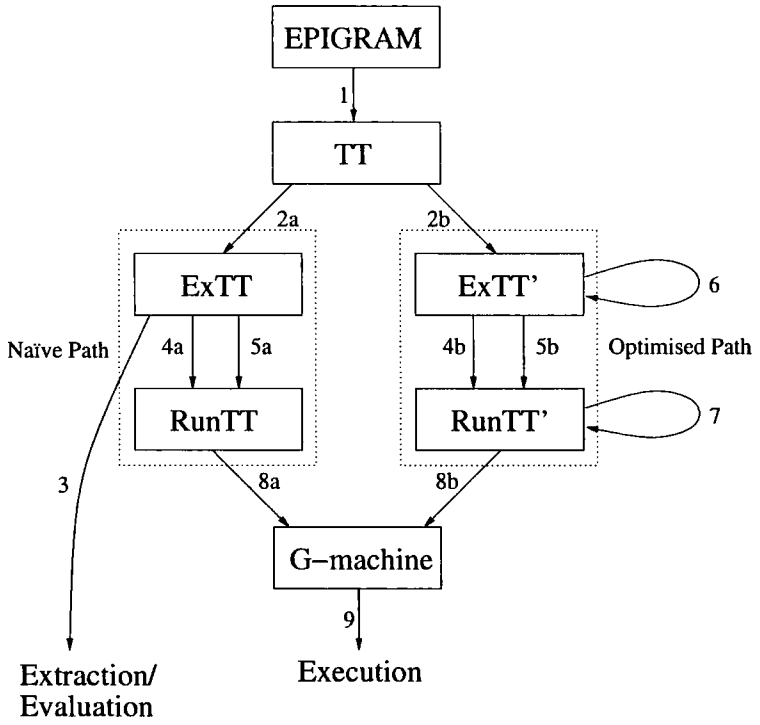


Figure 1.1: System Overview

Figure 1.1 shows an outline of the system. Between the EPIGRAM program and its execution there are several stages, and two possible paths through the system. The left path

represents a naïve approach to compilation, where no optimisation takes place and all terms are directly compiled from their elaborated form. An understanding of this path is necessary to explain the path on the right, which represents an optimising approach to compilation. Along this path, we remove duplicated and redundant information from data structures and make use of the rich type information to remove unreachable code. The phases indicated on the diagram are briefly summarised as follows:

- Step **1** in the diagram is the elaboration phase. This is described by [MM04b]; in this phase, programs in EPIGRAM notation are typechecked and elaborated into a core type theory, TT.
- We now either take a naïve approach, or an optimising approach. The naïve path proceeds as follows:
  - Step **2a** is a transformation into an execution language for the core type theory, called ExTT. In fact, in this approach, TT and ExTT are identical, so this is the identity transformation.
  - Step **3** is extraction of ExTT terms into Haskell, which is a simple to implement method of executing terms, but less efficient than compilation into an abstract machine language. It is less efficient because, to deal with terms which have an EPIGRAM type but no Haskell type, we must use an intermediate representation of values.
  - Steps **4a** and **5a** are two parts of the transformation into a run-time language of function definitions, RunTT. RunTT programs consist of **supercombinator** definitions; these are function definitions with no free variables and no inner lambdas, a form suitable for compilation to abstract machine code. Step **4a** translates user defined functions by Johnsson's supercombinator lifting algorithm, while step **5a** translates pattern matching elimination rules into RunTT using an adapted version of Augustsson's pattern matching compiler [Aug85].
  - Step **8a** translates the supercombinator language into G-machine code [Joh84], an efficient abstract machine language for the execution of lazy functional programs. Some minor modifications are made to account for compiling dependent types.

  The optimising path is the primary contribution of this thesis. The steps are similar to those in the naïve path, but the transformations between each stage are more involved. This path proceeds as follows:
  - Step **2b** translates TT into the execution language ExTT′, which here is a marked up version of TT. Parts of terms which are unused or duplicated (that is, considered redundant) are marked for deletion.
  - Steps **4b** and **5b** correspond to **4a** and **5a**, in that they convert ExTT′ into RunTT′ by lambda lifting and pattern matching compilation. The marking of step **2b**

means that these processes are not so simple — the lambda lifting process removes all terms which are marked for deletion; such terms must also be accounted for by the pattern matching compiler.

- Step **6** is a compilation-by-transformation phase on the execution language ExTT′. This makes some transformations for efficiency, in particular, making recursive calls direct rather than via an elimination rule.
- Step **7** is a second compilation-by-transformation phase on the supercombinator language. In this phase, source to source transformations are applied to RunTT′ which make use of the knowledge we have gained through typechecking — for example, removal of impossible cases. Also some standard transformations are made — inlining, and removal of unused arguments.
- Step **8b** translates RunTT′ into G-machine code. I introduce some new instructions to the G-machine for this phase to take advantage of the marking optimisations in ExTT′.

- Finally, step **9** involves the execution of G-machine code. There are several ways to achieve this — either by the implementation of an interpreter for the abstract machine, or a compiler from G-code to a more concrete target language such as C, machine code, or C-- [PNO97]. I give state transition rules for G-machine instructions, many of which are as originally defined by Johnsson, but some of which I introduce to implement the optimising features of ExTT′ and RunTT′.

1.5.2 Chapter Outline

The various stages in the compilation of EPIGRAM programs, as shown in figure 1.1 are presented in this thesis. The structure of the rest of this work is as follows:

Chapter 2 presents a background to the literature and the field of type theory and functional programming, and an introduction to programming with dependent types in the EPIGRAM notation.

Chapter 3 discusses execution environments and covers the naïve compilation path into G-machine code, adapting Johnsson's G-machine for use with TT.

Chapter 4 covers steps **2b**, **4b** and **5b**. In step **2b**, terms are marked up for later deletion. Marking takes place by means of three optimisations. The first of these is the **forcing** optimisation, which identifies parts of terms whose value is determined by another part of a term (and hence are redundant). Secondly, the **detagging** optimisation identifies where constructor choice in an elimination is determined by another value, meaning that the constructor tag need not be stored. The third optimisation is **collapsing** which identifies types with no computational meaning, which can be deleted entirely at run-time.

After step **2b** marks terms for deletion, these terms really are deleted in the supercombinator lifting process in step **4b**. Marking also affects the pattern matching compilation process, step **5b** — no case selection can take place on deleted terms. In this chapter we will see a modified pattern matching compiler algorithm which takes account of this and further takes advantage of the strong normalisation property of EPIGRAM.

Also in Chapter 4 are several examples, including an extended example of these techniques showing an inductive family based implementation of Augustsson and Carlsson's well-typed interpreter [AC99] and its run-time costs.

Chapter 5 considers the introduction of primitive types into the language, and the optimisation of the natural number representation $\mathbb{N}$ by transformation of ExTT′. This occurs in step **6** of the compilation process.

Chapter 6 covers additional optimisations. Firstly, a method for removing the abstraction layer of elimination rules is presented. By this method, recursion at run-time is implemented directly rather than by an elimination operator, effectively recovering the declared pattern matching behaviour of functions. As well as removing a layer of abstraction, this opens up the possibility of further optimisations such as tail recursion optimisation.

This chapter also considers optimisations which only apply in a dependently typed language of total functions — specifically, the elimination of impossible cases by *typing* rather than by global analysis. These optimisations take place during steps **6** and **7** of the compilation process.

Chapter 7 presents some conclusions. We will see how the features of EPIGRAM's type system contribute to a more efficient implementation of programs and consider some directions for further research.

Appendices A, B and C cover other technical details. Appendix A gives a detailed account of compiling a simple function, Appendix B presents some proofs of the properties of ExTT and Appendix C gives an implementation of a normalisation algorithm for ExTT.

1.5.3 Implementation Note

At the time of writing the EPIGRAM elaborator is still in development, although an early version has recently been released. In particular, this prototype has not implemented the *with* or *named with* notation described in section 2.2.7. The implementation documented by this thesis is of a *prototype* back-end for EPIGRAM. This includes an implementation of TT (including a simple theorem prover with tactics for building terms in TT), compilation to G-machine code via ExTT and RunTT, and extraction of Haskell programs from TT. This

prototype has served as an environment for experimentation with the implementation and optimisation techniques described here. Nevertheless, the techniques described will also be applicable to elaborated EPIGRAM programs, or indeed any language based on dependent type theory.

Since the front end is still in development, there are no large, real world, examples as yet. As a result there is no benchmark suite corresponding to Haskell's *nofib* suite [Par92], for example, against which to compare the results of the optimisations presented here. Instead, the results I present are in the form of comparisons between code generated by the naïve and optimised compilation paths and analysis of the run-time costs of the RunTT programs generated. These results themselves are encouraging, and suggest that it is indeed possible to build a feasible programming language on top of a dependent type theory.

Chapter 2

Epigram and its Core Type Theory

This chapter gives an introduction to the background of type theory and dependently typed functional programming and introduces the high level EPIGRAM notation along with the core type theory to which it elaborates. In the introduction we considered the benefits of dependent types for programming and some of the approaches taken by various languages and systems. We saw in the introduction that the characteristic feature of a dependent type system is the ability to predicate types on values, which leads to a more precise specification for programs, using list append as a motivating example. In this chapter, we will see in more detail how dependent types are used in EPIGRAM and its core language and consider several examples of EPIGRAM programs.

We will look first at the core language of EPIGRAM, which I call TT, since this is the language we will be compiling and optimising in later chapters. This core language, introduced in section 2.1, is a dependent type theory similar to Luo's ECC [Luo94] with the addition of definitions and inductive families. Tactics for developing programs in dependent type theory developed by McBride [McB00a] led to the design of the high level EPIGRAM notation. We will later see several examples of EPIGRAM programs and so in section 2.2 we introduce the high level notation and discuss some of the programming idioms this allows in section 2.3.

2.1 TT — The Core Type Theory

The first stage in the compilation of a programming language is translation to a core representation; in the case of a functional language this is often a form of the λ-calculus. For example, the core language of the Glasgow Haskell Compiler [GHC03], Core Haskell [TT01], is a subset of Haskell resembling the polymorphic λ-calculus. The core language of EPIGRAM is based on a dependently typed λ-calculus, similar to Luo's ECC [Luo94] with some minor

additions for practical programming. In this section, we will examine the details of the core language and look at some example programs.

2.1.1 The Core Language

The core language of EPIGRAM, which I call TT, is based on Luo's ECC with definitions, inductive families and equality. The syntax of TT is shown in figure 2.1. We may also abbreviate the function space $\forall x{:}S.\ T$ by $S \to T$ if x is not free in T. There is an infinite hierarchy of predicative universes, $\star_i : \star_{i+1}$. Universe levels can be left implicit and inferred by the machine, as in [HP91]. As such, when showing TT terms, we will generally leave out the universe level; for the majority of the examples in this thesis, $\star$ indicates $\star_0$.

$t ::=$	$\star_i$	(type universes)	$\mid$ x		(variable)
$\mid$	$\forall x{:}t.\ t$	(function space)	$\mid$ D		(inductive family)
$\mid$	$\lambda x{:}t.\ t$	(abstraction)	$\mid$ c		(constructor)
$\mid$	$t\ t$	(application)	$\mid$ D-**Elim**		(elimination rule)
$\mid$	$\underline{\text{let}}\ x \mapsto t\ :\ t\ \underline{\text{in}}\ t$	(let binding)			

Figure 2.1: The core language, TT

Remark: Although x, D, c and D-**Elim** all represent names of some form, it is convenient in an implementation to make this syntactic distinction as each one is treated differently in various parts of the system.

Contexts

The core language gives the syntax for both types and terms. In addition, we have a context Γ which binds names to types and values. A valid context is defined inductively as:

$$\frac{}{\mathcal{E} \vdash \underline{\text{valid}}} \qquad \frac{\Gamma \vdash S\ :\ \star_i}{\Gamma; x\ :\ S \vdash \underline{\text{valid}}} \qquad \frac{\Gamma \vdash s\ :\ S}{\Gamma; x \mapsto s\ :\ S \vdash \underline{\text{valid}}}$$

Where $\mathcal{E}$ denotes the empty context, $\Gamma; x\ :\ S$ denotes a context extended by a variable declaration x with its type S, and $\Gamma; x \mapsto s\ :\ S$ denotes a context extended by a variable definition x with its type S and value s. Computation and typechecking only make sense relative to a context. We write the typing judgement, which is a relation expressing that a term t has type T relative to a context Γ as follows:

$$\Gamma \vdash t\ :\ T$$

Where computation or typechecking takes place in the empty context, I shall write the typing judgement as follows, eliding the $\mathcal{E}$:

$$\vdash t\ :\ T$$

Computation

Conceptually, computation in the core language is defined by contraction rules, given in figure 2.2. **Contraction**, relative to a context Γ, is given by one of the following contraction schemes:

- β-contraction, which substitutes a value applied to a λ-binding for the bound variable in the scope of that binding. Since we have local definitions, by let bindings, then β-reduction is given by the scheme $\Gamma \vdash (\lambda x{:}S.\, t)\, s \rightsquigarrow \underline{\text{let}}\, x \mapsto s \,:\, S\, \underline{\text{in}}\, t$.
- η-contraction, which eliminates redundant λ abstractions. η-contraction is given by the scheme $\Gamma \vdash \lambda x{:}S.\, f\, x \rightsquigarrow f$.
- δ-contraction, which replaces a let bound variable by its value. δ-contraction is given by the scheme $\Gamma; x \mapsto s \,:\, S \vdash x \rightsquigarrow s$.

$$\begin{array}{ll} \beta\text{—contraction} & \dfrac{}{\Gamma \vdash (\lambda x{:}S.\, t)\, s \rightsquigarrow \underline{\text{let}}\, x \mapsto s \,:\, S\, \underline{\text{in}}\, t} \\ \eta\text{—contraction} & \dfrac{}{\Gamma \vdash \lambda x{:}S.\, f\, x \rightsquigarrow f} \\ \delta\text{—contraction} & \dfrac{}{\Gamma; x \mapsto s \,:\, S; \Gamma' \vdash x \rightsquigarrow s} \end{array}$$

Figure 2.2: Contraction schemes for TT

The terms of the form $(\lambda x : S.\, t)\, s$, $\lambda x : S.\, f\, x$ and x are called **β-redexes**, **η-redexes** and **δ-redexes** respectively. The terms $\underline{\text{let}}\, x \,:\, S \mapsto s\, \underline{\text{in}}\, t$, f and s are their **contractums**, respectively.

β-contraction is often presented as a substitution, i.e. $\Gamma \vdash (\lambda x.\, t)\, a \rightsquigarrow t[x/a]$. Here, we prefer to implement it in terms of let binding as in [MM04b], since this simplifies presentation of the theory; we use the following contextual closure rule to reduce a let binding by giving rise to a δ-redex:

$$\frac{\Gamma; x \mapsto s \,:\, S \vdash t \rightsquigarrow u}{\Gamma \vdash \underline{\text{let}}\, x \mapsto s \,:\, S\, \underline{\text{in}}\, t \rightsquigarrow u}$$

Reduction ($\rhd$) is the structural closure of contraction, and computation ($\rhd^*$) is the transitive closure of reduction. We also say that if a term x contains an occurrence of a redex y, and we replace y by its contractum, resulting in the term x', then x one-step reduces to x' ($\Gamma \vdash x \rhd_1 x'$)

Conversion, denoted $\simeq$, is the smallest equivalence relation closed under reduction and is defined in figure 2.3. If $\Gamma \vdash x \simeq y$, then y can be obtained from x in the context Γ by a finite (possibly empty) sequence of contractions and reversed contractions. Terms which are convertible are also said to be computationally equal. The conversion rule makes use of syntactic equivalence, denoted $\equiv$. If $\Gamma \vdash x \equiv y$, then the terms x and y are are identical

up to α-conversion. We avoid name capture problems in practice by referring to bound names by their de Bruijn indices [dB72] — the de Bruijn index of a variable is the number of variables bound more recently.

Definition: x is **convertible** to y relative to Γ ($\Gamma \vdash x \simeq y$)
if and only if there exist $x_1, \ldots, x_n$ ($n \geq 1$) such that $\Gamma \vdash x \equiv x_1, \Gamma \vdash y \equiv x_n$
and $\Gamma \vdash x_i \rhd_1 x_{i+1}$ or $\Gamma \vdash x_{i+1} \rhd_1 x_i$, for $i = 1, \ldots, n-1$

Figure 2.3: Conversion for TT

We say:

- A term is in **normal form** if and only if it contains no redexes. We denote the normal form of a term t relative to a context Γ by $\Gamma \vdash \text{NF}(t)$. A term t is strongly normalising, denoted $\Gamma \vdash \text{SN}(t)$, if every reduction sequence $t \rhd_1 t_1 \rhd_1 t_2 \rhd_1 \ldots$ reaches normal form in a finite number of reductions.
- A term is in **weak head-normal form**
  - If it is not a reducible expression.
  - If it is of the form $f\ a$ and f is a weak head-normal form.

  We denote the weak head-normal form of a term relative to a context Γ by $\Gamma \vdash \text{WHNF}(t)$.

Type Inference Rules

The type inference rules for TT are given in figure 2.4. Given the language and the typing rules, there are two problems for which we would like to have an algorithm (as with any type system):

- **Type Checking (TC)** Given a term t, a type T and a context mapping names to types Γ, can we determine that the term t has type T in the context Γ (written $\Gamma \vdash t : T$)?
- **Type Synthesis (TS)** Given a term t and a context Γ, can we infer a type T such that $\Gamma \vdash t : T$? This is also known as type inference.

A type synthesis algorithm for TT is given in figure 2.5 (TS). We use the following notation:

- $\Gamma \vdash t \Longrightarrow T$ means that t is assigned the type T.
- $\Gamma \vdash t \Longrightarrow T \twoheadrightarrow T'$ means that the type T assigned to term t has a weak head-normal form of T'.

Using this algorithm, we check a judgment $\Gamma \vdash a \ : \ A$ by synthesising types and checking for conversion in the standard way [Hue89, Coq96], as follows:

- $\Gamma \vdash A \Longrightarrow X \twoheadrightarrow \star_n$ (check A is a type)
- $\Gamma \vdash a \Longrightarrow B$ (infer a type for a)
- $\Gamma \vdash A \simeq B$ (check that the inferred and declared types are convertible)

$$\frac{\Gamma \vdash \underline{\text{valid}}}{\Gamma \vdash \star_n \ : \ \star_{n+1}} \ \textsf{Type}$$

$$\frac{\Gamma; x \ : \ S; \Gamma' \vdash \underline{\text{valid}}}{\Gamma; x \ : \ S; \Gamma' \vdash x \ : \ S} \ \textsf{Var}$$

(Similarly for c, D, D-**Elim**)

$$\frac{\Gamma; x \mapsto s \ : \ S; \Gamma' \vdash \underline{\text{valid}}}{\Gamma; x \mapsto s \ : \ S; \Gamma' \vdash x \ : \ S} \ \textsf{Val}$$

$$\frac{\Gamma \vdash f \ : \ \forall x{:}S.\, T \quad \Gamma \vdash s \ : \ S}{\Gamma \vdash f \ s \ : \ \underline{\text{let}}\ x \ : \ S \ \mapsto \ s \ \underline{\text{in}}\ T} \ \textsf{App}$$

$$\frac{\Gamma; x \ : \ S \vdash e \ : \ T \quad \Gamma \vdash \forall x{:}S.\, T \ : \ \star_n}{\Gamma \vdash \lambda x{:}S.e \ : \ \forall x{:}S.T} \ \textsf{Lam}$$

$$\frac{\Gamma; x \ : \ S \vdash T \ : \ \star_n \quad \Gamma \vdash S \ : \ \star_n}{\Gamma \vdash \forall x{:}S.T \ : \ \star_n} \ \textsf{Forall}$$

$$\frac{\Gamma \vdash e_1 \ : \ S \quad \Gamma; x \mapsto e_1 \ : \ S \vdash e_2 \ : \ T \quad \Gamma \vdash S \ : \ \star_n \quad \Gamma; x \mapsto e_1 \ : \ S \vdash T \ : \ \star_n}{\Gamma \vdash \underline{\text{let}}\ x \ : \ S \mapsto e_1 \ \underline{\text{in}}\ e_2 \ : \ \underline{\text{let}}\ x \ : \ S \mapsto e_1 \ \underline{\text{in}}\ T} \ \textsf{Let}$$

$$\frac{\Gamma \vdash x \ : \ A \quad \Gamma \vdash A' \ : \ \star_n \quad \Gamma \vdash A \simeq A'}{\Gamma \vdash x \ : \ A'} \ \textsf{Conv}$$

Figure 2.4: Typing rules for TT

Remark: The operational semantics of TT requires weak head normalisation — i.e., for reduction to proceed requires the machine to know whether a term is a λ or constructor headed. Some aspects of typechecking also require weak head-normal forms (for example checking if a term has a $\forall$ form at the head). Other aspects require conversion, which relies on reduction to normal form or weak head-normal form.

2.1.2 Inductive Datatypes

Datatypes in the core language TT are defined as inductive datatypes in the style of LEGO, COQ and ALF, and as presented by [Dyb94]. An inductive datatype is declared as a disjoint union of constructors, each with zero or more recursive and non-recursive arguments. An example of an inductive datatype is the type representing natural numbers, $\mathbb{N}$, which can be described in a natural deduction style with a type formation rule, and rules for each constructor, as follows:

$$\frac{\Gamma \vdash \underline{\text{valid}}}{\Gamma \vdash \star_n \Longrightarrow \star_{n+1}}$$

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad x \ : \ S \in \Gamma}{\Gamma \vdash x \Longrightarrow S}$$

(Similarly for c, D, D-**Elim**)

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad x \ : \ S \mapsto s \in \Gamma}{\Gamma \vdash x \Longrightarrow S}$$

$$\frac{\Gamma \vdash f \Longrightarrow X \twoheadrightarrow \forall x{:}S.\ T \quad \Gamma \vdash s \Longrightarrow S' \quad \Gamma \vdash S \simeq S'}{\Gamma \vdash f\ s \Longrightarrow \underline{\text{let}}\ x \ : \ S' \mapsto s\ \underline{\text{in}}\ T}$$

$$\frac{\Gamma; x \ : \ S \vdash e \Longrightarrow T \quad \Gamma \vdash \forall x{:}S.\ T \Longrightarrow X \twoheadrightarrow \star_n}{\Gamma \vdash \lambda x{:}S.\ e \Longrightarrow \forall x{:}S.\ T}$$

$$\frac{\Gamma; x \ : \ S \vdash T \Longrightarrow X \twoheadrightarrow \star_n \quad \Gamma \vdash S \Longrightarrow X' \twoheadrightarrow \star_n}{\Gamma \vdash \forall x{:}S.\ T \Longrightarrow X}$$

$$\frac{\Gamma \vdash S \Longrightarrow X \twoheadrightarrow \star_n \quad \Gamma \vdash e_1 \Longrightarrow S' \quad \Gamma \vdash S \simeq S' \quad \Gamma; x \ : \ S \mapsto e_1 \vdash e_2 \Longrightarrow T \quad \Gamma; x \ : \ S \mapsto e_1 \vdash T \Longrightarrow X' \twoheadrightarrow \star_n}{\Gamma \vdash \underline{\text{let}}\ x \ : \ S \mapsto e_1\ \underline{\text{in}}\ e_2 \Longrightarrow \underline{\text{let}}\ x \ : \ S \mapsto e_1\ \underline{\text{in}}\ T}$$

Figure 2.5: Type synthesis for TT

$$\underline{\text{data}} \quad \frac{}{\mathbb{N} \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{0} \ : \ \mathbb{N}} \quad \frac{n \ : \ \mathbb{N}}{\mathsf{s}\ n \ : \ \mathbb{N}}$$

This type introduces three constants to the context Γ, representing the type constructor ($\mathbb{N}$) and the two data constructors (0 and s).

$\mathbb{N} \ : \ \star \in \Gamma$
$\mathsf{0} \ : \ \mathbb{N} \in \Gamma$
$\mathsf{s} \ : \ \mathbb{N} \to \mathbb{N} \in \Gamma$

Inductive datatypes can also be parametrised over a value. Lists, for example, are parametrised over their element type. This can be described as follows:

$$\underline{\text{data}} \quad \frac{A \ : \ \star}{\mathsf{List}\ A \ : \ \star} \quad \underline{\text{where}} \quad \frac{\star}{\mathsf{nil} \ : \ \mathsf{List}\ A} \quad \frac{x \ : \ A \quad xs \ : \ \mathsf{List}\ A}{\mathsf{cons}\ x\ xs \ : \ \mathsf{List}\ A}$$

Note that we do not declare $A \ : \ \star$ in the premises for nil and cons, as its presence is inferable from the type formation rule. We adopt the convention, as in [MM04b], that constructor arguments with inferable types such as A need not be declared explicitly, for the sake of readability. Nevertheless, when the constructors are added to the context, we keep A as an argument to both nil and cons as it is required to preserve type correctness. The constants which are added to the context are:

$\mathsf{List} \ : \ \star \to \star \in \Gamma$
$\mathsf{nil} \ : \ \forall A{:}\star.\ \mathsf{List}\ A \in \Gamma$
$\mathsf{cons} \ : \ \forall A{:}\star.\ \forall x{:}A.\ \forall xs{:}\mathsf{List}\ A.\ \mathsf{List}\ A \in \Gamma$

In the definition of List, the value of the parameter A does not change across the structure; however, it is not necessary for each constructor to target the entire family as in List, nor is it necessary for the parameter to be a type. We could, for example, parametrise lists over their length as well as their element type. Vect is a datatype for lists parametrised over their length (vectors), and is described as follows:

$$\underline{\text{data}} \quad \frac{A : \star \quad n : \mathbb{N}}{\mathsf{Vect}\ A\ n\ :\ \star} \quad \underline{\text{where}} \quad \frac{}{\epsilon\ :\ \mathsf{Vect}\ A\ 0} \qquad \frac{x\ :\ A \quad xs\ :\ \mathsf{Vect}\ A\ k}{x{::}xs\ :\ \mathsf{Vect}\ A\ (\mathsf{s}\ k)}$$

Here we use an infix constructor for the non-empty vectors, similar to the infix constructor : used for Haskell lists. These rules state that empty lists have length zero and non-empty lists increase the length by one. Hence, as items are added to the vector, the length parameter increases. We call such parameters, which do change across the structure, **indices**. We say that Vect is an **inductive family**.

Note that each constructor targets a sub-family of Vect — this is the reason for using natural deduction style to introduce constructors, rather than a Haskell style `data` declaration. Again, there are implicit arguments to each constructor; the constants added to the context are as follows:

$$\mathsf{Vect}\ :\ \star \rightarrow \mathbb{N} \rightarrow \star \in \Gamma$$
$$\epsilon\ :\ \forall A{:}\star.\ \mathsf{Vect}\ A\ 0 \in \Gamma$$
$$::\ :\ \forall A{:}\star.\ \forall k{:}\mathbb{N}.\ \forall x : A.\ \forall xs{:}\mathsf{Vect}\ A\ k.\ \mathsf{Vect}\ A\ (\mathsf{s}\ k) \in \Gamma$$

The general scheme for declaration of an inductive family D with constructors c_i is given in figure 2.6. The $\vec{s}$ are the indices, and we split the constructor arguments into $\vec{a}$ (the non-recursive arguments) and $\vec{y}$ (the recursive arguments). The vector notation $\vec{x}$ [dB91] denotes the fact that there may be zero or more arguments in the form of x, and correspondingly x_i denotes the ith (zero based) entry in the vector $\vec{x}$. The constructors c_i can not be reduced further; we say that a term which is a fully applied constructor is in **canonical form**.

$$\underline{\text{data}} \quad \frac{\vec{i}\ :\ \vec{I}}{\mathsf{D}\ \vec{i}\ :\ \star}$$

$$\underline{\text{where}} \quad \frac{\vec{a}_1\ :\ \vec{A}_1 \quad y_{11}\ :\ \mathsf{D}\ r_{11}\ \ldots\ y_{1j}\ :\ \mathsf{D}\ r_{1j}}{\mathsf{c}_1\ \vec{a}_1\ \vec{y}_1\ :\ \mathsf{D}\ \vec{s}_1}$$

$$\ldots$$

$$\frac{\vec{a}_n\ :\ \vec{A}_n \quad y_{n1}\ :\ \mathsf{D}\ r_{n1}\ \ldots\ y_{nk}\ :\ \mathsf{D}\ r_{nk}}{\mathsf{c}_n\ \vec{a}_n\ \vec{y}_n\ :\ \mathsf{D}\ \vec{s}_n}$$

Figure 2.6: Inductive family declaration

A recursive argument may also be higher order, although figure 2.6 does not show this for the sake of clarity (i.e., it may be a function which computes a recursive argument, rather than simply a recursive argument), provided that it satisfies a condition which ensures that

computation over the datatype will terminate. This condition, known as **strict positivity**, states that if an argument to a constructor of a family D has type $\vec{T} \to \mathsf{D}\,\vec{s}$, then an instance of D may not occur in $\vec{T}$.

Dybjer's presentation of inductive families [Dyb94] also identifies the parameters of a datatype; in EPIGRAM we do not require the programmer to identify the parameters explicitly but rather look for values which cannot change across the structure. The $\vec{s}$ are the indices and parameters of the datatype; these may be computed from or predicated on the non-recursive arguments.

2.1.3 Elimination Rules

When we declare an inductive family D, we give the constructors which explain how to build objects in that family. Along with this, the machine generates an **elimination operator** D-**Elim** (the type of which we call the **elimination rule**) and corresponding reductions, which we call ι-**schemes**. These describe and implement the allowed reduction and recursion behaviour of terms in the family. The method for constructing elimination operators is well documented, in particular by [Dyb94, Luo94, McB00a].

Like [McB00a] I will give ι-schemes in pattern matching form. The general form of an elimination rule and its associated ι-schemes is shown in figure 2.7. Elimination rules reduce when they are fully applied and the target is in canonical form; we call this ι-**reduction**. The arguments to the elimination rule are as follows, using the nomenclature of [McB00a, MM04b]:

- x is the **target**, preceded by its parameters and indices, $\vec{i}$. The target is the object to be eliminated by the rule, and corresponds to the scrutinee of a case expression in a traditional functional language.
- P is the **motive** of the elimination. The motive is a function which computes the return type of the elimination from the target. The motive allows an elimination to return a different type depending on the value of the target, and hence distinguishes an elimination rule from a typical fold operator, where the return type is a polymorphic type variable.
- m_{c} is a **method** for the case of the constructor c. The method for c is the reduction chosen on elimination if the target is headed by the constructor c. The function takes arguments for each argument to c, and for each recursive argument y_i to c it takes an extra argument representing the value of the recursive call to D-**Elim** with y as the target.

Remark: We call an elimination operator applied to a target an **eliminator**. While in most presentations the arguments to an elimination operator are ordered *motive, methods, target*, we choose to put the target first (preceded by its parameters and indices, as it depends

$$
\begin{array}{lll}
\mathsf{D}\textbf{-Elim} : & \forall \vec{i}:\vec{I}.\, \forall x:\mathsf{D}\,\vec{i}. & \text{(target)} \\
& \forall P:\forall \vec{i}:\vec{I}.\, \mathsf{D}\,\vec{i} \to \star. & \text{(motive)} \\
& \forall m_{\mathsf{c}} : \ \forall \vec{a}:\vec{A}.\ \forall y_1:\mathsf{D}\,\vec{r}_1.\ \ \ldots\ \ \forall y_j:\mathsf{D}\,\vec{r}_j. & \\
& \qquad\qquad P\,\vec{r}_1\,y_1 \to\ \ldots \to\ P\,\vec{r}_j\,y_j \to\ P\,\vec{s}\,(\mathsf{c}\,\vec{a}\,\vec{y}). & \text{(methods)} \\
& \ldots & \\
& P\,\vec{i}\,x & \\
\mathsf{D}\textbf{-Elim}\,\vec{s}\,(\mathsf{c}\,\vec{a}\,\vec{y})\,P\,\vec{m} & \rightsquigarrow\ m_{\mathsf{c}}\,\vec{a}\,\vec{y}\,(\mathsf{D}\textbf{-Elim}\,\vec{r}_1\,y_1\,P\,\vec{m})\ \ldots\ (\mathsf{D}\textbf{-Elim}\,\vec{r}_j\,y_j\,P\,\vec{m}) &
\end{array}
$$

Figure 2.7: Elimination rule for D, with ι-scheme for c

on them) to support EPIGRAM's notion of eliminators for pattern matching, which we will see in section 2.2.2.

As an example, the elimination rule for $\mathbb{N}$ is as follows:

$\mathbb{N}$**-Elim** :	$\forall n:\mathbb{N}.$	Target
	$\forall P:\mathbb{N} \to \star.$	Motive
	$\forall m_0:P\,0.$	Method for 0
	$\forall m_{\mathsf{s}}:\forall k:\mathbb{N}.\forall ih:P\,k.P\,(\mathsf{s}:k).$	Method for s
	$P\,n$	Return type (motive instance)

The ι-schemes for $\mathbb{N}$**-Elim** which implement this elimination rule are given in pattern matching form as follows:

$$
\begin{array}{l}
\mathbb{N}\textbf{-Elim}\ \ 0\ \ P\,m_0\,m_{\mathsf{s}} \ \rightsquigarrow\ m_0 \\
\mathbb{N}\textbf{-Elim}\,(\mathsf{s}\,k)\,P\,m_0\,m_{\mathsf{s}} \ \rightsquigarrow\ m_{\mathsf{s}}\,k\,(\mathbb{N}\textbf{-Elim}\,k\,P\,m_0\,m_{\mathsf{s}})
\end{array}
$$

A simple example of a function which can be implemented in terms of this elimination rule is **plus**, defined as follows:

$$
\begin{array}{l}
\textbf{plus} \ : \ \forall n, m:\mathbb{N}.\,\mathbb{N} \\
\textbf{plus} \mapsto \lambda n, m:\mathbb{N}.\,\mathbb{N}\textbf{-Elim}\,n\,(\lambda n:\mathbb{N}.\,\mathbb{N})\,m\,(\lambda k:\mathbb{N}.\,\lambda ih:\mathbb{N}.\,\mathsf{s}\,ih)
\end{array}
$$

This is defined by recursion over the first argument n. When n is zero, the return value is m. When $n = \mathsf{s}\,k$ for some k we get an induction hypothesis ih which tells us the value of the recursive call (**plus** k n). In this case, we return the successor of the recursive call, s ih.

For a datatype where a parameter does not change across the whole structure, we can lift out the parameter from the arguments to the motive and methods. For example, the elimination rule for List does not pass A as an argument to the methods, since A does not change:

List-Elim :	$\forall A{:}\star.$	Parameter
	$\forall l{:}\mathsf{List}\ A.$	Target
	$\forall P{:}\mathsf{List}\ A \to \star.$	Motive
	$\forall m_{\mathsf{nil}}{:}P\ (\mathsf{nil}\ A).$	Method for nil
	$\forall m_{\mathsf{cons}}{:}\forall x:A.\ \forall xs{:}\mathsf{List}\ A.\ \forall ih{:}P\ xs.P\ (\mathsf{cons}\ A\ x\ xs).$	Method for cons
	$P\ l$	Return type

$$\textbf{List-Elim}\ A\ \ (\mathsf{nil}\ A)\ \ P\ m_{\mathsf{nil}}\ m_{\mathsf{cons}} \rightsquigarrow m_{\mathsf{nil}}$$

$$\textbf{List-Elim}\ A\ (\mathsf{cons}\ A\ x\ xs)\ P\ m_{\mathsf{nil}}\ m_{\mathsf{cons}} \rightsquigarrow m_{\mathsf{cons}}\ x\ xs\ (\textbf{List-Elim}\ A\ xs\ P\ m_{\mathsf{nil}}\ m_{\mathsf{cons}})$$

Recall that all arguments are kept explicit in TT, hence the A appears as an argument to nil and cons in this elimination rule. The elimination rule for Vect lifts the parameter A out of the motive and methods, but passes the length index through as it does change across the structure:

Vect-Elim :	$\forall A{:}\star.$	Parameter
	$\forall n{:}\mathbb{N}.$	Index
	$\forall v{:}\mathsf{Vect}\ A\ n.$	Target
	$\forall P{:}\forall n{:}\mathbb{N}.\ \mathsf{Vect}\ A\ n \to \star.$	Motive
	$\forall m_{\epsilon}{:}P\ 0\ (\epsilon\ A).$	Method for ϵ
	$\forall m_{::}{:}\forall k{:}\mathbb{N}.\ \forall x:A.\ \forall xs{:}\mathsf{Vect}\ A\ k.$	Method for ::
	$\quad \forall ih{:}P\ k\ xs.P\ (\mathsf{s}\ k)\ (::\ A\ k\ x\ xs).$	
	$P\ n\ v$	Return type

$$\textbf{Vect-Elim}\ A\ \ 0\ \ \ (\epsilon\ A)\ \ \ P\ m_{\epsilon}\ m_{::} \rightsquigarrow m_{\epsilon}$$

$$\textbf{Vect-Elim}\ A\ (\mathsf{s}\ k)\ (::\ A\ k\ x\ xs)\ P\ m_{\epsilon}\ m_{::} \rightsquigarrow m_{::}\ k\ x\ xs\ (\textbf{Vect-Elim}\ A\ k\ xs\ P\ m_{\epsilon}\ m_{::})$$

EPIGRAM also generates non-recursive eliminators (case analysis rules) for each type. These are the same as the recursive eliminators except that there are no additional arguments in the methods for the result of recursive calls. For $\mathbb{N}$, this would be as follows:

$\mathbb{N}$-Case :	$\forall n{:}\mathbb{N}.$	Target
	$\forall P{:}\mathbb{N} \to \star.$	Motive
	$\forall m_0{:}P\ 0.$	Method for 0
	$\forall m_{\mathsf{s}}{:}\forall k{:}\mathbb{N}.P\ (\mathsf{s}\ k).$	Method for s
	$P\ n$	Return type

It is not difficult to see how to prove this from **$\mathbb{N}$-Elim**, simply by not using the inductive hypotheses in the method calls. However, in practice, it is more efficient to define it directly as it removes a level of indirection. The general scheme for **D-Case** is shown in figure 2.8.

2.1.4 Equality

Thanks to the Curry Howard isomorphism, inductive families can represent not only data, but also proofs of propositions. An important such proposition is propositional equality,

$$
\begin{array}{lll}
\textbf{D-Case} : & \forall \vec{i}:\vec{I}.\, \forall x:\mathsf{D}\ \vec{i}. & \text{(target)} \\
& \forall P:\forall \vec{i}:\vec{I}.\, \mathsf{D}\ \vec{i} \to \star. & \text{(motive)} \\
& \left.\begin{array}{l} \forall m_{\mathsf{c}} : \ \forall \vec{a}:\vec{A}. \quad \forall y_1:\mathsf{D}\ \vec{r}_1. \quad \ldots \quad \forall y_j:\mathsf{D}\ \vec{r}_j.\, P\ \vec{s}\ (\mathsf{c}\ \vec{a}\ \vec{y}). \\ \ldots \end{array}\right\} & \text{(methods)} \\
& P\ \vec{i}\ x & \\
\textbf{D-Case}\ \vec{s}\ (\mathsf{c}\ \vec{a}\ \vec{y})\ P\ \vec{m} & \rightsquigarrow \quad m_{\mathsf{c}}\ \vec{a}\ \vec{y} \ : \ P\ \vec{s}\ (\mathsf{c}\ \vec{a}\ \vec{y}) &
\end{array}
$$

Figure 2.8: Non-recursive Elimination rule for D, with ι-scheme for c

which is defined using Martin-Löf's identity type declared as in figure 2.9 (using an infix notation for the type constructor $=$).

$$
\begin{array}{l}
\underline{\text{data}} \quad \dfrac{A : \star \quad a, b : A}{a = b : \star} \quad \underline{\text{where}} \quad \dfrac{A : \star \quad a : A}{\mathsf{refl}\ a : a = a} \\
=\textbf{-elim} : \forall A:\star.\, \forall a:A.\, \forall b:A. \\
\qquad \forall x:a = b.\, \forall P:a = b \to \star. \\
\qquad \forall m_{\mathsf{refl}}:P\ (\mathsf{refl}\ A\ a).\ P\ x \\
=\textbf{-elim}\ A\ a\ a\ (\mathsf{refl}\ A\ a)\ P\ m_{\mathsf{refl}} \ \rightsquigarrow \ m_{\mathsf{refl}}\ A\ a
\end{array}
$$

Figure 2.9: Martin-Löf's Equality

We can declare an equality between any two values in the same type, but we can only construct a proof of equality between two values which are equal. The constructor application $\mathsf{refl}\ a$ is a proof that $a = a$.

This equality relation is sufficient to describe equality between objects of the same type. However, with inductive families it is often useful to be able to describe equality between potentially different types. For example, it is impossible to declare an equality between two Vects with different indices, even if those indices are propositionally equal. It is intuitively clear that the following proposition (that :: respects equality) holds, however the definition of propositional equality we have is insufficient to express the theorem; there are type errors because the vectors involved have different indices.

$$
\begin{array}{lll}
\textbf{wrong} : & \forall A:\star.\, \forall m:\mathbb{N}.\, \forall x:A.\, \forall xs:\mathsf{Vect}\ A\ (\mathsf{s}\ m). & \\
& \qquad \forall n:\mathbb{N}.\, \forall y:A.\, \forall ys:\mathsf{Vect}\ A\ (\mathsf{s}\ n). & (\times) \\
& \qquad m = n \to x = y \to xs = ys \to (::\ A\ m\ x\ xs) \ = (::\ A\ n\ y\ ys) &
\end{array}
$$

Instead, we use McBride's heterogeneous definition of equality[1] [McB00a], declared as in figure 2.10. Using this definition, we can declare an equality between two values in *different* types, but we can only construct a proof of an equality between two identical values in the *same* type. Note that we do not declare this family with a data declaration but rather add the type formation and elimination rules to the core type theory as axioms, because

[1] McBride calls this "John Major" equality.

the default elimination rule given by the D-**Elim** scheme would not be suitable. The rule generated for a data declaration would be abstracted over both types A and B but we only want to be able to apply the rule when the types A and B are the same. Henceforth, $=$ is this heterogeneous equality.

$$\frac{A, B : \star \quad a : A \quad b : B}{a = b : \star} \qquad \frac{A : \star \quad a : A}{\mathsf{refl}\ a : a = a}$$

$$\begin{aligned} &=\textbf{-elim} : \forall A{:}\star.\, \forall a{:}A.\, \forall b{:}A. \\ &\qquad \forall x{:}a = b.\, \forall P{:}a = b \to \star. \\ &\qquad \forall m_{\mathsf{refl}} : P\ (\mathsf{refl}\ A\ a).\ P\ x \\ &=\textbf{-elim}\ A\ a\ a\ (\mathsf{refl}\ A\ a)\ P\ m_{\mathsf{refl}} \rightsquigarrow m_{\mathsf{refl}}\ A\ a \end{aligned}$$

Figure 2.10: Heterogeneous Equality for Dependent Types

2.1.5 Properties of TT

There are several metatheoretic properties which hold for UTT as shown by Goguen [Gog94], and hence we assume to hold for TT. These are:

- **Church Rosser.** If two terms s and t are convertible, then s and t have a common reduct, up to syntactic equivalence ($\equiv$).

  if $\Gamma \vdash s \simeq t$
  then there exists r, r' such that
  $\Gamma \vdash s \rhd^* r$ and $\Gamma \vdash t \rhd^* r'$ and $\Gamma \vdash r \equiv r'$

- **Strong normalisation.** All well-typed terms in TT are strongly normalising.

  if $\Gamma \vdash t : T$ then $\mathrm{SN}(t)$

- **Subject reduction.** If s reduces to t, then s and t have the same type.

$$\frac{\Gamma \vdash s : T \quad \Gamma \vdash s \rhd^* t}{\Gamma \vdash t : T}$$

- **Uniqueness of types.** A term only has one type, so if the same term is shown to have two types with respect to the context, then those two types must be convertible.

$$\frac{\Gamma \vdash s : T \quad \Gamma \vdash s : T'}{\Gamma \vdash T \simeq T'}$$

- **Adequacy.** In the empty context (that is, in the absence of any assumptions) the weak head-normal form of a term t is a constructor form.

$$\boxed{\begin{array}{l}\text{if } \vdash t : \mathsf{D}\ \vec{s} \\ \text{then } \textsc{whnf}(t) \equiv \mathsf{c}\ \vec{t} \text{ for some } \mathsf{c}, \vec{t}\end{array}}$$

Remark: η-contraction can cause problems with the metatheory, particularly with regard to the Church Rosser property. The counterexample which shows that Church Rosser fails is as follows (with $A \not\preceq B$):

$\lambda x\!:\!A.\ (\lambda x\!:\!B.\ x)x$

This reduces to $\lambda x : A.\ x$ by β-reduction, and $\lambda x : B.\ x$ by η-reduction. Of course, this term is not well-typed, but we still have a problem because Church Rosser is often shown by erasing types and showing the property for the untyped terms. Nevertheless, we are only interested in the well-typed terms, and the work of Geuvers [Geu93] and Jay and Ghani [JG95] leads us to believe that Church Rosser does hold for TT with η.

2.1.6 Universe Levels and Cumulativity

$$\boxed{\begin{array}{cc} \dfrac{\Gamma \vdash x \simeq y}{\Gamma \vdash x \preceq y} & \dfrac{\Gamma \vdash x \preceq y \quad \Gamma \vdash y \preceq z}{\Gamma \vdash x \preceq z} \\[2ex] \dfrac{}{\Gamma \vdash \star_n \preceq \star_{n+1}} & \dfrac{\Gamma \vdash S_1 \preceq S_2 \quad \Gamma; x : S_1 \vdash T_1 \preceq T_2}{\Gamma \vdash \forall x\!:\!S_1.\ T_1 \preceq \forall x\!:\!S_2.\ T_2} \end{array}}$$

Figure 2.11: Cumulativity

In TT, we have an infinite hierarchy of predicative universes, i.e $\star_n : \star_{n+1}$ for $n \geq 0$. In [MM04b], the core type theory also has cumulativity (figure 2.11), which allows us to embed values in higher universes — so if $A : \star_n$, we also have $A : \star_{n+k}$ for $k > 0$. The problem with defining cumulativity rules for the type theory, however, is that it breaks the uniqueness of types property. With cumulativity we can, for example, say the following:

$\mathbb{N} : \star_0$

$\mathbb{N} : \star_1$

From uniqueness of types, we could then conclude that $\star_0 = \star_1$, which is clearly not true. The uniqueness of types property will be crucial to later parts of this thesis, and so we do not have cumulativity in the core type theory. Nevertheless, there are programs for which cumulativity is useful. An example will be given in section 4.6; at that point I will suggest, in section 4.6.2, a solution to the cumulativity problem based on Tarski style universes, as implemented in Plastic [CL01].

2.1.7 TT Examples

To show how the core type theory is used, let us consider some small example programs. We have already seen **plus**, defined by elimination of its first argument:

$$\begin{aligned}&\textbf{plus} \;:\; \forall n, m{:}\mathbb{N}.\, \mathbb{N}\\ &\textbf{plus} \mapsto \lambda n, m{:}\mathbb{N}.\, \mathbb{N}\textbf{-Elim}\; n\; (\lambda n{:}\mathbb{N}.\, \mathbb{N})\; m\; (\lambda k{:}\mathbb{N}.\, \lambda ih{:}\mathbb{N}.\, \mathsf{s}\; ih)\end{aligned}$$

A more complex example is the append function on lists; this is similar in structure to **plus**. If the first list xs is empty, we simply return the second list ys. Otherwise, if the first list is of the form $\mathsf{cons}\; z\; zs$, we return $(\mathsf{cons}\; z\; (\textbf{append}\; zs\; ys))$, where the recursive call is represented by the inductive hypothesis ih.

$$\begin{aligned}\textbf{append} \;:\;& \forall A{:}\star.\, \forall xs, ys{:}\mathsf{List}\; A.\, \mathsf{List}\; A\\ \textbf{append} \mapsto\;& \lambda A{:}\star.\, \lambda xs, ys{:}\mathsf{List}\; A.\\ &\quad \mathsf{List}\textbf{-Elim}\; A\; xs\; (\lambda xs{:}\mathsf{List}\; A.\, \mathsf{List}\; A)\; ys\\ &\qquad (\lambda z{:}A.\, \lambda zs{:}\mathsf{List}\; A.\, \lambda ih{:}\mathsf{List}\; A.\, \mathsf{cons}\; A\; z\; ih)\end{aligned}$$

In Chapter 1 we considered the type safety of vector append as compared with list append. The definition of vector append in the core type theory is of the same structure as list append, although it does raise some issues about typechecking. The definition is as follows:

$$\begin{aligned}\textbf{vappend} \;:\;& \forall A{:}\star.\, \forall n, m{:}\mathbb{N}.\, \forall xs{:}\mathsf{Vect}\; A\; n.\, \forall ys{:}\mathsf{Vect}\; A\; m.\, \mathsf{Vect}\; A\; (\textbf{plus}\; n\; m)\\ \textbf{vappend} \mapsto\;& \lambda A{:}\star.\, \lambda n, m{:}\mathbb{N}.\, \lambda xs{:}\mathsf{Vect}\; A\; n.\, \lambda ys{:}\mathsf{Vect}\; A\; m.\\ &\quad \mathsf{Vect}\textbf{-Elim}\; A\; n\; xs\; (\lambda n{:}\mathbb{N}.\, \lambda xs{:}\mathsf{Vect}\; A\; n.\, \mathsf{Vect}\; A\; (\textbf{plus}\; n\; m))\; ys\\ &\qquad (\lambda k{:}\mathbb{N}.\, \lambda z{:}A.\, \lambda zs{:}\mathsf{Vect}\; A\; k.\, \lambda ih{:}\mathsf{Vect}\; A\; (\textbf{plus}\; k\; m).\\ &\qquad\quad ::\; A\; (\textbf{plus}\; k\; m)\; z\; ih)\end{aligned}$$

The issues with typechecking are based on the expected return types of the methods of Vect-**Elim**. The problems are:

- In the ϵ case, we expect a return type of $\mathsf{Vect}\; A\; (\textbf{plus}\; 0\; n)$. However, the return value ys has type $\mathsf{Vect}\; A\; n$.
- In the :: case, we expect a return type of $\mathsf{Vect}\; A\; (\textbf{plus}\; (\mathsf{s}\; k)\; n)$, however the return value of $::\; A\; (\textbf{plus}\; k\; n)\; z\; ih$ has type $\mathsf{Vect}\; A\; (\mathsf{s}\; (\textbf{plus}\; k\; n))$.

So why does the given definition of **vappend** typecheck? This definition typechecks because in conversion checking we are comparing normal forms (or weak head-normal forms) of terms, rather than the syntactic forms. For example, in checking the ϵ case, the normal form of $\textbf{plus}\; 0\; n$ is n — this is reducible because the first argument to **plus**, which is the one we pass to the elimination rule, is in canonical form (i.e. headed by a constructor). Hence, the ϵ case typechecks. The :: case typechecks for similar reasons. This is an important point about typechecking dependently typed programs — syntactic equality checking is not enough; we must reduce to normal form (or use some other method of conversion checking based on

reduction) before checking equality, hence why without strong normalisation typechecking becomes undecidable.

2.1.8 Labelled Types

Labelled types, introduced in [MM04b] are an extension to the core type theory which allow terms to be "labelled" by another term which describes its meaning. We extend the TT language of section 2.1 with syntax for labelled types as in figure 2.12. The typing and contraction (called ρ-reduction) rules for these syntax extensions are given in figure 2.13 and figure 2.14.

$$
\begin{array}{lcl}
t & ::= & \ldots \\
 & | & \langle l \ : \ t \rangle \\
 & | & \underline{\text{call}}\ \langle l \rangle\ t \\
 & | & \underline{\text{return}}\ t \\
 & & \\
l & ::= & n\ \vec{t} \quad \text{(A name applied to zero or more terms)}
\end{array}
$$

Figure 2.12: Extensions to TT for labelled types

$$
\frac{\Gamma \vdash T \ : \ \star_n}{\Gamma \vdash \langle l \ : \ T \rangle \ : \ \star_n}\ \text{Label}
$$

$$
\frac{\Gamma \vdash t \ : \ T}{\Gamma \vdash \underline{\text{return}}\ t \ : \ \langle l \ : \ T \rangle}\ \text{Return}
$$

$$
\frac{\Gamma \vdash t \ : \ \langle l \ : \ T \rangle}{\Gamma \vdash \underline{\text{call}}\ \langle l \rangle\ t \ : \ T}\ \text{Call}
$$

Figure 2.13: Typing and rules for labelled types

$$
\rho\text{-contraction} \qquad \Gamma \vdash \underline{\text{call}}\ \langle l \rangle\ (\underline{\text{return}}\ t) \rightsquigarrow t
$$

Figure 2.14: Contraction rule for labelled types

EPIGRAM programs are defined interactively, with metavariables (or holes, $\Box \ : \ T$) standing for parts of programs which have not yet been written, and their type. Labelled types allows the types of holes to be more informative; the system implicitly inserts a label into the return type of a function. So, if we are defining **plus** interactively, the type is

$$\textbf{plus} \ : \ \forall n{:}\mathbb{N}.\forall m{:}\mathbb{N}.\langle \textbf{plus}\ n\ m \ : \ \mathbb{N} \rangle$$

Now the types of recursive calls and return values give us some useful information, namely their *meaning* as well as their type. An incomplete definition of **plus**, with metavariables in place of the cases, is labelled as follows:

$$\begin{aligned}\textbf{plus} \;=\; & \lambda n, m{:}\mathbb{N}. \\ & \mathbb{N}\textbf{-Elim}\; n\; (\lambda n{:}\mathbb{N}.\langle \textbf{plus}\; n\; m \;:\; \mathbb{N}\rangle) \\ & \Box \;:\; \langle \textbf{plus}\; 0\; m \;:\; \mathbb{N}\rangle \\ & \Box \;:\; \forall k{:}\mathbb{N}.\forall ih{:}\langle \textbf{plus}\; k\; m \;:\; \mathbb{N}\rangle.\langle \textbf{plus}\; (\mathsf{s}\; k)\; m \;:\; \mathbb{N}\rangle\end{aligned}$$

Labelling the return type in this way tells us that when n is 0, the return value of the function is the value of **plus** $0\; m$, and when n is $\mathsf{s}\; k$, the recursive call we get is the value of **plus** $k\; m$ and the return value of the function is **plus** $(\mathsf{s}\; k)\; m$.

The purpose of the return keyword is to create a label, rather than a $\mathbb{N}$. Then, since the inductive hypothesis is now a label rather than a $\mathbb{N}$, the application of the inductive hypothesis is made with the call keyword.

A more detailed account of labelled types and their use in elaborating EPIGRAM terms is given in [MM04b]. I will in general leave labels out of terms — it is a simple transformation to change TT terms with labels to TT terms without labels. Eventually, I will use these labels to assist in efficient compilation. The details of this optimisation will be described in Chapter 6.

2.2 Programming in Epigram

This thesis concentrates on the efficient compilation of EPIGRAM programs and we will see many examples of EPIGRAM programs and their elaborated forms. Rather than writing programs directly in TT, EPIGRAM is a high level notation for programming which makes programs more readable and easier to develop. This section gives a tutorial introduction to programming with inductive families in the high level EPIGRAM notation, building on the core type theory of TT. For a complete specification of EPIGRAM see [MM04b]; a more comprehensive tutorial is given in [McB04].

2.2.1 Basic Notation

Data Type Declarations

Inductive datatypes and families are declared using a data declaration, as we have already seen in section 2.1.2:

$$\underline{\text{data}} \quad \frac{\vec{s} \ : \ \vec{S}}{\mathsf{D}\ \vec{s} \ : \ \star}$$

$$\underline{\text{where}} \quad \frac{\vec{a}_1 \ : \ \vec{A}_1 \quad \vec{y}_{11} \ : \ \mathsf{D}\ \vec{r}_{11} \ \ldots \ \vec{y}_{1j} \ : \ \mathsf{D}\ \vec{r}_{1j}}{\mathsf{c}_1\ \vec{a}_1\ \vec{y}_1 \ : \ \mathsf{D}\ \vec{s}_1}$$

$$\ldots$$

$$\frac{\vec{a}_n \ : \ \vec{A}_n \quad \vec{y}_{n1} \ : \ \mathsf{D}\ \vec{r}_{n1} \ \ldots \ \vec{y}_{nk} \ : \ \mathsf{D}\ \vec{r}_{nk}}{\mathsf{c}_n\ \vec{a}_n\ \vec{y}_n \ : \ \mathsf{D}\ \vec{s}_n}$$

The indices of each constructor may differ — such as in the Vect family (see section 2.1.2) where the constructors for the empty and non-empty vectors target different and disjoint branches of the family — so a Haskell style data declaration is insufficient to express many families.

The recursive arguments $\vec{y}$ may be higher order provided that they satisfy the strict positivity condition (see section 2.1.2). When a structure is strictly positive, we know that the recursive arguments can only represent smaller structures.

Function Definitions

A function definition takes the form of a type signature followed by the function body. Functions, like inductive datatypes, are declared in a natural deduction style, with the premises above the line (i.e., the argument types) and the conclusion below the line (i.e., the return type). This gives a convenient notation for dependent types because argument names can appear in the type of later arguments, and in the return type of the function.

$$\underline{\text{let}} \quad \frac{\vec{s} \ : \ \vec{S}}{\mathbf{f}\ \vec{s} \ : \ T} \qquad \mathbf{f}\ \vec{s} \ \mapsto \ \{body\}$$

In this declaration, $\vec{S}$ denotes the types of the arguments, and T the return type of the function. There may also be implicit arguments, as with data type declarations, whose values can be inferred from the given arguments $\vec{s}$. The type of elaborated $\mathbf{f}$ in the core TT is:

$$\mathbf{f} \ : \ \forall \vec{i} : \vec{I}.\, \forall \vec{s} : \textsc{elab}(\vec{S}).\ \textsc{elab}(T)$$

(Where $\textsc{elab}(p)$ denotes the elaboration of a high level program p, and $\vec{i} \ : \ \vec{I}$ are the implicit arguments.) We use the $\forall \vec{x} : \vec{S}$ notation, with explicit names for the arguments, since dependent types allow the $\vec{x}$ to occur in the return type of $\mathbf{f}$, T, in much the same way as the $\vec{x}$ are allowed to occur in the body of $\mathbf{f}$. Just as λ is a binder for function bodies in λ-calculus, the $\forall$ symbol is a binder for function types.

Finally, for function types $\forall x : S.\ T$, where x is not free in S we can use the more concise notation which will be familiar to Haskell or ML programmers:

$$\mathbf{f} \ : \ S \rightarrow T$$

2.2.2 Programming with Elimination Rules

As we saw in section 2.1.2, constructors provide a means for creating objects of an inductive datatype, and elimination rules provide a means for deconstructing those objects. In TT, elimination rules are the only means for examining datatypes and so the high level notation provides a convenient means for applying elimination rules.

Earlier, we saw the **plus** function on natural numbers defined in TT as follows:

$$\mathbf{plus} \ : \ \forall n, m{:}\mathbb{N}.\ \mathbb{N}$$
$$\mathbf{plus} = \lambda n, m{:}\mathbb{N}.\ \mathbb{N}\text{-}\mathbf{Elim}\ n\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ m\ (\lambda k{:}\mathbb{N}.\ \lambda ih{:}\mathbb{N}.\ (\mathsf{s}\ ih))$$

Let us consider how the EPIGRAM system allows us to define this function using high level notation, in an interactive style. We begin by declaring the type of **plus**:

$$\underline{\text{let}} \quad \frac{n, m \ : \ \mathbb{N}}{\mathbf{plus}\ n\ m \ : \ \mathbb{N}}$$

With this, EPIGRAM's interactive development system gives us a template for a function definition, with a "hole" for its body, $\Box$, indicating its type:

$$\mathbf{plus}\ n\ m \quad \Box \ : \ \mathbb{N}$$

We would like to define this function by recursion on the first argument, m, so we tell EPIGRAM to apply the elimination rule $\mathbb{N}$-**Elim** to n. The "by" operator ($\Leftarrow$) takes as its right hand side an eliminator (i.e. an elimination rule applied to its target). As a shorthand, we can access the appropriate eliminator for a term x with the notation $\underline{\text{elim}}\ x$. Applying the elimination rule gives two possible cases for n:

$$\underline{\text{let}} \quad \frac{n, m \ : \ \mathbb{N}}{\mathbf{plus}\ n\ m \ : \ \mathbb{N}}$$
$$\mathbf{plus} \quad n \quad m \ \Leftarrow \underline{\text{elim}}\ n$$
$$\mathbf{plus}\ \ 0 \quad m \ \mapsto \Box \ : \ \mathbb{N}$$
$$\mathbf{plus}\ (\mathsf{s}\ k)\ m \ \mapsto \Box \ : \ \mathbb{N}$$

The details of the elimination rule are hidden from the programmer; however, behind the scenes the system is building a term in TT, complete with labelled types. The labelled type of **plus** is:

$$\mathbf{plus} \ : \ \forall n, m{:}\mathbb{N}.\ \langle \mathbf{plus}\ n\ m \ : \ \mathbb{N} \rangle$$

The system knows if a recursive call is allowed by searching through the bindings in the context and checking for a term with a labelled type which matches the recursive call — this term is an inductive hypothesis. For this function, **plus** k m is an allowed recursive call in the $\mathsf{s}\ k$ case, since the type of the inductive hypothesis is $\langle \mathbf{plus}\ k\ m \ : \ \mathbb{N} \rangle$. We can complete the definition as follows:

$$\underline{\text{let}} \quad \frac{n, m \ : \ \mathbb{N}}{\mathbf{plus}\ n\ m \ : \ \mathbb{N}}$$

$$\begin{array}{lll} \mathbf{plus} & n & m \Leftarrow \underline{\text{elim}}\ n \\ \mathbf{plus} & 0 & m \mapsto m \\ \mathbf{plus}\ (\mathsf{s}\ k) & m & \mapsto \mathsf{s}\ (\mathbf{plus}\ k\ m) \end{array}$$

We therefore use elimination rules to generate readable pattern matching style functions. EPIGRAM programs are tree structured in that a call to an elimination rule breaks the program down into sub problems; we reflect this by indenting the program where there is an appeal to an elimination rule.

Remark: Using this approach, pattern matching is not hard-wired. Instead, there is a pattern matching style interface for programming with eliminators. Also, the interactive approach to program development means that the programmer does not have to type in the whole definition; the appropriate patterns are given by the elimination rule. This is particularly useful where case analysis on one argument tells us something about other arguments (case analysis on a Vect tells us which constructor was used to build its length index, for example). We will see some examples of this later, in particular in sections 2.3.2 and 2.3.3.

2.2.3 Impossible Cases

One of the most important features of the elaboration process is the elimination of cases which can be shown to be impossible through types. For example, consider how we might write a function which returns the tail of a non-empty vector. We declare the type, and get a template for the function as follows:

$$\underline{\text{let}} \quad \frac{v : \mathsf{Vect}\ A\ (\mathsf{s}\ n)}{\mathbf{vTail}\ v \ : \ \mathsf{Vect}\ A\ n}$$

$$\mathbf{vTail}\ v \ \mapsto \ \Box \ : \ \mathsf{Vect}\ A\ n$$

Clearly, the empty vector is not a valid input to this function — the type specifies that the input must have a non-zero length. As a result, when we declare that we wish to write the function by **Vect-Case** v (using the notation $\underline{\text{case}}\ v$ to access the non-recursive elimination rule), all the system gives us is the case for the non-empty vector:

$$\underline{\text{let}} \quad \frac{v : \mathsf{Vect}\ A\ (\mathsf{s}\ n)}{\mathbf{vTail}\ v \ : \ \mathsf{Vect}\ A\ n}$$

$$\begin{array}{lll} \mathbf{vTail} & v & \Leftarrow \underline{\text{case}}\ v \\ \mathbf{vTail}\ (a{::}v) & & \mapsto \ \Box \ : \ \mathsf{Vect}\ A\ n \end{array}$$

Completing this definition is straightforward:

$$\underline{\text{let}} \quad \frac{v : \mathsf{Vect}\ A\ (\mathsf{s}\ n)}{\mathbf{vTail}\ v \ : \ \mathsf{Vect}\ A\ n}$$

$$\begin{array}{lll} \mathbf{vTail} & v & \Leftarrow \underline{\text{case}}\ v \\ \mathbf{vTail}\ (a{::}v) & & \mapsto \ v \end{array}$$

By examining the input type Vect A (s n) we see that the empty vector ϵ is an impossible case, since it has the type Vect A 0 which does not unify with the input type. This much is clear for us to see, but how does the elaboration mechanism know that **vTail** $(a::v)$ is the only case and how does it produce a valid term in TT?

For this we use a technique described in [McB00b], elimination with a motive. To define a function in this way, the machine inserts equational constraints into the motive expressing the allowed values of the indices. This requires an empty type and a trivial type. The empty type is a type with no constructors:

$$\underline{\text{data}} \quad \frac{}{\mathsf{False} : \star} \quad \underline{\text{where}}$$

Since this type has no constructors, the elimination rule has no methods. As a result if we have an element of the empty type we can prove anything by passing any motive to the elimination rule.

$$\mathsf{False\text{-}Case} \ : \ \forall f : \mathsf{False}. \forall P : \mathsf{False} \rightarrow \star. P\ f$$

The trivial type has one constructor:

$$\underline{\text{data}} \quad \frac{}{\mathsf{True} : \star} \quad \underline{\text{where}} \quad \frac{}{() : \mathsf{True}}$$

The technique for eliminating impossible cases revolves around showing that the case is impossible, thereby producing an element of the empty type and returning a value of the appropriate type with False-**Case**. Checking impossible cases like this can be done automatically by elaboration and if a case is shown to be impossible it need not be written down. The elaboration of **vTail** is shown in detail in Appendix A. The result of this elaboration is shown in figure 2.15. Here, I have separated this into several functions for readability; in practice the system generates this as one definition.

2.2.4 Example — Vector lookup

Suppose we have a list, l, and an index, n, and we wish to retrieve the nth element of the list l. Traditionally, if we want to do this robustly, we might take the following steps:

1. Check the length of l.
2. Check that n is within the bounds of l.
   - If out of bounds, perform some error handling routine. In Haskell, we return $\perp$ in the error case, but this is not an option in EPIGRAM because of the strong normalisation property.
3. Perform the lookup.

The first two steps are potentially expensive, but if we leave them out we run the risk of a program error. Xi [Xi98] describes the use of constraints with dependent types to eliminate

$$
\begin{array}{ll}
\textbf{dMotive} : & \forall n{:}\mathbb{N}.\, \star \\
\textbf{dMotive} \mapsto & \lambda n{:}\mathbb{N}.\, \mathbb{N}\textbf{-Case}\ n\ (\forall n{:}\mathbb{N}.\, \star)\ \mathsf{False}\ (\lambda k{:}\mathbb{N}.\, \mathsf{True}) \\
\textbf{discriminate} : & \forall n{:}\mathbb{N}.\, \forall p{:}\mathsf{s}\ n\ =\ 0.\, \mathsf{False} \\
\textbf{discriminate} \mapsto & \lambda n{:}\mathbb{N}.\, \lambda p{:}\mathsf{s}\ n\ =\ 0. \\
 & \quad =\textbf{-elim}\ \mathbb{N}\ (\mathsf{s}\ n)\ p\ \textbf{dMotive}\ () \\
\textbf{emptyCase} : & \forall A{:}\star.\, \forall n{:}\mathbb{N}.\, (\mathsf{s}\ n\ =\ 0) \to \mathsf{Vect}\ A\ n \\
\textbf{emptyCase} \mapsto & \lambda A{:}\star.\, \lambda n{:}\mathbb{N}.\, \lambda p{:}\mathsf{s}\ n\ =\ 0. \\
 & \quad \mathsf{False}\textbf{-Elim}\ (\textbf{discriminate}\ n\ p)\ (\mathsf{Vect}\ A\ n) \\
\textbf{consCase} : & \forall A{:}\star.\, \forall n{:}\mathbb{N}.\, \forall k{:}\mathbb{N}.\, \mathsf{Vect}\ A\ k \to (\mathsf{s}\ n\ =\ \mathsf{s}\ k) \to \mathsf{Vect}\ A\ n \\
\textbf{consCase} \mapsto & \lambda A{:}\star.\, \lambda n{:}\mathbb{N}.\, \lambda k{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ k.\, \lambda p{:}k\ =\ n. \\
 & \quad =\textbf{-elim}\ \mathbb{N}\ k\ n\ (\textbf{S\_inj}\ k\ n\ (\textbf{eq\_sym}\ \mathbb{N}\ n\ k\ p))\ (\lambda n{:}\mathbb{N}.\, \mathsf{Vect}\ A\ n)\ v \\
\textbf{vTailAux} : & \forall n{:}\mathbb{N}.\, \forall A{:}\star.\, \forall k{:}\mathbb{N}.\, \forall v{:}\mathsf{Vect}\ A\ k.\, (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n \\
\textbf{vTailAux} \mapsto & \lambda n{:}\mathbb{N}.\, \lambda A{:}\star.\, \lambda k{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ k. \\
 & \quad \mathsf{Vect}\textbf{-Case}\ A\ k\ v \\
 & \quad (\lambda k{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ k.\, (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n) \\
 & \quad (\textbf{emptyCase}\ A\ n) \\
 & \quad (\lambda k{:}\mathbb{N}.\, \lambda a{:}A.\, \lambda v{:}\mathsf{Vect}\ A\ k.\, \textbf{consCase}\ A\ n\ k\ v) \\
\textbf{vTail} \mapsto & \lambda A{:}\star.\, \lambda n{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ (\mathsf{s}\ n). \\
 & (\lambda k{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ k. \\
 & \lambda P{:}\forall k{:}\mathbb{N}.\, \forall v{:}\mathsf{Vect}\ A\ k.\, (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n. \\
 & P\ (\mathsf{s}\ n)\ v\ (\mathsf{refl}\ (\mathsf{s}\ n))) \\
 & n\ v\ (\textbf{vTailAux}\ n\ A)
\end{array}
$$

Figure 2.15: Elaborated **vTail**

such bounds checks at run-time. Inductive families give us an alternative method. We begin by defining a family of finite sets. The finite sets, indexed over n, are sets with at most n elements and a natural use of this is to represent bounded numbers.

$$\underline{\text{data}} \quad \frac{n\ :\ \mathbb{N}}{\mathsf{Fin}\ n\ :\ \star}$$

$$\underline{\text{where}} \quad \frac{}{\mathsf{f0}\ :\ \mathsf{Fin}\ (\mathsf{s}\ n)} \qquad \frac{i\ :\ \mathsf{Fin}\ n}{\mathsf{fs}\ i\ :\ \mathsf{Fin}\ (\mathsf{s}\ n)}$$

We can see from the indices that it is not possible to create an element of Fin 0. To create such an object would be meaningless — Fin 0 is a set with no elements, corresponding to a type with no values.

The dependencies on Fin and Vect give us invariants which must hold in the definition of the lookup function. These invariants are verified at compile-time by the typechecker rather than at run-time by the run-time system. We declare the type of the lookup function with a let declaration:

$$\underline{\text{let}} \quad \frac{i\ :\ \mathsf{Fin}\ n \qquad v\ :\ \mathsf{Vect}\ A\ n}{\textbf{lookup}\ i\ v\ :\ A}$$

There are two extra arguments, n and A, which are left implicit as they can be inferred

from the types of i and v. There are some other constraints which we can infer just from the type:

- The value of n cannot be 0 in a well-typed application of **lookup**, in the empty context (i.e., when n is in canonical form). This is because it is impossible to create a canonical element of Fin 0. If n were equal to zero, then i would have to be of type Fin 0.
- As a result, the vector v must be non empty. This means that one possible error, that of looking up an element from an empty list *cannot happen* at run-time because attempting to call the function with an empty vector would be a *compile-time* error.

The function is written by recursion on i. If the value of i is zero then we return the first element in the list, otherwise we look in the tail of the list. I will again write the program by refinement, as directed by the EPIGRAM elaborator. The first step is to declare that we wish to write the program by recursion on i.

$$\begin{array}{lllll} \textbf{lookup} & i & v & \Leftarrow \underline{\text{elim}}\, i & \\ \textbf{lookup} & \mathsf{f0} & v & \mapsto \Box : A & \\ \textbf{lookup} & (\mathsf{fs}\, i) & v & \mapsto \Box : A & \end{array}$$

This gives us the possible patterns for i. The next step, for each subgoal, is case analysis on v. Here the elaborator establishes that the empty vector would violate the constraints in the type, as with the **vTail** function, and so we do not get a pattern for the empty vector. Note that giving two elimination rules on the right of $\Leftarrow$ means that the second rule will be applied immediately in *each case* generated by the first rule (c.f. the Then tactical in LEGO or sequencing with semicolon (;) in COQ).

$$\begin{array}{lllll} \textbf{lookup} & i & v & \Leftarrow \underline{\text{elim}}\, i \Leftarrow \underline{\text{case}}\, v & \\ \textbf{lookup} & \mathsf{f0} & (a :: v) & \mapsto a & \\ \textbf{lookup} & (\mathsf{fs}\, i) & (a :: v) & \mapsto \textbf{lookup}\, i\, v & \end{array}$$

We use **Vect-Case** rather than **Vect-Elim** because the recursion is on the finite set, rather than the list, so we do not need the recursive call on Vect.

The definition of **lookup** is now complete. Impossible cases were eliminated by the typechecker and constraints given by the invariants on the length of the vector and the bounds of the finite set mean that it is not possible to have an array bounds check error.

2.2.5 Alternative Elimination Rules

Sometimes the default elimination rule for a data type is not the elimination behaviour we want. We are not restricted to these default rules, however; any function with a motive and methods is considered an eliminator and we can therefore write down the pattern matching behaviour resulting from the methods.

A term e is an eliminator in a context Γ if:

- $\Gamma; \vec{t} \ : \ \vec{A} \vdash e : \forall P : (\forall \vec{a} : \vec{A}. \star). \forall m_{c_1} : (\forall \vec{a}_1 : \vec{A}_1. \ P \ \vec{s}_1). \ \ldots . \forall m_{c_n} : (\forall \vec{a}_1 : \vec{A}_1. \ P \ \vec{s}_n). \ P \ \vec{t}$
- $\Gamma; \vec{t} \ : \ \vec{A} \vdash \underline{\text{valid}}$
- $\Gamma; P \ : \ (\forall \vec{a} : \vec{A}. \star); \vec{a}_i \ : \ \vec{A}_i \vdash P \ \vec{s}_i \ : \ \star$, where $1 \leq i \leq n$.

This term e is a function which eliminates zero or more targets. The patterns which are allowed are given by the arguments $\vec{s}_i$ to the return type of each motive ($\vec{m}_c$.) Looking again at $\mathbb{N}$-**Elim**, we see how this fits the general scheme:

$$\begin{aligned}
\mathbb{N}\textbf{-Elim} \ : \ & \forall n : \mathbb{N}. \\
& \forall P : \mathbb{N} \to \star. \\
& \forall m_0 : P \ 0. \\
& \forall m_{\mathsf{s}} : \forall k : \mathbb{N}. \forall ih : P \ k. P \ (\mathsf{s} \ k). \\
& P \ n
\end{aligned}$$

The arguments to the motive P in the return type of the methods m_0 and m_{s} give the patterns which are allowed, which are 0 and $\mathsf{s} \ k$.

There is no reason why there should be only one target, and indeed in the case of indexed or parametrised families, the indices are effectively additional targets. The elimination rule for vectors illustrates this:

$$\begin{aligned}
\textsf{Vect-Elim} \ : \ & \forall A : \star. \\
& \forall n : \mathbb{N}. \forall v : \textsf{Vect} \ A \ n. \quad \text{(Targets)} \\
& \forall P : \forall n : \mathbb{N}. \textsf{Vect} \ A \ n \to \star. \\
& \forall m_{\epsilon} : P \ 0 \ \epsilon \\
& \forall m_{::} : \forall k : \mathbb{N}. \forall a : A. \forall v : \textsf{Vect} \ A \ k. \forall ih : P \ k \ v. P \ (\mathsf{s} \ k) \ (a :: v) \\
& P \ n \ v
\end{aligned}$$

The two arguments to the motive P indicate that this rule eliminates two values together. This makes sense, since the second value v depends on the first value n.

We can also write user defined elimination rules with this kind of behaviour. For example, we can write a double recursion rule which eliminates two natural numbers at once.

$$\begin{aligned}
\mathbb{N}\textbf{-double-elim} \ : \ & \forall n, m : \mathbb{N}. \\
& \forall m_{0n} : \forall n : \mathbb{N}. \ P \ 0 \ n. \\
& \forall m_{\mathsf{s}0} : \forall n : \mathbb{N}. \ P \ (\mathsf{s} \ n) \ 0. \\
& \forall m_{\mathsf{ss}} : \forall n : \mathbb{N}. \ \forall m : \mathbb{N}. \ P \ n \ m \to P \ (\mathsf{s} \ n) \ (\mathsf{s} \ m). \\
& P \ n \ m
\end{aligned}$$

User defined rules are implemented in terms of the elimination rules we already have, in this case by $\mathbb{N}$-**Elim**:

$$\begin{aligned}
&\mathbb{N}\textbf{-double-elim} \quad n \quad m \quad P \ m_{0n} \ m_{\mathsf{s}0} \ m_{\mathsf{ss}} \Leftarrow \underline{\text{elim}} \ n \\
&\mathbb{N}\textbf{-double-elim} \quad 0 \quad m \quad P \ m_{0n} \ m_{\mathsf{s}0} \ m_{\mathsf{ss}} \mapsto m_{0n} \ m \\
&\mathbb{N}\textbf{-double-elim} \quad (\mathsf{s} \ n) \quad m \quad P \ m_{0n} \ m_{\mathsf{s}0} \ m_{\mathsf{ss}} \Leftarrow \underline{\text{elim}} \ m \\
&\mathbb{N}\textbf{-double-elim} \quad (\mathsf{s} \ n) \quad 0 \quad P \ m_{0n} \ m_{\mathsf{s}0} \ m_{\mathsf{ss}} \mapsto m_{\mathsf{s}0} \ n \\
&\mathbb{N}\textbf{-double-elim} \quad (\mathsf{s} \ n) \ (\mathsf{s} \ m) \quad P \ m_{0n} \ m_{\mathsf{s}0} \ m_{\mathsf{ss}} \mapsto m_{\mathsf{ss}} \ n \ m \ (\mathbb{N}\textbf{-double-elim} \ n \ m)
\end{aligned}$$

Some functions are naturally recursive over two values, for example **max** which returns the larger of two natural numbers. **ℕ-double-elim** gives us a convenient pattern of recursion for writing this function:

$$\underline{\text{let}} \quad \frac{n, m \ : \ \mathbb{N}}{\mathbf{max}\ n\ m \ : \ \mathbb{N}}$$

$$\begin{array}{llll} \mathbf{max} & n & m & \Leftarrow \mathbb{N}\textbf{-double-elim}\ n\ m \\ \mathbf{max} & 0 & m & \mapsto m \\ \mathbf{max} & (\mathsf{s}\ n) & 0 & \mapsto \mathsf{s}\ n \\ \mathbf{max} & (\mathsf{s}\ n) & (\mathsf{s}\ m) & \mapsto \mathsf{s}\ (\mathbf{max}\ n\ m) \end{array}$$

2.2.6 Derived Eliminators and Memoisation

The elimination rules automatically generated for datatypes give us primitive recursion — a recursive call is allowed on recursive arguments of datatypes. This does not necessarily make all *structurally* recursive functions easy to define, however. Consider the function to return the nth element of the Fibonacci series; one way to write this in Haskell is as shown below:

```
fib :: Nat -> Nat
fib O = S O
fib (S O) = S O
fib (S (S k)) = plus (fib k) (fib (S k))
```

This is not a very efficient definition; there are two recursive calls, but it does not take advantage of sharing and some values of recursive calls will be computed repeatedly. Nevertheless, it represents a simple mathematical definition of the Fibonacci function. Unfortunately though, while it is structurally recursive, it is not primitive recursive and therefore cannot be defined directly using **ℕ-Elim**.

In COQ, structurally recursive functions can be defined using the primitive `Case` and `Fix` constructs, which separate the concepts of case analysis and recursion. A function defined using `Fix`, with a declared decreasing argument, can make recursive calls where the declared decreasing argument is structurally smaller. Giménez shows that elimination rules can be defined using `Case` and `Fix` and, conversely, all `Case`/`Fix` based functions can be defined using elimination rules [Gim94]. McBride mechanises the latter technique in his thesis [McB00a], and this is also implemented by EPIGRAM as described in [MM04b].

In EPIGRAM, as in COQ, the concepts of case analysis and recursion are separated. However, in EPIGRAM, elimination rules are used to implement the separation. Hence, for a family D, in addition to **D-Elim** and **D-Case**, an additional recursion operator is derived, called **D-Rec**. This operator carries within its motives a memo structure (**D-Memo**) which is a large tuple holding a value for the recursive call to each structurally smaller value:

$$\begin{array}{lll} \textsf{D-}\mathbf{Rec} & : & \forall \vec{a}\!:\!\vec{A}.\forall x\!:\!\textsf{D}\ \vec{a}.\forall P\!:\!(\forall \vec{s}\!:\!\vec{A}.\textsf{D}\ \vec{s} \to \star). \\ & & (\forall \vec{s}\!:\!\vec{A}.\forall y\!:\!\textsf{D}\ \vec{s}.\textsf{D-}\mathbf{Memo}\ (P\ y) \to P\ y) \to P\ x \end{array}$$

Note that this fits the form of elimination rules given in section 2.2.5. A call to the operator D-**Rec** for a term $d : \textsf{D}\ \vec{s}$ does not itself do case analysis, but rather gives access to recursive calls on values structurally smaller than d. To do the case analysis we require an additional application of D-**Case**.

The construction of such elimination operators is rather complex, and described in detail in [McB00a]. From the programmer's point of view, what it means is that any recursive calls on structurally smaller values are accessible via the memo structure. The definition of **fib** can therefore now be written by $\mathbb{N}$-**Rec** and $\mathbb{N}$-**Case**. We use the notation $\underline{\text{rec}}\ x$ to access the appropriate recursion rule.

$$\begin{array}{lll} \underline{\text{let}} & \dfrac{n\ :\ \mathbb{N}}{\mathbf{fib}\ n\ :\ \mathbb{N}} & \\ \mathbf{fib} & n & \Leftarrow \underline{\text{rec}}\ n \Leftarrow \underline{\text{case}}\ n \\ \mathbf{fib} & 0 & \mapsto 0 \\ \mathbf{fib} & (\textsf{s}\ k) & \Leftarrow \underline{\text{case}}\ k \\ \mathbf{fib} & (\textsf{s}\ 0) & \mapsto \textsf{s}\ 0 \\ \mathbf{fib}\ (\textsf{s}\ (\textsf{s}\ k')) & & \mapsto \mathbf{plus}\ (\mathbf{fib}\ k')\ (\mathbf{fib}\ (\textsf{s}\ k')) \end{array}$$

For reference, the construction of $\mathbb{N}$-**Rec** and its helper functions are shown in figure 2.16, figure 2.17 and figure 2.18. The fully elaborated **fib** function is shown in figure 2.19. This definition is large and barely readable, and is clearly a function we are happy to let the elaborator write for us. Note that the results of the recursive calls to **fib** are accessed by projecting them out of the tuple built by $\mathbb{N}$-**Memo**.

$$\begin{array}{lll} \underline{\text{let}} & \dfrac{n\ :\ \mathbb{N} \quad P\ :\ \mathbb{N} \to \star}{\mathbb{N}\textbf{-Memo}\ n\ P\ :\ \star} & \\ \mathbb{N}\textbf{-Memo} & n\ P & \Leftarrow \underline{\text{elim}}\ n \\ \mathbb{N}\textbf{-Memo} & 0\ P & \mapsto \textsf{True} \\ \mathbb{N}\textbf{-Memo}\ (\textsf{s}\ k)\ P & & \mapsto (P\ n \times \mathbb{N}\textbf{-Memo}\ k\ P) \end{array}$$

Figure 2.16: $\mathbb{N}$-**Memo** definition

$$\begin{array}{lll} \underline{\text{let}} & \dfrac{n\ :\ \mathbb{N} \quad P\ :\ \mathbb{N} \to \star \quad M\ :\ \forall n\!:\!\mathbb{N}.\ (\mathbb{N}\textbf{-Memo}\ n\ P) \to (P\ n)}{\mathbb{N}\textbf{-MemoGen}\ n\ P\ M\ :\ \mathbb{N}\textbf{-Memo}\ n\ P} & \\ \mathbb{N}\textbf{-MemoGen} & n\ P\ M & \Leftarrow \underline{\text{elim}}\ n \\ \mathbb{N}\textbf{-MemoGen} & 0\ P\ M & \mapsto () \\ \mathbb{N}\textbf{-MemoGen}\ (\textsf{s}\ k)\ P\ M & & \mapsto \underline{\text{let}}\ rec\ :\ (\mathbb{N}\textbf{-Memo}\ k\ P) \to (P\ k) \mapsto \\ & & \qquad \mathbb{N}\textbf{-MemoGen}\ k\ \underline{\text{in}}\ (M\ rec, rec) \end{array}$$

Figure 2.17: $\mathbb{N}$-**MemoGen** definition

$$\underline{\text{let}} \quad \frac{n : \mathbb{N} \quad P : \mathbb{N} \to \star \quad M : \forall n{:}\mathbb{N}.\, \mathbb{N}\textbf{-Memo}\ (P\ n) \to (P\ n)}{\mathbb{N}\textbf{-Rec}\ n\ P\ M\ :\ P\ n}$$
$$\mathbb{N}\textbf{-Rec}\ n\ P\ M\ \mapsto\ M\ n\ (\mathbb{N}\textbf{-MemoGen}\ P\ M\ n)$$

Figure 2.18: $\mathbb{N}$-**Rec** definition

$$\begin{array}{l}
\textbf{fib} \mapsto \lambda n{:}\mathbb{N}.\, \mathbb{N}\textbf{-Rec}\ n\ (\lambda x{:}\mathbb{N}.\, \mathbb{N}) \\
\qquad (\lambda n'{:}\mathbb{N}.\, \mathbb{N}\textbf{-Case}\ n'\ (\lambda x{:}\mathbb{N}.\, (\mathbb{N}\textbf{-Memo}\ (\lambda y{:}\mathbb{N}.\, \mathbb{N})\ x) \to \mathbb{N}) \\
\qquad\quad (\lambda u{:}\mathsf{True}.\, \mathsf{s0}) \\
\qquad\quad (\lambda k{:}\mathbb{N}.\, \mathbb{N}\textbf{-Case}\ k\ (\lambda x{:}\mathbb{N}.\, (\mathbb{N}\textbf{-Memo}\ (\lambda y{:}\mathbb{N}.\, \mathbb{N})\ (\mathsf{s}\ x)) \to \mathbb{N}) \\
\qquad\qquad ((\lambda x{:}\mathbb{N}\textbf{-Memo}\ (\lambda y{:}\mathbb{N}.\, \mathbb{N})\ \mathsf{s0}.\, \mathsf{s0})) \\
\qquad\qquad (\lambda k{:}\mathbb{N}.\, \lambda M{:}(\mathbb{N}, (\mathbb{N}, \mathbb{N}\textbf{-Memo}\ (\lambda x{:}\mathbb{N}.\, \mathbb{N})\ k)). \\
\qquad\qquad\quad \textbf{plus}\ (\textbf{fst}\ M)\ (\textbf{fst}\ (\textbf{snd}\ M)))))
\end{array}$$

Figure 2.19: Elaborated **fib**

2.2.7 Matching on Intermediate Values

The examples we have seen so far have performed pattern matching only on the arguments passed directly to the function. In practice though, we often create intermediate values in the process of computation. We could match on these by passing all of the pattern variables to a helper function, but [MM04b] also describes a more compact notation for this, the "with" construct (*lhs* | *expr* {*program*}). This construct adds *expr* to the values we are allowed to match on. Here we extend this notation to the "named with" construct (*lhs* | *var* ← *expr* {*program*}), which gives a name *var* for later case analysis.

An example of a function where such behaviour is useful is the `filter` function from the Haskell standard prelude. Filter removes any items from a list to which a given predicate does not apply. In Haskell, it is defined with guards to check the intermediate computation:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x:(filter p xs)
                | otherwise = filter p xs
```

In EPIGRAM we take $p\ x$ as an intermediate computation and match on its result with Bool-**Case** (figure 2.20). Pattern matching is on the result of the intermediate computation in a similar style to the pattern guards proposed for Haskell [EP00]. We will see several examples where this notation is useful; either making definitions more concise or removing the need for auxiliary functions.

```
let   f : A → Bool    xs : List A
      ─────────────────────────
      filter f xs : List A

  filter f     xs          ⇐ elim xs
  filter f     nil         ↦ nil
  filter f (cons x xs)  |  b ← f x    ⇐ case b
                        |    true     ↦ cons x (filter f xs)
                        |    false    ↦ filter f xs
```

Figure 2.20: **filter** definition

2.3 Programming Idioms

We have so far seen the basic syntax of EPIGRAM and some small example programs. These have been very similar in structure (if not in their typing) to the sort of programs we might write in a traditional functional language. However, with the stronger type system come new programming idioms. In this section, I will discuss some of these and the additional syntax EPIGRAM provides to support them. We will start by looking at two of the simpler idioms, dependent pairs and writing programs by induction over proofs, and move on to more complex and powerful idioms, views and techniques for showing termination.

2.3.1 Dependent Pairs

It is often the case in dependently typed programming that we do not know in advance of running a function which builds an instance of a family what the indices of that family will be. For example, if we write the **filter** function of the previous section over Vect rather than List (correspondingly calling it **vfilter**), what is its return type?

```
let   p : A → Bool    xs : Vect A n
      ─────────────────────────
      vfilter p xs : Vect A ?
```

In some cases, we can write a function which computes the required index in advance. If we are converting a List to a Vect, for example, we can calculate the length of the List, **length**, and build the index from that:

```
let   l : List A
      ──────────
      length l : ℕ

  length          l          ⇐ elim l
  length          nil        ↦ 0
  length (cons x xs)         ↦ s(length xs)
```

```
let   l : List A
      ──────────────────
      listToVect l : Vect A (length l)

  listToVect          l          ⇐ elim l
  listToVect          nil        ↦ ϵ
  listToVect (cons x xs)         ↦ x::(listToVect xs)
```

For **vfilter** however, we can only compute the index by running the function itself. In this case, we prefer to return a **dependent pair** of values. A dependent pair is a pair in which the type of the second item is predicated on the first value. This can be built into the core type theory as a primitive, as in Luo's ECC [Luo94], but inductive families mean that this is not necessary. In EPIGRAM we declare dependent pairs as an inductive family with the declaration in figure 2.21.

$$\begin{array}{ll} \underline{\text{data}} & \dfrac{A : \star \quad F : A \to \star}{\Sigma\, A\, F : \star} \\ \underline{\text{where}} & \dfrac{a : A \quad f : F\, a}{(a, f) : \Sigma\, A\, F} \end{array}$$

Figure 2.21: Dependent pair

Using a dependent pair, we can write **vfilter** as in figure 2.22. Note that there are additional matches on the results of recursive calls to **vfilter**, and that the first element of the pair can be inferred by the typechecker from the type of the second element. Using a dependent pair like this can provide a convenient layer of abstraction for an inductive family which hides the indices — the user of functions over the family need not know what the indices of the family are. The use of ? in the return values indicates that we expect the elaborator to be able to infer the values of these terms, as in each case there is only one value which would be well-typed.

$$\begin{array}{l} \underline{\text{let}} \quad \dfrac{f : A \to \mathsf{Bool} \quad xs : \mathsf{Vect}\, A\, n}{\mathbf{vfilter}\, f\, xs : \Sigma\, \mathbb{N}\, (\mathsf{Vect}\, A)} \\ \quad \mathbf{vfilter}\ f \quad xs \quad \Leftarrow \underline{\text{elim}}\ xs \\ \quad \mathbf{vfilter}\ f \quad \epsilon \quad \mapsto (?, \mathsf{nil}) \\ \quad \mathbf{vfilter}\ f\ (x{::}xs) \mid b \leftarrow f\ x \quad \Leftarrow \underline{\text{case}}\ b \\ \qquad\qquad\qquad\quad \mid \mathsf{true} \mid p \leftarrow \mathbf{vfilter}\ f\ xs \quad \Leftarrow \underline{\text{case}}\ p \\ \qquad\qquad\qquad\qquad\quad\ \ (\_, xs') \quad \mapsto (?, x{::}xs') \\ \qquad\qquad\qquad\quad \mid \mathsf{false} \mid p \leftarrow \mathbf{vfilter}\ f\ xs \quad \Leftarrow \underline{\text{case}}\ p \\ \qquad\qquad\qquad\qquad\quad\ \ (\_, xs') \quad \mapsto (?, xs') \end{array}$$

Figure 2.22: **vfilter** definition

Pairing like this is similar to the approach taken to vectors in the C++ standard template library [MSD01], in that the internal representation pairs the length with the list data itself, and operations on the vector class preserve length invariants. In C++, however, the length invariants are maintained by hand, rather than by the type system.

2.3.2 Induction Over Proofs

Properties can be expressed as inductive relations in EPIGRAM, which allows us to impose more constraints on the definition and use of functions. Sometimes it is difficult or impossible to express constraints using the indices of an inductive family alone. An example of where this is difficult is in defining the **minus** function on $\mathbb{N}$. It does not make sense to subtract a number from a smaller number since $\mathbb{N}$ does not represent negative numbers. We end up with either a mathematically incorrect definition of **minus** (by returning 0 if the result should be negative), or a function which is not defined for all of its inputs, which is impossible. The solution is to define a relation to express the constraint that a number must be subtracted from a larger or equal number. This is the less than or equal relation.

$$\underline{\text{data}} \quad \frac{x, y \ : \ \mathbb{N}}{x{\leq}y \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{leO} \ : \ 0{\leq}y} \qquad \frac{p \ : \ x{\leq}y}{\mathsf{leS}\, p \ : \ (\mathsf{s}\, x){\leq}(\mathsf{s}\, y)}$$

The **minus** function now takes three arguments; the two numbers n and m along with a proof that m is less than or equal to n. Then rather than defining the function by elimination of m or n, we define the function by elimination of p. By doing the recursion on the proof, we get patterns for m and n since they are the indices of the proof relation. This proof ensures that no invalid arguments can be passed to **minus**.

$$\underline{\text{let}} \quad \frac{n, m \ : \ \mathbb{N} \quad p \ : \ m \leq n}{\mathbf{minus}\ n\ m\ p \ : \ \mathbb{N}}$$

$$\begin{array}{llll} \mathbf{minus} & n & m & p & \Leftarrow \underline{\text{elim}}\ p \\ \mathbf{minus} & n & 0 & (\mathsf{leO}\ n) & \mapsto n \\ \mathbf{minus} & (\mathsf{s}\ n) & (\mathsf{s}\ m) & (\mathsf{leS}\ m\ n\ p) & \mapsto \mathbf{minus}\ n\ m\ p \end{array}$$

Remark: The main point here is that the patterns are generated not from the data directly, but from a proof of a property which must hold for that data. There is therefore only one case analysis required — on the proof — rather than case analysis on each of the numbers. Which case applies when we do case analysis on the proof affects the possible values of the numbers, an effect which we only begin to see when using dependent types.

2.3.3 Views

We have looked at alternative elimination rules in section 2.2.5, in order to give alternative pattern matching behaviour. Another method, once we know the alternative pattern matching behaviour we would like, is to write down an inductive family whose generated elimination rule has the behaviour we are looking for. Such an inductive family gives an alternative view of data; a family $\mathsf{D}\,\vec{s}$ is a **view** of its indices $\vec{s}$ if there is a covering function $\mathbf{d} \ : \ \forall \vec{s}{:}\vec{S}.\, \mathsf{D}\,\vec{s}$. Views were originally proposed by Wadler [Wad87] as a means of furnishing abstract types with pattern matching behaviour. The presentation here is as in [MM04b].

An example of the use of views is to give an informative comparison operation. Traditionally, we might have an $\underline{\text{if}}\ b\ \underline{\text{then}}\ t\ \underline{\text{else}}\ e$ construct, where b : Bool and $t, e \ : \ T$ for some

T, which is equivalent to Bool-**Case**. There are however two shortcomings of the typing of an if expression:

- There is no distinction between the types of the then and else branches, so there is no protection against accidentally writing the branches the wrong way round.
- We do not retain any information about the test in the type, either its result or any other information generated while performing the test.

For example, how might we compare two Ns? The conventional way would be to define an ordering function, returning an element of an Ordering type with constructors lt, eq and gt.

$$\underline{\text{let}} \quad \frac{n, m \ : \ \mathbb{N}}{\mathbf{Nord}\ n\ m \ : \ \mathsf{Ordering}}$$

$$\begin{array}{llll} \mathbf{Nord} & n & m & \Leftarrow \underline{\text{elim}}\ n, \underline{\text{elim}}\ m \\ \mathbf{Nord} & 0 & 0 & \mapsto \mathsf{eq} \\ \mathbf{Nord} & (\mathsf{s}\ n) & 0 & \mapsto \mathsf{gt} \\ \mathbf{Nord} & 0 & (\mathsf{s}\ m) & \mapsto \mathsf{lt} \\ \mathbf{Nord} & (\mathsf{s}\ n) & (\mathsf{s}\ m) & \mapsto \mathbf{Nord}\ n\ m \end{array}$$

However, this function is doing some extra work which is not reflected in the return value; it is effectively performing a subtraction of the smaller from the larger number and throwing the result away. If we later want to know the difference between the two numbers, this information has been lost, so we have to recalculate it. With a dependent type system, we can do better than this by making an elimination rule which eliminates numbers based on their difference:

$$\begin{array}{ll} \mathbf{Ncompare} \ : & \forall m, n{:}\mathbb{N}. \\ & \forall P{:}\mathbb{N} \to \mathbb{N} \to \star. \\ & \forall m_{\mathsf{lt}}{:}\forall x, y{:}\mathbb{N}.\ P\ x\ (\mathbf{plus}\ x\ (\mathsf{s}\ y)). \\ & \forall m_{\mathsf{eq}}{:}\forall x{:}\mathbb{N}.\ P\ x\ x. \\ & \forall m_{\mathsf{gt}}{:}\forall x, y{:}\mathbb{N}.\ P\ (\mathbf{plus}\ y\ (\mathsf{s}\ x))\ y. \\ & P\ m\ n \end{array}$$

This elimination rule, defined by recursion over m and n, finds which is the larger number and applies the appropriate method, but also each method type records which number is greater and by how much. Using this elimination rule, it is straightforward to write functions such as the following, **absDiff**, which finds the difference between two numbers:

$$\underline{\text{let}} \quad \frac{m, n \ : \ \mathbb{N}}{\mathbf{absDiff}\ m\ n \ : \ \mathbb{N}}$$

$$\begin{array}{llll} \mathbf{absDiff} & m & n & \Leftarrow \mathbf{Ncompare}\ m\ n \\ \mathbf{absDiff} & x & (\mathbf{plus}\ x\ (\mathsf{s}\ y)) & \mapsto \mathsf{s}\ y \\ \mathbf{absDiff} & x & x & \mapsto 0 \\ \mathbf{absDiff}\ (\mathbf{plus}\ y\ (\mathsf{s}\ x)) & & y & \mapsto \mathsf{s}\ x \end{array}$$

The patterns we get for the arguments of **absDiff** allow us to pick out directly what the difference between the arguments is, without doing any subtraction, since the subtraction has already been effectively performed by the elimination rule. Writing elimination rules such as **Ncompare** by hand is, however, cumbersome. Instead, EPIGRAM supports the use of views; the idea behind views is that the easiest way to get an elimination rule with the behaviour we want is to define a family whose default elimination rule has that behaviour. For example, the behaviour we want for **Ncompare** is given by the elimination rule for the Compare family in figure 2.23.

$$\underline{\text{data}} \quad \frac{m : \mathbb{N} \quad n : \mathbb{N}}{\mathsf{Compare}\ m\ n\ :\ \star}$$

$$\underline{\text{where}} \quad \frac{y : \mathbb{N}}{\mathsf{lt}\ y\ :\ \mathsf{Compare}\ x\ (\mathbf{plus}\ x\ (\mathsf{s}\ y))} \qquad \frac{}{\mathsf{eq}\ :\ \mathsf{Compare}\ x\ x} \qquad \frac{x : \mathbb{N}}{\mathsf{gt}\ x\ :\ \mathsf{Compare}\ (\mathbf{plus}\ y\ (\mathsf{s}\ x))\ y}$$

Figure 2.23: The Compare view of $\mathbb{N}$

When a family D is declared, as well as generating D-**Elim**, D-**Rec**, and D-**Case** as we have seen, EPIGRAM generates D-**View** as shown in figure 2.24. This is an elimination rule which generates patterns for the indices of D, but not D itself — it is easy to see how to build the definition of this from D-**Elim**, simply by dropping the target argument from the methods and motives. This rule is the non-dependent elimination rule for D.

$$\begin{array}{lll}
\text{D-}\mathbf{View} : & \forall \vec{i}{:}\vec{I}.\, \forall x{:}\mathsf{D}\ \vec{i}. & \text{(target)} \\
& \forall P{:}\forall \vec{i}{:}\vec{I}.\ \star. & \text{(motive)} \\
& \forall m_{\mathsf{c}} :\ \forall \vec{a}{:}\vec{A}.\ \forall y_1{:}\mathsf{D}\ \vec{r}_1.\ \ldots\ \forall y_j{:}\mathsf{D}\ \vec{r}_j. & \\
& \qquad P\ \vec{r}_1 \rightarrow\ \ldots \rightarrow\ P\ y_j \rightarrow\ P\ \vec{s}. & \text{(methods)} \\
& \ldots & \\
& P\ \vec{i} &
\end{array}$$

Figure 2.24: View rule for D

We can access the appropriate view rule for x by the notation $\underline{\text{view}}\ x$. Hence, if we have a view $\mathsf{D}\ \vec{s}$ with a covering function **d**, we can write a function by D-**View** with the following notation:

$$lhs \Leftarrow \underline{\text{view}}\ \mathbf{d}\ \vec{s}$$

To show that any two numbers are comparable by this view, we build a covering function **compare** as in figure 2.25. Note that in the recursive cases, we use the $\underline{\text{view}}$ notation for pattern matching.

$$\underline{\text{let}}\quad \frac{n, m \;:\; \mathbb{N}}{\mathbf{compare}\ n\ m \;:\; \mathsf{Compare}\ n\ m}$$

compare	n	m	$\Leftarrow \underline{\text{elim}}\ n, \underline{\text{elim}}\ m$
compare	0	0	$\mapsto \mathsf{eq}$
compare	$(\mathsf{s}\ n)$	0	$\mapsto \mathsf{gt}\ n$
compare	0	$(\mathsf{s}\ m)$	$\mapsto \mathsf{lt}\ m$
compare	$(\mathsf{s}\ n)$	$(\mathsf{s}\ m)$	$\Leftarrow \underline{\text{view}}\ \mathbf{compare}\ n\ m$
compare	$(\mathsf{s}\ x)$	$(\mathsf{s}(\mathbf{plus}\ (\mathsf{s}y)\ x))$	$\mapsto \mathsf{lt}\ y$
compare	$(\mathsf{s}\ x)$	$(\mathsf{s}\ x)$	$\mapsto \mathsf{eq}$
compare	$(\mathsf{s}(\mathbf{plus}\ (\mathsf{s}\ x)\ y))$	$(\mathsf{s}\ y)$	$\mapsto \mathsf{gt}\ x$

Figure 2.25: The covering function for Compare

Using the view notation notation, we can use the Compare view rather than **Ncompare** and get the appropriate patterns for the numbers in the definition of **absDiff**.

$$\underline{\text{let}}\quad \frac{m, n \;:\; \mathbb{N}}{\mathbf{absDiff}\ m\ n \;:\; \mathbb{N}}$$

absDiff	m	n	$\Leftarrow \underline{\text{view}}\ \mathbf{compare}\ m\ n$
absDiff	x	$(\mathbf{plus}\ x\ (\mathsf{s}\ y))$	$\mapsto \mathsf{s}\ y$
absDiff	x	x	$\mapsto 0$
absDiff	$(\mathbf{plus}\ y\ (\mathsf{s}\ x))$	y	$\mapsto \mathsf{s}\ x$

Note that the view notation suppresses the intermediate values created by the covering function **compare** $m\ n$, so we can concentrate on the patterns the elimination rule gives us. Applying view **compare** $m\ n$ has the same effect as would applying **Ncompare**, with the advantage that the definition of the new pattern matching rule is by first order programming.

2.3.4 Termination

We have seen that one of the requirements of being a well defined EPIGRAM function is that the function must terminate. This raises an important question, since it is impossible to decide in general if a general recursive function terminates — how big a restriction is this, and when can we show that a function which is not structurally recursive does nevertheless terminate?

Consider the **quicksort** function. For simplicity we will make this a monomorphic function and sort natural numbers in increasing order. In Haskell we might write the function as follows:

```
quicksort [] = []
quicksort (x:xs) = quicksort l ++ (x:quicksort r)
   where l = [y | y <- xs, y < x]
         r = [y | y <- xs, y >= x]
```

This is a nice concise definition with two auxiliary functions to partition the list into two halves, and a main function which reconstructs the sorted list from the sorted parts.

However, the recursion is not structural, so such a definition would not be accepted by EPIGRAM.

We do know that this function terminates (it can be shown by noting that the recursive calls are always on obviously smaller lists) — but how do we prove this to the language? I will briefly explain two possibilities for overcoming this sort of problem by defining the **quicksort** function in EPIGRAM, declared as follows:

$$\underline{\text{let}} \quad \frac{l \ : \ \mathsf{List}\,\mathbb{N}}{\mathbf{quicksort}\ l \ : \ \mathsf{List}\,\mathbb{N}}$$

Domain Predicates

General recursion in type theory can be achieved by means of a general accessibility predicate [Acz77]. A value a is **accessible** by a relation $\prec$ if there is no infinite decreasing sequence starting from a. A set A is **well-founded** with respect to $\prec$ if all of its elements are accessible by $\prec$. The accessibility predicate is defined in EPIGRAM as below:

$$\underline{\text{data}} \quad \frac{A \ : \ \star \qquad \prec \ : \ A \to A \to \star \qquad a \ : \ A}{\mathsf{Acc}\ A \ \prec \ a \ : \ \star}$$

$$\underline{\text{where}} \quad \frac{p \ : \ \forall x{:}A.(x \prec a) \to \mathsf{Acc}\ A \ \prec \ x}{\mathsf{acc}\ p \ : \ \mathsf{Acc}\ A \ \prec \ a}$$

The elimination rule for this predicate is known as the **rule of well-founded recursion**. Then, to guarantee that a general recursive algorithm terminates, we prove that it has a decreasing argument type which is well-founded and that the arguments to the recursive calls are smaller than the input.

Bove [Bov02a] and Capretta [Cap02, BC03] note that one general accessibility predicate gives no information that can help in a specific case. This often results in long and complicated proofs. Instead, they propose defining special purpose domain predicates for each general recursive function, and define the function by recursion over the domain predicate.

For the **quicksort** example, the function always terminates on the input nil, and terminates on the input consxxs if it also terminates on the inputs **filter**$(<x)xs$ and **filter**$(\geq x)xs$. This is expressed by the qsAcc predicate (figure 2.26).

$$\underline{\text{data}} \quad \frac{xs \ : \ \mathsf{List}\,\mathbb{N}}{\mathsf{qsAcc}\ xs \ : \ \star}$$

$$\underline{\text{where}} \quad \frac{}{\mathsf{qsNil} \ : \ \mathsf{qsAcc}\ \mathsf{nil}}$$

$$\frac{qsl \ : \ \mathsf{qsAcc}\ (\mathbf{filter}\ (<\ x)\ xs) \qquad qsr \ : \ \mathsf{qsAcc}\ (\mathbf{filter}\ (\geq\ x)\ xs)}{\mathsf{qsCons}\ qsl\ qsr \ : \ \mathsf{qsAcc}\ (\mathsf{cons}\ x\ xs)}$$

Figure 2.26: Domain predicate for **quicksort**

A **quicksort** helper function, **quicksort′** is defined by induction over this predicate

(figure 2.27). If we ignore the references to the predicate and concentrate simply on the lists, we see that this helper function is identical in structure to the Haskell definition.

```
quicksort'  xs          acc       ⇐ elim acc
quicksort'  nil         qsNil     ↦ nil
quicksort' (cons x xs) (qsCons qsl qsr)
  ↦ quicksort' (filter (< x) xs) qsl ++ cons x (quicksort' (filter (≥ x) xs) qsr)
```

Figure 2.27: Helper function for **quicksort**

To use this predicate and the helper function to define **quicksort**, we prove that all lists are accessible by the predicate, and hence that the domain of **quicksort** is the whole of List:

$$\underline{\text{let}} \quad \frac{zs \ : \ \mathsf{ListN}}{\mathbf{allQsAcc}\ zs \ : \ \mathsf{qsAcc}\ zs}$$

Given this function to build the predicate, the top level definition of **quicksort** is straightforward:

quicksort $xs \ \mapsto$ **quicksort'** xs (**allQsAcc** xs)

The difficulty with this method is in the definition of **allQsAcc**, which is where the details of the termination proof lie; this function is non-trivial to define. However, Bove and Capretta's method can be applied systematically to any terminating recursive function, including nested recursive calls and mutual recursive calls [BC01, Bov02b] leaving the user only to write a function to construct the accessibility predicate.

We could also consider qsAcc to be a view of lists, with **allQsAcc** as the covering function. This gives a clearer definition of **quicksort**, hiding away the domain predicate while still giving access to the same recursive calls. We have previously seen views used for alternative pattern matching — here we use views to generate different allowed recursive calls. The view based definition is shown in figure 2.28.

```
quicksort  xs      ⇐ view allQsAcc xs
quicksort  nil     ↦ nil
quicksort (cons x xs)
  ↦ quicksort (filter (< x) xs) ++ cons x (quicksort (filter (≥ x) xs))
```

Figure 2.28: Using qsAcc as a view of lists for recursion

Making the Function Structural

It would be preferable to avoid having to give a proof with every function which does not terminate through structural recursion, as with domain predicates. **quicksort** as defined in Haskell above had the drawback that it was relying on clever code, rather than an informative

data structure. The question to ask, therefore, is what is the data structure which gives the recursion behaviour we would like for quicksort?

There are two cases in the quicksort definition. There is the case of the empty list, and the case where we take out the head of the list, all items smaller than the head, and all items greater than the head. The corresponding data structure for this recursive behaviour, QuickSort, is shown in figure 2.29.

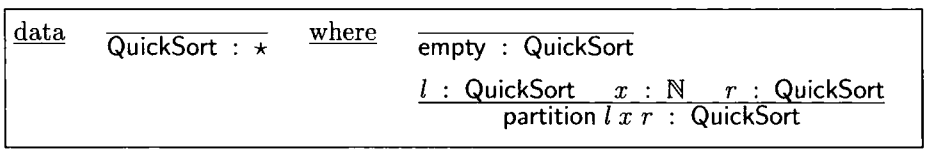


Figure 2.29: **quicksort** intermediate structure

We notice that the intermediate structure we have defined is nothing more than a binary tree. This should not be a surprise — tree-sort is merely quicksort with the recursive structure made explicit as intermediate data[2]. We can build a function which behaves like **quicksort** by composing a conversion function from lists to binary trees (**listToTree**) with a function converting back again (**flatten**).

$$\textbf{quicksort}\ x \ \mapsto\ \textbf{flatten}\ (\textbf{listToTree}\ x)$$

Is this function really **quicksort**? In one sense, no; it is tree sort, which is a slightly different algorithm in that it involves building an intermediate structure. However, the original Haskell function does not implement quicksort precisely either — Hoare's original imperative definition of quicksort [Hoa62] relied on a clever technique for in place sorting of lists, which we do not get in this definition. Turner notes in [Tur96] that for each version of quicksort there is a tree sort which performs exactly the same comparisons and has the same complexity. We also note that the tree data structure being built is the same as the structure which is built internally by the evaluation of the Haskell quicksort. It may not be the same definition or even exactly the same algorithm, but we have not lost anything in terms of complexity or behaviour from the Haskell definition.

Remark: Since we have dependent types, we could even refine the intermediate structure further, by including order invariants. Then we would be sure that **listToTree** constructs a binary search tree, and that **flatten**ing produces a sorted list.

2.4 Summary

In this chapter, we have seen the background to functional programming with dependent types using EPIGRAM and the underlying type theory. The EPIGRAM high level notation elaborates to a dependent type theory TT based on Luo's UTT with inductive families,

[2]In fact tree-sort was the first program proven correct by structural recursion in [Bur69]

heterogeneous equality and labelled types. We have seen examples of programs in TT — in particular, we should note that programming directly in TT leads to large and unreadable terms even for some very simple programs; **vTail** is a prime example.

The EPIGRAM elaborator exists to write these large and unreadable terms so that the programmer need not think about low level details such as how to prove certain cases are impossible and which variables (the inductive hypotheses) give the allowed recursive calls. Programming in the high level notation is based on using elimination rules to give pattern matching behaviour to functions. To support this, EPIGRAM generates several elimination rules for a family:

- D-**Elim** is the basic elimination rule which gives primitive recursion on D. All other elimination rules can be defined in terms of D-**Elim**. This rule is accessed by the notation elim x. (See section 2.1.3).
- D-**Case** gives case analysis on D, but no recursion. Although this can be defined in terms of D-**Elim** by ignoring the inductive hypotheses, it is more efficient to implement the reductions directly. This rule is accessed by the notation case$x$. (See section 2.1.3).
- D-**Rec** generates a memo structure which gives access to recursive calls on structurally smaller values. This rule is accessed by the notation rec x. (See section 2.2.6).
- D-**View** generates an elimination rule which gives recursion on the indices of D. This allows us to create new pattern matching behaviour for a family which is not necessarily based on constructor patterns. This rule is accessed by the notation view$x$. (See section 2.3.3).

Coquand notes that one of the drawbacks of programming with elimination rules is readability [Coq92], and proposes a pattern matching notation for dependent types. EPIGRAM's high level notation solves this readability problem by recovering the elimination rule based definitions from pattern matching definitions; this is possible because programming by pattern matching and programming by elimination rules are equivalent [Gim94, McB00a]. There is an additional benefit to the elimination rule based approach taken by EPIGRAM, which is that user defined elimination rules can be written by using views (or even directly) which gives more powerful pattern matching behaviour. The remaining drawback is that elimination rules, unlike direct pattern matching, impose an extra level of abstraction on programs. However, in Chapter 6, we will propose a method for overcoming this drawback.

Chapter 3

Compiling ExTT

In the last chapter I presented the core language of EPIGRAM and the high level notation. The core language is executed through a translation to an execution language, ExTT, and so in this chapter I will show a compilation scheme for ExTT. To begin with, we consider only the naïve path (see figure 1.1 on page 14), where the transformation from TT to ExTT is the identity transformation; in later chapters we will see how the compilation techniques can be modified in order to optimise evaluation via an optimising transformation to ExTT.

Compilation of any language involves translation to a machine language (or abstract machine language). Doing this directly for EPIGRAM is difficult, in particular because a typical machine does not have the same execution model as a functional language. Instead we translate via an intermediate representation which still has a functional flavour, yet is more amenable to translation to an abstract machine language. In this chapter we introduce an intermediate language which I call RunTT, and give a compilation scheme for translating RunTT into abstract machine code. The abstract machine we use is based on the G-machine [Joh84, Aug84], a well understood graph reduction machine.

Compilation of ExTT to G-machine code therefore consists of two high level steps; first we translate to the intermediate representation RunTT, then from RunTT to G-code. RunTT is a language of supercombinators, which are higher order functions with no free variables; removing free variables eliminates one difficulty from the compilation process. Each supercombinator sequence is then compiled to a G-code sequence which, when executed, builds the supercombinator body.

At the end of the chapter, we will look at some of the issues in designing a run-time system for a dependently typed language, specifically the overheads which are present when taking a naïve approach to compilation.

3.1 Execution Environments

Before we look at the details of the compilation of ExTT, let us consider the possible approaches we may take. The method used for evaluating an expression in a functional language depends on several things:

- Different techniques are used for interpretation and compilation. Compilation produces faster code, but interpretation is sometimes desirable, for example for fast prototyping and testing of individual functions.
- We should consider whether we want to reduce to a normal form, a head-normal form or a weak head-normal form.
- We should also make a choice between lazy evaluation, eager evaluation, or some hybrid approach as compilation techniques can differ substantially in each case.

With a dependently typed language, there is a new problem — we need some kind of evaluation mechanism at compile-time in order to implement the conversion check. We will therefore consider two environments for evaluation of terms, these being compile-time evaluation, where we reduce to the normal forms required by the conversion check, and run-time evaluation where we reduce in the empty context (with no free variables) and reduce to weak head-normal forms, doing only as much evaluation as is required by the programmer.

3.1.1 Normalisation by Evaluation

Normalisation by evaluation [BS91, BES98, Fil01], also known as reduction-free normalisation [AHS95] is a straightforward method for producing normal forms which relies on the meta-language's implementation of substitution. An implementation in Haskell, for example, pushes substitution through to the Haskell level. The basic technique is to build a meta-level representation of the term to be evaluated (`eval`), evaluate that term in the meta-language then reify the term back to an object level representation of normal forms (`quote`). Finally, we revert to the representation of ExTT (`forget`). Figure 3.1 shows an overview of the process of normalisation by evaluation for ExTT.

There are two main applications of normalisation by evaluation; firstly to provide a straightforward normalisation algorithm for the conversion check, and secondly for partial evaluation. The goal of partial evaluation is to simplify a function of multiple arguments where some arguments are known at compile-time; normalisation by evaluation is used to push the argument values through the body of the function. The main advantage of using normalisation by evaluation over other techniques such as compiled strong reduction [GL02] or the Krivine Machine [HMP96, WF03] is the ease of implementation; rather than implementing substitution by hand, we use the meta-language's implementation of substitution. It is not clear that normalisation by evaluation is more efficient than other methods, however.

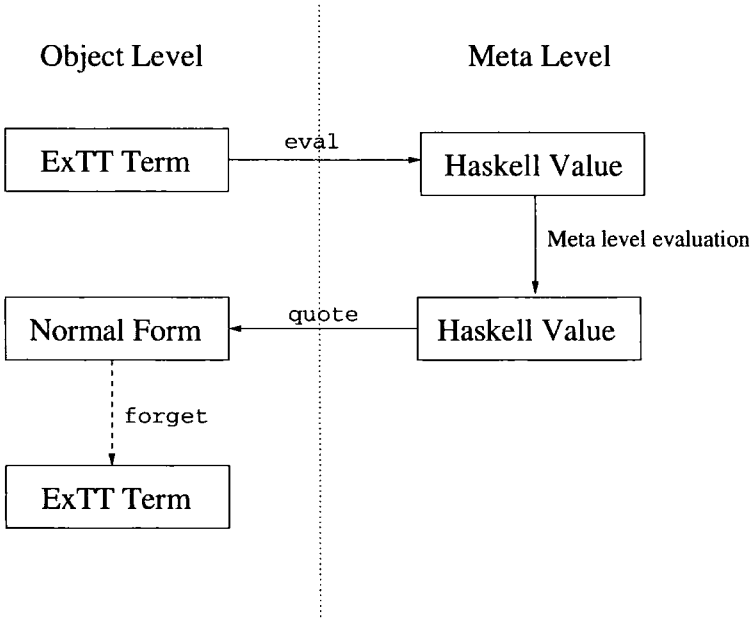


Figure 3.1: Normalisation By Evaluation

Normalisation by evaluation has not yet been proved correct for dependent type theory; however its correctness for simple type systems suggests we have no reason to think otherwise. Ultimately, however, if a dependently typed programming system is to use normalisation by evaluation and claim it is a safe system, then it must be shown to be correct.

In Appendix C, we will see an implementation in Haskell of normalisation by evaluation for ExTT.

3.1.2 Compilation

Compilation into machine language (whether a CPU's machine code or an abstract machine language) is a more efficient way of producing a normal form of a λ-term than interpreting or normalising directly, simply because analysis of the syntactic structure of the term is done in advance. As a result, decisions such as evaluation order are taken only once for each term and the choice encoded in machine language. Several different compilation methods have been developed, differing in particular in whether they perform lazy or eager evaluation.

Continuation Passing Style

Continuation passing style [App92], or CPS, is a method for evaluation in which functions return no value, but rather make tail calls which pass a continuation function explaining what to do with the result. This approach lends itself nicely to generating imperative

code since it makes sequencing explicit. [MWCG99], for example, describes the phases of compiling System-F to a typed assembly language via an intermediate CPS representation.

CPS is often used in the implementation of eager (call by value) languages, as it addresses problems such as repeated evaluation of an argument and ordering of side-effects. Lazy languages do not generally use CPS as an intermediate notation, partly due to tradition, but also because the explicit ordering makes it difficult to implement full laziness — i.e., avoiding evaluating a subterm more than once.

Abstract Machines

Compilation of lazy languages generally involves the implementation of an abstract machine which identifies an appropriate set of instructions for building graph representations of λ-terms and their weak head-normal forms.

Landin's SECD machine [Lan64] was the first abstract machine for reducing λ-terms. SECD stands for Stack, Environment, Control, Dump, which are the machine's internal registers. The memory of the SECD machine contains lists and integers, and the instruction set contains instructions for building lists and **closures**. A closure is a pair of the term and an environment containing representations of the free variables in the term; effectively this represents a suspended computation. This machine was originally used to implement ISWIM [Lan66], and a lazy version was developed for LispKit [HJJ82].

The Krivine machine is a well known method for normalisation of λ-terms [HMP96, WF03]. It is an evaluation machine which simulates weak-head reduction. In the Krivine machine, **closed** λ-terms are represented as closures. A closed term is a term containing no free variables. Representing terms this way avoids repeated substitution; the machine takes a closure and a stack, and returns a closure. At the end of the computation, there is a simple transformation (unloading) from closures to closed λ-terms which performs substitutions across the whole term in one pass.

Johnsson's G-machine [Joh84] shares several characteristics with the SECD machine. It too has stack, environment, control and dump registers; its novel features are the updating and sharing of graphs within the abstract machine and the inclusion of an output stream (although this is not an essential feature). A compiler based on the G-machine transforms function definitions into a set of supercombinators, which are functions with no free variables. G-code, the language executed by the G-machine, consists of instructions which build a graph representation of these supercombinators; a supercombinator is compiled into code which builds a graph and updates the root of the current reducible expression with that graph. The G-machine is a standard technique for implementing lazy functional languages, including Lazy ML, the Haskell B compiler and the `nhc` Haskell compiler [Röj95]. It is described further in [Pey87, PL92], and later in this chapter where it is used to implement ExTT.

Several variants and developments of the G-machine idea exist, such as the $\langle \nu, G \rangle$-

machine [AJ89] which is a modification geared towards parallel execution. Another abstract machine which takes several ideas from the G-machine is the ABC machine [SNvP91], used for the execution of Concurrent Clean. The design is very similar, but is focused on how the abstract machine code will ultimately be executed on a concrete machine. The Three Instruction Machine (TIM) [FW87] takes a different approach to representing function application nodes in the graph, preferring a spineless approach in which application nodes are represented as pairs of a code pointer and a tuple of arguments.

GHC is based on the Spineless Tagless G-machine (STG) [Pey92], which takes ideas from both the G-machine approach and the TIM approach. This machine deals with free variables internally, which eliminates the need for building supercombinators. Also, there is a uniform representation of closures which avoids the need for a distinction between constructor nodes and application nodes on the heap (hence the name tagless) — each closure is associated with a code pointer which evaluates and updates the closure; in the case of constructors, the closure is already evaluated so the code pointer points to a function which does nothing (in the simplest case) or returns a pointer to code for the appropriate case (in optimised cases). The STG machine has a more abstract code resembling a functional language, rather than an imperative instruction sequence like the G-machine. Nevertheless, STG code has an operational semantics which is translated into an internal representation called Abstract C, then finally into C or machine code.

A more recent development is GRIN (Graph Reduction Intermediate Notation, [BJ96, Boq99]) GRIN is a more low level highly optimisable notation for graph reduction. Its principal advantage is the ability to use heap analysis to eliminate unknown control flow due to evaluations and higher order functions, while still maintaining a functional style suitable for program transformations.

Strong Reduction

Grégoire and Leroy have developed a compiled implementation of strong reduction (using call by value semantics) within the Coq system [GL02]. Abstract machines are generally geared towards producing weak head-normal forms, not reducing under binders. However, when checking types in a dependent type theory such as the CIC implemented in Coq, we need to reduce under binders and deal with free variables. Grégoire and Leroy's abstract machine is a modification of the OCaml run-time machine, ZAM, extended with the ability to manipulate free variables. We could imagine this technique being used to implement strong reduction of TT using lazy evaluation by extending the G-machine in a similar way.

3.1.3 Program Extraction

Another possibility for producing executable code, rather than compiling to an abstract machine language, is to output code in another functional language by program extraction [PM89, Let02]. Extraction generally refers to the derivation of a simply typed program

from a proof of its specification — this involves stripping type expressions and proof irrelevant structures from code and could equally well apply to the translation of a dependently typed program into a simply typed form.

This method reduces the problem of compilation to a simpler problem, that of expressing a dependently typed term with a simple type. As a result, we get all the advantages of using the well tested, efficient and optimised run-time system of an already existing language. Unfortunately, it is not always possible to extract a term with an appropriate type (consider a function whose return type differs depending on its input, for example) and furthermore, we do not get the possibility of applying any low level optimisations based on dependent type information.

3.1.4 Execution of Epigram

Phase Distinction

The nature of a dependently typed programming language leads to there being some blurring of the distinction between compile-time and run-time, in that it is not immediately obvious which functions will be executed at compile-time and which functions will only be executed at run-time. Cardelli claimed in [Car88] that as a result types cannot be erased at run-time, although Augustsson showed for Cayenne that this was not the case [Aug98], since Cayenne has no means to analyse types at run-time (i.e., a casetype construct). Similarly, EPIGRAM has no way to examine types at run-time.

What happens is that there are two settings in which a function may be evaluated. In the first setting, during typechecking, functions are evaluated in order to check convertibility of terms. We will refer to this as "compile-time evaluation". In this phase, *strong* normalisation is important, as we may need to reduce terms containing free variables. In the second setting, "run-time evaluation", evaluation of functions is an end in itself; we only consider reduction to weak-head normal form and can safely assume that there are no free variables.

Evaluation Strategy

The evaluation strategy we have chosen for EPIGRAM is lazy evaluation. There are several reasons in favour of both strict and lazy evaluation, but we chose lazy evaluation initially because of the number of arguments to both functions and constructors which exist only for the purpose of ensuring type correctness; lazy evaluation ensures that these will never be evaluated at run-time. We will also take a lazy evaluation strategy at compile-time, for two reasons; firstly, for consistency with the run-time system and secondly since it allows us to take advantage of the substitution mechanism of the meta-language, Haskell, which itself is a lazy language. However, it is worth noting that for EPIGRAM the distinction is not crucial — since terms are strongly normalising, reduction will terminate at the same normal form whichever strategy we choose.

3.2 The Run-Time Language RunTT

3.2.1 Supercombinators and Lambda Lifting

RunTT is an intermediate language of supercombinators used to facilitate the compilation to abstract machine code. A **supercombinator s** is a λ-abstraction of the form:

$$\mathbf{s} \mapsto \lambda\vec{x}.E$$

where $\vec{x}$ is a series of zero or more arguments and E contains no λ-abstractions, such that **s** has no free variables. Having no free variables is a big advantage at compile-time — we can compile a fixed code sequence for each supercombinator without having to worry about external effects.

Supercombinators are an extension of **combinators**, which are λ-abstractions containing no occurrences of a free variable [Bar84]; the extension is that supercombinators can also be constants (that is, have no arguments). Only three combinators are required to represent any function[1], given appropriate base types and primitives. These are:

$$\mathbf{S} \mapsto \lambda f;\ g;\ x.\, f\ x\ (g\ x)$$
$$\mathbf{K} \mapsto \lambda x;\ y.\, x$$
$$\mathbf{I} \mapsto \lambda x.\, x$$

Early implementations of lazy functional languages such as Turner's SASL [Tur79] used a transformation into **S**, **K** and **I** as the basis of compilation, along with some other combinators for optimisation purposes. The advantages of using this fixed set of combinators are that such a small set can easily be implemented in hardware and the reduction machine is fairly simple to implement. This simplicity comes at a cost, however — since the granularity of execution is so small, the translation to **SKI** combinators can result in large programs. So instead of using a fixed set, we choose an appropriate set of supercombinators for each user defined function by a process known as **lambda lifting** [Hug84, Joh85]. The first step of compilation from ExTT is to lambda lift the ExTT terms into a run-time language, RunTT.

3.2.2 RunTT Syntax

The syntax of RunTT is presented in figure 3.2. The main features which distinguish RunTT from the execution language ExTT are:

- λ bindings appear only at the top level of terms; there are no inner λs and no free variables.
- All constructor applications (including type constructors) are fully applied.
- There is a case construct — in ExTT case analysis is performed by pattern matching the ι-schemes of elimination rules and implemented by ι-reduction; definition of elimination

[1] It is even possible with two, since $\mathbf{I} \mapsto \mathbf{S}\,\mathbf{K}\,\mathbf{K}$

rules in RunTT is via this case construct, which arise by compilation of the pattern matching ι-schemes. We call the term which is analysed by the case expression the **scrutinee**.

Type information, although it is not executable, is retained as a potential aid to optimisation; I will generally suppress the type label on λs since at this stage it serves no computational purpose.

$s ::=$	$\lambda \vec{a} : \vec{e}.\ e$	(supercombinator)			
$e ::=$	x	(bound variable)	$\mid$	f	(global name)
$\mid$	$\forall x{:}e.\ e$	(function space)	$\mid$	$\star_i$	(type of types)
$\mid$	$e\ e$	(function application)	$\mid$	$\mathsf{c}\langle\vec{e}\rangle$	(constructor application)
$\mid$	$\underline{\text{let}}\ a : e \mapsto e\ \underline{\text{in}}\ e$	(let binding)	$\mid$	$\mathsf{D}\langle\vec{e}\rangle$	(type constructor application)
$\mid$	$\underline{\text{case}}\ e\ \underline{\text{of}}\ \vec{alt}$	(case expression)			
$alt ::=$	$\mathsf{c}\ \langle\vec{x}\rangle \rightsquigarrow e$	(case alternative)			

Figure 3.2: The supercombinator language, RunTT

For simplicity of run-time representation, we ensure that all constructors are fully applied. This is straightforward to achieve, by η-expansion of all constructors which are not fully applied. The advantage of doing this is that at run-time we will always know, from the arity of a constructor, how much space to allocate for it. In a higher order language, it is not possible to do the same thing for function applications, and especially not in ExTT where the arity of a function may differ according to its input.

Constructor applications are given a separate syntax, $\mathsf{c}\langle\vec{e}\rangle$ to indicate that they are always fully applied. c itself is the **tag** of the constructor; I will present these as constructor names for readability, but in practice they are represented by integers. This integer can be used as an index into the jump table representing the alternatives in a case expression for which the constructor is the scrutinee.

RunTT is not strongly normalising, nor is it necessary or beneficial for it to be so. Since RunTT terms arise from programs in a strongly normalising language, we can be sure that programs in RunTT terminate (provided, of course, that the transformation to supercombinators is correct). In a naïve setting, we can also show termination by checking that case expressions make recursive calls on structurally smaller values. However, to require RunTT programs to be structurally recursive in general would give a lot less freedom for optimisation — in particular, we would not be able to remove the level of abstraction introduced by having to show termination for non-structurally recursive functions such as **quicksort**. Another consideration is that RunTT could potentially also be used as the run-time language for a language other than ExTT which may not be strongly normalising.

3.3 Translating Function Definitions to RunTT

We begin with a mapping from names to TT terms, `Defs`. A name maps to either a user defined function, a data or type constructor, or an elimination rule:

```
infix 1 :::
data thing ::: type = thing ::: type

data NameDef = Fun Term
             | Con Int
             | TyCon Int
             | Elim Patterns
type Defs = [(Name,(NameDef ::: Term))]
```

The type `Defs` describes the global context Γ. Pairing of terms with their types is implemented by the infix constructor `:::` rather than simply a tuple, for clarity. The type `NameDef` describes each possible entry in the context, taking function definitions, data constructors, type constructors and pattern matching elimination rule definitions separately.

For each name which maps to a term (i.e., implemented with the `Fun` constructor of `NameDef`), that term is either a CAF (a constant applicative form) taking no arguments, or a λ-binding. How do we translate these definitions to RunTT? There are some intermediate steps and representations involved; let us now consider these steps.

3.3.1 Grouping λ-abstractions

Before we start, we ensure that all data and type constructors are fully applied. This is simple to achieve, by η-expansion, if any are not fully applied. Since RunTT expects all constructors to be fully applied, it is wise to apply this step while we are still allowed inner λ-abstractions, rather than complicating lambda lifting further.

The first stage in translating a function definition into RunTT is to allow λ-bindings of more than one argument. In TT, all λ-abstractions are of arity 1. So, for example, in a binding $\lambda x : X.\ \lambda y : Y.\ e$, if x appears in e then x is free in e. It is more convenient if λ-bindings of more than one argument are allowed — here this results in the binding $\lambda x : X; y : Y.\ e$, where x and y are both bound in e.

We achieve this by repeatedly applying a grouping transformation $[\![\cdot]\!]_G$ to the term:

$$[\![\lambda \vec{x} : \vec{X}.\ \lambda y : Y.\ e]\!]_G \implies [\![\lambda \vec{x} : \vec{X}; y : Y.\ e]\!]_G$$
$$[\![\lambda \vec{x} : \vec{X}.\ e]\!]_G \implies \lambda \vec{x} : \vec{X}.\ e$$

The default case of $[\![\cdot]\!]_G$ traverses the term looking for λ-bindings. This transformation identifies where the scope of a λ-binding is itself a λ-binding, and merges them into one λ-binding. Naturally, terms here fit neither into the syntax of TT (since λ binds multiple arguments) nor RunTT (since there may be inner λs) so we use an intermediate representation

in which the only difference from TT is to allow binding of multiple arguments.

The notation $\lambda x : X; y : Y$, with the arguments separated by a semicolon, denotes that x and y are both bound by the same λ. In this setting, we use de Bruijn levels rather than de Bruijn indices to represent variable names, as we can then think of the index i as the ith argument to a function. (e.g. in $\lambda x : X; y : Y$, x is represented by 0 and y by 1).

As an example of the grouping step, consider the **plus** function, defined in TT as follows:

$\mathbf{plus} \ : \ \forall n{:}\mathbb{N}.\forall m{:}\mathbb{N}.\mathbb{N}$
$\mathbf{plus} \mapsto \lambda n, m{:}\mathbb{N}.\ \mathbb{N}\text{-}\mathbf{Elim}\ n\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ m\ (\lambda k{:}\mathbb{N}.\lambda ih{:}\mathbb{N}.\mathsf{s}\ ih)$

Grouping the arguments results in the following definition:

$\mathbf{plus} \mapsto \lambda n; m \ : \ \mathbb{N}.\ \mathbb{N}\text{-}\mathbf{Elim}\ n\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ m\ (\lambda k; ih{:}\mathbb{N}.\ \mathsf{s}\ ih)$

3.3.2 Lambda Lifting

The second stage is to lift out all remaining inner λ-abstractions and let bindings to the top level, removing free variables. We do this by giving each inner abstraction and binding a unique new name and replacing their occurrence with their name. In the case of the **plus** function, this results in the following (assuming the names **plus1** and **plus2** are not defined elsewhere:

$\mathbf{plus} \mapsto \lambda n; m \ : \ \mathbb{N}.\ \mathbb{N}\text{-}\mathbf{Elim}\ n\ \mathbf{plus1}\ m\ \mathbf{plus2}$
$\mathbf{plus1} \mapsto \lambda n{:}\mathbb{N}.\ \mathbb{N}$
$\mathbf{plus2} \mapsto \lambda k; ih{:}\mathbb{N}.\ \mathsf{s}\ ih$

In this case, the resulting function definitions contain no free variables. However, this is not always the case. Consider the following (uncurried) definition:

$\mathbf{f} \mapsto \lambda x; y \ : \ \mathbb{N}.\ \underline{\text{let}}\ z \ : \ \mathbb{N} \mapsto \mathbf{plus}\ x\ y\ \underline{\text{in}}\ \mathbf{plus}\ z\ z$

Lifting out the inner let binding results in the following set of top level definitions:

$\mathbf{f} \mapsto \lambda x; y \ : \ \mathbb{N}.\ \mathbf{plus\ f1\ f1}$
$\mathbf{f1} \mapsto \mathbf{plus}\ x\ y$

There is clearly a problem here — x and y are free in **f1**; the function has no hope of accessing the appropriate x and y unless it is given more information. The solution is to add x and y as arguments to **f1**, and change the application in **f** to pass through appropriate x and y:

$\mathbf{f} \mapsto \lambda x; y \ : \ \mathbb{N}.\ \mathbf{plus}\ (\mathbf{f1}\ x\ y)\ (\mathbf{f1}\ x\ y)$
$\mathbf{f1} \mapsto \lambda x; y \ : \ \mathbb{N}.\ \mathbf{plus}\ x\ y$

Johnsson describes an effective algorithm for determining the free variables in a set of function definitions by solving set equations in [Joh85]. Although in this example, it was clear which arguments were to be abstracted out, the optimal solution is not always so obvious, especially where there are nested let bindings or mutually recursive functions.

We should note that in this example, we have lost full laziness (i.e., avoiding executing any subterm more than once) by lifting z out — evaluating $\mathbf{f}$ will involve evaluating $\mathbf{f1}\ x\ y$ twice! This kind of problem can be solved by a separate full laziness pass [PL91b] which identifies maximal free expressions prior to lambda lifting.

The purpose of lambda lifting is to remove free variables in order to make compilation easier. In the STG machine, however, free variables are kept; Santos notes in [San95] that there is a performance penalty in the resulting code where free variables are removed. Conversely, GRIN [BJ96] does compile from supercombinators generated by the hbcc Haskell compiler and gets encouraging results, yielding code several times faster than that produced by the STG machine in many cases. GRIN's performance comes largely from the ability to eliminate unknown control flow from programs (due in part to higher order functions) and therefore allowing a more sophisticated heap analysis.

3.3.3 Tidying up

The final step, now that we have top level functions with no inner λ-abstractions and no free variables, is to translate the definition into RunTT syntax. The only difference now is the constructor syntax which represents fully applied constructors only — we have already ensured that all constructors are fully applied, so there is a simple mapping to RunTT. In the case of **plus**, we get the following RunTT supercombinators:

$$\mathbf{plus} \mapsto \lambda n; m\ :\ \mathbb{N}\langle\rangle.\ \mathbb{N}\textbf{-Elim}\ n\ \mathbf{plus1}\ m\ \mathbf{plus2}$$
$$\mathbf{plus1} \mapsto \lambda n{:}\mathbb{N}\langle\rangle.\ \mathbb{N}\langle\rangle$$
$$\mathbf{plus2} \mapsto \lambda k; ih{:}\mathbb{N}\langle\rangle.\ \mathsf{s}\langle ih\rangle$$

3.3.4 Arity

What is the arity of the function **adder** in the following EPIGRAM declaration?

$$\underline{\text{let}}\quad \frac{n\ :\ \mathbb{N}}{\mathbf{adderType}\ n\ :\ \star}\quad \begin{array}{lll} \mathbf{adderType} & n & \Leftarrow \underline{\text{elim}}\ n \\ \mathbf{adderType} & 0 & \mapsto \mathbb{N} \\ \mathbf{adderType} & (\mathsf{s}\ k) & \mapsto \mathbb{N} \rightarrow \mathbf{adderType}\ k \end{array}$$

$$\underline{\text{let}}\quad \frac{n, a\ :\ \mathbb{N}}{\mathbf{adder}\ n\ a\ :\ \mathbf{adderType}\ n}\quad \begin{array}{lll} \mathbf{adder} & n\ a & \Leftarrow \underline{\text{elim}}\ n \\ \mathbf{adder} & 0\ a & \mapsto a \\ \mathbf{adder} & (\mathsf{s}\ k)\ a & \mapsto \lambda n{:}\mathbb{N}.\ \mathbf{adder}\ k\ (\mathbf{plus}\ a\ n) \end{array}$$

The arity depends on the input n; the number of arguments expected is $n + 1$. At run-time it is often helpful to know the arity of a supercombinator to check whether it is fully applied. Do dependent types cause some difficulty here? The lambda lifted version of **adder** in RunTT is as follows (eliding the argument types):

$\textbf{adder} \mapsto \lambda n.\ \mathbb{N}\textbf{-Elim}\ n\ \textbf{adder1}\ \textbf{adder2}\ \textbf{adder3}$
$\textbf{adder1} \mapsto \lambda m.\ \mathbb{N}\langle\rangle \to \textbf{adderType}\ m$
$\textbf{adder2} \mapsto \lambda a.\ a$
$\textbf{adder3} \mapsto \lambda k; ih; a; n.\ ih\ (\textbf{plus}\ a\ n)$

Conveniently, due to lambda lifting, each of these supercombinators are of known arity, as is $\mathbb{N}$-**Elim** which is called by **adder**. What happens is that **adder** returns a function if given $\mathsf{s}\ k$, or a constructor if given $\mathsf{0}$. We can get the arity of a supercombinator simply by counting the variables bound by the λ.

3.4 Translating Elimination Rules to RunTT

In addition to translating TT functions into compilable supercombinators, we need a way to translate pattern matching elimination rules into a compilable form. For this, we translate into Augustsson style case expressions [Aug85]. For elimination rules, the algorithm for doing so is rather simpler than Augustsson's algorithm since we know in advance that we can make the necessary case distinction on the target of the elimination rule; for each ι-scheme, the pattern in the target argument's position is a different constructor form. We know this must be the case, because ι-schemes are machine generated and built only from data declarations. Given a set of ι-schemes for a family $\mathsf{D}\ \vec{s}$:

$$\mathsf{D}\textbf{-Elim}\ \vec{s}\ (\mathsf{c}_1\ \vec{a}_1\ \vec{y}_1)\ P\ \vec{m} \rightsquigarrow m_{\mathsf{c}_1}\ \ldots$$
$$\ldots$$
$$\mathsf{D}\textbf{-Elim}\ \vec{s}\ (\mathsf{c}_n\ \vec{a}_n\ \vec{y}_n)\ P\ \vec{m} \rightsquigarrow m_{\mathsf{c}_n}\ \ldots$$

Case distinction is made on the constructors of the target, c_i, and we know that the right hand side refers only to the arguments of these constructors and the names of the other arguments. Thus, we take the target of the elimination rule as the scrutinee of the case expression, and translate into RunTT as follows:

$$\begin{aligned}\mathsf{D}\textbf{-Elim} \mapsto\ & \lambda\vec{s};\ c;\ P;\ \vec{m}.\ \underline{\text{case}}\ c\ \underline{\text{of}}\\ & \quad \mathsf{c}_1\langle\vec{a}_1, \vec{y}_1\rangle \rightsquigarrow m_{\mathsf{c}_1}\ \ldots\\ & \ldots\\ & \quad \mathsf{c}_n\langle\vec{a}_n, \vec{y}_n\rangle \rightsquigarrow m_{\mathsf{c}_n}\ \ldots\end{aligned}$$

D-Elim in this form is a lambda lifted supercombinator, since there are no inner lambda abstractions. For example, elimination on natural numbers, $\mathbb{N}$-**Elim**, is translated to the following case expression:

$$\begin{aligned}\mathbb{N}\textbf{-Elim} \mapsto\ & \lambda n;\ P; m_0; m_\mathsf{s}.\ \underline{\text{case}}\ n\ \underline{\text{of}}\\ & \quad \mathsf{0}\langle\rangle \rightsquigarrow m_0\\ & \quad \mathsf{s}\langle k\rangle \rightsquigarrow m_\mathsf{s}\ k\ (\mathbb{N}\textbf{-Elim}\ k\ P\ m_0\ m_\mathsf{s})\end{aligned}$$

3.5 The G-machine

Later in this thesis I will be discussing optimisations and transformations at the ExTT and RunTT levels rather than at the lower level of abstract machine. Nevertheless, let us consider the design of an abstract machine for RunTT in order that we may see what effect the design decisions we make at the RunTT level have on the abstract machine. Further implementation details of the G-machine are given in Appendix D.

The principle behind the G-machine [Joh84, Aug84] is to build graph bodies from a series of sequentially executed abstract machine instructions, called G-code. Each supercombinator is compiled to a sequence of instructions which instantiates the body of the supercombinator in the machine's memory; this results in several optimisations. The decision of which subexpression to reduce is made at compile-time rather than run-time. Also, the abstract language is finally in an imperative form which allows a more direct mapping into a real machine code or programming language.

Choice of Abstract Machine

We use the G-machine here since it is a standard, well-understood and well-documented approach for implementing run-time systems for lazy languages. This is not the fastest or most modern abstract machine (GRIN [Boq99] and STG code [Pey92] are more efficient), but is relatively straightforward to implement for experimentation with higher level optimisations based on the type system of TT. It is not an essential feature of EPIGRAM, ExTT or even RunTT that the G-machine is used as a back end, nor is it essential to any of the optimisations we will present later.

One problem in particular with the G-machine is that it is not abstract enough; G-code is fairly low level and therefore does not necessarily map well onto any specific CPU (e.g., it is stack based whereas many CPUs are register based). This problem is addressed by the STG machine whose language has a more functional flavour. An interesting topic of further research would be to examine how dependent type systems might affect the implementation of an abstract machine. We will later be making some modifications to the G-machine, in particular to deal with elimination rules efficiently, and will identify some things which ought to be taken into consideration in the design of an abstract machine for a dependently typed language.

3.5.1 Graph Representation

Each supercombinator is compiled to a series of abstract machine instructions which, when executed, construct an instance of the supercombinator body. To this end, we need to consider how a supercombinator body is represented at run-time. A supercombinator can build any of the following:

- A closure representing an unevaluated (suspended) function application.

- A function applied to no arguments.
- A fully applied constructor.
- A constant (in RunTT as it stands, these are only the type universes $\star_i$).

A graph node can therefore be one of:

- APP f a, where f and a are graphs representing a function body and its argument
- FUN n, where n is the name of a function.
- CON t xs, where t is the constructor tag, and xs is a list of known length.
- TYPE, which stands for any type. As there is no casetype construct or equivalent form of universe elimination, there is no way to eliminate on types so distinguishing between them in the evaluation graph would serve no purpose. There is only one such node; all references to it are shared. We could, however, imagine extending the machine so that it did allow elimination over types, by adding heap nodes for representing type constructors; doing so may help with the implementation of polymorphic functions as in [HM95].

These graphs are stored on the **heap**, which is a garbage collected global store.

3.5.2 Machine State

The G-machine state is a tuple, $\langle C, S, G, E, D \rangle$ where

- C is the code sequence currently being executed. This is a list of G-machine instructions.
- S is a stack of node names pointing into the graph.
- G is the graph, which maps node names to heap nodes.
- E is the global environment mapping function names to a pair of their arity and their code.
- D is a dump for recursive evaluations, effectively a call stack. This is a stack of pairs, where each pair holds a stack of node names (S before the evaluation) and a G-code sequence (C before the evaluation).

Johnsson's original G-machine was a 7-tuple, the extra elements being o, an output stream to which the result of evaluation is printed and V, a stack of basic (primitive) values for storing the results of intermediate computations. I have left out the output stream to concentrate on the evaluation of graphs. Our language of supercombinators (at the moment) has only constructors of inductive families as canonical forms so I omit V. I will discuss

the addition of primitive types into the language in Chapter 5 — we can generate suitable forms for output by introducing strings as a primitive and writing a **show** function for each type.

3.5.3 Informal Semantics

G-machine instructions can be divided into several groups. There are instructions for managing the stack, instructions for building and deconstructing graphs and instructions for controlling evaluation and execution. The basic form of any G-machine program is to build a graph and evaluate it.
The stack management instructions include:

- PUSH i, which pushes the value at the offset i from the top of the stack onto the top of the stack. This results in two copies of the value on the stack.
- PUSHFUN f, which pushes the value FUN f onto the top of the stack.
- MOVE i, which moves the value at the top of the stack to the offset i from the top of the stack, which has the effect of reordering the stack.
- DISCARD n, which discards the top n stack items, which may be garbage collected later.
- SLIDE i, which discards the i stack items below the top item (that is, leaving the top item intact, it discards from item 1 to item $i + 1$).

Graph construction and deconstruction instructions include:

- MKAP, which builds an application node applying the second item on the stack (the function) to the first item on the stack (the argument), placing the application node on the stack.
- MKCON i c, which builds a constructor application node applying the constructor c to the top i items on the stack.
- MKTYPE, which creates a reference to the graph TYPE.
- SPLIT n, which, assuming the graph at the top of the stack is of the form CON$\langle x_1, \ldots, x_n \rangle$, pushes $\vec{x}$ onto the top of the stack, with x_n pushed last.

The presence of MKTYPE may be surprising, since EPIGRAM and ExTT have no means of examining types, which suggests that all types can be erased. It is not completely clear that this is the case however; whether it is possible depends to some extent on the implementation of universes, for example. In the naïve compilation path, therefore, we do not remove types. Later, in the optimised compilation path, we will see some methods for removing types which can be shown never to be examined.
Evaluation and execution control instructions include:

- EVAL, which evaluates the item at the top of the stack to canonical form (that is, head-normal form).
- JUMP l, which jumps to the label l.
- CASEJUMP $(c_1, l_1), \ldots, (c_n, l_n)$, which examines the top stack item (which is assumed to be in canonical form) and jumps to the label appropriate to the constructor at the head of the graph.
- LABEL l, which defines the target of a JUMP or CASEJUMP instruction.
- UPDATE i, which updates the item at offset i from the top of the stack with the item at the top of the stack.
- RET n, which discards n stack items and continues execution from the point where the previous EVAL was made.

The instructions give the basic evaluation behaviour of the G-machine, on which the translation scheme I will present next is based. I will shortly add further instructions to cover proper tail recursion, and later extend the G-machine with instructions to implement elimination rules efficiently.

3.5.4 Operational Semantics

Since the G-machine is a state machine, its formal semantics are defined by state transition rules, presented in figure 3.3. I use the following notational conventions:

- The code sequence, C, is presented as a sequence of instructions separated by semicolons, as in the translation scheme, and terminated by a pair of brackets, e.g. $i_0; i_1; \ldots; ()$.
- The stack, S, is presented as a sequence of names which are pointers into the graph G, e.g. $n_0.n_1.()$.
- The graph G is the memory of the G-machine; $G[n = v]$ indicates that the name n refers to the value v in G. An empty graph is represented as $\{\}$, and update of a node n in the graph with a value v is denoted by $G\{n = v\}$.
- The environment E is a mapping from names to pairs of arity and code. $E[f = (a, c)]$ indicates that the supercombinator f has arity a and is built by the code sequence c.
- The dump D is effectively a call stack, presented as a sequence of pairs of code and a stack (i.e. closures).

Note that there is an additional instruction accounted for in this presentation, UNWIND. The machine is put into the UNWIND state by both EVAL and RET to unwind the spine of an application onto the stack.

$$
\begin{array}{lcl}
\langle \mathsf{PUSH}\ i;\ c, n_0 \ldots n_i.S, G, E, D\rangle & \Longrightarrow & \langle c, n_i.n_0 \ldots n_i.S, G, E, D\rangle \\
\langle \mathsf{PUSHFUN}\ f;\ c, S, G, E, D\rangle & \Longrightarrow & \langle c, n.S, G\{n = \mathsf{FUN}\ f\}, E, D\rangle \\
\langle \mathsf{MOVE}\ i;\ c, n_0 \ldots n_i.S, G, E, D\rangle & \Longrightarrow & \langle c, n_1 \ldots n_{i-1}.n_0.S, G, E, D\rangle \\
\langle \mathsf{SLIDE}\ i;\ c, n_0 \ldots n_i.S, G, E, D\rangle & \Longrightarrow & \langle c, n_0.S, G, E, D\rangle \\
\langle \mathsf{DISCARD}\ i;\ c, n_0 \ldots n_{i-1}.S, G, E, D\rangle & \Longrightarrow & \langle c, S, G, E, D\rangle \\
\langle \mathsf{MKAP};\ c, a.f.S, G, E, D\rangle & \Longrightarrow & \langle c, n.S, G\{n = \mathsf{APP}\ f\ a\}, E, D\rangle \\
\langle \mathsf{MKCON}\ i\ t;\ c, n_0 \ldots n_{i-1}.S, G, E, D\rangle & \Longrightarrow & \langle c, n'.S, G\{n' = \mathsf{CON}\ t\ (n_0 \ldots n_{i-1})\}, E, D\rangle \\
\langle \mathsf{MKTYPE};\ c, S, G, E, D\rangle & \Longrightarrow & \langle c, n.S, G[n = \mathsf{TYPE}], E, D\rangle \\
\langle \mathsf{SPLIT}\ i;\ c, n.S, G[n = \mathsf{CON}\ t\ (n_0 \ldots n_{i-1})], E, D\rangle & & \\
 & \Longrightarrow & \langle c, n_{i-1} \ldots n_0.n.S, G, E, D\rangle \\
\langle \mathsf{EVAL};\ c, n.S, G[n = \mathsf{APP}\ f\ a], E, D\rangle & \Longrightarrow & \langle \mathsf{UNWIND};\ (), n.(), G, E, (c, S).D\rangle \\
\langle \mathsf{EVAL};\ c, n.S, G[n = \mathsf{FUN}\ f], E, D\rangle & \Longrightarrow & \langle \mathsf{UNWIND};\ (), n.(), G, E, (c, S).D\rangle \\
\langle \mathsf{EVAL};\ c, n.S, G[n = \mathsf{CON}\ t\ \vec{a}], E, D\rangle & \Longrightarrow & \langle c, n.S, G, E, D\rangle \\
\langle \mathsf{EVAL};\ c, n.S, G[n = \mathsf{TYPE}], E, D\rangle & \Longrightarrow & \langle c, n.S, G, E, D\rangle \\
\langle \mathsf{CASEJUMP}\ (\mathsf{t}_1, l_1), \ldots, (\mathsf{t}_n, l_n); & & \\
\quad \mathsf{LABEL}\ l_1; c_1\ \ldots \mathsf{LABEL}\ l_n;\ c_n, & & \\
\quad n.S, G[n = \mathsf{CON}\ \mathsf{t}_i\ \vec{a}], E, D\rangle & \Longrightarrow & \langle c_i, n.S, G, E, D\rangle \\
\langle \mathsf{JUMP}\ l;\ \ldots \mathsf{LABEL}\ l;\ c, S, G, E, D\rangle & \Longrightarrow & \langle c, S, G, E, D\rangle \\
\langle \mathsf{LABEL}\ l;\ c, S, G, E, D\rangle & \Longrightarrow & \langle c, S, G, E, D\rangle \\
\langle \mathsf{UPDATE}\ i;\ c, n_0 \ldots n_i.S, G[n_0 = N_0], E, D\rangle & \Longrightarrow & \langle c, n_1 \ldots n_i.S, G\{n_i = N_0\}, E, D\rangle \\
\langle \mathsf{RET}\ i;\ c, n_0 \ldots n_{i-1}.n.S, G[n = \mathsf{APP}\ f\ a], E, D\rangle & \Longrightarrow & \langle \mathsf{UNWIND};\ (), n.S, G, E, D\rangle \\
\langle \mathsf{RET}\ i;\ c, n_0 \ldots n_{i-1}.n.S, G[n = \mathsf{FUN}\ f], E, D\rangle & \Longrightarrow & \langle \mathsf{UNWIND};\ (), n.S, G, E, D\rangle \\
\langle \mathsf{RET}\ i;\ c, n_0 \ldots n_{i-1}.n.S, G[n = \mathsf{CON}\ t\ a], E, (c', S').D\rangle & \Longrightarrow & \langle c', n.S', G, E, D\rangle \\
\langle \mathsf{RET}\ i;\ c, n_0 \ldots n_{i-1}.n.S, G[n = \mathsf{TYPE}], E, (c', S').D\rangle & \Longrightarrow & \langle c', n.S', G, E, D\rangle \\
\langle \mathsf{UNWIND};\ (), n.S, G[n = \mathsf{APP}\ f\ a], E, D\rangle & \Longrightarrow & \langle \mathsf{UNWIND};\ (), f.n.S, G[n = \mathsf{APP}\ f\ a], E, D\rangle \\
\langle \mathsf{UNWIND};\ (), n_0 \ldots n_i.S, & & \\
\quad G[n_0 = \mathsf{FUN}\ f, n_1 = \mathsf{APP}\ n_1'\ n_1'', \ldots, n_i = \mathsf{APP}\ n_i'\ n_i''], & & \\
\quad E[f = (i, c)], D\rangle & & \\
 & \Longrightarrow & \langle c, n_1'' \ldots n_i''.n_i.S, G, E, D\rangle \\
\langle \mathsf{UNWIND};\ (), n_0 \ldots n_i.(), G[n_0 = \mathsf{FUN}\ f], E[f = (a, c')], (c, S).D\rangle \text{ where } i < a & & \\
 & \Longrightarrow & \langle c, n_i.S, G, E, D\rangle
\end{array}
$$

Figure 3.3: State transitions for the G-machine

3.5.5 Translation Scheme

The translation scheme from RunTT to G-code is rather smaller than the translation scheme given in the original G-machine papers [Joh84, Aug84, Aug85] primarily because of the lack of primitive types in TT (however, see Chapter 5 for extensions in ExTT which implement low-level behaviour arising from high level EPIGRAM declarations). Canonical forms in TT consist only of constructor forms and basic types.

The top level translation scheme $\mathcal{S}[\![\cdot]\!]$, given in figure 3.4 gives code which reduces the body of a top level supercombinator to canonical form.

The $\mathcal{E}[\![\cdot]\!]$ translation scheme, given in figure 3.5, gives code to compute the canonical form of an expression and leaves the value on the top of the stack. This is the scheme which translates the body of function definitions.

The $\mathcal{C}[\![\cdot]\!]$ translation scheme, given in figure 3.6 gives code to construct the graph of an expression and leaves a pointer to the graph on the top of the stack. This scheme is called by the $\mathcal{E}[\![\cdot]\!]$ scheme for constructing graphs which are to be evaluated later, giving lazy semantics.

Given an environment of supercombinators E, and a RunTT supercombinator e to evaluate, the initial state of a G-machine to evaluate the supercombinator e is $\langle \mathcal{S}[\![e]\!], (), \{\}, E, () \rangle$.

3.5.6 Example — plus and ℕ-Elim

For the **plus** function compilation proceeds as follows:

$\mathbf{plus} \mapsto \lambda n; m.\ \mathbb{N}\text{-}\mathbf{Elim}\ n\ \mathbf{plus1}\ m\ \mathbf{plus2}$
$\mathcal{S}[\![\lambda n; m.\ \mathbb{N}\text{-}\mathbf{Elim}\ n\ \mathbf{plus1}\ m\ \mathbf{plus2}]\!]$
$\quad \Longrightarrow \mathcal{E}[\![\mathbb{N}\text{-}\mathbf{Elim}\ n\ \mathbf{plus1}\ m\ \mathbf{plus2}]\!]\ r\,3; \mathsf{UPDATE}\,3; \mathsf{RET}\,2$
$\quad \Longrightarrow \mathcal{C}[\![\mathbb{N}\text{-}\mathbf{Elim}\ n\ \mathbf{plus1}\ m\ \mathbf{plus2}]\!]\ r\,3; \mathsf{UPDATE}\,3; \mathsf{RET}\,2$
$\quad \Longrightarrow \mathcal{C}[\![\mathbb{N}\text{-}\mathbf{Elim}]\!]\ r\,3; \mathcal{C}[\![n]\!]\ r\,4; \mathsf{MKAP}; \mathcal{C}[\![\mathbf{plus1}]\!]\ r\,4; \mathsf{MKAP};$
$\qquad \mathcal{C}[\![m]\!]\ r\,4; \mathsf{MKAP}; \mathcal{C}[\![\mathbf{plus2}]\!]\ r\,4; \mathsf{MKAP}; \mathsf{UPDATE}\,3; \mathsf{RET}\,2$
$\quad \Longrightarrow \mathsf{PUSHFUN}\ \mathbb{N}\text{-}\mathbf{Elim}; \mathsf{PUSH}\,2; \mathsf{MKAP}; \mathsf{PUSHFUN}\ \mathbf{plus1}; \mathsf{MKAP};$
$\qquad \mathsf{PUSH}\,1; \mathsf{MKAP}; \mathsf{PUSHFUN}\ \mathbf{plus2}; \mathsf{MKAP}; \mathsf{UPDATE}\,3; \mathsf{RET}\,2$

The function r is defined such that $r(n) = 3$ and $r(m) = 2$. Compilation of **plus1** and **plus2** is rather simpler; the compilation to G-code of **plus**, **plus1** and **plus2** are shown in figure 3.7

The compilation of $\mathbb{N}$-**Elim** requires dealing with a case expression. The G-machine code for $\mathbb{N}$-**Elim** is given in figure 3.8.

3.5.7 Implementing a G-machine Compiler With Dependent Types

The G-machine compiler we have seen here has been implemented as a simply typed translation, since its implementation is in Haskell. What benefit would we get from implementing this program in a dependently typed language? Let us consider the invariants which need to be maintained in the compiler and look at the sort of errors which could occur. In the course of developing the G-machine compiler, the main sources of errors were:

- Incorrect stack manipulation (for example, stack overflows due to incorrect variable indexing).
- Attempting case analysis on a value which is not yet in canonical form (due to a missing EVAL).

Instances of this kind of error can be reduced by giving a dependently typed representation to G-code. Here, we will briefly consider how this might be achieved. Given the main sources of error, occurring in stack manipulation and in analysing non-canonical values, we

$$\mathcal{S}[\![\lambda \vec{a} : \vec{E}.\ e]\!] \Longrightarrow \mathcal{E}[\![e]\!]\ r\ (m+1);\ \mathsf{UPDATE}\ m+1;\ \mathsf{RET}\ m$$
$$\text{where } m \Longrightarrow \textsc{length}(\vec{a})$$
$$r(a_i) \Longrightarrow (m+2) - i$$

Figure 3.4: The $\mathcal{S}[\![\cdot]\!]$ translation scheme

$$\mathcal{E}[\![x]\!]\ r\ n \Longrightarrow \mathsf{PUSH}\ n - r(x);\ \mathsf{EVAL}$$
$$\mathcal{E}[\![\underline{\text{case}}\ e\ \underline{\text{of}}\ \mathsf{c}_1\langle\vec{a}_1\rangle \rightsquigarrow e_1 \ldots \mathsf{c}_n\langle\vec{a}_n\rangle \rightsquigarrow e_n]\!]\ r\ n \Longrightarrow$$
$$\quad \mathcal{E}[\![e]\!]\ r\ n;\ \mathsf{CASEJUMP}\ (\mathsf{c}_1, l_1)\ (\mathsf{c}_2, l_2) \ldots (\mathsf{c}_n, l_n);$$
$$\quad\quad \mathsf{LABEL}\ l_1;\ \mathsf{SPLIT}\ n_1;\ \mathcal{E}[\![e_1]\!]\ d_1\ n + n_1;\ \mathsf{MOVE}\ n_1 + 1; \mathsf{DISCARD}\ n_1 + 1;\ \mathsf{JUMP}\ l$$
$$\quad\quad \ldots$$
$$\quad \mathsf{LABEL}\ l$$
$$\quad\quad \text{where } d_n(a_{ij}) \Longrightarrow n + j$$
$$\quad\quad\quad d_n(x) \Longrightarrow r(x)$$
$$\quad\quad\quad n_k = \textsc{length}(\vec{a}_k)$$
$$\mathcal{E}[\![\underline{\text{let}}\ a \mapsto e_1\ \underline{\text{in}}\ e_2]\!]\ r\ n \Longrightarrow \mathcal{C}[\![e_1]\!]\ r\ n;\ \mathcal{E}[\![e_2]\!]\ r'\ (n+1);\ \mathsf{SLIDE}\ 1$$
$$\quad \text{where } r'(a) \Longrightarrow n + 1$$
$$\quad\quad r'(x) \Longrightarrow r(x)$$
$$\mathcal{E}[\![e]\!]\ r\ n \Longrightarrow \mathcal{C}[\![e]\!]\ r\ n$$

Figure 3.5: The $\mathcal{E}[\![\cdot]\!]$ translation scheme

$$\mathcal{C}[\![f]\!]\ r\ n \Longrightarrow \mathsf{PUSHFUN}\ f$$
$$\mathcal{C}[\![x]\!]\ r\ n \Longrightarrow \mathsf{PUSH}\ n - r(x)$$
$$\mathcal{C}[\![\star_i]\!]\ r\ n \Longrightarrow \mathsf{MKTYPE}$$
$$\mathcal{C}[\![\forall x{:}e_1.\ e_2]\!] \Longrightarrow \mathsf{MKTYPE}$$
$$\mathcal{C}[\![\mathsf{c}\langle e_1, e_2, \ldots, e_i\rangle]\!]\ r\ n \Longrightarrow \mathcal{C}[\![e_1]\!]\ r\ n;\ \mathcal{C}[\![e_2]\!]\ r\ (n+1); \ldots;$$
$$\mathcal{C}[\![e_i]\!]\ r\ (n+i-1);\ \mathsf{MKCON}\ \mathsf{c}\ i$$
$$\mathcal{C}[\![\mathsf{D}\langle\vec{e}\rangle]\!] \Longrightarrow \mathsf{MKTYPE}$$
$$\mathcal{C}[\![e_1\ e_2]\!]\ r\ n \Longrightarrow \mathcal{C}[\![e_1]\!]\ r\ n;\ \mathcal{C}[\![e_2]\!]\ r\ n+1;\ \mathsf{MKAP}$$
$$\mathcal{C}[\![\underline{\text{let}}\ a \mapsto e_1\ \underline{\text{in}}\ e_2]\!]\ r\ n \Longrightarrow \mathcal{C}[\![e_1]\!]\ r\ n;\ \mathcal{C}[\![e_2]\!]\ r'\ (n+1);\ \mathsf{SLIDE}\ 1$$
$$\quad \text{where } r'(a) \Longrightarrow n + 1$$
$$\quad\quad r'(x) \Longrightarrow r(x)$$

Figure 3.6: The $\mathcal{C}[\![\cdot]\!]$ translation scheme

$$
\begin{array}{l}
\textbf{plus} \mapsto \lambda n; m.\ \mathbb{N}\textbf{-Elim}\ m\ \textbf{plus1}\ n\ \textbf{plus2} \\
\quad \mathcal{S}[\![\lambda n; m.\ \mathbb{N}\textbf{-Elim}\ m\ \textbf{plus1}\ n\ \textbf{plus2}]\!] \\
\qquad \Longrightarrow \mathsf{PUSHFUN}\ \mathbb{N}\textbf{-Elim};\ \mathsf{PUSH}\ 2;\ \mathsf{MKAP};\ \mathsf{PUSHFUN}\ \textbf{plus1};\ \mathsf{MKAP}; \\
\qquad\quad \mathsf{PUSH}\ 1;\ \mathsf{MKAP};\ \mathsf{PUSHFUN}\ \textbf{plus2};\ \mathsf{MKAP};\ \mathsf{UPDATE}\ 3;\ \mathsf{RET}\ 2 \\
\textbf{plus1} \mapsto \lambda n.\ \mathbb{N}\langle\rangle \\
\quad \mathcal{S}[\![\lambda n.\ \mathbb{N}\langle\rangle]\!] \Longrightarrow \mathsf{MKTYPE};\ \mathsf{UPDATE}\ 2;\ \mathsf{RET}\ 1 \\
\textbf{plus2} \mapsto \lambda k; ih.\ \mathsf{s}\langle ih\rangle \\
\quad \mathcal{S}[\![\lambda k; ih.\ \mathsf{s}\langle ih\rangle]\!] \Longrightarrow \mathsf{PUSH}\ 0; \mathsf{MKCON}\ \mathsf{s}\ 1; \mathsf{UPDATE}\ 3;\ \mathsf{RET}\ 2
\end{array}
$$

Figure 3.7: Compilation of **plus** to G-machine code

$$
\begin{array}{l}
\mathbb{N}\textbf{-Elim} \mapsto \lambda n;\ P; m_0; m_\mathsf{s}.\ \underline{\mathsf{case}}\ n\ \underline{\mathsf{of}} \\
\qquad\qquad 0 \rightsquigarrow m_0 \\
\qquad\qquad \mathsf{s}\langle k\rangle \rightsquigarrow m_\mathsf{s}\ k\ (\mathbb{N}\textbf{-Elim}\ k\ P\ m_0\ m_\mathsf{s}) \\
\quad \mathcal{S}[\![\lambda n;\ P; m_0; m_\mathsf{s}.\ \underline{\mathsf{case}}\ n\ \underline{\mathsf{of}} \\
\qquad 0 \rightsquigarrow m_0 \\
\qquad \mathsf{s}\langle k\rangle \rightsquigarrow m_\mathsf{s}\ k\ (\mathbb{N}\textbf{-Elim}\ k\ P\ m_0\ m_\mathsf{s})]\!] \\
\qquad \Longrightarrow \mathsf{PUSH}\ 3; \mathsf{EVAL}; \mathsf{CASEJUMP}\ (0, l_0)\ (\mathsf{s}, l_\mathsf{s}); \\
\qquad\qquad \mathsf{LABEL}\ l_0;\ \mathsf{SPLIT}\ 0;\ \mathsf{PUSH}\ 2;\ \mathsf{MOVE}\ 1; \mathsf{DISCARD}\ 1; \mathsf{JUMP}\ l \\
\qquad\qquad \mathsf{LABEL}\ l_\mathsf{s};\ \mathsf{SPLIT}\ 1;\ \mathsf{PUSH}\ 2;\ \mathsf{PUSH}\ 1;\ \mathsf{MKAP}; \\
\qquad\qquad\qquad\qquad \mathsf{PUSHFUN}\ \mathbb{N}\textbf{-Elim};\ \mathsf{PUSH}\ 2;\ \mathsf{MKAP}; \\
\qquad\qquad\qquad\qquad \mathsf{PUSH}\ 6; \mathsf{MKAP};\ \mathsf{PUSH}\ 5;\ \mathsf{MKAP}; \\
\qquad\qquad\qquad\qquad \mathsf{PUSH}\ 4;\ \mathsf{MKAP};\ \mathsf{MKAP}; \\
\qquad\qquad\qquad\qquad \mathsf{MOVE}\ 2; \mathsf{DISCARD}\ 2; \mathsf{JUMP}\ l \\
\qquad\quad \mathsf{LABEL}\ l;\ \mathsf{UPDATE}\ 5;\ \mathsf{RET}\ 4
\end{array}
$$

Figure 3.8: Compilation of $\mathbb{N}$-**Elim** to G-machine code

implement a datatype representing G-code sequences indexed over the canonicity of contents of the stack. A value can either be in canonical form or a redex, and we represent the stack contents as a vector which explains the canonicity of each item in the stack.

$$
\begin{array}{l}
\underline{\mathsf{data}}\ \ \overline{\mathsf{Canonicity} : \star}\ \ \underline{\mathsf{where}}\ \ \overline{\mathsf{Canonical} : \mathsf{Canonicity}}\ \ \overline{\mathsf{Redex} : \mathsf{Canonicity}} \\
\textbf{Stack} = \lambda n{:}\mathbb{N}.\ \mathsf{Vect}\ \mathsf{Canonicity}\ n
\end{array}
$$

Now we define a datatype Gcode which represents G-code sequences and is indexed over the stack. As a result, the index on each instruction describes how that function affects the stack.

$$\underline{\text{data}} \quad \frac{s \ : \ \mathbf{Stack}\ n}{\mathsf{Gcode}\ s \ : \ \star} \quad \underline{\text{where}} \quad \frac{i \ : \ \mathsf{Fin}\ n \qquad g \ : \ \mathsf{Gcode}\ s}{g; \mathsf{PUSH}\ i \ : \ \mathsf{Gcode}\ (\mathbf{lookup}\ i\ s)::s}$$

$$\frac{g \ : \ \mathsf{Gcode}\ (a::f::s)}{g; \mathsf{MKAP} \ : \ \mathsf{Gcode}\ \mathsf{Redex}::s}$$

$$\frac{g \ : \ \mathsf{Gcode}\ (x::s)}{g; \mathsf{EVAL} \ : \ \mathsf{Gcode}\ \mathsf{Canonical}::s}$$

...

Note how, of the instructions given here, the indices describe some detail of the operational semantics of each instruction, with respect to the stack. These indices ensure the following properties:

- With PUSH, the index must be within the bounds of the stack, since the **lookup** operation requires its argument to be a Fin bounded by the vector size.
- With MKAP, there must be two arguments on the stack so there can be no stack overflow.
- With EVAL, we are guaranteed to end up with a canonical value on the stack. There is also a potential optimisation here, of removing unnecessary EVALs when we know a value is already in canonical form due to the stack contents.

For the moment, however, we have implemented the compilation schemes in Haskell, using a list to represent the byte-code. Further work which will be possible when the EPIGRAM front end is stable will be to implement this translation scheme using dependent types and therefore showing several correctness properties in a straightforward way,

3.6 Proper Tail Recursion

A problem with the G-machine in its current presentation is that many functions build closures which are immediately evaluated when the function returns. This has two principal disadvantages — it creates garbage unnecessarily, and it creates an extra stack frame. A **tail call** helps to avoid this problem. If the last thing a function does is return a *fully applied* function, there is no need to build the closure; the code for that function can be executed immediately with a tail call. Assuming $\text{ARITY}(\mathbf{g}) = 2$, this definition of $\mathbf{f}$ makes a tail call:

$$\mathbf{f} \mapsto \lambda x.\, \mathbf{g}\, 0\, x$$

The $\mathcal{S}[\![\cdot]\!]$ compilation scheme builds the following G-code for $\mathbf{f}$:

$$\mathcal{S}[\![\lambda x.\, \mathbf{g}\, 0\, x]\!] \Longrightarrow \mathsf{PUSHNAME}\,\mathbf{g};\ \mathsf{MKCON}\,0\,0;\ \mathsf{MKAP};\ \mathsf{PUSH}\,1;\ \mathsf{MKAP};\ \mathsf{UPDATE}\,3;\ \mathsf{RET}\,2$$

On reaching the RET instruction, the closure built by $\mathbf{f}$ is entered. If $\mathbf{g}$ is fully applied, however, it would clearly make more sense to jump to $\mathbf{g}$ directly and avoid building the intermediate closure.

Where a function is fully applied, we can simply squeeze out the i stack elements which refer to the current function's local variables, keeping the m elements which are passed the tail call. This introduces a new G-code instruction, SQUEEZE m i, also introduced by Johnsson [Joh84]. Tail calls are made by the JFUN f instruction, which jumps directly to the code for the function name f. The operational semantics of these instructions are shown in figure 3.9.

$$\begin{array}{lcl} \langle \mathsf{SQUEEZE}\ m\ i;\ c, n_0 \ldots n_{m-1} \ldots n_{m+i-1}.S, G, E, D\rangle & \Longrightarrow & \langle c, n_0 \ldots n_{m-1}.S, G, E, D\rangle \\ \langle \mathsf{JFUN}\ f;\ c, n_0 \ldots n_{i-1}.S, G, E[f = (i, c')], D\rangle & \Longrightarrow & \langle c', n_0 \ldots n_{i-1}.S, G, E, D\rangle \end{array}$$

Figure 3.9: State transitions for SQUEEZE and JFUN

f can now be compiled more efficiently to the following G-code:

$$\mathcal{S}[\![\lambda x.\, \mathbf{g}\ 0\ x]\!] \Longrightarrow \mathsf{CON}\ 0\ 0;\ \mathsf{PUSH}\ 1;\ \mathsf{SQUEEZE}\ 2\ 1;\ \mathsf{JFUN}\ \mathbf{g}$$

Dealing with tail calls efficiently requires some modifications to the $\mathcal{E}[\![\cdot]\!]$ compilation scheme. I introduce a separate compilation scheme, $\mathcal{R}[\![\cdot]\!]$ which returns a value and is presented in figure 3.10. If the value returned is a fully applied function it can be made into a tail call, otherwise the $\mathcal{E}[\![\cdot]\!]$ scheme is used.

$$\begin{array}{l} \mathcal{R}[\![\mathbf{f}\ a_1\ a_2\ \ldots\ a_m]\!]\ r\ n \Longrightarrow \mathcal{C}[\![a_1]\!]\ r\ n;\ \mathcal{C}[\![a_2]\!]\ r\ (n+1); \ldots;\ \mathcal{C}[\![a_m]\!]\ r\ (n+m-1); \\ \qquad\qquad\qquad\qquad \mathsf{SQUEEZE}\ m\ (n-1);\ \mathsf{JFUN}\ f \\ \qquad\qquad\qquad\qquad\quad \underline{\text{if}}\ \text{ARITY}(\mathbf{f}) = m \\ \mathcal{R}[\![e]\!]\ r\ n \Longrightarrow \mathcal{E}[\![e]\!]\ r\ n;\ \mathsf{UPDATE}\ n;\ \mathsf{RET}\ (n-1) \end{array}$$

Figure 3.10: The $\mathcal{R}[\![\cdot]\!]$ compilation scheme

The top level $\mathcal{S}[\![\cdot]\!]$ compilation scheme (figure 3.11) now returns a value, rather than evaluating its body.

$$\begin{array}{l} \mathcal{S}[\![\lambda \vec{a} : \vec{E}.\, e]\!] \Longrightarrow \mathcal{R}[\![e]\!]\ r\ (m+1) \\ \text{where } m \Longrightarrow \text{LENGTH}(\vec{a}) \\ \qquad\quad r(a_i) \Longrightarrow (m+2) - i \end{array}$$

Figure 3.11: The $\mathcal{S}[\![\cdot]\!]$ compilation scheme, with tail calls

3.7 Run-time Considerations

In this chapter we have seen a naïve method for compiling ExTT to an abstract machine code; ExTT arises from the identity transformation from TT terms in their raw form, without considering particular features of TT which make the resulting code potentially large and

inefficient. The type safety, totality and provability of terms in EPIGRAM relies on adding extra information to terms in the language which would not be present in a simply typed language; particularly worrying is the machinery required to eliminate impossible cases, as we saw in the **vTail** example in Chapter 2 (repeated here in figure 3.12).

$$
\begin{array}{l}
\underline{\mathsf{let}} \quad \dfrac{v : \mathsf{Vect}\ A\ (\mathsf{s}\ n)}{\mathbf{vTail}\ v\ :\ \mathsf{Vect}\ A\ n} \\
\quad \mathbf{vTail}\ (a{::}\ v)\ \mapsto\ v \\
\\
\quad \mathbf{dMotive}\ :\quad \forall n{:}\mathbb{N}.\ \star \\
\quad \mathbf{dMotive}\ \mapsto\quad \lambda n{:}\mathbb{N}.\ \mathbb{N}\text{-}\mathbf{Case}\ n\ (\forall n{:}\mathbb{N}.\ \star)\ \mathsf{False}\ (\lambda k{:}\mathbb{N}.\ \mathsf{True}) \\
\quad \mathbf{discriminate}\ :\quad \forall n{:}\mathbb{N}.\ \forall p{:}\mathsf{s}\ n\ =\ 0.\ \mathsf{False} \\
\quad \mathbf{discriminate}\ \mapsto\quad \lambda n{:}\mathbb{N}.\ \lambda p{:}\mathsf{s}\ n\ =\ 0. \\
\qquad\qquad\qquad\quad =\text{-}\mathbf{elim}\ \mathbb{N}\ (\mathsf{s}\ n)\ p\ \mathbf{dMotive}\ () \\
\quad \mathbf{emptyCase}\ :\quad \forall A{:}\star.\ \forall n{:}\mathbb{N}.\ (\mathsf{s}\ n\ =\ 0) \rightarrow \mathsf{Vect}\ A\ n \\
\quad \mathbf{emptyCase}\ \mapsto\quad \lambda A{:}\star.\ \lambda n{:}\mathbb{N}.\ \lambda p{:}\mathsf{s}\ n\ =\ 0. \\
\qquad\qquad\qquad\quad \mathsf{False}\text{-}\mathbf{Elim}\ (\mathbf{discriminate}\ n\ p)\ (\mathsf{Vect}\ A\ n) \\
\quad \mathbf{consCase}\ :\quad \forall A{:}\star.\ \forall n{:}\mathbb{N}.\ \forall k{:}\mathbb{N}.\ \mathsf{Vect}\ A\ k \rightarrow (\mathsf{s}\ n\ =\ \mathsf{s}\ k) \rightarrow \mathsf{Vect}\ A\ n \\
\quad \mathbf{consCase}\ \mapsto\quad \lambda A{:}\star.\ \lambda n{:}\mathbb{N}.\ \lambda k{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ k.\ \lambda p{:}k\ =\ n. \\
\qquad\qquad\qquad\quad =\text{-}\mathbf{elim}\ \mathbb{N}\ k\ n\ (\mathbf{S\_inj}\ k\ n\ (\mathbf{eq\_sym}\ \mathbb{N}\ n\ k\ p))\ (\lambda n{:}\mathbb{N}.\ \mathsf{Vect}\ A\ n)\ v \\
\quad \mathbf{vTailAux}\ :\quad \forall n{:}\mathbb{N}.\ \forall A{:}\star.\ \forall k{:}\mathbb{N}.\ \forall v{:}\mathsf{Vect}\ A\ k.\ (\mathsf{s}\ n\ =\ k) \rightarrow \mathsf{Vect}\ A\ n \\
\quad \mathbf{vTailAux}\ \mapsto\quad \lambda n{:}\mathbb{N}.\ \lambda A{:}\star.\ \lambda k{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ k. \\
\qquad\qquad\qquad\quad \mathsf{Vect}\text{-}\mathbf{Case}\ A\ k\ v \\
\qquad\qquad\qquad\quad (\lambda k{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ k.\ (\mathsf{s}\ n\ =k) \rightarrow \mathsf{Vect}\ A\ n) \\
\qquad\qquad\qquad\quad (\mathbf{emptyCase}\ A\ n) \\
\qquad\qquad\qquad\quad (\lambda k{:}\mathbb{N}.\ \lambda a{:}A.\ \lambda v{:}\mathsf{Vect}\ A\ k.\ \mathbf{consCase}\ A\ n\ k\ v) \\
\quad \mathbf{vTail}\ \mapsto\quad \lambda A{:}\star.\ \lambda n{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ (\mathsf{s}\ n). \\
\qquad\qquad\qquad\quad (\lambda k{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ k. \\
\qquad\qquad\qquad\quad \lambda P{:}\forall k{:}\mathbb{N}.\ \forall v{:}\mathsf{Vect}\ A\ k.\ (\mathsf{s}\ n\ =\ k) \rightarrow \mathsf{Vect}\ A\ n. \\
\qquad\qquad\qquad\quad P\ (\mathsf{s}\ n)\ v\ (\mathsf{refl}\ (\mathsf{s}\ n))) \\
\qquad\qquad\qquad\quad n\ v\ (\mathbf{vTailAux}\ n\ A)
\end{array}
$$

Figure 3.12: **vTail** and its elaboration

There are several efficiency problems which we might note in developing the run-time system for the language. Consider the version of **vTail** as written by the programmer (at the top of figure 3.12), and the fully elaborated term. The programmer's version suggests that a target machine might proceed along these lines:

- Get a pointer to v, the argument. v consists of a pointer to the head of the vector h and a pointer to the tail of the vector t.
- Return the pointer to t.

There is only one possible case here; we know from type checking that the vector must be non-empty so there should be no need to examine v to check whether it even has a head

or tail. However, the fully elaborated **vTail** tells a rather different story. There is a proof of equality constructed, an appeal to the elimination operator of vectors and the element type and length of the vector are passed implicitly although never used. How can we get the target machine to compile to the simple two step procedure above from this code? Problems such as this which arise in the execution of dependently typed terms will be addressed in the rest of this thesis.

There are several overheads which we can immediately identify which we ought to pay close attention to in the design of an optimised run-time system for TT.

3.7.1 Invariants of Inductive Families

The indices of inductive families express the properties of elements of that family. The indices of Vect, for example, express the fact that all elements must be the same type and that the length increases by one every time we add an item. These indices are left implicit in the declaration of Vect since they can be inferred by the type checker, but what happens to them at run-time? Figure 3.13 shows the optimal (that is, with perfect sharing) storage of a vector $a::b::c::\epsilon$, on the heap (the instances of A are also shared, but I have omitted the pointers for clarity).

This is with perfect sharing; it is possible (and indeed likely) for the numbers representing vector length to be separate instances. Although the programmer writes down only two arguments to the :: constructor, the typechecker has inferred that it is a vector of element type A and that the inner vector has length s (s 0). These values are stored on the heap along with the vector. To make matters worse, when the vector grows a length argument is stored with each :: node, even though the type tells us that the length must be one more than the length of the inner vector. There are 25 cells here used to represent the vector, its length and its element type. Just removing the extra pointers to the length and element type reduces this to 18 cells.

A naïve representation of inductive families would store all of the values in the structure, simply because they are part of the structure whether implicit or not. A practical implementation must consider methods for removing implicit information, whether it be inferable at run-time (like the length of the Vect) or simply not used (like the element type). Since implicit information is implicit exactly because it is *duplicated* in some other part of the term this amounts to removing subterms whose values are already known. We would like to be able to remove these duplicated terms at run-time, but we must take some care, for the following reasons:

- If duplicated values are removed from the representation of families, the compilation of elimination rules to RunTT as in section 3.4 is not so straightforward. We will no longer find all variables used on the right hand side of the reduction simply by examining the target — we might also need to look at the indices.

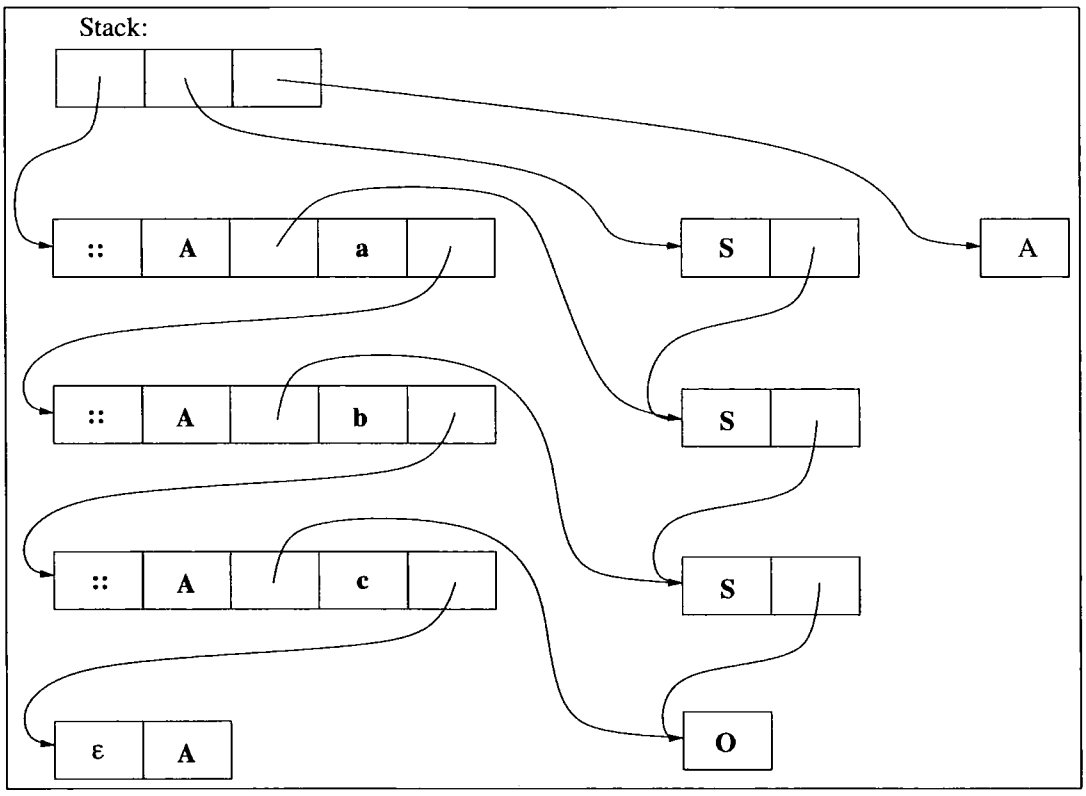


Figure 3.13: Storage of $a::b::c::\epsilon$

- We need to bear in mind the difference between compile-time evaluation for type checking, and run-time evaluation. Are there any terms which can be removed in one setting but not the other?
- We need to be sure that the transformed program has the same operational behaviour as the original program. A transformation which is not guaranteed to preserve the behaviour of a program is of no practical use.

3.7.2 Proofs

Dependently typed functional programs can include proofs of equations both as additional checks on invariants and in order to assist the type checker. In fact, elimination with a motive [McB00b], which is used in the definition of **vTail** to help remove impossible cases, relies on inserting equality proofs into the motive of an elimination rule.

These proofs assist the type checker and help assert properties of a program. At run-time, however, they have served their purpose and have no computational meaning so can safely be removed. This does not just apply to equality proofs but to *any* inductive relation which shows some computationally irrelevant property. The difficulty here is in identifying

which inductive families are computationally irrelevant and which may serve a purpose at run-time.

The Coq system approaches this problem by making a distinction between computational families and logical families. Set is an element of Type and is a universe of computational structures, and Prop, also an element of Type, is a universe of logical structures. It is not possible within Coq to move from the Prop universe to the Set universe by induction over a type in Prop, but we are allowed to generate a Prop by induction over a type in Set. The practical result of this is that no Prop (with the exception of singleton types, such as equality, since they have informative content) can produce a computational structure and so it is guaranteed that a Prop will not be used at run-time. The extraction mechanism [PM89, Let02], which creates ML or Haskell programs from Coq terms, exploits this by removing all (non-singleton) instances of Prop from the extracted code.

$$\underline{\text{let}} \quad \frac{n, m : \mathbb{N} \quad p : m \leq n}{\textbf{minus}\ n\ m\ p : \mathbb{N}}$$

$$\begin{array}{llll} \textbf{minus} & n & m & p & \Leftarrow \underline{\text{elim}}\ p \\ \textbf{minus} & n & 0 & (\mathsf{leO}\ n) & \mapsto n \\ \textbf{minus} & (\mathsf{s}\ n) & (\mathsf{s}\ m) & (\mathsf{leS}\ m\ n\ p) & \mapsto \textbf{minus}\ n\ m\ p \end{array}$$

Figure 3.14: Programming by induction over a proof

In section 2.3.2 I gave an example of programming by induction over a proof (See figure 3.14). In Coq, it would not be possible to write such a program using the default $\leq$ relation since it inhabits the Prop universe. To write this program would require adding a separate $\leq$ relation as a member of Set, which would result in the relation being present in the extracted code. Since we get patterns for the arguments n and m from the induction rule, however, it would seem intuitively obvious that the proof is not needed at run-time. We would like to find a way to be able to program by induction over a proof, but still remove that proof at run-time if the resulting patterns allow us to do so. The domain predicates used to show termination (see section 2.3.4) are an important example of a situation where we would like to write programs by induction over a proof, but we would still like to be able to remove such termination proofs at run-time.

3.7.3 Number Representation

So far, we have been using a unary representation of natural numbers:

$$\underline{\text{data}} \quad \frac{}{\mathbb{N} : \star} \quad \underline{\text{where}} \quad \frac{}{0 : \mathbb{N}} \quad \frac{n : \mathbb{N}}{\mathsf{s}\ n : \mathbb{N}}$$

With this declaration, we write functions **plus** and **mult** and are able to prove characteristic properties of these functions in a straightforward manner.

An experienced programmer, or anyone thinking about the internal representation of programs, might wonder whether this does not cause significant overheads, and of course it does. After all, computers have arithmetic operations built in and we can, we would hope, be reasonably confident of their correctness! So why do we use this representation, and can we do better?

Although operationally disastrous, the unary representation of $\mathbb{N}$ is conceptually useful. The benefits of $\mathbb{N}$ are that:

- It is naturally structurally recursive, which machine integers are not. This allows us to relate other structures (such as Vect) to natural numbers. Also, it allows us to implement a kind of bounded representation corresponding to a `for` loop in an imperative language.
- It is, at least in theory, unbounded, unlike machine integers which have some upper and lower bound.
- As a result proving properties of $\mathbb{N}$, functions over it, and families indexed over it, is more straightforward.

Leaving primitive types such as integers, characters, strings and arrays out of the core language gives us a small, clean, theoretically sound core. While this facilitates checking program correctness, it fails to take advantage of the architecture of the underlying machine. What we would like is a compilation scheme which changes the theoretically sound implementation of $\mathbb{N}$ into an unbounded big number type based on machine integers along with a justification of the correctness of this compilation scheme. Then we keep the compile time advantages of the $\mathbb{N}$ structure (by continuing to program with $\mathbb{N}$ in the high level notation), while still taking advantage of the underlying machine (by translating to an appropriate low level representation).

3.7.4 Dead Code In Impossible Cases

In section 2.2.3 I showed the elaboration of the **vTail** function which takes the tail of a non-empty vector, where the empty vector case is impossible. The machinery required to prove this is quite complex and leaves a lot of computationally redundant information in the **vTail** term. Some of the problems here are due to equality proofs, as described in section 3.7.2, but even if we overcome this problem there is still some redundant information:

- The whole term is wrapped in an outer λ-abstraction in order to introduce an equality into the motive.
- One of the cases in the helper function **vTailAux**, performs elimination on the empty type. Since it is not possible to have an element of the empty type we can be sure this case will never be executed.

- Having one impossible case leaves only one case which can apply. This suggests that it might be nice to shortcut the application to **Vect-Case** somehow so that no check is made at run-time.

Finding a way to overcome these three problems would lead to a target machine version of **vTail** close to that suggested at the beginning of this section.

3.7.5 Intermediate Data Structures

We saw intermediate data structures used to assist computation by the use of views in section 2.3.3, where list reversal and $\mathbb{N}$ comparison were both implemented as data structures. Also, an intermediate data structure was used in section 2.3.4 to represent the computational behaviour of the **quicksort** function. While these structures give us the relevant patterns on the left hand side of a function definition, there is a small overhead in creating and matching on the intermediate structures.

A possible approach to removing these intermediate values is Wadler's deforestation technique [Wad84, Wad90]. In general such structures will not cause a large performance hit, particularly since using a lazy evaluation strategy means that the structures need not be computed in their entirety before pattern matching. It may even be preferable not to remove these structures; where the same structure is examined more than once, lazy evaluation caches the intermediate result.

3.8 Summary

In this chapter, we have seen how ExTT terms can be compiled to an abstract machine code (G-machine code), which gives code for run-time only evaluation of a term. We have looked at the compilation process via an intermediate language of run-time supercombinators, RunTT and shown how to translate these supercombinators into G-code. This is a standard technique which has been applied in lazy functional languages for many years, and adapts to dependently typed programming with only minor modifications. Other virtual machines, such as the ABC machine and the $\langle \nu, G \rangle$-machine, are built on similar concepts and so such machines should adapt easily to dependently typed programming languages.

The approach to evaluation we have taken in this chapter has largely been naïvely adapted from techniques for implementing simply typed lazy functional languages — but we have also briefly looked at some of the run-time considerations of *dependently* typed programming. The naïve approaches we have taken in this chapter, both to normalisation of terms for typechecking and to compilation, clearly have several overheads which are not a problem in simply typed functional languages. In the following chapters, we shall look at ways of optimising the naïve compilation scheme to take account of these considerations.

Chapter 4

Optimising Inductive Families

(Much of the material in this chapter, except sections 4.4 and 4.6, has previously appeared as [BMM04]).

Machine generated elimination rules are the basic method by which EPIGRAM programs make decisions, perform recursion and compute results and therefore their efficient implementation, and the efficient storage of the data they examine, is very important to the efficiency of EPIGRAM programs. The building of elimination rules from inductive definitions is well understood and described in [Dyb94, Luo94, McB00a] among others. The computation behaviour of the rules is often presented directly as pattern matching ι-schemes similar to those we might find in Haskell, but with the possibility of repeated arguments and arbitrary terms on the left hand side where type dependency dictates the form of these terms. We can think of these elimination rules as a particularly special kind of pattern matching function whose behaviour and definition we know more about than we might reasonably know about pattern matching functions in general. For example, we know that functions are total, so we need not perform any run-time checks for incomplete function definitions — if some patterns are not covered, it is because the type dictates that those patterns are impossible.

In this chapter I will talk about how to take advantage of these special features of elimination rules to optimise their implementation. First, we will look again at the general form of elimination rules and examine an important property — namely that in a well-typed application, repeated arguments must be convertible. Given this, we go on to look at methods for implementing elimination rules, taking advantage of their properties in order to streamline their definition and hence programs which elaborate in terms of them. In particular, we observe that since an elimination rule for a family D is the only function allowed to examine the internal structure of D, we are free to choose any internal representation for D provided that it gives enough information to implement the elimination rule. We will use this observation to remove redundant data from the representation of families in several ways, and show several examples of data structures which can be optimised by these techniques.

In the naïve compilation path presented in the previous chapter, we used the identity transformation to translate from TT to ExTT. In this chapter, however, we will add annotations to ExTT which mark terms for optimisation, and specify optimisations by translation rules from TT to ExTT. The marking up of terms in this way leads to the need for a more sophisticated translation from ExTT to RunTT, especially regarding the compilation of elimination rules to simple case expressions. A compilation scheme for this is presented, along with associated modifications to the G-machine. Finally, we will see a larger example of the use of dependent types — a well-typed interpreter in the style of [AC99] — and how the optimisations presented in this chapter apply to this example.

4.1 Elimination Rules and Their Implementation

4.1.1 Form of Elimination Rules

Recall that an inductive family D, with constructors c_i is declared as below:

$$\underline{\text{data}} \quad \frac{\vec{i} : \vec{I}}{\mathsf{D}\,\vec{i} : \star}$$

$$\underline{\text{where}} \quad \frac{\vec{a}_1 : \vec{A}_1 \quad \vec{y}_1 : \mathsf{D}\,\vec{r}_1}{\mathsf{c}_1\,\vec{a}_1\,\vec{y}_1 : \mathsf{D}\,\vec{s}_1} \quad \cdots \quad \frac{\vec{a}_n : \vec{A}_n \quad \vec{y}_n : \mathsf{D}\,\vec{r}_n}{\mathsf{c}_n\,\vec{a}_n\,\vec{y}_n : \mathsf{D}\,\vec{s}_n}$$

When a family D is declared, EPIGRAM generates a basic elimination rule D-**Elim** and three other rules derived from it, D-**Case**, D-**View** and D-**Rec**, which together are used to implement functions defined with the high level pattern matching notation. The elimination operators (i.e., the implementations of these rules) are the only functions which are allowed to examine an instance of D directly.

We have already seen elimination operators used for programming in Chapter 2 and built a compilation scheme for programs written in this way in Chapter 3. However, such a naïve compilation scheme has its disadvantages, as noted at the end of Chapter 3. How can we take advantage of the properties of elimination operators so that the compiler produces a more efficient implementation?

I will take Vect as a running example. Recall that elaborating the declaration of Vect results in a type declaration $\mathsf{Vect} : \forall A{:}\star.\, \forall n{:}\mathbb{N}.\, \star$, and constructors:

$$\epsilon \;:\; \forall A{:}\star.\, \mathsf{Vect}\; A\; 0$$
$$:: \;:\; \forall A{:}\star.\, \forall k{:}\mathbb{N}.\, \forall a{:}A.\, \forall v{:}\mathsf{Vect}\; A\; k.\, \mathsf{Vect}\; A\; (\mathsf{s}\; k)$$

The variables left implicit in the data declaration have become explicitly quantified arguments. In naïve implementations these take up space, as shown at the end of the previous chapter — every $\mathsf{Vect}\; A\; n$ stores the sequence $0, \ldots, n-1$, and n references to A. The space implications for families with more complex invariants are quite drastic if this problem is left unchecked.

The ι-schemes generated for Vect are as follows:

$$\textsf{Vect-Elim}\ A\ \ 0\ \ \ \ (\epsilon\ A)\ \ \ P\ m_\epsilon\ m_{::} \rightsquigarrow m_\epsilon$$
$$\textsf{Vect-Elim}\ A\ (\mathsf{s}\ k)\ (::\ A\ k\ a\ v)\ P\ m_\epsilon\ m_{::} \rightsquigarrow m_{::}\ k\ a\ v\ (\textsf{Vect-Elim}\ A\ k\ v\ P\ m_\epsilon\ m_{::})$$

The most important thing to observe here about this pattern matching definition is that there are repeated arguments on the left hand side. That is, A appears twice in the first ι-scheme, and A and k appear twice in the second scheme. What are the semantics of such definitions? This appears to require non-linear pattern matching — in Haskell this would be illegal; here we might expect to have to do a run-time conversion check to make sure that arguments with the same name really are convertible. Even then, what should happen if the conversion check fails, since there is no possibility of failure (i.e., $\bot$ is not a value) in a language of total functions? The important property of elimination operators is that if the application is *well-typed*, such a conversion check *cannot* fail at run-time. This property is applied in the Plastic proof assistant to avoid checking of repeated arguments [CL99].

The type of an application of an elimination operator (eliding the method types for clarity) is:

$$\textsf{D-Elim}\ \vec{s}\ :\ \forall z{:}\mathsf{D}\ \vec{s}.\ \forall P{:}(\forall \vec{i}{:}\vec{I}.\ \mathsf{D}\ \vec{i} \rightarrow \star).\ \ldots\ \rightarrow P\ \vec{s}\ z$$

The type of a typical constructor, to which this operator will be applied, is:

$$\mathsf{c}\ \vec{a}\ \vec{y}\ :\ \mathsf{D}\ \vec{t}$$

If an application $\Gamma \vdash \textsf{D-Elim}\ \vec{s}\ (\mathsf{c}\ \vec{a}\ \vec{y})$ is well typed, $\Gamma \vdash \mathsf{D}\ \vec{s} \simeq \mathsf{D}\ \vec{t}$ must hold, since **D-Elim** $\vec{s}$ expects an argument whose type is convertible with $\mathsf{D}\ \vec{s}$. Hence $\vec{s} \simeq \vec{t}$ must also hold (by the Church Rosser property and the definition of the conversion relation) so there is no need to repeat the conversion check at run-time — duplicated pattern variables are guaranteed to be matched by convertible terms in a well-typed application. This property has important consequences; effectively it tells us that the naïve implementation is passed duplicate information — surely we can erase all but one instance of each repeated argument?

Another important observation, of which we can take advantage in the implementation of an elimination rule, is that the form of one argument can tell us something about other arguments. In the case of **Vect-Elim**, for example, if the target is headed by ϵ, we know that the length index must be 0 — no other value would be well-typed, so there is no need to deal with those cases.

The elimination rule **D-Elim** is the basic means TT provides for inspecting data in the inductive family D, and the other elimination rules can be implemented in terms of it. Therefore if we optimise **D-Elim**'s reduction behaviour, we optimise the programs which elaborate in terms of it. Moreover, if any data in the representation of D's elements is not needed by **D-Elim**, then it is *never* needed at run-time and can be erased from the representation — only the elimination rule has direct access to the arguments of D's constructors.

4.1.2 Pattern Syntax and its Run-Time Semantics

In Chapter 2 we saw that EPIGRAM generates an elimination rule for each inductive datatype which implements ι-reduction for that datatype in terms of ι-schemes. Let us now examine in more detail how ι-schemes are implemented in order to establish how to erase data from structures.

We write a set of ι-schemes in pattern matching style with a fixed arity,

$$\textsf{D-Elim}\ \vec{p}_i \rightsquigarrow e_i$$

where each $\vec{p}_i$ has the given arity, p_{ij} is a **pattern** and e_i is a term over $\vec{p}_i$'s **pattern variables**. The rule set is then compiled into an efficient case-expression. In the naïve implementation of Chapter 3, we compiled ι-schemes in a fixed manner by case analysis on the target. Now, however, we take a more general approach, annotating the patterns to direct the compilation, with parts of patterns which are *presupposed* to match marked by $[\cdot]$. This pattern syntax is presented in figure 4.1.

$p ::=$	x	(pattern variable)	$\mid$	$\mathsf{c}\ \vec{p}$	(constructor pattern)
$\mid$	$[t]$	(presupposed term)	$\mid$	$[\mathsf{c}]\ \vec{p}$	(presupposed constructor pattern)

Figure 4.1: Pattern syntax

The marking of a pattern $[x]$ indicates that in a *well typed* pattern, x may be *presupposed* to match, without checking. Such markings are made using the observations from section 4.1.1, that only one occurrence of a repeated argument need be matched, and that we can tell the form of some terms by matching on other arguments. We also mark terms which are not in constructor form, since it is not possible to determine x from $\mathbf{f}\,x$ for arbitrary $\mathbf{f}$. Such terms can also be presupposed to match by the fact that the application of the elimination rule must be well typed. We define an operation $|p|$ which strips these presupposition marks from a pattern, as in figure 4.2.

$$\begin{array}{ll} |x| & \Longrightarrow x \\ |\mathsf{c}\ \vec{p}| & \Longrightarrow \mathsf{c}\ |\vec{p}| \\ |[t]| & \Longrightarrow t \\ |[\mathsf{c}]|\ \vec{p} & \Longrightarrow \mathsf{c}\ |\vec{p}| \\ |p\ \vec{p}| & \Longrightarrow |p|\ |\vec{p}| \end{array}$$

Figure 4.2: $|p|$; removing presupposition marks from a pattern

The partial function MATCH (figure 4.3) specifies when a pattern and term yield a **matching substitution** (MATCHES lifts MATCH to argument sequences by composing the substitution built from the first argument with the substitutions built from the rest of the arguments). MATCH is a meta-operation, i.e. it is an operation on syntax.

$$
\begin{array}{l}
\text{MATCH}(\ x\ ,t) \implies t/x \\
\text{MATCH}(\ \mathsf{c}\ \vec{p}\ ,t) \implies \text{MATCHES}(\vec{p},\vec{t}) \ \ \underline{\text{if}}\ \text{WHNF}(t) \implies \mathsf{c}'\ \vec{t}\ \underline{\text{and}}\ \mathsf{c}=\mathsf{c}' \\
\text{MATCH}(\ [t']\ ,t) \implies \text{ID} \\
\text{MATCH}([\mathsf{c}]\ \vec{p},t) \implies \text{MATCHES}(\vec{p},\vec{t}) \ \ \underline{\text{if}}\ \text{WHNF}(t) \implies \mathsf{c}'\ \vec{t} \\
\text{MATCHES}(\ \mathsf{nil}\ ,\ \mathsf{nil}) \implies \text{ID} \\
\text{MATCHES}(p\ \vec{p},t\ \vec{t}) \implies \text{MATCH}(p,t) \circ \text{MATCHES}(\vec{p},\vec{t})
\end{array}
$$

Figure 4.3: Pattern matching semantics

The first two lines of MATCH test constructors and bind pattern variables as is usual in implementations of pattern matching. The remaining two lines, however, presuppose the successful outcome of testing. To justify these presuppositions, we shall require that each ι-scheme is **respectful** of well typed instances, as defined in figure 4.4. The respectfulness condition states that if a set of patterns with presupposition marks matches an argument sequence $\vec{t}$, yielding substitutions σ, then applying those substitutions to the unmarked patterns, $|\vec{p}_i|$, yields the original argument sequence $\vec{t}$.

$$
\begin{array}{l}
\underline{\text{if}}\ \Gamma \vdash \mathsf{D}\text{-}\mathbf{Elim}\ \vec{t}\ :\ T \\
\quad \underline{\text{and}}\ \text{MATCHES}(\vec{p}_i,\vec{t}) \implies \sigma \\
\quad \underline{\text{then}}\ \Gamma \vdash \mathsf{D}\text{-}\mathbf{Elim}\ \sigma|\vec{p}_i| \equiv \mathsf{D}\text{-}\mathbf{Elim}\ \vec{t}\ :\ T
\end{array}
$$

Figure 4.4: The respectfulness condition for ι-schemes

Remark: Respectfulness is not an issue in simply typed programming, because no analysis of arguments can be left out — i.e., it is not possible to learn the form of any argument by looking at other arguments. With dependent types, on the other hand, examining one argument can restrict the possible forms of other arguments.

A set of ι-schemes, $\mathsf{D}\text{-}\mathbf{Elim}\ \vec{p}_i \rightsquigarrow e_i$ is **well-defined** according to the criterion in figure 4.5. An application of a well-defined set of schemes yields ι-reduction $\mathsf{D}\text{-}\mathbf{Elim}\ \vec{t} \rightsquigarrow \sigma e_i$, which is the matching substitution σ applied to the right hand side of the appropriate ι-scheme, e_i. In a well-defined set of ι-schemes, there is *exactly one* scheme which matches when the rule is fully applied and the target is constructor headed; the elimination rule is implemented by a total function with non-overlapping patterns.

$$
\begin{array}{l}
\underline{\text{if}}\ \Gamma \vdash \mathsf{D}\text{-}\mathbf{Elim}\ \vec{t}\ :\ T\text{, where D-Elim is fully applied with a constructor headed target} \\
\quad \underline{\text{then}}\ \text{MATCHES}(\vec{p}_i,\vec{t}) \implies \sigma\ \text{ for exactly one } i
\end{array}
$$

Figure 4.5: The well-definedness condition for ι-schemes

The implementation of an elimination rule must be both respectful and well-defined to be acceptable, since these conditions ensure that the implementation has the desired behaviour;

well-definedness preserves totality, and respectfulness ensures that reduction correctly implements the ι-schemes. Respectfulness also preserves subject reduction.

4.1.3 Standard Implementation

The **standard implementation** of a rule D-**Elim** corresponds to the scheme given in section 3.4; that is, evaluation proceeds by inspecting the target of the elimination and ignoring the indices — the indices are presupposed given the target, since the indices are computed by the arguments of the constructor. For $\mathsf{D} : \forall \vec{i} : \vec{I}.\ \star$, with typical constructor

$\mathsf{c} \ : \ \forall \vec{a} : \vec{A}.\ \mathsf{D}\ \vec{r}_1 \to \ldots \to \mathsf{D}\ \vec{r}_j \to \mathsf{D}\ \vec{s}$,

our typical ι-scheme has a standard implementation as shown in figure 4.6.

For typical $\mathsf{c} \ : \ \forall \vec{a} : \vec{A}.\ \mathsf{D}\ \vec{r}_1 \to \ldots \to \mathsf{D}\ \vec{r}_j \to \mathsf{D}\ \vec{s}$

$$\textbf{D-Elim}\ [\vec{s}]\ (\mathsf{c}\ \vec{a}\ \vec{y})\ P\ \vec{m} \ \rightsquigarrow \ m_{\mathsf{c}}\ \vec{a}\ \vec{y}\ (\textbf{D-Elim}\ \vec{r}_1\ y_1\ P\ \vec{m})\ \ldots\ (\textbf{D-Elim}\ \vec{r}_j\ y_j\ P\ \vec{m})$$

Figure 4.6: Standard implementation of D-**Elim**

Theorem 4.1. *The standard implementation of* D-***Elim*** *is well-defined and respectful.*

Proof. For any Γ, if $\Gamma \vdash \textbf{D-Elim}\ \vec{s}'\ (\mathsf{c}\ \vec{a}'\ \vec{y}')\ P'\ \vec{m}' \ : \ T$ then

$\textsc{matches}([\vec{s}]\ (\mathsf{c}\ \vec{a}\ \vec{y})\ P\ \vec{m}, \vec{s}'\ (\mathsf{c}\ \vec{a}'\ \vec{y}')\ P'\ \vec{m}') \Longrightarrow \sigma$,
where σ is $\vec{a}'/\vec{a} \circ \vec{y}'/\vec{y} \circ P'/P \circ \vec{m}'/\vec{m}$

but matching the other ι-schemes fails, so these schemes are well-defined. Typechecking, we get $\mathsf{c}\ \vec{a}'\ \vec{y}' \ : \ \mathsf{D}\ (\vec{a}'/\vec{a} \circ \vec{y}'/\vec{y})\vec{s} = \mathsf{D}\ \sigma\vec{s}$. Hence $\sigma\vec{s} = \vec{s}'$ as $\textbf{D-Elim}\ \vec{s}'\ (\mathsf{c}\ \vec{a}'\ \vec{y}')$ is well-typed. Hence our typical scheme is respectful. □

The standard implementation is well-defined — we have exactly one scheme explicitly matching each of D's constructors — and respectful, by inversion of the typing rules. This is just as well, because there is no guarantee that the indices $\vec{s}$ will take the constructor form which explicit matching requires.

For example, the standard implementation of **Vect-Elim** is given in figure 4.7.

$$\begin{array}{l}\textbf{Vect-Elim}\ [A]\ [\mathsf{0}]\quad\ (\epsilon\ A)\quad\ P\ m_\epsilon\ m_{::} \ \rightsquigarrow \ m_\epsilon \\ \textbf{Vect-Elim}\ [A]\ [\mathsf{s}\ k]\ (::\ A\ k\ a\ v)\ P\ m_\epsilon\ m_{::} \ \rightsquigarrow \ m_{::}\ k\ a\ v\ (\textbf{Vect-Elim}\ A\ k\ v\ P\ m_\epsilon\ m_{::})\end{array}$$

Figure 4.7: Standard implementation of **Vect-Elim**

4.1.4 Alternative Implementations

In the standard implementation, we do not examine the indices at all; they are presupposed as they are computed from the arguments to the constructor. However, where these indices are themselves constructor or variable patterns we *can* examine them in the reduction rule and so we are free to consider alternative implementations of the corresponding ι-schemes. We may certainly presuppose a pattern variable in the target if we can recover it by matching an index. For example, this implementation of Vect-**Elim** is also respectful and well-defined:

$$\begin{array}{l}\textbf{Vect-Elim}\ A\ \ 0\ \ \ \ (\epsilon\ [A])\ \ \ \ P\ m_\epsilon\ m_{::}\ \rightsquigarrow\ m_\epsilon \\ \textbf{Vect-Elim}\ A\ (\mathsf{s}\ k)\ (::\ [A]\ [k]\ a\ v)\ P\ m_\epsilon\ m_{::}\ \rightsquigarrow\ m_{::}\ k\ a\ v\ (\textbf{Vect-Elim}\ A\ k\ v\ P\ m_\epsilon\ m_{::})\end{array}$$

But this implementation still does more work than is necessary; there is no need to check the constructor tags on *both* the length and the target — one check will do. If, for example, the target is ϵ, we already know that the second argument must be 0, since the declaration of the ϵ constructor tells us that this index can only take the value 0 for constructor ϵ. Likewise, if the target is ::, the second argument must be headed by s.

We may either examine the constructor of the vector, as in the first implementation in figure 4.8 or instead privilege index length over vector contents, as in the second implementation.

1. $\begin{array}{l}\textbf{Vect-Elim}\ A\ \ [0]\ \ \ \ (\epsilon\ [A])\ \ \ \ P\ m_\epsilon\ m_{::}\ \rightsquigarrow\ m_\epsilon \\ \textbf{Vect-Elim}\ A\ ([\mathsf{s}]\ k)\ (::\ [A]\ [k]\ a\ v)\ P\ m_\epsilon\ m_{::}\ \rightsquigarrow\ m_{::}\ k\ a\ v\ (\textbf{Vect-Elim}\ A\ k\ v\ P\ m_\epsilon\ m_{::})\end{array}$
2. $\begin{array}{l}\textbf{Vect-Elim}\ A\ \ 0\ \ \ \ ([\epsilon]\ [A])\ \ \ \ P\ m_\epsilon\ m_{::}\ \rightsquigarrow\ m_\epsilon \\ \textbf{Vect-Elim}\ A\ (\mathsf{s}\ k)\ ([::]\ [A]\ [k]\ a\ v)\ P\ m_\epsilon\ m_{::}\ \rightsquigarrow\ m_{::}\ k\ a\ v\ (\textbf{Vect-Elim}\ A\ k\ v\ P\ m_\epsilon\ m_{::})\end{array}$

Figure 4.8: Alternative implementations of Vect-**Elim**

In the following sections, we show how to choose alternative implementations for elimination operators by systematically exploiting the presence of constructor symbols in indices. The implementation of an elimination rule is chosen so that it examines as little of the *target* as possible. Since only the elimination rule has direct access to the target, this leads naturally to space optimisations, where we do not merely "comment out" unnecessary data from patterns — we delete them entirely from the representation of datatypes.

4.2 ExTT and Its Properties

Previously, in the naïve compilation path, the mapping from TT to ExTT was the identity transformation. For the optimising path, we augment ExTT's syntax with **deleted** terms $\{t\}$ and its operational semantics with corresponding deleted patterns (figure 4.9). The intention of deleted terms and patterns is to exploit the fact that, as shown in the previous section, we

do not need to examine all of the left hand side of an elimination rule in order to ι-reduce. Deleted patterns match only deleted arguments, and yield the identity substitution:

$\text{MATCH}(\{t\}, \{t'\}) \Longrightarrow \text{ID}$

$p ::=$	x	(pattern variable)	$\mid \mathsf{c}\ \vec{p}$	(constructor pattern)
$\mid$	$[t]$	(presupposed term)	$\mid [\mathsf{c}]\ \vec{p}$	(presupposed constructor pattern)
$\mid$	$\{t\}$	(deleted term)	$\mid \{\mathsf{c}\}\ \vec{p}$	(deleted constructor pattern)
$t ::=$	$\ldots$			
	$\{t\}$	(deleted term)	$\mid \forall\{x{:}t\}.\ t$	(deleted function)

$\text{MATCH}(\{p\}, \{t\}) \Longrightarrow \text{ID}$

Figure 4.9: Extensions in ExTT

ExTT terms arise only by mappings from TT, so we think of ExTT as a family of languages $\mathsf{ExTT}(S)$, parametrised over a set of mappings S from TT. In the naïve compilation path, therefore, we compiled $\mathsf{ExTT}(\emptyset)$.

We define a forgetful mapping operation $|\cdot|$ which removes the deletion marks from ExTT terms, giving a TT term. $|p|$ removes the deletion marks from patterns, as defined in figure 4.10. Correspondingly, we define an operation $|t|$ which removes deletion marks from terms, defined in figure 4.11.

$$
\begin{array}{ll}
|x| & \Longrightarrow x \\
|\mathsf{c}\ \vec{p}| & \Longrightarrow \mathsf{c}\ |\vec{p}| \\
|\{t\}| & \Longrightarrow t \\
|\{\mathsf{c}\}\ \vec{p}| & \Longrightarrow \mathsf{c}\ |\vec{p}| \\
|[t]| & \Longrightarrow t \\
|[\mathsf{c}]\ \vec{p}| & \Longrightarrow \mathsf{c}\ |\vec{p}|
\end{array}
$$

Figure 4.10: $|p|$; removing deletion marks from a pattern

$$
\begin{array}{ll}
|\star_n| & \Longrightarrow \star_n \\
|x| & \Longrightarrow x \\
|\forall x{:}S.\ T| & \Longrightarrow \forall x{:}|S|.\ |T| \\
|\lambda x{:}S.\ e| & \Longrightarrow \lambda x{:}|S|.\ |e| \\
|\underline{\text{let}}\ x \mapsto v\ \underline{\text{in}}\ e| & \Longrightarrow \underline{\text{let}}\ x \mapsto |v|\ \underline{\text{in}}\ |e| \\
|\{t\}| & \Longrightarrow |t| \\
|\forall\{x{:}S\}.\ T| & \Longrightarrow \forall x{:}|S|.\ |T| \\
|f\ a| & \Longrightarrow |f|\ |a|
\end{array}
$$

Figure 4.11: $|t|$; removing deletion marks from a term.

4.2.1 Properties of ExTT

With the additions to ExTT come additional equality, conversion and computation rules. To distinguish these from the rules for TT, we annotate the turnstile as follows:

- Syntactic equality for ExTT is denoted by $\Gamma \overset{\text{Ex}}{\vdash} x \overset{\text{Ex}}{\equiv} y$.
- Conversion for ExTT is denoted by $\Gamma \overset{\text{Ex}}{\vdash} x \overset{\text{Ex}}{\simeq} y$.
- Reduction for ExTT is denoted by $\Gamma \overset{\text{Ex}}{\vdash} x \overset{\text{Ex}}{\rhd^*} y$.

Likewise, we annotate the turnstile on TT judgments as $\Gamma \overset{\text{TT}}{\vdash} J$. Where there is no ambiguity, we will omit the annotation.

Contraction is as for TT, except that deleted terms $\{\cdot\}$ do not reduce (i.e., $\{\cdot\}$ is not a reducible expression, and so $\{t\}$ is a normal form for all t). We also say that $\Gamma \overset{\text{Ex}}{\vdash} \{x\} \overset{\text{Ex}}{\equiv} \{y\}$ for all x, y. Strong normalisation holds trivially for ExTT, since $\{t\}$ is a normal form for all t and all ExTT reductions have a corresponding TT reduction (see Lemma B.3 in Appendix B).

We extend the definition of contexts to annotate variables which are expected to be deleted. Contexts are defined as in figure 4.12.

$$\frac{}{\mathcal{E} \vdash \underline{\text{valid}}} \qquad \frac{\Gamma \vdash S \ : \ \star_i}{\Gamma; x \ : \ S \vdash \underline{\text{valid}}} \qquad \frac{\Gamma \vdash s \ : \ S}{\Gamma; x \mapsto s \ : \ S \vdash \underline{\text{valid}}}$$

$$\frac{\Gamma \vdash \mathsf{D}\ \vec{s} \ : \ \star_n}{\Gamma; \{x\} \ : \ \forall \vec{a}{:}\vec{A}.\ \mathsf{D}\ \vec{s} \vdash \underline{\text{valid}}} \text{ if } \Gamma; \{y\} \ : \ \forall \vec{b}{:}\vec{B}.\ \mathsf{D}\ \vec{t}; \Gamma' \vdash \underline{\text{valid}} \text{ then } \exists i.\ \textsc{disjoint}(s_i, t_i)$$

Figure 4.12: Contexts in ExTT

The side condition on the last rule ensures that a name can only be added with deletion marks if the indices of its type are disjoint with all other deleted names in the context. We will postpone discussion of the DISJOINT operation and the purpose of this rule until section 4.3.2. Again, we use $|\Gamma|$ to remove deletion marks from entries in Γ, as defined in figure 4.13.

$$\begin{array}{lll} |\mathcal{E}| & \Longrightarrow \mathcal{E} \\ |\Gamma; x \ : \ S| & \Longrightarrow |\Gamma|; x \ : \ |S| \\ |\Gamma; x \mapsto s \ : \ S| & \Longrightarrow |\Gamma|; x \mapsto |s| \ : \ |S| \\ |\Gamma; \{x\} \ : \ S| & \Longrightarrow |\Gamma|; x \ : \ |S| \end{array}$$

Figure 4.13: $|\Gamma|$; removing deletion marks from all entries in the context

We also have a conversion rule corresponding to that for TT (figure 4.14).

Definition: x is **convertible** to y relative to Γ ($\Gamma \overset{\mathsf{Ex}}{\vdash} x \overset{\mathsf{Ex}}{\simeq} y$)
if and only if there exist $x_1, \ldots, x_n$ ($n \geq 1$) such that $\Gamma \overset{\mathsf{Ex}}{\vdash} x \overset{\mathsf{Ex}}{\equiv} x_1, \Gamma \overset{\mathsf{Ex}}{\vdash} y \overset{\mathsf{Ex}}{\equiv} x_n$
and $\Gamma \overset{\mathsf{Ex}}{\vdash} x_i \overset{\mathsf{Ex}}{\triangleright_1} x_{i+1}$ or $\Gamma \overset{\mathsf{Ex}}{\vdash} x_{i+1} \overset{\mathsf{Ex}}{\triangleright_1} x_i$, for $i = 1, \ldots, n-1$

Figure 4.14: Conversion for ExTT

4.2.2 Defining Optimisations

We will consider a variety of optimisations of inductive families and their elimination rules. Optimisations are defined by mappings from TT to ExTT — an optimisation for a family D is given by:

- A substitution $[\![\cdot]\!]$ from each constructor of the family to an ExTT term.
- An optimised ι-scheme for each constructor of the family.
- An updated entry in Γ for each constructor of the family.

For original ι-scheme $\Gamma \vdash \mathsf{D}\text{-}\mathbf{Elim}\ \vec{t_i} \rightsquigarrow e_i$, the optimised ι-scheme has the form $\mathsf{D}\text{-}\mathbf{Elim}\ \vec{p_i} \rightsquigarrow d_i$, where $|\vec{p_i}| = \vec{t_i}$, $|d_i| = e_i$ and every undeleted free variable in d_i is a pattern variable in $\vec{p_i}$. That is, unmarking the optimised scheme yields the original scheme. The optimised schemes must be well-defined in that exactly one scheme must match when D-**Elim** is fully applied with a constructor headed target, and respectful in that

if $\Gamma \vdash \mathsf{D}\text{-}\mathbf{Elim}\ \vec{t} : T$ and $\textsc{matches}(\vec{p_i}, [\![\vec{t}]\!]) \Longrightarrow \sigma$
then there exists a substitution τ such that $\Gamma \vdash \tau\, |\sigma(\mathsf{D}\text{-}\mathbf{Elim}\ \vec{p_i})| = \mathsf{D}\text{-}\mathbf{Elim}\ \vec{t} : T$

That is, applying the optimised elimination rule and unmarking the result yields the same result as applying the original rule. The rôle of τ is to instantiate the variables free in e_i, but deleted in d_i — these are deleted since they are not needed when executing ExTT terms, hence they need not be matched.

4.2.3 Typechecking via ExTT

There are two settings in which evaluation takes place in EPIGRAM (namely, compile-time and run-time, as described in section 3.1.4), and hence there are two settings to consider for optimisations. While ExTT is primarily designed as a language for efficient run-time evaluation of TT programs, we can also get some benefit at compile-time, by using ExTT for typechecking.

Figure 4.15 gives a type synthesis algorithm for ExTT. The intention is to use this type synthesis algorithm to check TT judgements, bypassing the TT type synthesis algorithm entirely. Since typechecking relies to some extent on reduction (in the conversion check), we can optimise typechecking by avoiding the reduction of marked terms in ExTT.

$$\frac{\Gamma \vdash \underline{\text{valid}}}{\Gamma \vdash \star_n \Longrightarrow \star_{n+1}}$$

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad x \ : \ S \in \Gamma}{\Gamma \vdash x \Longrightarrow S}$$

(Similarly for c, D, D-**Elim**)

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad x \ : \ S \mapsto s \in \Gamma}{\Gamma \vdash x \Longrightarrow S}$$

$$\frac{\Gamma \vdash f \Longrightarrow X \twoheadrightarrow \forall x{:}S.\ T \quad \Gamma \vdash s \Longrightarrow S' \quad \Gamma \vdash S \simeq S'}{\Gamma \vdash f\ s \Longrightarrow \underline{\text{let}}\ x \ : \ S' \mapsto s\ \underline{\text{in}}\ T}$$

$$\frac{\Gamma \vdash f \Longrightarrow X \twoheadrightarrow \forall \{x{:}S\}.\ T \quad \Gamma \vdash s \Longrightarrow S' \quad \Gamma \vdash S \simeq S'}{\Gamma \vdash f\ \{s\} \Longrightarrow \underline{\text{let}}\ x \ : \ S' \mapsto s\ \underline{\text{in}}\ T}$$

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad \{f\} \ : \ \forall x{:}S.\ T \in \Gamma \quad \Gamma \vdash s \Longrightarrow S' \quad \Gamma \vdash S \simeq S'}{\Gamma \vdash \{f\}\ s \Longrightarrow \underline{\text{let}}\ x \ : \ S' \mapsto s\ \underline{\text{in}}\ T}$$

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad \{f\} \ : \ \forall \{x{:}S\}.\ T \in \Gamma \quad \Gamma \vdash s \Longrightarrow S' \quad \Gamma \vdash S \simeq S'}{\Gamma \vdash \{f\}\ \{s\} \Longrightarrow \underline{\text{let}}\ x \ : \ S' \mapsto s\ \underline{\text{in}}\ T}$$

$$\frac{\Gamma; x \ : \ S \vdash e \Longrightarrow T \quad \Gamma \vdash \forall x{:}S.\ T \Longrightarrow \star_n}{\Gamma \vdash \lambda x{:}S.\ e \Longrightarrow \forall x{:}S.\ T}$$

$$\frac{\Gamma; x \ : \ S \vdash T \Longrightarrow X \twoheadrightarrow \star_n \quad \Gamma \vdash S \Longrightarrow X' \twoheadrightarrow \star_n}{\Gamma \vdash \forall x{:}S.\ T \Longrightarrow \star_n}$$

$$\frac{\begin{array}{c}\Gamma \vdash S \Longrightarrow X \twoheadrightarrow \star_n \quad \Gamma \vdash e_1 \Longrightarrow S' \quad \Gamma \vdash S \simeq S' \\ \Gamma; x \ : \ S \mapsto e_1 \vdash e_2 \Longrightarrow T \quad \Gamma; x \ : \ S \mapsto e_1 \vdash T \Longrightarrow X' \twoheadrightarrow \star_n\end{array}}{\Gamma \vdash \underline{\text{let}}\ x \ : \ S \mapsto e_1\ \underline{\text{in}}\ e_2 \Longrightarrow \underline{\text{let}}\ x \ : \ S \mapsto e_1\ \underline{\text{in}}\ T}$$

Figure 4.15: Type synthesis for ExTT

If we want to check a judgement $\Gamma \overset{\mathsf{TT}}{\vdash} a \ : \ A$ using the ExTT type synthesis algorithm, we must ensure that the translation from TT to ExTT satisfies certain properties. In particular, for an optimisation to be valid at compile-time we require the following three properties to hold:

Property 1. *If* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$ *then* $\exists A.\ \Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ *and* $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$

Property 2. *If* $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ *then* $\exists B.$

$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$ *and*

$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![A]\!]$ *and*

$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$

Property 3. *If* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\simeq} B$ *then* $\Gamma \overset{\mathsf{TT}}{\vdash} A \simeq |B|$

These properties ensure that we can check a judgement $\Gamma \overset{\mathsf{TT}}{\vdash} a \ : \ A$ by checking the following:

- $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\Longrightarrow} X \overset{\mathsf{Ex}}{\twoheadrightarrow} \star_n$
- $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$
- $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\simeq} B$

Properties 1, 2 and 3 ensure the soundness and completeness of this algorithm. This is demonstrated by Theorems B.1 and B.2 in Appendix B which show that type synthesis in ExTT is equivalent to type synthesis in TT.

4.3 Building Efficient Implementations

The generation of alternative implementations of elimination rules relies on three optimising transformations, which are called **forcing**, **detagging** and **collapsing**, from [BMM04].

- **Forcing** implements the commenting out of constructor arguments which are also retrievable from the indices by pattern matching. This relies on the *injectivity* property of constructors, and the respectfulness and well-definedness of elimination rules.
- **Detagging** implements the commenting out of constructor tags of the target where the choice of ι-scheme can be determined by the indices alone. This relies on the *disjointness* property of constructors, and the respectfulness and well-definedness of elimination rules.
- **Collapsing** is a run-time only optimisation which implements the commenting out of entire data structures. This applies when forcing and detagging comment out all but the recursive arguments of a structure. As we will see, collapsing applies only when evaluation takes place in the empty context.

4.3.1 Eliding Redundant Constructor Arguments

The first alternative implementation of Vect-**Elim** in figure 4.8 above matches A and k in the indices rather than the target:

$$\begin{array}{l}\mathsf{Vect\text{-}Elim}\ A\ \ [0]\qquad(\epsilon\,[A])\qquad P\ m_\epsilon\ m_{::} \rightsquigarrow m_\epsilon \\ \mathsf{Vect\text{-}Elim}\ A\ ([\mathsf{s}]\ k)\ (::[A]\ [k]\ a\ v)\ P\ m_\epsilon\ m_{::} \rightsquigarrow m_{::}\ k\ a\ v\ (\mathsf{Vect\text{-}Elim}\ A\ k\ v\ P\ m_\epsilon\ m_{::})\end{array}$$

In general, when can we comment out an argument of a constructor?

If we have two constructor headed terms, $\mathsf{c}\ \vec{a}$, $\mathsf{c}\ \vec{b}$ in the same type $\mathsf{D}\ \vec{s}$ and the value of the ith argument of c is determined only by (or forced by) the indices $\vec{s}$, such that $a_i \simeq b_i$, we say that the ith argument of c is **forceable** (figure 4.16). For example, the A argument to ϵ is forceable since if $\epsilon\ a$, $\epsilon\ b$: $\mathsf{Vect}\ A\ 0$ then clearly $a \simeq b \simeq A$; no other value would be well typed. For ::, A and k are forceable in the same way.

The ith argument of a constructor c is **forceable**
if $\Gamma \vdash \mathsf{c}\ \vec{a},\ \mathsf{c}\ \vec{b}\ :\ \mathsf{D}\ \vec{s}$ implies $\Gamma \vdash a_i \simeq b_i$

Figure 4.16: Forceable arguments

To say whether an argument is forceable is, in general, difficult, and likely to be undecidable since it relies on the injectivity of a function, and knowing the inverse of that function. However, it is possible to identify *some* forceable arguments. In particular, constructor arguments which are repeated in an ι-scheme are forceable. This is to be expected; such repeated arguments arise from the patterns describing constructor indices. Constructors are injective, and since they cannot be reduced it is trivial to compute what the arguments must have been given a constructor application in normal form.

Consider a typical constructor, fully applied to variables, $\mathsf{c}\ \vec{a}\ \vec{y}\ :\ \mathsf{D}\ \vec{s}$. If we express $\vec{s}$ as $|\vec{p}|$, where $\vec{p}$ arises by marking the presupposed terms in patterns built from $\vec{s}$, then any a_i appearing as a pattern variable in $\vec{p}$ is forceable, by injectivity of constructors. We call these arguments **concretely forceable** (figure 4.17) since they can be retrieved in constant time by pattern matching on the indices.

For fully applied $\mathsf{c}\ \vec{a}\ \vec{y}\ :\ \mathsf{D}\ \vec{s}$, where $\vec{s} = |\vec{p}|$
if a_i appears in $\vec{p}$ as a pattern variable then a_i is **concretely forceable**

Figure 4.17: Concretely forceable arguments

Lemma 4.2. *If a_i is concretely forceable in $\mathsf{c}\ \vec{a}\ \vec{y}$, then the ith argument of* c *is forceable.*

Proof. We need to show that substitution instances of concretely forceable variables in patterns are convertible.

For $\mathsf{c}\ \vec{a}\ \vec{y}\ :\ \mathsf{D}\ \vec{s}$, a_i is concretely forceable if it appears as a pattern variable in $\vec{p}$ where $|\vec{p}| = \vec{s}$. a_i is determined by a pattern variable appearing in p_j. So if two terms matching p_j are convertible, then the two terms matching a_i must also be convertible, by respectfulness of elimination rules. Therefore the substitution instances (determined by MATCH) must also be convertible.

□

To express $\vec{s}$ as $|\vec{p}|$, we write a program PAT to extract from a term a linear pattern with its variable set and PATS, which lifts PAT across argument sequences, shown in figure 4.18. V is an accumulator containing the variable set built so far (which is initialised to the empty set $\emptyset$); the second argument is the index in $\vec{s}$.

The helper operation LAZY exploits the fact that we need not examine the constructors at the head of the indices to implement the reduction, given that it can be implemented by examining the constructors at the head of the target.

$$
\begin{array}{l}
\textsc{pat}\ (\ V,\ x\ ) \Longrightarrow (x \cup V, x)\ \underline{\text{if}}\ x \notin V \\
\textsc{pat}\ (\ V, \mathsf{c}\ \vec{t}\ ) \Longrightarrow (V', \textsc{lazy}(\mathsf{c}, \vec{p}))\ \underline{\text{if}}\ \textsc{pats}\ (V, \vec{t}) \Longrightarrow (V', \vec{p}) \\
\textsc{pat}\ (\ V,\ t\ ) \Longrightarrow (V, [t]) \\
\textsc{pats}(\ V, \mathsf{nil}) \Longrightarrow (V, \mathsf{nil}) \\
\textsc{pats}(\ V, t\ \vec{t}\ ) \Longrightarrow (V'', p\ \vec{p}) \\
\qquad\qquad \underline{\text{if}}\ \textsc{pat}\ (V, t) \Longrightarrow (V', p)\ \underline{\text{and}}\ \textsc{pats}\ (V', \vec{t}) \Longrightarrow (V'', \vec{p}) \\
\textsc{lazy}(\ \mathsf{c}, [\vec{p}]) \Longrightarrow [\mathsf{c}\ \vec{p}] \\
\textsc{lazy}(\ \mathsf{c},\ \vec{p}\ ) \Longrightarrow [\mathsf{c}]\ \vec{p}\ \ \underline{\text{otherwise}}
\end{array}
$$

Figure 4.18: Extracting patterns from a constructor's indices

For our typical constructor c, $\textsc{pats}\,(\emptyset, \vec{s})$ gives us $(V, \vec{p})$ where V is the set of arguments of c which are forced by $\vec{s}$, and $\vec{p}$ are the patterns which D-**Elim** will match. If an argument $x_i \in V$ then x_i is concretely forceable. Then we may create an alternative implementation for the ι-scheme which matches c as follows:

$$
\begin{array}{lll}
\textbf{D-Elim}\ \vec{p}\ (\mathsf{c}\ \vec{a}^{[V]}\ \vec{y})\ P\ \vec{m} \rightsquigarrow m_{\mathsf{c}}\ \cdots & \underline{\text{where}}\ a^{[V]} \Longrightarrow [a]\ \underline{\text{if}}\ a \in V \\
& \qquad\qquad a^{[V]} \Longrightarrow a\ \ \underline{\text{otherwise}}
\end{array}
$$

The helper operation $a^{[V]}$ comments out the variable a in the patterns if it appears in the set of concretely forceable arguments V.

Lemma 4.3. *If* $\mathsf{c}\ \vec{a}\ \vec{y}\ :\ \mathsf{D}\ \vec{s}$, *and* $\textsc{pats}\ (\emptyset, \vec{s}) = (V, \vec{p})$ *then* $\forall a \in V$, a *is a concretely forceable argument of* c.

Proof. PATS traverses patterns inserting pattern variables into V. By definition, these are concretely forceable arguments of c. □

The Forcing Optimisation

The forcing optimisation on a constructor c marks the concretely forceable arguments of c for deletion; it generates a substitution on the identifier c, which gives a term in ExTT. Also, we get an optimised ι-scheme in ExTT for c. To be meaningful, this optimisation is applied to all constructors of a family D. The general scheme is given in figure 4.19, and the instance for Vect in figure 4.20. Note that types are elided in the λ-bindings; this is to avoid distracting attention from the optimisation itself — $\lambda a; b; \ldots . e$ is used here as a shorthand for $\lambda a{:}A.\ \lambda b{:}B.\ \ldots . e$.

When forcing, we also update the context by $[\![\Gamma]\!]$, so that the type of the constructor c in the optimised context binds deleted arguments. Since we change all applications of c so that the appropriate arguments are deleted, the optimised code is well-typed in ExTT. The types of the constructors in the Vect example are as follows:

$$
\begin{array}{lll}
\epsilon & : & \forall\{A{:}\star\}.\ \mathsf{Vect}\ A\ \mathsf{0} \\
:: & : & \forall\{A{:}\star\}.\ \forall\{k{:}\mathbb{N}\}.\ \forall a{:}A.\ \forall v{:}\mathsf{Vect}\ A\ k.\ \mathsf{Vect}\ A\ (\mathsf{s}\ k)
\end{array}
$$

For each $\mathsf{c} : \forall \vec{a}{:}\vec{A}.\, \mathsf{D}\ \vec{r}_1 \to \ldots \to \mathsf{D}\ \vec{r}_j \to \mathsf{D}\ \vec{s}$ where PATS $(\emptyset, \vec{s}) \Longrightarrow (V, \vec{p})$
take $[\![\mathsf{c}]\!] \Longrightarrow \lambda \vec{a}; \vec{y}.\, \mathsf{c}\ \vec{a}^{\{V\}}\ \vec{y}$
$\quad \mathsf{D}\text{-}\mathbf{Elim}\ \vec{p}\ (\mathsf{c}\ \vec{a}^{\{V\}}\ \vec{y})\ P\ \vec{m} \mapsto m_{\mathsf{c}}\ \vec{a}\ \vec{y}\ (\mathsf{D}\text{-}\mathbf{Elim}\ \vec{r}_1\ y_1\ P\ \vec{m})\ \ldots\ (\mathsf{D}\text{-}\mathbf{Elim}\ \vec{r}_j\ y_j\ P\ \vec{m})$
$\quad$ where $a^{\{V\}} \Longrightarrow \{a\}$ if $a \in V$
$\qquad\quad a^{\{V\}} \Longrightarrow a$ otherwise
and $\mathsf{c} : \forall \vec{a}{:}\vec{A}^{\{V\}}.\, \forall \vec{y}{:}[\![\vec{Y}]\!].\, \mathsf{D}\ [\![\vec{s}]\!] \in [\![\Gamma]\!]$
$\quad$ where $\forall a{:}A^{\{V\}} \Longrightarrow \forall \{a{:}[\![A]\!]\}$ if $a \in V$
$\qquad\quad \forall a{:}A^{\{V\}} \Longrightarrow \forall a{:}[\![A]\!]$ otherwise

Figure 4.19: The Forcing Optimisation

$[\![\epsilon]\!] \Longrightarrow \lambda A.\, \epsilon\ \{A\}$
$[\![::]\!] \Longrightarrow \lambda A; k; a; v.\, ::\ \{A\}\ \{k\}\ a\ v$

$\mathsf{Vect}\text{-}\mathbf{Elim}\ A\ [0]\ (\epsilon\ \{A\})\ P\ m_\epsilon\ m_{::} \rightsquigarrow m_\epsilon$
$\mathsf{Vect}\text{-}\mathbf{Elim}\ A\ ([\mathsf{s}]\ k)\ (::\ \{A\}\ \{k\}\ a\ v)\ P\ m_\epsilon\ m_{::} \rightsquigarrow m_{::}\ k\ a\ v\ (\mathsf{Vect}\text{-}\mathbf{Elim}\ A\ k\ v\ P\ m_\epsilon\ m_{::})$

Figure 4.20: Forcing for Vect

So rather than merely commenting out concretely forceable arguments using $a^{[V]}$, the forcing optimisation marks such arguments for deletion with $a^{\{V\}}$. Note in the Vect-**Elim** rule that the constructor tags 0 and s are commented out (but not marked for deletion) to indicate that they are not inspected; these tags are commented out by the LAZY operation in figure 4.18.

In the transformation from ExTT to RunTT, the deleted arguments really are removed from the fully applied constructors. This is safe because these terms are only decomposed by Vect-**Elim**, the new implementation of which does not expect the deleted arguments.

Properties of Forcing

Forcing satisfies the required properties of a compile-time optimisation. The elimination rule is respectful and well-defined, and typechecking the resulting terms in ExTT is equivalent to typechecking in TT.

Theorem 4.4. *The forcing implementation of* D-***Elim*** *is respectful and well-defined.*

Proof. Clearly, $|\vec{p}| = \vec{s}$ and $\left|\mathsf{c}\ \vec{a}^{\{V\}}\ \vec{y}\right| = \mathsf{c}\ \vec{a}\ \vec{y}$, so if $\Gamma \vdash \mathsf{D}\text{-}\mathbf{Elim}\ \vec{s}'\ (\mathsf{c}\ \vec{a}'\ \vec{y}')\ P'\ \vec{m}' : T$ then, as before, $\vec{s}' = (\vec{a}'/\vec{a} \circ \vec{y}'/\vec{y})\vec{s}$. Now,

$\text{MATCHES}(\vec{p}, (\vec{a}'/\vec{a} \circ \vec{y}'/\vec{y})\vec{s}) \Longrightarrow \circ_{a_i \in V}(a_i'/a_i)$
$\text{MATCHES}(\mathsf{c}\ \vec{a}^{\{V\}}\ \vec{y}, \mathsf{c}\ \vec{a}'\ \vec{y}') \Longrightarrow \circ_{a_i \notin V}(a_i'/a_i) \circ \vec{y}'/\vec{y}$

Hence any matching substitution σ for the left-hand side satisfies

$\text{ID}\left|\sigma(\mathsf{D}\text{-}\mathbf{Elim}\ \vec{p}\ (\mathsf{c}\ \vec{a}^{\{V\}}\ \vec{y})\ P\ \vec{m})\right| = \mathsf{D}\text{-}\mathbf{Elim}\ \vec{s}'\ (\mathsf{c}\ \vec{a}'\ \vec{y}')\ P'\ \vec{m}'$

So these schemes are respectful. They are clearly well-defined, as they discriminate on the target's constructor. □

Theorem 4.5. *Forcing satisfies Properties 1, 2 and 3.*

Proof. See Theorems B.7, B.8 and B.9 in Appendix B. □

Remark: How can we display elements of D accurately if we erase parts of the structure? Information which is dropped by the forcing optimisation can always be retrieved by writing a function in terms of the elimination rule, and so displaying a term does not need direct access to the term's representation; display (or at least conversion to a textual representation) can be implemented in terms of the elimination rule, writing a function similar to the **show** function in Haskell. Assuming the existence of a String type, we might write a **show** function for D by the following scheme:

$$\underline{\text{let}} \quad \frac{d \ : \ \mathsf{D}\,\vec{i}}{\mathbf{show}\ d \ : \ \mathsf{String}}$$

$$\begin{aligned}
&\mathbf{show}_{\vec{i}} \quad d \quad \Leftarrow \underline{\text{elim}}\ d \\
&\mathbf{show}_{\vec{s}}\ (\mathsf{c}_1\ \vec{a}_1\ \vec{y}_1) \ \mapsto\ \text{"}\mathsf{c}_1\text{"} +\!\!+ (\mathbf{show}\ \vec{a}_1) +\!\!+ (\mathbf{show}_{\vec{r}_1}\ \vec{y}_1) \\
&\ldots \\
&\mathbf{show}_{\vec{s}}\ (\mathsf{c}_n\ \vec{a}_n\ \vec{y}_n) \ \mapsto\ \text{"}\mathsf{c}_n\text{"} +\!\!+ (\mathbf{show}\ \vec{a}_n) +\!\!+ (\mathbf{show}_{\vec{r}_n}\ \vec{y}_n)
\end{aligned}$$

This assumes appropriate **show** functions for each of the $\vec{a}$, but in principle we see that displaying structures, including their erased elements, is straightforward.

4.3.2 Eliding Redundant Constructor Tags

In the second alternative implementation of **Vect-Elim** in figure 4.8, case selection is by analysis of the length index rather than the target itself:

$$\begin{aligned}
&\mathbf{Vect\text{-}Elim}\ A\ \ 0 \qquad ([\epsilon]\ [A]) \qquad P\ m_\epsilon\ m_{::} \ \rightsquigarrow\ m_\epsilon \\
&\mathbf{Vect\text{-}Elim}\ A\ (\mathsf{s}\ k)\ ([::]\ [A]\ [k]\ a\ v)\ P\ m_\epsilon\ m_{::} \ \rightsquigarrow\ m_{::}\ k\ a\ v\ (\mathbf{Vect\text{-}Elim}\ A\ k\ v\ P\ m_\epsilon\ m_{::})
\end{aligned}$$

For which types can we do case selection on an argument other than the target?

If we have two constructor headed terms $\mathsf{c}\,\vec{a}, \mathsf{c}'\,\vec{b}$ in a type $\mathsf{D}\,\vec{s}$, and the constructor choice is determined only by (or forced by) the indices $\vec{s}$, such that $\mathsf{c} \equiv \mathsf{c}'$ we say that the family D is **detaggable** (figure 4.21). i.e. the constructor tag is determined only by $\vec{s}$; given $\vec{s}$, we can tell what the constructor tag must be. Vect is detaggable because the length index determines whether the constructor is ϵ (if the length index is 0) or :: (if the length index is $\mathsf{s}\ k$).

Again, there is no method in general to tell whether a family is detaggable, but we can use properties of constructors to identify some families as detaggable. For any set of ι-schemes, if the index patterns are already mutually exclusive, we can decide which scheme applies without checking the target's constructor tag. The DISJOINT operation (figure 4.22) checks if two patterns are guaranteed to match disjoint sets of terms.

A family D is **detaggable**
if $\Gamma \vdash \mathsf{c}\ \vec{a}, \mathsf{c}'\ \vec{b} : \mathsf{D}\ \vec{s}$ implies $\mathsf{c} \equiv \mathsf{c}'$

Figure 4.21: Detaggable families

$$
\begin{array}{l}
\text{DISJOINT}(\ \mathsf{c}\ \vec{p}\ ,\ \mathsf{c}'\ \vec{q}\ ) \Longrightarrow \mathsf{true}\ \underline{\text{if}}\ \mathsf{c} \neq \mathsf{c}' \\
\text{DISJOINT}(\ \mathsf{c}\ \vec{p}\ ,\ \mathsf{c}\ \vec{q}\ ) \Longrightarrow \exists i.\text{DISJOINT}(p_i, q_i) \\
\text{DISJOINT}(\ [\mathsf{c}]\ \vec{p}, [\mathsf{c}']\ \vec{q}) \Longrightarrow \exists i.\text{DISJOINT}(p_i, q_i) \\
\text{DISJOINT}(\ p\ ,\ q\ ) \Longrightarrow \mathsf{false}\ \underline{\text{otherwise}}
\end{array}
$$

Figure 4.22: The DISJOINT meta-operation

Of course if we are to match on the indices then we must actually examine their constructors, so the previous lazy definition of PATS is not sufficient. We compute the patterns we need for this optimisation with EPATS (figure 4.23) — the same as PATS but with LAZY replaced by EAGER. EAGER generates patterns without commented out constructors, to indicate to the pattern matching compiler that it may inspect these tags.

$$
\begin{array}{l}
\text{EPAT}\ (\ V,\ x\ ) \Longrightarrow (x \cup V, x)\ \underline{\text{if}}\ x \notin V \\
\text{EPAT}\ (\ V, \mathsf{c}\ \vec{t}) \Longrightarrow (V', \text{EAGER}(\mathsf{c}, \vec{p}))\ \underline{\text{if}}\ \text{EPATS}\ (V, \vec{t}) \Longrightarrow (V', \vec{p}) \\
\text{EPAT}\ (\ V,\ t\ ) \Longrightarrow (V, [t]) \\
\text{EPATS}(\ V, \mathsf{nil}) \Longrightarrow (V, \mathsf{nil}) \\
\text{EPATS}(\ V, t\ \vec{t}) \Longrightarrow (V'', p\ \vec{p}) \\
\qquad\qquad \underline{\text{if}}\ \text{EPAT}\ (V, t) \Longrightarrow (V', p)\ \underline{\text{and}}\ \text{EPATS}\ (V', \vec{t}) \Longrightarrow (V'', \vec{p}) \\
\text{EAGER}(\ \mathsf{c}, \vec{p}) \Longrightarrow \mathsf{c}\ \vec{p}
\end{array}
$$

Figure 4.23: Extracting eager patterns

For each constructor $\mathsf{c}_i : \forall \vec{x} : \vec{X}_i.\ \mathsf{D}\ \vec{s}_i$ of a family D, EPATS $(\emptyset, \vec{s}_i)$ gives us $(V_i, \vec{p}_i)$, where V_i is the set of arguments of c_i forced by $\vec{s}_i$ and $\vec{p}_i$ are the patterns which D-**Elim** will match. If the patterns $\vec{p}_i$ generated from the indices are mutually exclusive, we say D is **concretely detaggable** (figure 4.24).

The pattern sets are mutually exclusive if the following property holds:

$$\forall\, i \neq j.\ \exists k.\ \text{DISJOINT}(p_{ik}, p_{jk}) \Longrightarrow \mathsf{true}$$

That is, for every pair of ι-schemes, one of the indices is matched in each scheme by disjoint patterns; this ensures that by examining all of the indices we have reduced the number of possible ι-schemes to one. In order to implement detagging, we extend ExTT's operational semantics with deleted constructor patterns $\{\mathsf{c}\}\ \vec{p}$. A deleted constructor pattern $\{\mathsf{c}\}\ \vec{p}$ matches a term t if the canonical form of t is a deleted constructor application $\{\mathsf{c}\}\ \vec{t}$

For a family D with i constructors of the form
$\mathsf{c}_i \ : \ \forall \vec{x} : \vec{X}_i.\ \mathsf{D}\ \vec{s}_i$
Where for each i, EPATS $(\emptyset, \vec{s}_i) \implies (V_i, \vec{p}_i)$
if $\forall\, i \neq j.\ \exists k.$ DISJOINT(p_{ik}, p_{jk}) then D is **concretely detaggable**

Figure 4.24: Concretely detaggable families

and $\vec{p}$ also matches $\vec{t}$.

$$\text{MATCH}(\{\mathsf{c}\}\ \vec{p}, t) \implies \text{MATCHES}(\vec{p}, \vec{t}) \text{ if } \text{WHNF}(t) \implies (\{\mathsf{c}\}\ \vec{t})$$

We are careful to distinguish $(\{\mathsf{c}\}\ \{\vec{t}\})$, which is a trivial canonical form with its constructor and all of its arguments deleted, from $\{\mathsf{c}\ \vec{t}\}$, which is deleted altogether.

Lemma 4.6. *If* D *is concretely detaggable then* D *is detaggable.*

Proof. For two constructors of D, c_i and c_j, the patterns $\vec{p}_i$ and $\vec{p}_j$ are generated by EPATS. No set of terms can match both sets of patterns unless $i = j$, by the definition of concretely detaggable.

If we have $\Gamma \vdash \mathsf{c}\ \vec{a}, \mathsf{c}'\ \vec{b}\ :\ \mathsf{D}\ \vec{s}$, then we have EPATS $(\emptyset, \vec{s}) \implies (V, \vec{p})$. Since no term can match patterns for more than one constructor, $\vec{p}$ determines the constructor, so $\Gamma \vdash \mathsf{c} \equiv \mathsf{c}'$. □

The Detagging Optimisation

The detagging optimisation scheme is given in figure 4.25. Note that this optimisation subsumes the forcing optimisation by marking $\vec{x}$ with $a^{\{V\}}$. Detagging for vectors is given in figure 4.26. The types of the constructors in ExTT are as for the forcing optimisation; however, they are added to the context with deletion marks:

$$\{\epsilon\} \ : \ \forall \{A{:}\star\}.\, \mathsf{Vect}\ A\ 0$$
$$\{:\} \ : \ \forall \{A{:}\star\}.\, \forall \{k{:}\mathbb{N}\}.\, \forall a{:}A.\, \forall v{:}\mathsf{Vect}\ A\ k.\, \mathsf{Vect}\ A\ (\mathsf{s}\ k)$$

Recall that the definition of contexts only allows us to add constructors of a family with deletion marks to the context if the indices of the type are pairwise disjoint with previously added constructors of the same family. This side condition holds for detaggable families, since detaggability is decided by pairwise disjointness of indices.

We achieve this space optimisation at the cost of using eager rather than lazy patterns. The number of constructor tests required increases by a constant factor (possibly zero if, as in the case of Vect, there is another index with disjoint patterns across all ι-schemes) and indices may sometimes be computed where they would previously be ignored. In practice, we take a greedy approach to minimising the number of eager patterns required to make the distinction, by checking the index with the most disjoint constructor tags first.

The number of constructor tests required is a factor in deciding whether to apply this optimisation, the balance being between speed and storage requirements. If we are more

$[\![\mathsf{c}_i]\!] \Longrightarrow \lambda \vec{x}.\ \{\mathsf{c}_i\}\ \vec{x}^{\{V\}}$
$\mathsf{D\text{-}Elim}\ \vec{p}_i\ (\{\mathsf{c}_i\}\ \vec{x}^{\{V\}})\ P\ \vec{m} \rightsquigarrow e_i$
where e_i is the right hand side from the standard implementation of D-**Elim**.
and $\{\mathsf{c}_i\} : \forall \vec{a} : \vec{A}^{\{V\}}.\ \forall \vec{y} : [\![\vec{Y}]\!].\ \mathsf{D}\ [\![\vec{s}]\!] \in [\![\Gamma]\!]$
where $\forall a : A^{\{V\}} \Longrightarrow \forall \{a : [\![A]\!]\}$ if $a \in V$
$\forall a : A^{\{V\}} \Longrightarrow \forall a : [\![A]\!]$ otherwise

Figure 4.25: The detagging optimisation

$[\![\epsilon]\!] \Longrightarrow \lambda A.\ \{\epsilon\}\ \{A\}$
$[\![::]\!] \Longrightarrow \lambda A; k; a; v.\ \{::\}\ \{A\}\ \{k\}\ a\ v$
$\mathsf{Vect\text{-}Elim}\ A\ \ 0\ \ (\{\epsilon\}\ \{A\})\ \ P\ m_\epsilon\ m_{::} \rightsquigarrow m_\epsilon$
$\mathsf{Vect\text{-}Elim}\ A\ (\mathsf{s}\ k)\ (\{::\}\ \{A\}\ \{k\}\ a\ v)\ P\ m_\epsilon\ m_{::} \rightsquigarrow m_{::}\ k\ a\ v\ (\mathsf{Vect\text{-}Elim}\ A\ k\ v\ P\ m_\epsilon\ m_{::})$

Figure 4.26: Detagging for Vect

concerned with speed, we might prefer to limit the number of constructor tests on the indices to one, or even not allow detagging at all to avoid the overhead of eager pattern matching. However, if we are more concerned with space, we might not want a limit on the number of constructor tests at all.

As with many optimisations, it is difficult to decide on a single best approach for all cases and it may even be preferable to leave the maximum acceptable number of constructor tests as an option for the programmer.

Properties of Detagging

Detagging, like forcing, satisfies the required properties of a compile-time optimisation. The elimination rule is respectful and well-defined, and typechecking the resulting terms in ExTT is equivalent to typechecking in TT.

Theorem 4.7. *The detagging implementation of* D-***Elim*** *is respectful and well-defined*

Proof. These schemes are respectful for all Γ by the same argument as for forcing—the switch to eager patterns does not affect the set of variables matched from the indices, nor the success of matching well-typed values. Deleting the constructor in the target can only improve the possibility of a match, but the disjointness condition directly ensures that the schemes remain well-defined. □

Theorem 4.8. *Detagging satisfies Properties 1, 2 and 3.*

Proof. See Theorems B.12, B.13 and B.14 in Appendix B. □

4.3.3 Collapsing Content Free Families

Consider the less than or equal relation, declared and elaborated as follows:

$$\underline{\text{data}}\quad \frac{x,y\ :\ \mathbb{N}}{x{\leq}y\ :\ \star}\quad \underline{\text{where}}\quad \frac{}{\mathsf{leO}\ :\ 0{\leq}y}\qquad \frac{p\ :\ x{\leq}y}{\mathsf{leS}\,p\ :\ \mathsf{s}\,x{\leq}\mathsf{s}\,y}$$

$\leq\ :\ \mathbb{N}\to\mathbb{N}\to\star$

$\mathsf{leO}\ :\ \forall y{:}\mathbb{N}.\leq 0\,y$

$\mathsf{leS}\ :\ \forall x,y{:}\mathbb{N}.\leq x\,y\to\leq(\mathsf{s}\,x)\,(\mathsf{s}\,y)$

Note that we use prefix notation for $\leq$ when it is in elaborated form, and infix for the higher level notation. The $\leq$ family describes a property of its indices and stores no other data. It is not surprising therefore to find that much of its content can be deleted. The detagging optimisation on $\leq$ (with concretely forced arguments also deleted) is given in figure 4.27.

$$\begin{array}{l}
[\![\mathsf{leO}]\!] \Longrightarrow \lambda y.\ (\{\mathsf{leO}\}\ \{y\}) \\
[\![\mathsf{leS}]\!] \Longrightarrow \lambda x\,y\,p.\ (\{\mathsf{leS}\}\ \{x\}\ \{y\}\ p) \\
\quad \leq\textbf{-Elim}\ 0\qquad y\qquad (\{\mathsf{leO}\}\ \{y\})\quad P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}}\ \rightsquigarrow\ m_{\mathsf{leO}}\ y \\
\quad \leq\textbf{-Elim}\ (\mathsf{s}\,x)\ (\mathsf{s}\,y)\ (\{\mathsf{leS}\}\ \{x\}\ \{y\}\ p)\ P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}} \\
\qquad\qquad\qquad\qquad \rightsquigarrow\ m_{\mathsf{leS}}\ x\ y\ p\ (\leq\textbf{-Elim}\ x\ y\ p\ P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}})
\end{array}$$

Figure 4.27: Optimisation of $\leq$

Now we are left with only one undeleted argument, the recursive p in leS. This argument serves two purposes — firstly it is the target of the recursive call and secondly it is passed to the method m_{leS}. We might think that p can also be elided — ultimately it can only by examined directly by $\leq$**-Elim** which, by induction, can be shown *never* to examine it (since the target is not examined at all in the base case, and the recursive argument is passed as the target to each recursive call). In compile-time evaluation, however, where we may reduce under binders, we must at least check that the target is canonical for reduction to be possible. If not, we run the risk of reducing a proof of something which cannot be constructed, such as $5{\leq}4$!

Compile-Time vs. Run-Time Implementation

In our **Vect-Elim** example, we deleted both ϵ and its argument. We might be tempted to go a step further, and comment out that entire target, since the A and 0 indices tell us exactly what the canonical form of the target must be.

$$\textbf{Vect-Elim}\ A\ 0\ [\{\epsilon\}\ \{A\}]\ P\ m_\epsilon\ m_{::}\ \rightsquigarrow\ m_\epsilon$$

However, this ι-scheme is not respectful and breaks subject reduction thus:

$$\begin{array}{rcl}
\ldots;x : \mathsf{Vect}\ A\ 0 \vdash\ \textbf{Vect-Elim}\ A\ 0\ x\ P\ m_\epsilon\ m_{::} & : & P\ 0\ x \\
\rightsquigarrow\ m_\epsilon & : & P\ 0\ \epsilon
\end{array}$$

The pattern $(\{\epsilon\}\,\{A\})$ may not test tags or extract arguments, but it still only matches targets whose weak head-normal forms are constructor applications. The forcing and detagging optimisations are safe to use in any context, and we need to reduce under binders (that is, in a non-empty context) when performing the conversion checks which ensure that EPIGRAM programs elaborate to well typed terms.

However, at run-time, we can employ a much more restricted notion of computation, reducing only in the *empty* context. The nature of run-time evaluation is that we produce only as much of a normal form as the programmer requires. While we *can* produce a strongly normalised term due to the termination property of TT, we only reduce the scope of a binding if it is applied (i.e. bound) to a canonical form.

In the run-time scenario, we can exploit the **adequacy** property of TT (figure 4.28) to gain further optimisations, not available in a general context; in the empty context, t must reduce to some constructor form.

$$\boxed{\begin{array}{l}\underline{\text{if}}\ \ \vdash t : \mathsf{D}\,\vec{s} \\ \underline{\text{then}}\ \textsc{whnf}(t) = \mathsf{c}\,\vec{t}\ \text{for some}\ \vec{t}\end{array}}$$

Figure 4.28: Adequacy of TT

The adequacy property ensures that in the empty context, there is no non-canonical normal form to which t can reduce; the only normal forms available are canonical forms. In effect, we may employ weaker criteria for alternative implementations of elimination operators in run-time execution, since such execution always takes place in the empty context. The respectfulness condition at run-time (figure 4.29) is the same as respectfulness, with the additional constraint that it holds only in the empty context.

$$\boxed{\begin{array}{l}\underline{\text{if}}\ \ \vdash \mathsf{D}\textbf{-Elim}\,\vec{t} : T \\ \quad \text{and}\ \textsc{matches}(\vec{p}_i, \vec{t}) \Longrightarrow \sigma \\ \quad \underline{\text{then}}\ \ \vdash \mathsf{D}\textbf{-Elim}\,\sigma|\vec{p}_i| \equiv \mathsf{D}\textbf{-Elim}\,\vec{t} : T\end{array}}$$

Figure 4.29: The run-time respectfulness condition for ι-schemes

We also have a weaker criterion for well-definedness (figure 4.30) which takes into account that all values passed to a fully applied function are in canonical form.

$$\boxed{\begin{array}{l}\underline{\text{if}}\ \vdash \mathsf{D}\textbf{-Elim}\,\vec{t} : T\text{, where D-Elim is fully applied} \\ \quad \underline{\text{then}}\ \ \textsc{matches}(\vec{p}_i, \vec{t}) \Longrightarrow \sigma\ \text{ for exactly one } i\end{array}}$$

Figure 4.30: The run-time well-definedness condition for ι-schemes

The adequacy property tells us that the target will always match a constructor pattern

at run-time, hence we may safely presuppose a pattern from which no information is gained, as suggested above. Moreover, by applying this observation inductively, we can sometimes extract another, more drastic optimisation from the guarantee of adequacy at run-time — collapsing of content free data structures.

4.3.4 The Collapsing Optimisation

Let us reconsider the optimisation of $\leq$ in a run-time scenario. At run-time, always reducing in the empty context, we never need to check that the recursive argument p is canonical because the adequacy property tells us that it must be. Hence, at run-time, we no longer need to store the recursive argument — the entire family collapses. This optimisation is given in figure 4.31.

$$
\begin{array}{l}
[\![\mathsf{leO}]\!] \Longrightarrow \lambda y.\,(\{\mathsf{leO}\ y\}) \\
[\![\mathsf{leS}]\!] \Longrightarrow \lambda x; y; p.\,(\{\mathsf{leS}\ x\ y\ p\}) \\
\quad \leq\textbf{-Elim}\ 0 \qquad y \qquad \{\mathsf{leO}\ y\} \quad P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}} \quad \rightsquigarrow\ m_{\mathsf{leO}}\ y \\
\quad \leq\textbf{-Elim}\ (\mathsf{s}\ x)\ (\mathsf{s}\ y)\ \{\mathsf{leS}\ x\ y\ p\}\ P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}} \\
\qquad\qquad\qquad\qquad \rightsquigarrow\ m_{\mathsf{leS}}\ x\ y\ (\{p\})\ (\leq\textbf{-Elim}\ x\ y\ \{p\}\ P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}})
\end{array}
$$

Figure 4.31: Run-time optimisation of $\leq$

Note that $(\{p\})$ remains an argument to the m_{leS} method, although after deletion we pass the trivial canonical object; since m_{leS} can be instantiated by any function of an appropriate type we must take into account the possibility that it is instantiated by a polymorphic function, where it is unknown at compile-time whether an argument is collapsible or not.

For which families can we do this run-time optimisation?

If we have two terms a, b in a family $\mathsf{D}\ \vec{s}$, and the *values* of a and b are determined entirely by $\vec{s}$, such that there is at most one element of $\mathsf{D}\ \vec{s}$, then we say D is **collapsible** (figure 4.32). The relation $\leq$ is collapsible because there is only one way of constructing any value in $x \leq y$ for given indices x and y.

A family D is **collapsible**
if $\vdash x, y\ :\ \mathsf{D}\ \vec{s}$ implies $\vdash x \simeq y$

Figure 4.32: Collapsible families

Again, deciding whether a family is collapsible is likely to be undecidable in general, but we can apply a more restricted notion which identifies collapsible families which can be reconstructed from their indices. We say a family is **concretely collapsible** (figure 4.33) if it is concretely detaggable (which accounts for reconstructing the constructor choice from the indices), and for each constructor $\mathsf{c}\ :\ \forall \vec{a} : \vec{A}.\ \mathsf{D}\ \vec{r}_1 \rightarrow \ldots \rightarrow \mathsf{D}\ \vec{r}_j \rightarrow \mathsf{D}\ \vec{s}$, EPATS $(\emptyset, \vec{s})$

gives $(\vec{a}, \vec{p})$ — that is, *all* of the non-recursive arguments $\vec{a}$ appear in the set of concretely forceable variables.

For a concretely detaggable family D with i constructors of the form
$\mathsf{c}_i : \forall \vec{a} : \vec{A}_i.\ \mathsf{D}\ \vec{r}_{i1} \to \ldots \to \mathsf{D}\ \vec{r}_{ij} \to \mathsf{D}\ \vec{s}$
If for each i, EPATS $(\emptyset, \vec{s}) \implies (\vec{a}, \vec{p})$ then D is **concretely collapsible**

Figure 4.33: Concretely collapsible families

The general case for the collapsing optimisation is given in figure 4.34. The original D-**Elim**, which is passed an argument in the family D, is transformed into a new version of D-**Elim** which has that argument dropped. The motive still has the same type as in the standard D-**Elim**, but the only value which will be passed in the target position will be the trivial canonical value, $\langle\rangle$.

$$\begin{array}{l}
\mathsf{D}\text{-}\mathbf{Elim}\ \vec{p}\ \{\mathsf{c}\ \vec{a}\ \vec{y}\}\ P\ \vec{m} \\
\qquad \leadsto m_{\mathsf{c}}\ \vec{a}\ (\{y_1\})\ \ldots\ (\{y_n\})\ (\mathsf{D}\text{-}\mathbf{Elim}\ \vec{r}_1\ \{y_1\}\ P\ \vec{m})\ \ldots\ (\mathsf{D}\text{-}\mathbf{Elim}\ \vec{r}_n\ \{y_n\}\ P\ \vec{m}) \\
[\![\mathsf{c}]\!] \implies \lambda \vec{a}; \vec{y}.\ (\{\mathsf{c}\ \vec{a}\ \vec{y}\}) \\
[\![\mathsf{D}\text{-}\mathbf{Elim}]\!] \implies \lambda \vec{i}; x; P; \vec{m}.\ \mathsf{D}\text{-}\mathbf{Elim}\ \vec{i}\ \{x\}\ P\ \vec{m}
\end{array}$$

Figure 4.34: The collapsing optimisation

Theorem 4.9. *The collapsing implementation of* D-***Elim*** *is respectful at run-time and well-defined at run-time.*

Proof. These schemes are well-defined at run-time (in the empty context) by the same argument as for detagging. They are respectful at run-time because the only possible left-hand sides have the form $\vdash \mathsf{D}\text{-}\mathbf{Elim}\ \vec{s}'\ (\mathsf{c}\ \vec{a}'\ \vec{y}')\ P'\ \vec{m}'$, hence, by disjointness, the only possible match, even with the target deleted, is with the scheme for c, with matching substitution $\sigma = \vec{a}'/\vec{a} \circ P'/P \circ \vec{m}'/\vec{m}$, binding all the undeleted free variables on the right-hand side because EPATS $(\emptyset, \vec{s}) \implies (\vec{a}, \vec{p})$. Taking $\tau = \vec{y}'/\vec{y}$, we see that

$$\vdash \tau\,|\sigma(\mathsf{D}\text{-}\mathbf{Elim}\ \vec{p}\ \{\mathsf{c}\ \vec{a}\ \vec{y}\}\ P\ \vec{m})| = \mathsf{D}\text{-}\mathbf{Elim}\ \vec{s}'\ (\mathsf{c}\ \vec{a}'\ \vec{y}')\ P'\ \vec{m}'$$

hence these schemes are respectful at run-time. □

Trade-offs

For a concretely collapsible family, the constructor tag and all the non-recursive arguments are *cheaply* recoverable from the indices. "Cheaply" means that the arguments can be retrieved in constant time by matching on the fully evaluated indices, and the constructor tag can be determined by inspecting a (user determined) small number of the constructor tags on the indices.

There is a trade-off in all of these optimisations between storage requirements and speed. Even though arguments can be retrieved in constant time, for non-trivial indices — e.g. s(s(s(s(s n))))) — the cost of recovery increases, as recovering the value n in this case involves analysing the argument of each successor symbol. Another issue is that indices may also be computed as the result of a function; in a lazy evaluation setting, an effect of forcing here may be to compute a value which would otherwise remain unused. We have not yet explored the space/time trade-offs of these optimisations for such complex indices, in particular because the dependently typed programs we have investigated so far have not had such complex indices.

The possibility of collapsing data structures is the main advantage of the detagging optimisation; detagging is a necessary step towards collapsing. In general, the space saving in not storing the tag of a family at run-time is small in comparison to the fact that we are now committed to retaining some indices in order to discriminate between ι-reductions. Otherwise, as we will see with some of the optimisations in Chapter 6, we may be able to discard these indices. If detagging leads to collapsing of an otherwise redundant data structure however, it is beneficial.

4.3.5 Interaction Between Optimisations

While these optimisations work well on inductive families in isolation, we should consider how optimisations will interact when several constructors and elimination rules are transformed. There is one consideration in particular — earlier, I stated that the elimination rule for a family D was the *only* function allowed to examine D directly. Clearly, in the presence of detagging, this is no longer the case. Now, *any* elimination rule is able to examine D, if D forms an index of a concretely detaggable family. For example, the detagged Vect-**Elim** rule has direct access to the constructors of $\mathbb{N}$.

What problems might this cause? Consider the following data structure, an association list which links a Vect of keys (B) to their values (A):

$$\underline{\text{data}} \quad \frac{A, B : \star \quad v : \mathsf{Vect}\ B\ n}{\mathsf{aVect}\ A\ B\ v : \star}$$

$$\underline{\text{where}} \quad \frac{}{\mathsf{aNil} : \mathsf{aVect}\ A\ B\ \epsilon} \qquad \frac{a : A \quad l : \mathsf{aVect}\ A\ B\ v}{\mathsf{aCons}\ a\ l : \mathsf{aVect}\ A\ B\ (b{::}v)}$$

A first look at this suggests it might be a detaggable family; each constructor's Vect index is disjoint, surely? However, since Vect itself is detaggable, we can no longer discriminate on its constructors! The elaboration of aVect is shown in figure 4.35.

That is, the current set of substitutions from TT to ExTT are applied immediately the family is elaborated. Notice that although we can not discriminate on the constructors of Vect, by indexing over a Vect we must also index over Vect's indices! And so, this family is also detaggable, by disjointness of Vect's indices.

In general, if constructors of a family D are indexed by disjoint constructors of a de-

$$
\begin{array}{l}
\mathsf{aNil} \;:\; \forall A, B{:}\star.\, \mathsf{aVect}\ A\ B\ \{\epsilon\} \\
\mathsf{aCons} \;:\; \forall A, B{:}\star.\, \forall a{:}A.\, \forall b{:}B.\, \forall n{:}\mathbb{N}.\, \forall v{:}\mathsf{Vect}\ B\ n.\, \forall l{:}\mathsf{aVect}\ A\ B\ v. \\
\qquad\qquad \mathsf{aVect}\ A\ B\ (\{{:}\}\ \{B\}\ \{n\}\ b\ v) \\
\quad \mathsf{aVect\text{-}}\mathbf{Elim}\ A\ B\ \ 0\ \ \ (\{\epsilon\}\ \{B\})\ \ \ \mathsf{aNil}\ \ \ P\ m_{\mathsf{aNil}}\ m_{\mathsf{aCons}} \rightsquigarrow m_{\mathsf{aNil}} \\
\quad \mathsf{aVect\text{-}}\mathbf{Elim}\ A\ B\ (\mathsf{s}\ k)\ (\{{:}\}\ \{B\}\ \{k\}\ b\ v)\ (\mathsf{aCons}\ A\ B\ a\ b\ k\ l)\ P\ m_{\mathsf{aNil}}\ m_{\mathsf{aCons}} \\
\qquad\qquad \rightsquigarrow m_{\mathsf{aCons}}\ A\ B\ a\ b\ k\ v\ l\ (\mathsf{aVect\text{-}}\mathbf{Elim}\ A\ B\ k\ v\ l\ P\ m_{\mathsf{aNil}}\ m_{\mathsf{aCons}})
\end{array}
$$

Figure 4.35: Elaboration of aVect

taggable family X, D is also detaggable because the case distinction which discriminates between X's constructors can also be used to discriminate between D's constructors. We must, however, be careful to apply substitutions as we go so as not to attempt pattern matching on these detagged constructors.

4.3.6 Using the Standard Implementation

It is also possible that we could take a more straightforward approach to implementing optimised elimination rules, even using the standard implementation. Recall the standard implementation of Vect-**Elim**, which marks the indices as presupposed:

$$
\begin{array}{l}
\mathsf{Vect\text{-}}\mathbf{Elim}\ [A]\ [0]\ \ \ (\epsilon\ A)\ \ \ P\ m_{\epsilon}\ m_{::} \rightsquigarrow m_{\epsilon} \\
\mathsf{Vect\text{-}}\mathbf{Elim}\ [A]\ [\mathsf{s}\ k]\ (::\ A\ k\ a\ v)\ P\ m_{\epsilon}\ m_{::} \rightsquigarrow m_{::}\ k\ a\ v\ (\mathsf{Vect\text{-}}\mathbf{Elim}\ A\ k\ v\ P\ m_{\epsilon}\ m_{::})
\end{array}
$$

In the forcing, detagging and collapsing optimisations, we have exploited the presence of constructor symbols in indices to remove arguments from data structures. An alternative optimisation, however, would arise from the observation that the indices are never examined in the body of the standard implementation of D-**Elim** so need not be passed to the elimination operator at all. This would save space, in that there would not be extra references to the indices on the stack, but we would also expect to save time since building an application of the elimination rule would require fewer MKAP instructions.

In the current implementation, we prefer to use the alternative implementations generated by the forcing, detagging and collapsing optimisations, rather than the standard implementation, for two reasons:

- Firstly, if we remove arguments from the application of the elimination operator, rather than the application of the constructor, then there will still be pointers to the indices at each level of a recursive data structure. If on the other hand we remove arguments from the constructor, there are only pointers to the indices at the top level application of the elimination operator — these applications may, of course, occupy a significant amount of memory in a lazy implementation.

- Secondly, as we will see in Chapter 6, we have further techniques for optimising applications of elimination operators which can in many cases remove arguments to the

operator as well as constructor arguments. If we make the choice too early between using the standard implementation and an alternative implementation, we will be denied these optimisations.

Nevertheless, there are many issues to consider in optimising a program, and it is not clear whether the techniques presented above are optimal in all cases. For example, building an application of an eliminator is more expensive than building a constructor application, since it requires more steps (MKAP applies a function to only one argument, MKCON applies a constructor to all of its arguments, since it can assume that constructors are fully applied). Further work is required to determine how other implementation choices (for example, lazy versus eager evaluation) affect the optimisations.

4.4 Compilation Scheme for ExTT

4.4.1 Extensions to RunTT

The language of supercombinators, RunTT, is now built from marked terms in ExTT. The translation into ExTT is an analysis phase and the actual erasure is applied in the lifting on RunTT supercombinators. Marked terms, $\{t\}$, are simply omitted as part of the supercombinator lifting algorithm. By erasing the same arguments from constructors and patterns of ι-schemes, we ensure that deleted arguments are matched only by deleted patterns and therefore both can safely be removed.

Compiling elimination rules into RunTT is no longer so straightforward as compilation to a case on the target of the elimination rule. Previously, we used case analysis on the target to extract constructor arguments. Having elided some of these owing to their repetition, we need another means of retrieving their values. One way to do this is with multiple case expressions. However, this is potentially expensive. As a result of the well-definedness property of ι-schemes, if we know which ι-scheme applies, we also know the form of each argument to the elimination rule. As a result, we would like to be able to project subterms out of these arguments without checking their form. We would like to do only enough case analysis to establish which ι-scheme applies. We also now introduce a "match anything" pattern for case analysis which is useful in the compilation of ι-schemes for detagged families.

To avoid case analysis where we already know the form of an argument, I introduce **argument projection** into RunTT. Where a term t in RunTT is known to have the form $\mathsf{c}\langle e_0, e_1, \ldots e_n\rangle$, $t!i$ projects the ith argument out of the tuple if $i \leq n$, and is undefined otherwise. However, we know that $i \leq n$ must hold, so there is no run-time check.

As a result of the detagging optimisation, we would also like to delete constructor tags from the RunTT representation. As well as tagged constructor applications, $\mathsf{c}\langle\vec{e}\rangle$, I introduce untagged constructor applications, $\langle\vec{e}\rangle$. case analysis on such forms is not allowed; instead, argument projection is used to retrieve arguments.

The syntax of this extended RunTT is given in figure 4.36.

$s ::=$	$\lambda \vec{a} : \vec{e}.\ e$	(supercombinator)			
$e ::=$	x	(bound variable)	$\mid$	f	(global name)
$\mid$	$\forall x : e.\ e$	(function space)	$\mid$	$\star_i$	(type of types)
$\mid$	$e\ e$	(function application)	$\mid$	$\mathsf{c}\langle\vec{e}\rangle$	(constructor application)
$\mid$	$\underline{\text{let}}\ a : e \mapsto e\ \underline{\text{in}}\ e$	(let binding)	$\mid$	$\mathsf{D}\langle\vec{e}\rangle$	(type constructor application)
$\mid$	$e!i$	(argument projection)	$\mid$	$\langle\vec{e}\rangle$	(untagged constructor)
$\mid$	$\underline{\text{case}}\ e\ \underline{\text{of}}\ \vec{alt}$	(case expression)			
$alt ::=$	$\mathsf{c}\ \langle\vec{x}\rangle \rightsquigarrow e$	(case alternative)			
$\mid$	$\_ \rightsquigarrow e$	(match anything)			

Figure 4.36: RunTT with extensions

4.4.2 Compiling Elimination Rules

Translating into the supercombinator representation from ExTT is relatively straightforward, using the algorithm presented in Chapter 3 with the additional requirement that marked terms be removed. Translation of pattern matching ι-schemes is less straightforward. The main problem arises where a family has been detagged since we are no longer guaranteed that there is an argument for which all patterns have disjoint constructor tags. Another problem is that case analysis on the target does not necessarily retrieve all of the arguments which will be passed to the method, as some may have been removed by the forcing optimisation.

Pattern matching ι-schemes can be compiled into $\underline{\text{case}}$ expressions using Augustsson's pattern matching compiler algorithm [Aug85, Pey87] with some modifications and simplifications. The priority is to establish which ι-scheme applies with as few case expressions as possible. Well-definedness at run-time of elimination rules tells us that exactly one of the cases must match; there is no error case to handle. This property makes optimisation of pattern matching much easier, compared with other optimisations of the pattern matching compiler algorithm — optimisations such as those described in [SR00, FM01] make use of exhaustiveness information (i.e. checking that the patterns cover all cases) or reordering constructor tests. In particular, the algorithm we present here tests each constructor only once.

The pattern matching compiler is defined by the $\mathcal{I}$ compilation scheme. This scheme takes two arguments; firstly, a sequence of names of the i arguments to the elimination rule, $e_1 \ldots e_i$. The second argument is a list of patterns and their reductions, each one corresponding to a case of the elimination rule.

$$\mathcal{I}(e_1 \ldots e_i, \left\{ \begin{array}{ll} p_{11} \cdots p_{1i} & \rightsquigarrow\ x_1 \\ \ldots & \\ p_{n1} \cdots p_{ni} & \rightsquigarrow\ x_n \end{array} \right\})$$

This scheme compiles a *respectful* and *well defined* (non-overlapping and no error case — exactly one set of patterns matches in all cases) set of ι-schemes of the form

$$\begin{array}{ll} \mathbf{f} & p_{11} \ldots p_{1i} \rightsquigarrow x_1 \\ \ldots & \\ \mathbf{f} & p_{n1} \ldots p_{ni} \rightsquigarrow x_n \end{array}$$

where the arguments are given unique names $e_1 \ldots e_i$.

There are some preliminaries to consider before applying this method. Firstly, we must consider how to project arguments from constructors. For each pattern argument to each ι-scheme, p_{ij}, we extract its variable set $\vec{v}$ (that is, the names which appear as pattern variables in p_{ij}), together with, for each variable v in that variable set, a term t which projects the value of that variable from the argument e_j matched by the pattern p_{ij}. Then the right hand side of the ι-scheme x_i is modified by substituting the term t for the variable v. We define the meta-operation PROJECT, which computes the mappings from ExTT names to RunTT terms, as in figure 4.37.

Given a pattern p, and the name of the argument which matches on that pattern n, PROJECT generates a list of pairs (x, t), where x is a pattern variable and t is the RunTT term which projects the value of x from the argument matched by the pattern. The argument f is a function passed to recursive calls of PROJECT; when looking for names in a nested pattern, f is the RunTT term which retrieves the term matched by the nested pattern.

PROJARGS is a helper operation which retrieves names from nested patterns — i is the index of the argument being examined. For each (unmarked) argument x, PROJARGS calls PROJECT on x with an argument projection composed with f.

$$\begin{array}{l} \text{PROJECT}(n,\ x) \implies \text{PROJECT'}(n, (\lambda a \implies a), x) \\ \\ \text{PROJECT'}(n, f,\ \ x\ \ ) \implies [(x, f\ n)] \\ \text{PROJECT'}(n, f,\ (\mathsf{c}\ \vec{e})\ ) \implies \text{PROJARGS}(n, f, 0, \vec{e}) \\ \text{PROJECT'}(n, f,\ (\{\mathsf{c}\}\ \vec{e})) \implies \text{PROJARGS}(n, f, 0, \vec{e}) \\ \text{PROJECT'}(n, f,\ \ [x]\ \ ) \implies [] \\ \\ \text{PROJARGS}(n, f, i,\ \ \ [] \ \ \ ) \implies [] \\ \text{PROJARGS}(n, f, i,\ (\{x\} : xs)) \implies \text{PROJARGS}\ n\ f\ i\ xs \\ \text{PROJARGS}(n, f, i,\ \ (x : xs)\ ) \\ \quad \implies \text{PROJECT'}(n, ((\lambda a \implies (a!i)) \circ f), x) +\!\!+ \text{PROJARGS}(n, f, (i+1), xs) \end{array}$$

Figure 4.37: The PROJECT operation

For example, if we have a pattern $(\{::\}\ \{A\}\ \{k\}\ a\ v)$ for an argument x, we can extract RunTT terms to retrieve a and v from x with $\text{PROJECT}(x, (\{::\}\ \{A\}\ \{k\}\ a\ v))$. Evaluation of this proceeds as follows:

$$\begin{aligned}
\textsc{project}(x, (\{::\}\ \{A\}\ \{k\}\ a\ v)) &\Rightarrow \textsc{project'}(x, (\lambda x \implies x), (\{::\}\ \{A\}\ \{k\}\ a\ v)) \\
&\Rightarrow \textsc{projargs}(x, (\lambda x \implies x), 0, \langle \{A\}\ \{k\}\ a\ v \rangle) \\
&\Rightarrow \textsc{projargs}(x, (\lambda x \implies x), 0, \langle \{k\}\ a\ v \rangle) \\
&\Rightarrow \textsc{projargs}(x, (\lambda x \implies x), 0, \langle a\ v \rangle) \\
&\Rightarrow \textsc{project'}(x, (\lambda a \implies a!0)\ a) \\
&\qquad +\!\!+ \textsc{projargs}(x, (\lambda x \implies x), 1, \langle v \rangle) \\
&\Rightarrow [(a, x!0)] +\!\!+ \textsc{projargs}(x, (\lambda x \implies x), 1, \langle v \rangle) \\
&\Rightarrow [(a, x!0)] +\!\!+ \textsc{project'}(x, (\lambda a \implies a!1), v) \\
&\Rightarrow [(a, x!0), (v, x!1)]
\end{aligned}$$

Arguments may be nested inside constructors, such as the n in $\mathsf{s}\,(\mathsf{s}\,n)$. In this case, the argument is projected out as follows:

$$\textsc{project}(x, \mathsf{s}\,(\mathsf{s}\,n)) \Rightarrow [(n, (x!0)!0)]$$

For each pattern p_{ij}, a mapping from variable names matched to the RunTT terms which retrieve those variables is given by an application of $\textsc{project}(e_j, (\lambda x \implies x)\ p_{ij})$. Then these terms are substituted in the right hand side of the pattern for the argument names.

For example, the $\mathbb{N}$-**Elim** rule

$$\begin{aligned}
&\mathbb{N}\textbf{-Elim}\ \ 0\ \ P\ m_0\ m_\mathsf{s} &&\rightsquigarrow\ m_0 \\
&\mathbb{N}\textbf{-Elim}\ (\mathsf{s}\ k)\ P\ m_0\ m_\mathsf{s} &&\rightsquigarrow\ m_\mathsf{s}\ k\ (\mathbb{N}\textbf{-Elim}\ k\ P\ m_0\ m_\mathsf{s})
\end{aligned}$$

is compiled to a RunTT case expression by invoking the $\mathcal{I}$ compilation scheme as follows:

$$\mathbb{N}\textbf{-Elim} \mapsto \mathcal{I}(\langle n, P, m_0, m_\mathsf{s} \rangle, \left\{ \begin{array}{lll} 0\ P\ m_0\ m_\mathsf{s} & \rightsquigarrow & m_0 \\ (\mathsf{s}\ k)\ P\ m_0\ m_\mathsf{s} & \rightsquigarrow & m_\mathsf{s}\ (n!0)\ (\mathbb{N}\textbf{-Elim}\ (n!0)\ P\ m_0\ m_\mathsf{s}) \end{array} \right\})$$

A second consideration is how to optimise the rule so that the minimum number of case analyses are required. To achieve this, we reorder the $\vec{e}$ such that the argument where most patterns are disjoint (i.e., the greatest number of disjoint constructors) is examined first. This is a greedy approach, the intention being that one case analysis will suffice in the maximum possible cases. [SR00] describes heuristics for minimising the number of constructor tests, but for elimination rules for non-detaggable families (and even many for detaggable families), there will be an argument where *all* patterns are disjoint.

The $\mathcal{I}$ compilation scheme, summarised in figure 4.38, proceeds by examining the leftmost patterns $p_{11} \ldots p_{1n}$, which represent the patterns which the first argument e_1 could match. It is a recursive function, $\vec{e}$ decreasing in length on each recursive call, which shows its termination. There are several cases to consider.

Case 1: n=1; only one possible pattern

In this case, no further checking need be done, as we have eliminated all but one case. Since the elimination rule is total, this must be the case which applies. No case is needed, as the variables in the patterns are extracted by argument projection in x_1.

Case 1 Only one possible ι-scheme

$$\mathcal{I}(e_1 \ldots e_i, \{p_{11} \ldots p_{1i} \rightsquigarrow x_1\}) \Longrightarrow x_1$$

Case 2 Leftmost patterns are all disjoint

$$\mathcal{I}(e_1 \ldots e_i, \begin{Bmatrix} (\mathsf{c}_1\ \vec{a}_1)\ p_{12} \ldots p_{1i} \rightsquigarrow x_1 \\ \ldots \\ (\mathsf{c}_n\ \vec{a}_n)\ p_{n2} \ldots p_{ni} \rightsquigarrow x_n \end{Bmatrix}) \Longrightarrow \begin{array}{l} \underline{\text{case}}\ e_1\ \underline{\text{of}} \\ \quad (\mathsf{c}_1\ \vec{a}_1) \rightsquigarrow x_1 \\ \quad \ldots \\ \quad (\mathsf{c}_n\ \vec{a}_n) \rightsquigarrow x_n \end{array}$$

Case 3 No pair of leftmost patterns are disjoint

$$\mathcal{I}(e_1 \ldots e_m, \begin{Bmatrix} p_{11} \ldots p_{1i} \rightsquigarrow x_1 \\ \ldots \\ p_{n1} \ldots p_{ni} \rightsquigarrow x_i \end{Bmatrix}) \Longrightarrow \mathcal{I}(e_2 \ldots e_n, \begin{Bmatrix} p_{12} \ldots p_{1i} \rightsquigarrow x_1 \\ \ldots \\ p_{n2} \ldots p_{ni} \rightsquigarrow x_i \end{Bmatrix})$$

Case 4 At least one pair of leftmost patterns is disjoint

Take P to be the smallest set such that $p_{i1} \in P$ if p_{i1} is in constructor form. Then:

$$\mathcal{I}(e_1 \ldots e_i, \begin{Bmatrix} p_{11} \ldots p_{1i} \rightsquigarrow x_1 \\ \ldots \\ p_{n1} \ldots p_{ni} \rightsquigarrow x_n \end{Bmatrix}) \Longrightarrow$$

$\underline{\text{case}}\ e_1\ \underline{\text{of}}$

$$p \rightsquigarrow \mathcal{I}(e_2 \ldots e_n, \begin{Bmatrix} p_{k2} \ldots p_{ki} \rightsquigarrow x_1 \\ \ldots \end{Bmatrix})$$

[where $\forall p \in P,\ \forall k.\ p_{k1} \notin P$ or $(p_{k1} = \mathsf{c}\langle\vec{e}\rangle$ and $p = \mathsf{c}\langle\vec{e}'\rangle)$]

$$\ldots$$

$$\_ \rightsquigarrow \mathcal{I}(e_2 \ldots e_n, \begin{Bmatrix} p_{k2} \ldots p_{ki} \rightsquigarrow x_1 \\ \ldots \end{Bmatrix})$$

[where $\forall k. p_{k1} \notin P$]

Figure 4.38: $\mathcal{I}$ compilation scheme

$$\mathcal{I}(e_1 \ldots e_i, \{p_{11} \ldots p_{1i} \rightsquigarrow x_1\}) \Longrightarrow x_1$$

Case 2: $p_{11} \ldots p_{n1}$ are all disjoint patterns

In this case, distinction can be made on the first argument alone. If $\forall i \neq j.\text{DISJOINT}(p_{i1}, p_{j1})$, p_{i1} is constructor headed for all i, such that $\mathsf{c}_i\ \vec{a}_i = pi1$ and the RunTT $\underline{\text{case}}$ expression is built as follows:

$$\mathcal{I}(e_1 \ldots e_i, \begin{Bmatrix} (\mathsf{c}_1\ \vec{a}_1)\ p_{12} \ldots p_{1i} \rightsquigarrow x_1 \\ \ldots \\ (\mathsf{c}_n\ \vec{a}_n)\ p_{n2} \ldots p_{ni} \rightsquigarrow x_n \end{Bmatrix}) \Longrightarrow \begin{array}{l} \underline{\text{case}}\ e_1\ \underline{\text{of}} \\ \quad (\mathsf{c}_1\ \vec{a}_1) \rightsquigarrow x_1 \\ \quad \ldots \\ \quad (\mathsf{c}_n\ \vec{a}_n) \rightsquigarrow x_n \end{array}$$

Case 3: No pair in $p_{11} \dots p_{n1}$ is headed by disjoint constructors

In this case, no distinction can be made on this argument, so we move on. If a term is presupposed, this means we don't even examine it because we already know what it is; examining it in the compiled code would break our specification of MATCH. The RunTT expression is built as follows:

$$\mathcal{I}(e_1 \dots e_m, \begin{Bmatrix} p_{11} \dots p_{1i} & \rightsquigarrow & x_1 \\ \dots & & \\ p_{n1} \dots p_{ni} & \rightsquigarrow & x_i \end{Bmatrix}) \Longrightarrow \mathcal{I}(e_2 \dots e_n, \begin{Bmatrix} p_{12} \dots p_{1i} & \rightsquigarrow & x_1 \\ \dots & & \\ p_{n2} \dots p_{ni} & \rightsquigarrow & x_i \end{Bmatrix})$$

In practice, the optimisation of reordering the $\vec{e}$ in descending order of the number of disjoint constructor patterns will ensure that this case never applies.

Case 4: Two or more of $p_{11} \dots p_{n1}$ are headed by disjoint constructors

This is the most complex case, and is a generalisation of case 2. Here, some ι-schemes can be eliminated, but no definite choice can be made. We make recursive calls to $\mathcal{I}$, leaving out the schemes which cannot match. We take P to be the smallest set of patterns such that $p_{i1} \in P$ if $p_{i1} = \mathsf{c}\langle\vec{e}\rangle$ for some c and $\vec{e}$.

Then the RunTT case expression is built by:

$$\mathcal{I}(e_1 \dots e_i, \begin{Bmatrix} p_{11} \dots p_{1i} & \rightsquigarrow & x_1 \\ \dots & & \\ p_{n1} \dots p_{ni} & \rightsquigarrow & x_n \end{Bmatrix}) \Longrightarrow$$

$$\underline{\text{case}}\ e_1\ \underline{\text{of}}$$

$$p \rightsquigarrow \mathcal{I}(e_2 \dots e_n, \begin{Bmatrix} p_{k2} \dots p_{ki} & \rightsquigarrow & x_1 \\ \dots & & \end{Bmatrix})$$

$$[\text{where } \forall p \in P,\ \forall k.\ p_{k1} \notin P \text{ or } (p_{k1} = \mathsf{c}\langle\vec{e}\rangle \text{ and } p = \mathsf{c}\langle\vec{e}'\rangle)]$$

$$\dots$$

$$\_ \rightsquigarrow \mathcal{I}(e_2 \dots e_n, \begin{Bmatrix} p_{k2} \dots p_{ki} & \rightsquigarrow & x_1 \\ \dots & & \end{Bmatrix})$$

$$[\text{where } \forall k.p_{k1} \notin P]$$

That is to say, if e_1 matches a pattern p, we can rule out the cases where the pattern for e_1 is headed by a different constructor, but we cannot rule out the cases where the pattern for e_1 is a variable.

If there is only one pattern variable e_1 left to consider, all patterns must be disjoint, or an error has occurred. If a family is detaggable, it is on the understanding that case distinction can be made on the indices. Otherwise, case distinction can always be made on the target.

There is a question remaining of how to compile a rule with no ι-schemes, such as with the elimination rule for the empty type:

$$\underline{\text{data}}\ \ \frac{}{\mathsf{False} : \star}\ \ \underline{\text{where}}$$

$\mathsf{False}\text{-}\mathbf{Elim} : \forall x{:}\mathsf{False}.\, \forall P{:}\mathsf{False} \to \star.\, P\ x$

This type has no constructors, and hence the elimination rule has no ι-schemes. Of course, in practice, this rule can never be executed, since there is no canonical form of False on which to apply it. I will postpone discussion of an effective way to handle this problem until section 6.2.4 — for the moment, it suffices to say that False-**Elim** can not reduce.

Example — Vect

Recall the detagged Vect elimination rule:

$$\begin{array}{l}\mathsf{Vect}\text{-}\mathbf{Elim}\ A\ \ 0\ \ \ \ (\{\epsilon\}\ \{A\})\ \ \ \ P\ m_\epsilon\ m_{::} \rightsquigarrow m_\epsilon \\ \mathsf{Vect}\text{-}\mathbf{Elim}\ A\ (\mathsf{s}\ k)\ (\{::\}\ \{A\}\ \{k\}\ a\ v)\ P\ m_\epsilon\ m_{::} \rightsquigarrow m_{::}\ k\ a\ v\ (\mathsf{Vect}\text{-}\mathbf{Elim}\ A\ k\ v\ P\ m_\epsilon\ m_{::})\end{array}$$

The first step in compiling this to pattern matching form is to give each argument a unique name. For readability, let us take A, n, x, P, m_ϵ and $m_{::}$ (rather than $e_1 \ldots e_6$). Then for each ι-scheme, we compute the terms required to extract pattern variables from the left hand side with PROJECT. This is trivial in the ϵ case. For ::, applying PROJECT to each argument yields:

$$\begin{array}{l}\textsc{project}(A, A) \implies [(A, A)] \\ \textsc{project}(n, (\mathsf{s}\ k)) \implies [(k, n!0)] \\ \textsc{project}(x, (\{::\}\ \{A\}\ \{k\}\ a\ v)) \implies [(a, x!0), (v, x!1)] \\ \textsc{project}(P, P) \implies [(P, P)] \\ \textsc{project}(m_\epsilon, m_\epsilon) \implies [(m_\epsilon, m_\epsilon)] \\ \textsc{project}(m_{::}, m_{::}) \implies [(m_{::}, m_{::})]\end{array}$$

Then we get a RunTT term for Vect-**Elim** by applying $\mathcal{I}$ as follows:

$$\mathsf{Vect}\text{-}\mathbf{Elim} \mapsto \lambda A;\ n;\ x;\ P;\ m_\epsilon;\ m_{::}.\ \mathcal{I}(n, A, x, P, m_\epsilon, m_{::}, \left\{\begin{array}{l}0\ A\ (\{\epsilon\}\ \{A\})\ P\ m_\epsilon\ m_{::} \rightsquigarrow m_\epsilon \\ (\mathsf{s}\ k)\ A\ (\{::\}\ \{A\}\ \{k\}\ a\ v)\ P\ m_\epsilon\ m_{::} \rightsquigarrow \\ \quad m_{::}\ (n!0)\ (x!0)\ (x!1) \\ \quad\quad (\mathsf{Vect}\text{-}\mathbf{Elim}\ A\ (n!0)\ (x!1)\ P\ m_\epsilon\ m_{::})\end{array}\right\})$$

which reduces to the following <u>case</u> expression:

$$\begin{array}{l}\mathsf{Vect}\text{-}\mathbf{Elim} \mapsto \lambda A;\ n;\ x;\ P;\ m_\epsilon;\ m_{::}. \\ \quad \underline{\text{case}}\ n\ \underline{\text{of}} \\ \quad\quad 0 \rightsquigarrow m_\epsilon \\ \quad\quad (\mathsf{s}\langle k\rangle) \rightsquigarrow m_{::}\ (n!0)\ (x!0)\ (x!1) \\ \quad\quad\quad\quad (\mathsf{Vect}\text{-}\mathbf{Elim}\ A\ (n!0)\ (x!1)\ P\ m_\epsilon\ m_{::})\end{array}$$

A curious effect of this compilation algorithm as that, although the <u>case</u> expression binds the k argument of s, it is not used in the right hand side; rather, $n!0$ is used to get k. This happens because we do not know before $\mathcal{I}$ compilation which names can be bound by <u>case</u>

expressions and which we need to get by argument projection. It is a simple transformation to reinstate names which *are* bound by case afterwards, by reversing the mapping generated by the PROJECT operation. In this case, we have $\mathsf{s}\,k$, and $\text{PROJECT}(n, (\mathsf{s}\,k)) \Longrightarrow [(k, (n!0))]$; reversing the mapping from k to $(n!0)$ gives:

$$\begin{array}{lll}\textsf{Vect-Elim} \mapsto & \lambda A;\ n;\ x;\ P;\ m_\epsilon;\ m_{::}. \\ & \underline{\text{case}}\ n\ \underline{\text{of}} \\ & \quad 0 \rightsquigarrow m_\epsilon \\ & \quad (\mathsf{s}\langle k\rangle) \rightsquigarrow m_{::}\ k\ (x!0)\ (x!1) \\ & \qquad\qquad (\textbf{Vect-Elim}\ A\ k\ (x!1)\ P\ m_\epsilon\ m_{::})\end{array}$$

However, we do not do this immediately after compilation of the patterns — to do so is a premature optimisation[1]. Instead we wait until other optimisations have been applied to the RunTT term.

Example — between

A more complex example results from the between relation over three numbers m, n, p, which expresses the property that $m \leq n \leq p$:

$$\underline{\text{data}}\quad \frac{m, n, p\ :\ \mathbb{N}}{\mathsf{between}\ m\ n\ p\ :\ \star}$$

$$\underline{\text{where}}\quad \frac{}{\mathsf{bO}\ :\ \mathsf{between}\ 0\ 0\ 0} \qquad \frac{b\ :\ \mathsf{between}\ 0\ 0\ m}{\mathsf{bOOs}\ b\ :\ \mathsf{between}\ 0\ 0\ (\mathsf{s}\ m)}$$

$$\frac{b\ :\ \mathsf{between}\ 0\ m\ n}{\mathsf{b0ss}\ b\ :\ \mathsf{between}\ 0\ (\mathsf{s}\ m)\ (\mathsf{s}\ n)} \qquad \frac{b\ :\ \mathsf{between}\ m\ n\ p}{\mathsf{bsss}\ b\ :\ \mathsf{between}\ (\mathsf{s}\ m)\ (\mathsf{s}\ n)\ (\mathsf{s}\ p)}$$

To show that this relation really does represent the property we want, we can prove the lemma $m \leq n \rightarrow n \leq p \rightarrow \mathsf{between}\, m\, n\, p$. This can be proved by induction over the variables m, n and p, then inversion over the relations. The ι-schemes for between are shown in figure 4.39.

$$\begin{array}{lllll}\textbf{between-Elim} & 0 & 0 & 0 & \mathsf{bO}\ \ P\ m_{\mathsf{bO}}\ m_{\mathsf{bOOs}}\ m_{\mathsf{b0ss}}\ m_{\mathsf{bsss}} \rightsquigarrow m_{\mathsf{bO}} \\ \textbf{between-Elim} & 0 & 0 & (\mathsf{s}\ m) & (\mathsf{bOOs}\ m\ b)\ \ P\ m_{\mathsf{bO}}\ m_{\mathsf{bOOs}}\ m_{\mathsf{b0ss}}\ m_{\mathsf{bsss}} \\ & & \rightsquigarrow m_{\mathsf{bOOs}}\ m\ b\ (\textbf{between-Elim}\ 0\ 0\ m\ b\ P\ m_{\mathsf{bO}}\ m_{\mathsf{bOOs}}\ m_{\mathsf{b0ss}}\ m_{\mathsf{bsss}}) \\ \textbf{between-Elim} & 0 & (\mathsf{s}\ m) & (\mathsf{s}\ n) & (\mathsf{b0ss}\ m\ n\ b)\ \ P\ m_{\mathsf{bO}}\ m_{\mathsf{bOOs}}\ m_{\mathsf{b0ss}}\ m_{\mathsf{bsss}} \\ & & \rightsquigarrow m_{\mathsf{b0ss}}\ m\ n\ b\ (\textbf{between-Elim}\ 0\ m\ n\ b\ P\ m_{\mathsf{bO}}\ m_{\mathsf{bOOs}}\ m_{\mathsf{b0ss}}\ m_{\mathsf{bsss}}) \\ \textbf{between-Elim} & (\mathsf{s}\ m) & (\mathsf{s}\ n) & (\mathsf{s}\ p) & (\mathsf{bsss}\ m\ n\ p\ b)\ \ P\ m_{\mathsf{bO}}\ m_{\mathsf{bOOs}}\ m_{\mathsf{b0ss}}\ m_{\mathsf{bsss}} \\ & & \rightsquigarrow m_{\mathsf{bsss}}\ m\ n\ p\ b\ (\textbf{between-Elim}\ m\ n\ p\ b\ P\ m_{\mathsf{bO}}\ m_{\mathsf{bOOs}}\ m_{\mathsf{b0ss}}\ m_{\mathsf{bsss}})\end{array}$$

Figure 4.39: ι-schemes for **between-Elim**

between is concretely detaggable, since

$\forall\, i \neq j.\ \exists k.\ \text{DISJOINT}(p_{ik}, p_{jk}) \Longrightarrow \mathsf{true}$

[1]The reason why projection is preferred is explained in section 6.2.4.

That is, it is possible to establish which constructor applies purely by examining the indices. In addition, between is concretely collapsible. The implementation of the (run-time) elimination rule is given by the marked-up ι-schemes in figure 4.40.

$$
\begin{array}{l}
\textsf{between-}\mathbf{Elim}\ \ 0\ \ 0\ \ 0\ \ \{\textsf{b0}\}\ \ P\ m_{\textsf{b0}}\ m_{\textsf{b00s}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}} \rightsquigarrow m_{\textsf{b0}} \\
\textsf{between-}\mathbf{Elim}\ \ 0\ \ 0\ \ (\textsf{s}\ m)\ \ \{\textsf{b00s}\ m\ b\}\ \ P\ m_{\textsf{b0}}\ m_{\textsf{b00s}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}} \\
\quad \rightsquigarrow m_{\textsf{b00s}}\ m\ (\{b\})\ (\textsf{between-}\mathbf{Elim}\ 0\ 0\ m\ \{b\}\ P\ m_{\textsf{b0}}\ m_{\textsf{b00s}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}}) \\
\textsf{between-}\mathbf{Elim}\ \ 0\ \ (\textsf{s}\ m)\ \ (\textsf{s}\ n)\ \ \{\textsf{b0ss}\ m\ n\ b\}\ \ P\ m_{\textsf{b0}}\ m_{\textsf{b00s}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}} \\
\quad \rightsquigarrow m_{\textsf{b0ss}}\ m\ n\ (\{b\})\ (\textsf{between-}\mathbf{Elim}\ 0\ m\ n\ \{b\}\ P\ m_{\textsf{b0}}\ m_{\textsf{b00s}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}}) \\
\textsf{between-}\mathbf{Elim}\ (\textsf{s}\ m)\ (\textsf{s}\ n)\ \ (\textsf{s}\ p)\ \ \{\textsf{bsss}\ m\ n\ p\ b\}\ \ P\ m_{\textsf{b0}}\ m_{\textsf{b00s}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}} \\
\quad \rightsquigarrow m_{\textsf{bsss}}\ m\ n\ p\ (\{b\})\ (\textsf{between-}\mathbf{Elim}\ m\ n\ p\ \{b\}\ P\ m_{\textsf{b0}}\ m_{\textsf{b00s}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}})
\end{array}
$$

Figure 4.40: ι-schemes for between-**Elim** after collapsing

Applying the $\mathcal{I}$ compilation scheme, which repeatedly applies case 4, yields the super-combinator definition shown in figure 4.41. Note again that since between is concretely collapsible, instances passed to the methods are replaced with the trivial canonical empty tuple, $\langle\rangle$.

$$
\begin{array}{l}
\textsf{between-}\mathbf{Elim} \mapsto \lambda m;\ n;\ p;\ P;\ m_{\textsf{b0}};\ m_{\textsf{b00s}};\ m_{\textsf{b0ss}};\ m_{\textsf{bsss}}. \\
\quad \underline{\textsf{case}}\ m\ \underline{\textsf{of}} \\
\quad\quad 0 \rightsquigarrow \underline{\textsf{case}}\ n\ \underline{\textsf{of}} \\
\quad\quad\quad\quad 0 \rightsquigarrow \underline{\textsf{case}}\ p\ \underline{\textsf{of}} \\
\quad\quad\quad\quad\quad\quad 0 \rightsquigarrow m_{\textsf{b0}} \\
\quad\quad\quad\quad\quad\quad \textsf{s}\langle k\rangle \rightsquigarrow m_{\textsf{b00s}}\ k\ \langle\rangle\ \ldots \\
\quad\quad\quad\quad \textsf{s}\langle k\rangle \rightsquigarrow m_{\textsf{b0ss}}\ k\ (p!0)\ \langle\rangle\ \ldots \\
\quad\quad \textsf{s}\langle k\rangle \rightsquigarrow m_{\textsf{bsss}}\ k\ (n!0)\ (p!0)\ \langle\rangle\ \ldots
\end{array}
$$

Figure 4.41: Compiled ι-schemes for between-**Elim**

4.4.3 Extensions to the G-machine

The extensions made to RunTT in the previous section necessitate some alterations to the G-machine. There are two principal changes:

- Implementation of constructor argument projection ($e!i$) is possible via the CASEJUMP instruction, but this is inefficient since the purpose is to avoid unnecessary case analysis at run-time. We ought to implement this operation more efficiently.
- Case analysis now exists only to establish which ι-scheme to execute, not to project out arguments. We can therefore imagine a simpler alternative to CASEJUMP. In addition, since case is now not necessarily on the target of an elimination rule, some cases may be impossible. RunTT includes a "match anything" case alternative, so this also needs to be handled.

New Compilation Schemes

To handle these additions to the language, we need to make additions to the $\mathcal{E}[\![\cdot]\!]$ and $\mathcal{C}[\![\cdot]\!]$ compilation schemes, and to the heap representation of the G-machine.

There are some alternative approaches to dealing with argument projection in the G-machine. The effect of projecting the nth argument from a graph G could be to either push a new graph node onto the stack for later evaluation, PROJ n G (the lazy approach) or to push the graph pointed to by the nth argument of a G in canonical form onto the stack (the eager approach). I choose the eager approach because, in general, the projection will not be made more than twice in an ι-scheme (once as an argument to the method, and once in the recursive call). The overhead of constructing the graph node is too much for the laziness to compensate for this; and even so, an optimisation which lifts out common subexpressions can ensure that the projection is evaluated only once.

Construction of untagged structures is relatively straightforward. Corresponding to CON t xs, there is a new graph node type:

- TUP xs, where xs is a list of known length, which represents a detagged constructor as a tuple of the arguments xs.

Two new instructions are added to the G-machine. PROJ i projects the ith argument out of the (canonical) object on top of the stack, replacing the top stack value. MKTUP i constructs an untagged constructor from the top i stack elements. The G-machine state transition rules for these instructions are given in figure 4.42.

$$\langle \text{MKTUP } i;\ c, n_0 \ldots n_{i-1}.S, G, E, D\rangle \implies \langle c, n'.S, G[n' = \text{TUP } (n_0 \ldots n_{i-1})], E, D\rangle$$
$$\langle \text{PROJ } i;\ c, n_0.S, G[n_0 = \text{CON } t\ (x_0 \ldots x_i \ldots x_a), E, D\rangle \implies \langle c, x_i.S, G, E, D\rangle$$
(where a is the number of arguments to the constructor)
$$\langle \text{PROJ } i;\ c, n_0.S, G[n_0 = \text{TUP } (x_0 \ldots x_i \ldots x_a), E, D\rangle \implies \langle c, x_i.S, G, E, D\rangle$$
(where a is the number of arguments in the tuple)

Figure 4.42: State transitions for MKTUP and PROJ

The additions to the $\mathcal{E}[\![\cdot]\!]$ compilation scheme are given in figure 4.43. Firstly, evaluating an argument projection $e!i$ involves evaluation of e (to get it into a canonical form) then projection of the ith argument of e with PROJ. We also account for evaluation of untagged tuples and case expressions with defaults.

The CASEJUMP instruction has slightly different behaviour to account for the changes to RunTT. It examines the target and jumps to the appropriate label, as before, but there is also a default case to account for the "match anything" pattern. There is still no error case; typechecking accounts for the fact that this can't happen.

Figure 4.44 gives the additions to the $\mathcal{C}[\![\cdot]\!]$ compilation scheme. For argument projection, note that the projection itself is evaluated eagerly; e is compiled by the $\mathcal{E}[\![\cdot]\!]$ scheme to ensure

$$
\begin{array}{l}
\mathcal{E}[\![e!i]\!]\ r\ n \implies \mathcal{E}[\![e]\!]\ r\ n; \mathsf{PROJ}\ i;\ \mathsf{EVAL} \\
\mathcal{E}[\![\langle e_1, e_2, \ldots, e_i\rangle]\!]\ r\ n \implies \mathcal{C}[\![e_1]\!]\ r\ n;\ \mathcal{C}[\![e_2]\!]\ r\ (n+1); \ldots; \\
\qquad\qquad\qquad\qquad \mathcal{C}[\![e_i]\!]\ r\ (n+i-1);\ \mathsf{MKTUP}\ i \\
\mathcal{E}[\![\underline{\mathsf{case}}\ e\ \underline{\mathsf{of}}\ \mathsf{c}_1\langle\vec{a}_1\rangle \rightsquigarrow e_1\ \ldots\ \mathsf{c}_n\langle\vec{a}_n\rangle \rightsquigarrow e_n\ ,\ \_ \rightsquigarrow e_{\mathsf{def}}]\!]\ r\ n \implies \\
\quad \mathcal{E}[\![e]\!]\ r\ n;\ \mathsf{CASEJUMP}\ (\mathsf{c}_1, l_1)\ (\mathsf{c}_2, l_2) \ldots (\mathsf{c}_n, l_n)\ l_{\mathsf{def}}; \\
\qquad \mathsf{LABEL}\ l_1;\ \mathsf{SPLIT}\ n_1;\ \mathcal{E}[\![e_1]\!]\ d_1\ n+n_1;\ \mathsf{MOVE}\ n_1+1; \mathsf{DISCARD}\ n_1+1;\ \mathsf{JUMP}\ l \\
\qquad \ldots \\
\qquad \mathsf{LABEL}\ l_{\mathsf{def}};\ \mathcal{E}[\![e_{\mathsf{def}}]\!]\ r\ n; \\
\quad \mathsf{LABEL}\ l \\
\qquad \text{where } d_n(a_{ij}) \implies n+j \\
\qquad\qquad\ d_n(x) \implies r(x) \\
\qquad\qquad\ n_k = \textsc{length}(\vec{a}_k)
\end{array}
$$

Figure 4.43: Extension to the $\mathcal{E}[\![\cdot]\!]$ scheme

that the object of the projection is in canonical form. This scheme also includes construction of untagged tuples.

$$
\begin{array}{l}
\mathcal{C}[\![e!i]\!]\ r\ n \implies \mathcal{E}[\![e]\!]\ r\ n; \mathsf{PROJ}\ i \\
\mathcal{C}[\![\langle e_1, e_2, \ldots, e_i\rangle]\!]\ r\ n \implies \mathcal{C}[\![e_1]\!]\ r\ n;\ \mathcal{C}[\![e_2]\!]\ r\ (n+1); \ldots; \\
\qquad\qquad\qquad\qquad \mathcal{C}[\![e_i]\!]\ r\ (n+i-1);\ \mathsf{MKTUP}\ i
\end{array}
$$

Figure 4.44: Extensions to the $\mathcal{C}[\![\cdot]\!]$ scheme

4.5 Examples

We can see the effect that the transformations described in this chapter have on programs by running the programs on a G-machine both with and without the transformations applied. There are several quantities which we may choose to measure, such as the number of instructions executed, memory allocations, memory usage, processor cycles used or time taken. The quantities we choose to measure for each run, naïve and optimised, are the following:

- The number of G-machine instructions executed.
- The number of thunks (suspended computations) created.
- The number of memory accesses (instructions which analyse a heap cell).
- The number of cells allocated for data storage.

We choose number of instructions executed above processor cycles or time taken because of the nature of the implementation of the G-machine, and the size of the examples; since the

examples are small and run quickly, we can get a more precise measure of the time taken this way. We choose thunks and cell allocations to give an idea of how much storage is required, which gives a picture of how well the optimisations perform as storage optimisations.

The only optimisations applied are those presented in this chapter; there is, for example, no strictness analysis or inlining or any form of tail recursion transformation. This is to see how the forcing, detagging and collapsing optimisations work independently of any other analysis. Some of the results we will see may seem surprising, particular with regard to the number of instructions executed. This is largely due to the inefficiency of number representation in TT, using an unary representation of $\mathbb{N}$ — this problem will be addressed in Chapter 5. The extra layer of abstraction imposed by elimination rules, particularly arguments unused at run-time such as the motive, also adds significant overheads which will be addressed in Chapter 6. There is also an overhead in outputting results (which we do by converting the result of each program to a string), a trivial implementation detail not addressed in this thesis.

4.5.1 The Finite Sets

The finite sets, indexed over a natural number n, are a family of types with n elements. Effectively, they are a representation of bounded numbers and are declared as follows:

$$\underline{\text{data}} \quad \frac{n \ : \ \mathbb{N}}{\mathsf{Fin}\ n \ : \ \star}$$

$$\underline{\text{where}} \quad \frac{}{\mathsf{f0} \ : \ \mathsf{Fin}\ (\mathsf{s}\ n)} \qquad \frac{i \ : \ \mathsf{Fin}\ n}{\mathsf{fs}\ i \ : \ \mathsf{Fin}\ (\mathsf{s}\ n)}$$

The forcing optimisation elides the indices from the elaborated constructors:

$$[\![\mathsf{f0}]\!] \Longrightarrow \lambda n.\, \mathsf{f0}\ \{n\}$$
$$[\![\mathsf{fs}]\!] \Longrightarrow \lambda n; i.\, \mathsf{fs}\ \{n\}\ i$$

After stripping the forceable arguments, the shape of the resulting type matches that of $\mathbb{N}$ — that is, the base constructor takes no arguments and the step constructor takes a single recursive argument. This is to be expected; Fin and $\mathbb{N}$ represent the same thing (natural numbers), but Fin also maintains an invariant representing an upper bound on the number which is not part of the data structure.

An expression, **lookup**(fs(fsf0))((s(s0))::(s0)::0::ϵ), was evaluated and printed before and after applying the transformations. The results of evaluating and printing this expression are shown in figure 4.45.

4.5.2 Comparison of Natural Numbers

The Compare family from [MM04b] represents the result of comparing two numbers, storing which is the greater and by how much:

Program	Version	Instructions	Thunks	Memory Accesses	Cells
Vector lookup	Naïve	549	300	166	39
	Optimised	537	300	166	27
	Change	**-2.23%**	-	-	**-30.76%**

Figure 4.45: Run-time costs of Fin and Vect

$$\underline{\text{data}} \quad \frac{m : \mathbb{N} \quad n : \mathbb{N}}{\mathsf{Compare}\ m\ n\ :\ \star}$$

$$\underline{\text{where}} \quad \frac{y\ :\ \mathbb{N}}{\mathsf{lt}\ y\ :\ \mathsf{Compare}\ x\ (\mathbf{plus}\ x\ (\mathsf{s}\ y))}$$

$$\frac{}{\mathsf{eq}\ :\ \mathsf{Compare}\ x\ x}$$

$$\frac{x\ :\ \mathbb{N}}{\mathsf{gt}\ x\ :\ \mathsf{Compare}\ (\mathbf{plus}\ y\ (\mathsf{s}\ x))\ y}$$

Compare is an example of a family which is collapsible, but not concretely collapsible. Clearly there is only one possible element of Compare m n for each m and n, and given this element we can extract their difference in constant time. If we were to collapse Compare we would replace this simple inspection by the recomputation of the difference each time the same value was used. We restrict concretely collapsible families to those where the recomputation of values is cheap.

Nonetheless, by forcing, Compare need only store which index is larger and by how much:

$[\![\mathsf{lt}]\!] \Longrightarrow \lambda x; y.\ \mathsf{lt}\ \{x\}\ y$
$[\![\mathsf{eq}]\!] \Longrightarrow \lambda x.\ \mathsf{eq}\ \{x\}$
$[\![\mathsf{gt}]\!] \Longrightarrow \lambda x; y.\ \mathsf{gt}\ x\ \{y\}$

The results of applying this optimisation to a program which computes the **gcd** of two Ns by using view Compare are shown in figure 4.46.

Program	Version	Instructions	Thunks	Memory Accesses	Cells
gcd 6 3	Naïve	37896	18864	12293	2749
	Optimised	37486	18636	12293	2567
	Change	**-1.08%**	**-1.20%**	-	**-6.62%**

Figure 4.46: Run-time costs of **gcd**, written by view Compare

For reference, the **gcd** program is presented in figure 4.47. A view **plusrec** is defined to give recursion on numbers which are shown to be smaller by their presence as an argument to **plus**. The "?" used as an argument to plusRec indicates that the typechecker is expected to work out what this argument should be — it is often the case in writing dependently typed programs (particularly those which express proofs) that the typechecker can work out what an argument should be, purely from its type.

```
data    n : ℕ              where   R : ∀a, b:ℕ. (n = s(plus a b))  →  PlusRec b
        ──────────                 ─────────────────────────────────────────────
        PlusRec n                            plusRec R : PlusRec n

         n : ℕ    R : ∀a, b:ℕ. ∀eq:n = s (plus a b). PlusRec b
                  a, b : ℕ    eq′ : s n = s (plus a n)
let     ───────────────────────────────────────────────────────
                  plusrecs n R a b eq′ : PlusRec b
  plusrecs        n         R     a  b eq′  ⇐ case eq′
   plusrecs   (plus a b)    R     a  b eq′  ⇐ case a
    plusrecs      b         R     0  b refl ↦ plusRec R
    plusrecs (s (plus a b)) R (s a)  b refl ↦ R a b refl

let            n : ℕ
        ──────────────────
        plusrec n : PlusRec n
  plusrec   n    ⇐ elim n
   plusrec  0    ↦ plusRec ?
   plusrec (s k) ↦ plusRec (plusrecs k (λa, b:ℕ. λeq:k = s (plus a b). ?))

let      m, n : ℕ
        ──────────
        gcd m n : ℕ
  gcd          m               n            ⇐ view plusrec m ⇐ view plusrec n
                                              ⇐ compare m n
   gcd         x         (plus x (sy))      ⇐ case x
    gcd        0              (s y)                              ↦ s y
    gcd      (s x)     (plus (s x) (s y))                        ↦ gcd (s x) (s y)
   gcd         x               x                                 ↦ x
   gcd   (plus y (s x))        y            ⇐ case y
    gcd      (s x)             0                                 ↦ s x
    gcd (plus (s y) (s x))   (s y)                               ↦ gcd (s y) (s x)
```

Figure 4.47: Computing the greatest common divisor of two integers

4.5.3 Domain Predicates

In [BC03], Bove and Capretta use domain predicates to prove termination of general recursive functions, an example of which we have already seen in section 2.3.4. Domain predicates are inductive families which express the termination criteria for each possible input to a function.

The **quicksort** function terminates on the input nil, and terminates on the input cons$x\,xs$ if it terminates on the inputs **filter** $(<\ x)\ xs$ and **filter** $(\geq\ x)\ xs$. This is expressed by the qsAcc predicate; this gives the termination criteria for each input cons $x\ xs$ and nil:

```
data     l : List ℕ
        ────────────
        qsAcc l : ⋆

where   ─────────────────
        qsNil : qsAcc nil

        qsl : qsAcc (filter (< x) xs)    qsr : qsAcc (filter (≥ x) xs)
        ───────────────────────────────────────────────────────────────
                     qsCons qsl qsr : qsAcc (cons x xs)
```

The main part of **quicksort** is defined by induction over this predicate; the details of the termination proof lie in converting a list to an instance of the domain predicate. A naïve implementation of this method would need to store the predicate, since **quicksort** is, in their method, implemented by induction over it. However, qsAcc is concretely collapsible, hence it need not be stored at run-time:

$$\begin{aligned} [\![\mathsf{qsNil}]\!] &\implies \{\mathsf{qsNil}\} \\ [\![\mathsf{qsCons}]\!] &\implies \lambda x; xs; qsl; qsr.\ \{\mathsf{qsCons}\ x\ xs\ qsl\ qsr\} \end{aligned}$$

In fact the optimisation replaces computation over qsAcc by computation over its indices, restoring the intended operational semantics of the original program!

We should expect Bove-Capretta domain predicates to be collapsible because they are constructed mechanically from pattern matching programs in the first place. Further, domain predicates generated to show termination of a function defined by non-overlapping patterns are *concretely* collapsible.

Take a non-structurally recursive $\mathbf{f}$, defined by pattern matching:

$$\underline{\text{let}} \quad \frac{x\ :\ \mathsf{D}\ \vec{s}}{\mathbf{f}\ x\ :\ T} \qquad \begin{aligned} \mathbf{f}\ \ \vec{p}_1 &\mapsto e_1 \\ &\cdots \\ \mathbf{f}\ \ \vec{p}_n &\mapsto e_n \end{aligned}$$

where e_i may include any number of arbitrary recursive calls to $\mathbf{f}\ \vec{x}$, for arbitrary terms $\vec{x}$ where all variables in $\vec{x}$ are retrievable from $\vec{p}$ by pattern matching. For the predicate to be concretely collapsible, the patterns $\vec{p}_i$ must be non-overlapping and complete, i.e.:

$$\forall\, i \neq j.\ \exists k.\ \textsc{disjoint}(p_{ik}, p_{jk}) \implies \mathsf{true}$$

This is the same condition which ensures that the elimination rule for detaggable families is respectful and well-defined.

For each of the cases $\vec{p}_i$, we have that $\mathbf{f}$ terminates if all recursive calls in e_i terminate. The domain predicate generated for $\mathbf{f}$ is of this form:

$$\underline{\text{data}} \quad \frac{x\ :\ \mathsf{D}\ \vec{s}}{\mathsf{fAcc}\ x\ :\ \star}$$

$$\underline{\text{where}} \quad \frac{\vec{r}_1\ :\ \mathsf{fAcc}\ \vec{x}_1 \ldots \vec{r}_m\ :\ \mathsf{fAcc}\ \vec{x}_m}{\mathsf{fp}_1\ \vec{r}\ :\ \mathsf{fAcc}\ \vec{p}_1} \quad \cdots \quad \frac{\vec{r}_1\ :\ \mathsf{fAcc}\ \vec{x}_1 \ldots \vec{r}_l\ :\ \mathsf{fAcc}\ \vec{x}_l}{\mathsf{fp}_n\ \vec{r}\ :\ \mathsf{fAcc}\ \vec{p}_n}$$

The fAcc predicate gives termination conditions for each of the cases of $\mathbf{f}$. A case i, with patterns $\vec{p}_i$, terminates if every recursive call made by that case terminates. The $\vec{x}_j$ indices of the nested fAccs are the arguments to the recursive calls — any variables in these terms are forceable arguments, since they are retrieved from $\vec{p}_i$ by pattern matching.

Recall that a family is concretely collapsible if it is concretely detaggable and if, for each constructor, all of the non-recursive arguments are forceable. Now observe that this is always true for any fAcc:

- It is *detaggable*, because the condition that the patterns $\vec{p}_i$ are non-overlapping in the definition of **f** is the same as the condition for the indices of a detaggable family. Since the $\vec{p}_i$ are the indices of fAcc, it is a detaggable family.
- All non-recursive arguments for each constructor (the variables appearing in $\vec{x}_j$) are forceable, since they are retrieved from $\vec{p}_i$ by pattern matching.

Figure 4.48 shows how the collapsing transformation affects the run-time costs of **quicksort**.

Program	Version	Instructions	Thunks	Memory Accesses	Cells
Quicksort	Naïve	175649	86600	55221	17268
	Optimised	171264	85586	55189	13900
	Change	**-2.50%**	**-1.17%**	**-0.05%**	**-19.50%**

Figure 4.48: Run-time costs of quicksort

4.5.4 Non-repeating Lists

We can use the List family to build a representation of lists in which duplicate values are not permitted. To do this, we build a new datatype indexed over lists and including an additional proof which verifies that in each non-empty list, the head is not an element of the tail. We define the **elem** function, which tests whether a value is an element of a list as follows; the f argument is instantiated with a function to test equality between two values of the parameter type. This is similar to Haskell's type class system, in which, internally, a dictionary would be passed to **elem** containing the appropriate instance of the equality function.

$$\underline{\text{let}} \quad \frac{a : A \quad l : \mathsf{List}\ A \quad f : A \to A \to \mathsf{Bool}}{\mathbf{elem}\ a\ l\ f : \mathsf{Bool}}$$

$$\begin{array}{lll}
\mathbf{elem}\ a & l & f \Leftarrow \underline{\text{elim}}\ l \\
\mathbf{elem}\ a & \mathsf{nil} & f \mapsto \mathsf{false} \\
\mathbf{elem}\ a\ (\mathsf{cons}\ x\ xs) & f & \left|\begin{array}{ll} f\ a\ x & \\ \quad p & \Leftarrow \underline{\text{case}}\ p \\ \quad \mathsf{true} & \mapsto \mathsf{true} \\ \quad \mathsf{false} & \mapsto \mathbf{elem}\ a\ xs\ s \end{array}\right.
\end{array}$$

The So family is a predicate which says that its argument is always true — it is indexed over Bool but only an element of So true can be instantiated:

$$\underline{\text{data}} \quad \frac{b : \mathsf{Bool}}{\mathsf{So}\ b : \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{oh} : \mathsf{So}\ \mathsf{true}}$$

So is trivially concretely collapsible; there is only one possible value. Now we can use the So family to prove that a value added to a list is not already an element of that list. The DList family represents lists with no duplicate elements, and is indexed over the list which

holds the actual data, as well as an equality function used for testing for presence of a value in the list:

$$\underline{\text{data}} \quad \frac{f : A \to A \to \mathsf{Bool} \quad l : \mathsf{List}\ A}{\mathsf{DList}\ A\ f\ l : \star}$$

$$\underline{\text{where}} \quad \frac{}{\emptyset : \mathsf{DList}\ A\ f\ \mathsf{nil}}$$

$$\frac{x : A \quad s : \mathsf{DList}\ A\ f\ xs \quad p : \mathsf{So}\ (\mathbf{not}\ (\mathbf{elem}\ x\ xs\ f))}{\mathsf{insert}\ x\ s\ p : \mathsf{DList}\ A\ f\ (\mathsf{cons}\ x\ xs)}$$

There are several advantages to indexing this structure over lists. Essentially, it *is* a list but with extra preconditions; indexing over lists means that we can still use List functions such as **elem** over DLists. Indexing over List also makes DList concretely collapsible, so the run-time representation is simply the underlying List coupled with the equality function. The disadvantage is that the user of this type has to maintain invariant properties of the underlying list — this is important in defining DList functions, but we would prefer to abstract details of these invariants away from users of the type. Rather than make the user use the DList type directly, we use a dependent pair to expose an interface:

$$\mathsf{DListTop}\ A\ f \ \mapsto\ \Sigma\ (\mathsf{List}\ A)\ (\mathsf{DList}\ A\ f)$$

The collapsing optimisation yields the following substitutions for constructors of DList:

$$[\![\emptyset]\!] \implies \lambda A; f; l.\ \{\emptyset\}\ \{A\}\ \{f\}\ \{l\}$$
$$[\![\mathsf{insert}]\!] \implies \lambda A; f; x; xs; s; p.\ \{\mathsf{insert}\}\ \{A\}\ \{f\}\ \{x\}\ \{xs\}\ \{s\}\ \{p\}$$

The run-time costs of a function over a DList are shown in figure 4.49. This function takes a DList of Ns and totals all numbers in that DList.

Program	Version	Instructions	Thunks	Memory Accesses	Cells
Totalling a DList	Naïve	69612	28218	30695	2494
	Optimised	66333	27278	29774	1622
	Change	**-4.71%**	**-3.33%**	**-3.00%**	**-34.96%**

Figure 4.49: Run-time costs of adding values in a DList

4.5.5 Simply Typed λ-calculus

We define the simply typed λ-calculus in a similar fashion to [MM04b], making extensive use of inductive families to specify invariants on the data structures. We begin with STy, an unindexed type representing simple monomorphic types with a base type and function spaces:

$$\underline{\text{data}} \quad \frac{}{\mathsf{STy} : \star} \quad \underline{\text{where}} \quad \frac{}{\iota : \mathsf{STy}} \quad \frac{s, t : \mathsf{STy}}{s \Rightarrow t : \mathsf{STy}}$$

We represent contexts by Vects of types, $\mathsf{Env} = \mathsf{Vect}\ \mathsf{STy}$. The explicit size allows us to give a safe de Bruijn representation of variables, themselves rendered by the the Fin family. Hence our the family Expr, represents non-checked but well-scoped terms.

$$\underline{\text{data}} \quad \frac{n : \mathbb{N}}{\mathsf{Expr}\ n : \star}$$

$$\underline{\text{where}} \quad \frac{i : \mathsf{Fin}\ n}{\mathsf{eVar}\ i : \mathsf{Expr}\ n} \qquad \frac{S : \mathsf{STy} \quad t : \mathsf{Expr\ s}\ n}{\mathsf{eLam}\ S\ t : \mathsf{Expr}\ n} \qquad \frac{f, s : \mathsf{Expr}\ n}{\mathsf{eApp}\ f\ s : \mathsf{Expr}\ n}$$

Clearly n is forceable for each of these constructors, by its appearance as a variable in the index of each constructor.

In order to give types to variables, we introduce the Var relation, representing membership of a context. $\mathsf{Var}\ G\ i\ T$ states that the ith member of the context G has type T, and is concretely collapsible.

$$\underline{\text{data}} \quad \frac{G : \mathsf{Env}\ n \quad i : \mathsf{Fin}\ n \quad T : \mathsf{STy}}{\mathsf{Var}\ G\ i\ T : \star}$$

$$\underline{\text{where}} \quad \frac{}{\mathsf{stop} : \mathsf{Var}\ (S::G)\ \mathsf{f0}\ S} \qquad \frac{v : \mathsf{Var}\ G\ i\ T}{\mathsf{pop}\ v : \mathsf{Var}\ (S::G)\ (\mathsf{fs}\ i)\ T}$$

Finally, we have a type of well typed terms. This is indexed over a context, the raw term it arises from and its type. This gives us a particularly safe representation — it is not possible to write a typechecker which gives rise to the wrong well typed term. This indexing also enables us to synchronise terms safely with value environments during evaluation in the style of Augustsson and Carlsson [AC99].

$$\underline{\text{data}} \quad \frac{G : \mathsf{Env}\ n \quad e : \mathsf{Expr}\ n \quad T : \mathsf{STy}}{\mathsf{Term}\ G\ e\ T : \star}$$

$$\underline{\text{where}} \quad \frac{v : \mathsf{Var}\ G\ i\ T}{\mathsf{var}\ v : \mathsf{Term}\ G\ (\mathsf{eVar}\ i)\ T} \qquad \frac{b : \mathsf{Term}\ (S::G)\ e\ T}{\mathsf{lam}\ b : \mathsf{Term}\ G\ (\mathsf{eLam}\ S\ e)\ (S \Rightarrow T)}$$

$$\frac{f : \mathsf{Term}\ G\ fe\ (S \Rightarrow T) \quad a : \mathsf{Term}\ G\ ae\ S}{\mathsf{app}\ f\ a : \mathsf{Term}\ G\ (\mathsf{eApp}\ fe\ ae)\ T}$$

A naïve implementation of Term gives rise to a horrifying amount of duplication. Fortunately, many of the arguments are forceable and thanks to the indexing over raw terms, Term is also detaggable. After these optimisations and the collapsing of Var, this is all that remains:

$$\begin{aligned}
[\![\mathsf{var}]\!] &\Longrightarrow \lambda n; G; i; T; v.\ \{\mathsf{var}\}\ \{n\}\ \{G\}\ \{i\}\ \{T\}\ \{v\} \\
[\![\mathsf{lam}]\!] &\Longrightarrow \lambda n; G; S; e; T; b.\ \{\mathsf{lam}\}\ \{n\}\ \{G\}\ \{S\}\ \{e\}\ \{T\}\ b \\
[\![\mathsf{app}]\!] &\Longrightarrow \lambda n; G; fe; S; T; f; ae; a.\ \{\mathsf{app}\}\ \{n\}\ \{G\}\ \{fe\}\ S\ \{T\}\ f\ \{ae\}\ a
\end{aligned}$$

The only non-recursive arguments which survive are the domain types of applications. Typechecking thus consists of ensuring that these can be determined.

The run-time costs of running the typechecker on a simple term, applying the identity function to an element of base type, are shown in figure 4.50.

Program	Version	Instructions	Thunks	Memory Accesses	Cells
Typechecking $(\lambda x : \iota.\ x)\iota$	Naïve	23232	13746	6910	1620
	Optimised	20891	11820	6722	1136
	Change	**-10.08%**	**-14.01%**	**-2.72%**	**-29.88%**

Figure 4.50: Run-time costs of typechecking $(\lambda x : \iota.\ x)\ \iota$

4.5.6 Results summary

Program	Version	Instructions	Thunks	Memory Accesses	Cells
Vector lookup	Naïve	549	300	166	39
	Optimised	537	300	166	27
	Change	**-2.23%**	-	-	**-30.76%**
gcd 6 3	Naïve	37896	18864	12293	2749
	Optimised	37486	18636	12293	2567
	Change	**-1.08%**	**-1.20%**	-	**-6.62%**
Quicksort	Naïve	175649	86600	55221	17268
	Optimised	171264	85586	55189	13900
	Change	**-2.50%**	**-1.17%**	**-0.05%**	**-19.50%**
Totalling a DList	Naïve	69612	28218	30695	2494
	Optimised	66333	27278	29774	1622
	Change	**-4.71%**	**-3.33%**	**-3.00%**	**-34.96%**
Typechecking $(\lambda x : \iota.\ x)\iota$	Naïve	23232	13746	6910	1620
	Optimised	20891	11820	6722	1136
	Change	**-10.08%**	**-14.01%**	**-2.72%**	**-29.88%**

Figure 4.51: Results of marking optimisation

The results are summarised in figure 4.51. In each case there is a significant reduction in the number of cell allocations made on the heap (this being where data is stored). Correspondingly, there is a reduction in the number of instructions executed; this is not surprising, since fewer heap nodes need to be created. The transformations are intended as storage optimisations and are applied here with some success — the number of memory accesses, however, remains largely the same. This is not surprising, because the optimisations are intended to avoid duplication of data rather than to remove data outright. It is however also good to see that a result of the space optimisation is also a slight reduction in the number of overall instructions executed. It would be surprising to see anything other than a reduction in space, given the nature of the transformations; each transformation removes subterms rather than rearranging subterms so it is almost certain that we should see a saving somewhere. Nevertheless, these results show that, at least for these simple examples, these optimisations are not at the expense of time.

Since this is an experimental implementation, we do not necessarily get an accurate picture of run-time just from the number of instructions executed; in particular, some in-

structions are more expensive to execute than others. We can, however, see from the nature of the transformations that in general they remove instructions, rather than replacing several cheap instructions with one expensive one. i.e., the transformations prune ExTT terms, in that they simply remove constructor arguments. Hence in the G-code, there are fewer PUSH instructions, and MKCON builds smaller data structures. The only way in which these transformations can cause performance to get worse (in terms of time, using a lazy evaluation strategy) is if they cause an index to be evaluated which otherwise would not be - this would happen if the index is needed for some other computation (i.e. PROJ needs to be evaluated) or an index is used for discrimination. However, in general, we choose indices because their values are related to the family's values and so construction of the family is closely related to computation of the index, hence this problem is unlikely to arise. Also, Ennals notes in [EP03, Enn03] that most values are eventually evaluated to normal form, so there is rarely a penalty in "speculatively" evaluating a value.

These results are obtained by applying the forcing, detagging and collapsing optimisations in isolation, and therefore do not present a full picture of the run-time costs of EPIGRAM programs. We should also consider that these programs are all run to completion; in many situations, particularly with lazy evaluation, we may expect production and consumption of data to be interleaved. In future work, when there is a significant body of EPIGRAM code to experiment with, it will be interesting to investigate how other optimisations interact with the optimisations presented here. In particular, Jones' root optimisation [Jon94], which takes advantage of arguments which do not change in recursive calls, may have a beneficial effect on the implementation of elimination rules. Serious G-machine implementations also do some re-ordering of arguments; this kind of technique may also improve the effect of the root optimisation.

Dependent types present us with another approach to reasoning about optimisations; in future work, we may wish to model potential optimisations by creating a representation of ExTT code in EPIGRAM itself, indexed over its cost, similar to Santos' cost semantics [San95]. Using such an approach, we could predict the cost of the original and transformed code, and compare the prediction with actual results.

4.6 A larger example — A well-typed interpreter

Some of the advantages of Cayenne [Aug98] are demonstrated by Augustsson and Carlsson's well-typed interpreter [AC99]. The interpreter they implement has the following features:

- It implements addition of integers, boolean comparisons, λ-abstraction and function application, returning an element of a Cayenne type.
- The return type depends on the program being interpreted. For example, the return type of an addition operation is an integer, but the return type of a comparison is a boolean. In a simply typed language, this would be achieved through a tagged union,

with a tag indicating the return type. Augustsson and Carlsson demonstrate that the overhead of the tag is unnecessary when using dependent types.

- Only well-typed expressions can be interpreted, achieved through the using of a well-typing predicate passed to the interpreter.
- Type dependency is also used to verify the synchronisation between type environments (that is, the types passed to a λ-abstraction) and value environments (the values applied to a λ-abstraction). If, for example, the first element in the type environment is $\mathbb{N}$, the first element of the value environment can only be an element of $\mathbb{N}$.

In this section, I implement the same program using inductive families to represent well-typedness (removing the need for a well-typing predicate) and synchronisation of type and value environments. I also show how the marking optimisations of this chapter lead to an efficient RunTT implementation.

4.6.1 The language

The language to be interpreted is a simply typed λ-calculus with integers, booleans, addition and comparison. I will refer to this language as λ_{AC}. Its syntax is shown in figure 4.52 and its typing rules in figure 4.53. This language augments Augustsson and Carlsson's implementation with a primitive recursion operator for natural numbers, primrec.

$e ::=$	$\lambda a : s.\ e$	λ-abstraction	$\mid$	$e_1\ e_2$	application
$\mid$	a	bound variable	$\mid$	$e_1 + e_2$	addition
$\mid$	$e_1 \leq e_2$	less than or equal	$\mid$	e_1 and e_2	boolean and
$\mid$	n	number	$\mid$	b	boolean value
$\mid$	primrec $e_1\ e_2\ e_3$	primitive recursion			

Figure 4.52: The interpreter language, λ_{AC}

$$\frac{}{\Gamma \vdash n : \mathbb{N}} \qquad \frac{}{\Gamma \vdash b : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e_1 : s \to t \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1\ e_2 : t} \qquad \frac{\Gamma, a : s \vdash e : t}{\Gamma \vdash \lambda a : s.\ e : s \to t}$$

$$\frac{\Gamma \vdash e_1 : \mathbb{N} \quad \Gamma \vdash e_1 : \mathbb{N}}{\Gamma \vdash e_1 + e_2 : \mathbb{N}} \qquad \frac{\Gamma \vdash e_1 : \mathbb{N} \quad \Gamma \vdash e_1 : \mathbb{N}}{\Gamma \vdash e_1 \leq e_2 : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_1 : \mathsf{Bool}}{\Gamma \vdash e_1 \text{ and } e_2 : \mathsf{Bool}} \qquad \frac{}{\Gamma, a : t \vdash a : t}$$

$$\frac{\Gamma \vdash x : \mathbb{N} \quad z : A \quad s : \mathbb{N} \to A \to A}{\Gamma \vdash \mathsf{primrec}\ x\ z\ s : A}$$

Figure 4.53: Typing rules for λ_{AC}

4.6.2 Representation

This language can be represented as an inductive family which, by indexing over the type environment and the type of an expression, ensures that only well-typed expressions can be built.

Since the value returned by the interpreter is a type in the implementation language, we implement type environments as a vector of types. As with the simply typed λ-calculus example of section 4.5.5, we represent type environments as vectors of types, and membership of a type environment as a relation (figure 4.54).

$$\underline{\text{let}} \quad \frac{n : \mathbb{N}}{\mathsf{Env}\ n : \star} \quad \mathsf{Env}\ n \mapsto \mathsf{Vect} \star n$$

$$\underline{\text{data}} \quad \frac{G : \mathsf{Env}\ n \quad i : \mathsf{Fin}\ n \quad t : \star}{\mathsf{Var}\ G\ i\ t : \star}$$

$$\underline{\text{where}} \quad \frac{}{\mathsf{stop} : \mathsf{Var}\ (s::G)\ \mathsf{f0}\ s} \qquad \frac{v : \mathsf{Var}\ G\ i\ t}{\mathsf{pop}\ v : \mathsf{Var}\ (s::G)\ (\mathsf{fs}\ i)\ t}$$

Figure 4.54: Type environments

The declaration of the family representing λ_{AC} is as shown in figure 4.55. There is a clear resemblance between this declaration and the typing rules in figure 4.53. There is no need for a well-typing predicate; indexing over the type means that if a term can be built at all it must be well typed.

$$\underline{\text{data}} \quad \frac{G : \mathsf{Env}\ n \quad A : \star}{\mathsf{Expr}\ G\ A : \star} \quad \underline{\text{where}}$$

$$\frac{k : \mathbb{N}}{\mathsf{enat}\ k : \mathsf{Expr}\ G\ \mathbb{N}} \qquad \frac{b : \mathsf{Bool}}{\mathsf{ebool}\ b : \mathsf{Expr}\ G\ \mathsf{Bool}}$$

$$\frac{f : \mathsf{Expr}\ G\ (s \to t) \quad a : \mathsf{Expr}\ G\ s}{\mathsf{eapp}\ f\ a : \mathsf{Expr}\ G\ t} \qquad \frac{e : \mathsf{Expr}\ (s::G)\ t}{\mathsf{elam}\ e : \mathsf{Expr}\ G\ (s \to t)}$$

$$\frac{a, b : \mathsf{Expr}\ G\ \mathbb{N}}{\mathsf{eadd}\ a\ b : \mathsf{Expr}\ G\ \mathbb{N}} \qquad \frac{a, b : \mathsf{Expr}\ G\ \mathbb{N}}{\mathsf{ele}\ a\ b : \mathsf{Expr}\ G\ \mathsf{Bool}}$$

$$\frac{a, b : \mathsf{Expr}\ G\ \mathsf{Bool}}{\mathsf{eand}\ a\ b : \mathsf{Expr}\ G\ \mathsf{Bool}} \qquad \frac{v : \mathsf{Var}\ G\ i\ t}{\mathsf{evar}\ v : \mathsf{Expr}\ G\ t}$$

$$\frac{x : \mathsf{Expr}\ G\ \mathbb{N} \quad z : \mathsf{Expr}\ G\ A \quad s : \mathsf{Expr}\ G\ (\mathbb{N} \to A \to A)}{\mathsf{eprimrec}\ x\ z\ s : \mathsf{Expr}\ G\ A}$$

Figure 4.55: Interpreter type declaration

The interpreter has a value environment in which to look up the values of variables. Since variables in the environment may have different types, using a Vect is not appropriate. Instead, we synchronise it with the type environment; each value in the value environment gets its type from the corresponding entry in the type environment. The declaration of the

value environment is given in figure 4.56, along with a lookup function.

```
data  G : Env n / ValEnv G : ⋆   where   / empty : ValEnv ϵ
                                          t : T   r : ValEnv G / extend t r : ValEnv (T::r)

let   v : Var G i T   ve : ValEnv G / envLookup v ve : T

  envLookup   v          ve          ⇐ elim v
  envLookup   stop       (extend t r)  ↦ t
  envLookup   (pop v)    (extend t r)  ↦ envLookup v r
```

Figure 4.56: Value environments

Note, in **envLookup**, that it is only possible to look up values in a non-empty environment. This is ensured by the type of v, which is indexed over i : Fin n, making i and n implicit arguments to **envLookup**. Since i cannot take an index of zero, the ValEnv cannot be indexed over a non-empty type environment. The type of **envLookup** ensures that the value we retrieve from the value environment will have the type given by the corresponding entry in the type environment.

Remark — Universes

In the interpreter, we represent type environments as a vector of $\star$. There is a difficulty here with universe levels, however, as we do not have cumulativity; here we use Vect to contain elements of type $\star_0$ (i.e., the parameter type is itself of type $\star_1$), whereas earlier we have used Vect to contain elements of some type in $\star_0$. With the type theory as it stands, we can only do this with two separate declarations of Vect, at different universe levels, which is at best inconvenient — it would be preferable for Vect to allow element types at all levels of the universe hierarchy. In this example, we assume that this is the only use of Vect and allow it to contain elements of $\star_0$. Nevertheless, the problem of how to deal with universe hierarchies must ultimately be addressed.

A possible solution is to use Tarski style universes [ML85], as discussed in [Luo94] and implemented in Plastic [CL01]. However, such decisions about the representation of universes do not affect the optimisations presented in this thesis, and so we will not discuss them further here.

4.6.3 The interpreter

The implementation of the interpreter for λ_{AC} is shown in figure 4.57 — this program can also be viewed as a proof that evaluation of expressions in λ_{AC} terminates, since the implementation is in a strongly normalising language. **primrec** is a helper function which implements the primitive recursion over $\mathbb{N}$.

interp itself is written by structural recursion over the input expression x. It returns a semantic representation, as an EPIGRAM term, of the input expression. So, for example, the interpretation of a λ-abstraction in λ_{AC} (elam) is an EPIGRAM function which implements that λ-abstraction. Interpretation of an application then simply applies the function to the interpretation of its argument. Note that in the case for elam, we use the implicit argument s to establish the input type of the function. This approach is similar to normalisation by evaluation (see Appendix C) in that we construct a semantic representation of the term to be interpreted, but there is no reification back to the object language here.

$$
\begin{array}{l}
\underline{\mathsf{let}}\ \dfrac{x : \mathsf{Expr}\ G\ T \quad ve : \mathsf{ValEnv}\ G}{\mathbf{interp}\ x\ ve : T} \\
\quad \mathbf{interp}\ \ x\ \ ve \Leftarrow \underline{\mathsf{elim}}\ x \\
\quad \mathbf{interp}\ \ (\mathsf{enat}\ k)\ \ ve \mapsto k \\
\quad \mathbf{interp}\ \ (\mathsf{ebool}\ b)\ \ ve \mapsto b \\
\quad \mathbf{interp}\ \ (\mathsf{eapp}\ f\ a)\ \ ve \mapsto (\mathbf{interp}\ f\ ve)\ (\mathbf{interp}\ a\ ve) \\
\quad \mathbf{interp}\ \ (\mathsf{elam}_s\ e)\ \ ve \mapsto \lambda a : s.\ \mathbf{interp}\ e\ (\mathsf{extend}\ a\ ve) \\
\quad \mathbf{interp}\ \ (\mathsf{eadd}\ a\ b)\ \ ve \mapsto \mathbf{plus}\ (\mathbf{interp}\ a\ ve)\ (\mathbf{interp}\ b\ ve) \\
\quad \mathbf{interp}\ \ (\mathsf{ele}\ a\ b)\ \ ve \mapsto \mathbf{le}\ (\mathbf{interp}\ a\ ve)\ (\mathbf{interp}\ b\ ve) \\
\quad \mathbf{interp}\ \ (\mathsf{eand}\ a\ b)\ \ ve \mapsto \mathbf{and}\ (\mathbf{interp}\ a\ ve)\ (\mathbf{interp}\ b\ ve) \\
\quad \mathbf{interp}\ \ (\mathsf{evar}\ v)\ \ ve \mapsto \mathbf{envLookup}\ v\ ve \\
\quad \mathbf{interp}\ (\mathsf{eprimrec}\ x\ z\ s)\ ve \mapsto \mathbf{primrec}\ (\mathbf{interp}\ x\ ve)\ (\mathbf{interp}\ z\ ve)\ (\mathbf{interp}\ s\ ve) \\
\underline{\mathsf{let}}\ \dfrac{n : \mathbb{N} \quad z : A \quad s : \mathbb{N} \to A \to A}{\mathbf{primrec}\ n\ z : s : A} \\
\quad \mathbf{primrec}\ \ n\ \ z\ s \Leftarrow \underline{\mathsf{elim}}\ n \\
\quad \mathbf{primrec}\ \ 0\ \ z\ s \mapsto z \\
\quad \mathbf{primrec}\ (\mathsf{s}\ k)\ z\ s \mapsto s\ k\ (\mathbf{primrec}\ k\ z\ s)
\end{array}
$$

Figure 4.57: The interpreter

The **plus** function which implements the eadd operation is defined elsewhere; the boolean operations **le** and **and** have straightforward implementations (figure 4.58).

$$
\begin{array}{l}
\underline{\mathsf{let}}\ \dfrac{n, m : \mathbb{N}}{\mathbf{le}\ n\ m : \mathsf{Bool}} \\
\quad \mathbf{le}\ \ n\ \ m \Leftarrow \underline{\mathsf{elim}}\ n \\
\quad \mathbf{le}\ \ 0\ \ m \mapsto \mathsf{true} \\
\quad \mathbf{le}\ \ (\mathsf{s}\ n)\ \ m \Leftarrow \underline{\mathsf{elim}}\ m \\
\quad \mathbf{le}\ (\mathsf{s}\ n)\ \ 0 \mapsto \mathsf{false} \\
\quad \mathbf{le}\ (\mathsf{s}\ n)\ (\mathsf{s}\ m) \mapsto \mathbf{le}\ n\ m \\
\underline{\mathsf{let}}\ \dfrac{x, y : \mathsf{Bool}}{\mathbf{and}\ x\ y : \mathsf{Bool}} \\
\quad \mathbf{and}\ \ x\ \ y \Leftarrow \underline{\mathsf{case}}\ x \\
\quad \mathbf{and}\ \mathsf{true}\ y \mapsto y \\
\quad \mathbf{and}\ \mathsf{false}\ y \mapsto \mathsf{false}
\end{array}
$$

Figure 4.58: Implementation of **le** and **and**

4.6.4 Optimisation

We have already seen the optimisations which apply to Fin and Vect, which can also be applied in this case. We also observe that Var is concretely collapsible. The transformations which arise by marking the forceable arguments to Expr's constructors are shown in figure 4.59.

$$
\begin{array}{l}
[\![\mathsf{enat}]\!] \Longrightarrow \lambda n;\, G;\, k.\ \mathsf{enat}\ \{n\}\ \{G\}\ k \\
[\![\mathsf{ebool}]\!] \Longrightarrow \lambda n;\, G;\, b.\ \mathsf{ebool}\ \{n\}\ \{G\}\ b \\
[\![\mathsf{eapp}]\!] \Longrightarrow \lambda n;\, G;\, s;\, t;\, f;\, a.\ \mathsf{eapp}\ \{n\}\ \{G\}\ s\ \{t\}\ f\ a \\
[\![\mathsf{elam}]\!] \Longrightarrow \lambda n;\, G;\, s;\, t;\, e.\ \mathsf{elam}\ \{n\}\ \{G\}\ s\ t\ e \\
[\![\mathsf{eadd}]\!] \Longrightarrow \lambda n;\, G;\, a;\, b.\ \mathsf{eadd}\ \{n\}\ \{G\}\ a\ b \\
[\![\mathsf{ele}]\!] \Longrightarrow \lambda n;\, G;\, a;\, b.\ \mathsf{ele}\ \{n\}\ \{G\}\ a\ b \\
[\![\mathsf{eand}]\!] \Longrightarrow \lambda n;\, G;\, a;\, b.\ \mathsf{eand}\ \{n\}\ \{G\}\ a\ b \\
[\![\mathsf{evar}]\!] \Longrightarrow \lambda n\ G;\, i;\, t;\, v.\ \mathsf{evar}\ \{n\}\ \{G\}\ i\ \{t\}\ \{v\} \\
[\![\mathsf{eprimrec}]\!] \Longrightarrow \lambda n;\, G;\, A;\, x;\, z;\, s.\ \mathsf{eprimrec}\ \{n\}\ \{G\}\ \{A\}\ x\ z\ s
\end{array}
$$

Figure 4.59: Optimisation of Expr

There are several things to note about these transformations. In particular, the type environment which is stored at every node of an expression in a naïve implementation of Expr is removed from the term entirely so will appear only as an argument to Expr-**Elim**. We also see that some of the $\star$ arguments have not been marked, however — s and t are still arguments to elam, s is still an argument to eapp. These are not forceable since, as they do not appear in the indices of these constructors, they will not appear as pattern variables in Expr-**Elim**. However, as there is no casetype construct, these arguments can never be used — it is conceivable that a later optimisation can remove such arguments.

If we also optimise out the unusable type arguments which remain, this structure is the same as the one used in [AC99], and the same as a structure we might consider using to represent terms in a simply typed language. In this example, there is no run-time storage overhead caused by indexing the family over several invariants.

4.6.5 Results

The run-time cost of the interpreter is assessed by evaluating four λ_{AC} expressions of varying size and complexity. First, we define two functions; **plus** which applies the primitive addition operator to its two arguments, and **mult** which applies **plus** recursively to implement multiplication. In λ_{AC}, these are defined as follows:

$$\mathbf{plus} \mapsto \lambda x.\, \lambda y.\, x + y$$
$$\mathbf{mult} \mapsto \lambda x.\, \lambda y.\, \mathsf{primrec}\ x\ 0\ (\lambda k.\, \lambda ih.\, \mathbf{plus}\ y\ ih)$$

The four expressions we interpret are, in increasing order of complexity, 2, 2+3, **plus** 2 3 and **mult** 2 3. The run-time cost of each of these evaluations is shown in figure 4.60.

Program	Version	Instructions	Thunks	Memory Accesses	Cells
2	Naïve	1009	744	225	35
	Optimised	997	744	225	23
	Change	**-1.18%**	-	-	**-34.29%**
2+3	Naïve	9569	6876	2419	526
	Optimised	9389	6876	2419	346
	Change	**-1.88%**	-	**0**	**-34.22%**
plus 2 3	Naïve	31661	21120	8236	3037
	Optimised	30218	20904	8104	1846
	Change	**-4.55%**	**-1.02%**	**-1.60%**	**-39.21%**
mult 2 3	Naïve	199832	113916	54669	27766
	Optimised	187473	112620	53637	16919
	Change	**-6.18%**	**-1.14%**	**-1.89%**	**-39.07%**

Figure 4.60: Run-time costs of the interpreter

Clearly, the biggest gain on applying the optimisation is the reduction in the number of cells required to store data. This is not surprising since it is precisely the purpose of the forcing and detagging optimisations. In each case, the optimisation removes 35-40% of the allocations. A natural consequence of this is to reduce the total number of G-machine instructions executed — there are fewer arguments to constructors, so fewer stack operations required. For the larger expressions we also see a slight reduction in the number of thunks built — this occurs as a result of RunTT functions which build constructor applications needing fewer arguments, e.g.:

$\lambda n; G; k.$ enat n G k optimises to $\lambda k.$ enat k

The optimised version builds fewer application nodes, hence fewer thunks.

4.7 Summary

We have seen in this chapter how the properties of elimination rules lead to the optimisation of the data structures eliminated by those rules and the programs which elaborate in terms of those rules. We have defined an extended execution language for TT, which we call ExTT. Terms in ExTT can arise only by applying an optimising transformation for the original TT. In particular, we apply three optimisations based on the form of an elimination rule:

- The forcing optimisation arises from the observation that arguments which are repeated in an elimination rule must be convertible. We only need to keep one copy of such arguments — given the choice between keeping the copy passed as an index to the elimination rule and keeping the copy stored within the data structure, we keep the copy passed to the elimination rule, firstly because this appears only in the top level application and secondly because we will have further opportunities to remove this if it remains unused.

- The detagging optimisation arises from the observation that elimination rules are well-defined (that is, complete and non-overlapping). Hence, if we can determine which ι-scheme to choose based on the constructors of something other than the target, we need never store the constructors of the target itself.

- The collapsing optimisation arises from the observation that evaluation at run-time is in the empty context and hence all observable terms are in canonical form. If we never need to examine the canonical form of an object, we need not store that object at all.

The collapsing optimisation is only valid at run-time, which means that different transformations are used for constructors and elimination rules of collapsible families depending on whether we are in a compile-time or run-time setting. We cannot, therefore, simply apply the transformation from TT to ExTT once at compile-time only — if we want to get the full benefit of the collapsing optimisation, we have to apply a second set of transformations for the run-time setting.

These are remarkably straightforward optimisations, but they only present themselves because we are taking inductive families seriously as data structures. The purpose of the forcing optimisation is largely to overcome the space penalties of adopting dependent types in the first place, but detagging derives new benefit from static information unavailable in a simply typed setting. For example, in the development of a typechecker for the simply typed λ-calculus as presented in section 4.5.5, it is clear that there must be a link between the raw terms and the well-typed terms. In a simply typed language, this is inexpressible, but the indexing of the well-typed terms over the raw terms not only expresses the link, but leads to an optimisation of the representation of well-typed terms. Collapsing, too, derives further benefit — we can delete accessibility arguments and equational reasoning from run-time code not because we deem them to be proof-irrelevant, but because they actually *are* irrelevant. This allows us to build new structures on top of old structures, with additional invariants (such as the non-repeating list example of section 4.5.4), without any overhead.

The forcing, detagging and collapsing optimisations necessitate a more sophisticated compilation scheme for elimination rules than we used in Chapter 3, as we saw in section 4.4.2. This is a modified version of Augustsson's pattern matching compiler algorithm [Aug85, Pey87]. The modifications are made to take advantage of the respectfulness and well-definedness of elimination rules — we only do enough case analysis to identify which ι-scheme applies, and use constructor argument projection ($x!i$) to project out arguments where we already know (due to well-definedness) what form an object must take.

As with all optimisations, there are various trade-offs to consider when applying these optimisations. For example, forcing is a storage optimisation, but we must consider the possible time penalty in reconstructing the forced arguments from the indices, where the indices are sufficiently complex. With detagging and collapsing, we must consider whether removing the tag leads to an overly complex implementation of the elimination rule, due to increased difficulty in discriminating between constructors. Evaluation strategy also has

an effect; it is possible that these storage optimisations cause terms to be evaluated which would otherwise remain unused in a lazy evaluation setting. With the examples we have seen in this chapter, it is relatively easy to project out constructor arguments and discriminate on elimination rules, since the indices on the families we have considered are not particularly complex. However, we are just beginning to learn how to write dependently typed programs, and it remains to be seen whether the programs we have seen in this chapter are representative of dependently typed programs as a whole.

Chapter 5

Number Representation

Paul Graham notes in "The Hundred Year Language" [Gra03] that in a programming language, just as in Mathematics, the fewer axioms the better. He even asks "Could a programming language go so far as to get rid of numbers as a fundamental data type?" The original core definition of Lisp as proposed by McCarthy [McC60] did not have numbers as primitives, after all, and this is what we have done so far with EPIGRAM, defining natural numbers just as any other inductive datatype. While this is convenient for programming thanks to the natural structure it has and the elimination behaviour it generates, it is not practical for computation with large numbers due to space and time complexity. What we look for with natural numbers, and potentially with any data structure which can be represented in a more compact fashion, is an efficient internal representation in RunTT and transformation rules from the TT definition to the efficient internal representation.

A practical programming language includes certain datatypes as primitives, from which the user can build more complex data structures. Such primitives typically include integer and real numbers, characters and strings. These primitive types can be equipped with primitive operations such as comparison, arithmetic in the case of numbers, and various manipulation operators in the case of strings. The choice of primitive types in a programming language is often based on the data which the underlying machine has a representation for, numbers being the obvious example. Landin considers a family of languages, ISWIM [Lan66], parametrised over the choice of primitive types (in Landin's words, "a basic set of given things") with a common structure ("a way of expressing things in terms of other things") where the choice of primitives affects the application domain of a language.

In EPIGRAM, however, there are no primitive types — only a "way of expressing things in terms of other things" — and all data structures are built by hand via inductive datatypes. As a consequence, the core language has no access to the machine's efficient implementation of primitive types. We may define types with similar behaviour to the structures provided by the CPU, such as $\mathbb{N}$, but with far worse performance, in terms of both speed and space. In this chapter, we will consider ways to improve this situation, first considering an imple-

mentation of binary arithmetic purely in EPIGRAM, then an external implementation which uses the CPU's representation of numbers as the underlying representation of $\mathbb{N}$.

5.1 Representing Numbers in Type Theory

In Chapter 3, I mentioned the inefficiency of number representation as an overhead which we must take into consideration in the design of a run-time system for EPIGRAM. The advantages to the programmer are that the unary structure of $\mathbb{N}$ gives rise to obvious recursion behaviour and, correspondingly, straightforward proofs of properties of functions over $\mathbb{N}$ and data structures indexed over $\mathbb{N}$. These advantages are perhaps outweighed by efficiency considerations. The size of the representation of a number n is proportional to n itself; compare this with a binary representation where the size is proportional to $\log n$. Correspondingly, the unary nature of the structure means that arithmetic operations take time proportional to n whereas operations on a binary representation take time proportional to $\log n$.

An example of the limitations of the $\mathbb{N}$ representation arises in [CO01]. This work is primarily interested in using an external oracle to provide witnesses for applying Pocklington's Criterion to show primality of large integers; this works by using a Java program to generate COQ tactic scripts. Caprotti and Oostdijk note that while they are able to generate tactic scripts for large numbers, the theorem prover is not able to process the data structures required to store these numbers.

Work by Magaud and Bertot [MB01] improves the situation within COQ. Here, they present a technique for transforming data structures and their proofs into a more efficient representation using $\mathbb{N}$ as their example. This involves mapping the constructors and elimination rule of $\mathbb{N}$ to a binary setting bin, relying on a proof of an isomorphism between $\mathbb{N}$ and bin. While clearly an improvement, this technique still relies on $\mathbb{N}$ as an intermediate structure for implementing the elimination rule. In a practical programming language we would like to avoid such overheads, and take advantage of the underlying machine directly.

5.1.1 What is $\mathbb{N}$ used for?

There are three main uses for natural numbers in EPIGRAM programs:

1. The structure of a $\mathbb{N}$ allows it to be used to specify size-based invariants of data structures; Vect is an example of this, in that adding an item to a vector corresponds to adding a s symbol to a $\mathbb{N}$. Many properties of $\mathbb{N}$ and operations on it can be proved inductively in order to verify properties of structures which are indexed over $\mathbb{N}$.

2. Recursion over $\mathbb{N}$ gives bounded iteration, corresponding to a for loop in an imperative language. $\mathbb{N}$-**Elim** n represents performing an operation n times. This is similar to the motivation for the Church numeral representation [Chu41] which represents application of a function n times.

3. $\mathbb{N}$ and its basic operations **plus** and **mult** can be used as a straightforward implementation of unsigned integer arithmetic.

The most important of these is the first; using $\mathbb{N}$ in this way gives us a method for verifying size-based properties of programs without having to execute $\mathbb{N}$ based programs at run-time (since the properties are verified once and for all at compile-time). The second purpose, using $\mathbb{N}$-**Elim** to perform an operation n times, gives a method for repetition with guaranteed termination (as with a `for` loop in an imperative language). In this case, we need not worry that the structure of $\mathbb{N}$ is of order n, because n is exactly how many times we want to execute an operation. In the third case, however, using $\mathbb{N}$ to implement arithmetic, the structure of the number representation is unimportant; **plus** and **mult** are abstract operations for which the programmer is not interested in the internal representation or implementation. It is unreasonable to consider $\mathbb{N}$ an appropriate structure where arithmetic is an end in itself.

There are therefore two separate settings to consider; where the structure is important (as in verification of properties and bounded iteration) and where the structure is unimportant (as in arithmetic). These are two separate aims, and it therefore makes sense to choose two separate representations for each.

5.2 The Word family

In this section I present an implementation of binary numbers in EPIGRAM taking advantage of inductive families and views. The main aim of this representation is to provide a space efficient representation and efficient arithmetic operations entirely within the core language of EPIGRAM.

There have been proposals for methods of representing numbers efficiently in the λ-calculus. [Gol00] chooses a representation based on a list of bits with predicates for zero and successor and a predecessor function and defines efficient arithmetic functions. A representation in COQ is given by [MB01], based on numbers of the form 0, 1, $2 \times x$ and $2 \times x + 1$.

Many different approaches to number representation are suggested by [Knu69]. I choose a dichotomous representation for numbers; a number is a represented as a tree of digits, with the individual bits at the leaves. Every number, other than the base cases, has a more significant word and a less significant word. This representation[1] has simplicity in mind. It leads to a straightforward definition of the successor, addition and multiplication functions since there are only two cases to deal with — one digit and two digit numbers.

[1] originally due to James McKinna

5.2.1 Word n

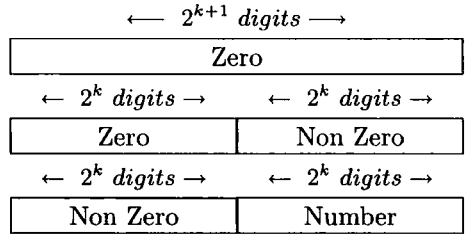
Word n is a family representing non-zero numbers of length 2^n digits. I also define a parametrised type $(\cdot)_0$ which adds a zero element to any type. Representing non-zero numbers separately, while slightly complicating the data structure, has some advantages. Firstly, it leads to a certain amount of compression; large numbers of leading zeroes are collapsed. Secondly, it allows a more precise definition of the types of certain functions, including successor and predecessor.

The $(\cdot)_0$ family (figure 5.1) adds a zero element to any family; a value is either zero, or any value in the original family. This is the same in structure as the `Maybe` type in Haskell.

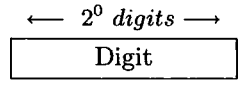
$$\underline{\text{data}} \quad \frac{T : \star}{(T)_0 : \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{O} : (T)_0} \quad \frac{t : T}{\boxed{t} : (T)_0}$$

Figure 5.1: Lifting a zero element

Non zero words are indexed over n, such that their length is 2^n. This means that for $n \neq 0$, the number can be broken down into two halves; a more significant word and a less significant word. Also, they are parametrised over the digit type, D. The digit represents the base of the number system. Informally, a number composed of 2^{k+1} digits can be any of the following:



The base case, for numbers of 2^0 digits, is clearly simply a digit:



The EPIGRAM declaration which builds such a structure is given in figure 5.2.

For simplicity, in what follows I will take $D = \{1\}$ and elide it, so that Word n contains a binary representation of numbers. There is no reason why D should not be any other base, including 32 bit machine integers.

The indexing of numbers over n is crucial to this representation for two reasons in particular. Without it, it would be possible to build badly formed numbers where the left and right halves were of different lengths, necessitating either run-time checks on the length, or needless complexity in function definitions. Also, the index provides a useful structure

$$\begin{array}{ll} \underline{\text{data}} & \dfrac{D : \star \quad n : \mathbb{N}}{\mathsf{Word}\ D\ n\ :\ \star} \\ \underline{\text{where}} & \dfrac{d\ :\ D}{\mathsf{Wd}\ d\ :\ \mathsf{Word}\ D\ \mathsf{0}} \\ & \dfrac{w\ :\ \mathsf{Word}\ D\ n}{\mathsf{W0}\ w\ :\ \mathsf{Word}\ D\ (\mathsf{s}\ n)} \\ & \dfrac{w\ :\ \mathsf{Word}\ D\ n \quad w_0\ :\ (\mathsf{Word}\ D\ n)_0}{\mathsf{W@}\ w\ w_0\ :\ \mathsf{Word}\ D\ (\mathsf{s}\ n)} \end{array}$$

Figure 5.2: Word declaration

for recursive calls — it is possible to write functions with a base[2] case (dealing with the single digit numbers) and a recursive case (dealing with the two digit numbers). Indexing over n means that numbers are inherently bounded by the index, unlike $\mathbb{N}$ which is (at least theoretically) unbounded. While this may be a disadvantage if we really want to represent unbounded numbers, it is in harmony with bounded machine arithmetic.

Remark: Using $(\cdot)_0$ to insert a zero element into Word has the unfortunate problem of making the recursive argument w_0 to W@ non strictly positive. While there is a simple transformation to get around this problem, namely using two separate families for zero and non-zero Words, we will continue using $(\cdot)_0$ for clarity of presentation.

5.2.2 The Split view of Word (s n)

I have said that this representation of numbers is *dichotomous*, which suggests that any number of length greater than one digit can be regarded as a two digit number. We might say that in this representation, every number has at most two digits. Dichotomous representations have been used in the past to implement numbers of arbitrary size, such as in early versions of LeLisp and as possible hardware representations[3]. To take advantage of this property, I introduce a view of numbers which gives their two digits separately, splitting them into a more significant word and a less significant word. To begin, I define a "glue" function $(x|y)$ which appends two digits:

$$\underline{\text{let}} \quad \frac{w, v\ :\ (\mathsf{Word}\ n)_0}{w \mid v\ :\ (\mathsf{Word}\ \mathsf{s}\ n)_0}$$

$$\begin{array}{lcl} w \mid v & \Leftarrow \underline{\text{case}}\ w \\ \mathsf{O} \mid v & \Leftarrow \underline{\text{case}}\ v \\ \mathsf{O} \mid \mathsf{O} & \mapsto \mathsf{O} \\ \mathsf{O} \mid \boxed{w} & \mapsto \boxed{\mathsf{W0}\ w} \\ \boxed{w} \mid v & \mapsto \boxed{\mathsf{W@}\ w\ v} \end{array}$$

Then the Split view of a number (figure 5.3) gives the more significant and less significant halves of that number. The covering function for this view is straightforward to define (figure

[2] "Base" being a particularly appropriate word in this context.
[3] Jean Vuillemin, personal communication.

5.4); it is basically the inverse of the glue function defined above.

$$\underline{\text{data}} \quad \frac{w \ : \ (\mathsf{Word\ s}\ n)_0}{\mathsf{Split}\ w \ : \ \star}$$

$$\underline{\text{where}} \quad \frac{msw, lsw \ : \ (\mathsf{Word}\ n)_0}{\mathsf{digits}\ msw\ lsw \ : \ \mathsf{Split}\ (msw \mid lsw)}$$

Figure 5.3: The Split view

$$\begin{array}{llll} \underline{\text{let}} & \dfrac{w \ : \ (\mathsf{Word\ s}\ n)_0}{\mathbf{split}\ w \ : \ \mathsf{Split}\ w} & \mathbf{split} \quad w & \Leftarrow \underline{\text{case}}\ w \\ & & \mathbf{split} \quad \mathsf{O} & \mapsto \mathsf{digits\ O\ O} \\ & & \mathbf{split} \quad \boxed{b} & \Leftarrow \underline{\text{case}}\ b \\ & & \mathbf{split} \quad \boxed{\mathsf{W0}\ w} & \mapsto \mathsf{digits\ O}\ \boxed{w} \\ & & \mathbf{split}\ \boxed{\mathsf{W@}\ w\ v} & \mapsto \mathsf{digits}\ \boxed{w}\ v \end{array}$$

Figure 5.4: Covering function for Split

This view gives a convenient form for pattern matching on two digit numbers. Using $\underline{\text{view}}\ \mathbf{split}\ w$ for recursion over w gives a pattern containing the two "digits" of the number:

$$\begin{array}{llll} \underline{\text{let}} & \dfrac{w \ : \ (\mathsf{Word\ s}\ n)_0}{f\ w \ : \ \mathsf{SomeType}} & f \quad w & \Leftarrow \underline{\text{view}}\ \mathbf{split}\ w \\ & & f\ (msw|lsw) & \mapsto \ \ldots \end{array}$$

This view is used extensively in the definition of arithmetic over Words. In particular, when defining functions by induction over the length of the Word, n, splitting numbers in this way gives us access to the recursive calls on the smaller Words.

5.2.3 The successor function

Since numbers are inherently bounded by their index, it is possible for the successor function to overflow. What, then, is an appropriate type for successor? I represent the possibility of overflow with the $(\,\cdot\,)^\infty$ type (figure 5.5), which, like $(\,\cdot\,)_0$, is the same in structure as the `Maybe` type in Haskell.

$$\underline{\text{data}} \quad \frac{T \ : \ \star}{(T)^\infty \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{\infty \ : \ (T)^\infty} \qquad \frac{t \ : \ T}{\lceil t \rceil \ : \ (T)^\infty}$$

Figure 5.5: Overflow type

It would be good for the successor function to be surjective, since then it has an inverse, the predecessor function. The zerolessness of Word means that it is possible to give it an

appropriate type. The definition is by induction over the index, n, where the $\mathsf{s}\ n$ case is defined by the Split view. Note that the index is an implicit argument to this function; since recursion is on this index, I have subscripted it in the definition (figure 5.6). Note also that there is a separate function, **sucDigit** for implementing the base case (one digit numbers).

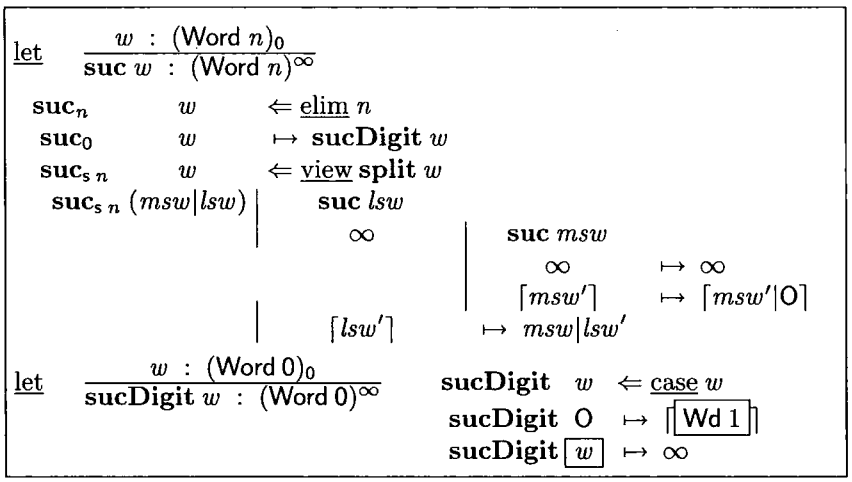


Figure 5.6: The successor function

5.2.4 Addition

Like successor, addition on bounded numbers can overflow. The typical way to capture this with hardware implementations of binary arithmetic is with a carry flag. This is perhaps reminiscent of the way we were taught to add up two digit numbers in primary school, with the form shown in figure 5.7.

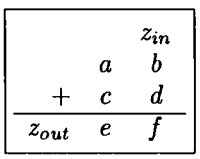


Figure 5.7: General form of addition

We begin by adding the less significant digits, and getting an intermediate carry, $(z_{mid}, f) = b +_{z_{in}} d$. Then we add the more significant digits using the intermediate carry, $(z_{out}, e) = a +_{z_{mid}} c$. The carry flag is represented by the type Carry, with two constructors yes and no. The return type is a pair of the carry flag and the binary number — this is not a dependent pair, so we use the simpler tuple type Carry × (Word n)$_0$ rather than Σ Carry (λz : Carry. (Word n)$_0$).

With the dichotomous representation, this maps nicely into a case for the base digits, and a recursive case. Again, recursion is on the length index of the arguments. This function is surjective — the base case is clearly surjective by examining all the possibilities, and the recursive case (figure 5.8) is surjective because it simply glues the result of (surjective) recursive calls together. The base case (figure 5.9) does nothing more than tabulate the eight possible base cases for add with carry on a single bit number.

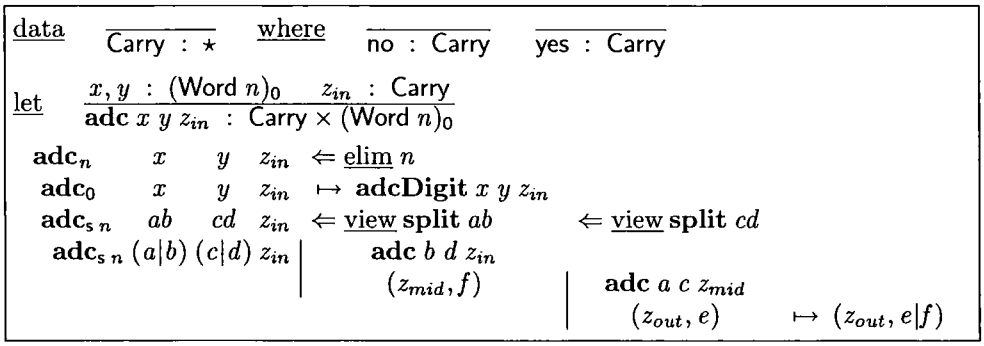


Figure 5.8: Definition of **adc**

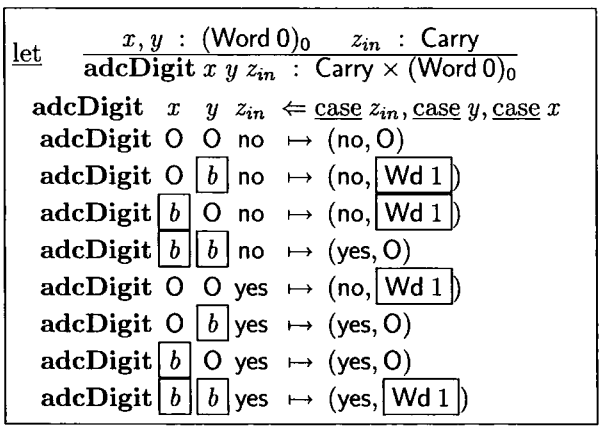


Figure 5.9: Base case of **adc**

The simplicity of this definition is due entirely to the choice of representation. At the expense of doing a little extra work to build an appropriate elimination rule (by the Split view) for two digit Words, we get a simple implementation for addition.

5.2.5 Multiplication

Let us go back to school again, and consider how we were taught to multiply two two-digit numbers using long multiplication. The general form can be presented as in figure 5.10.

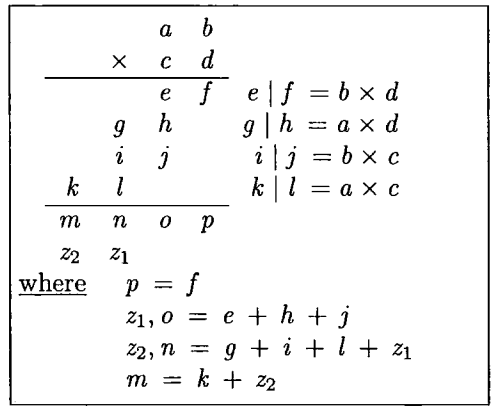


Figure 5.10: General form of multiplication

This is an approach we can consider taking for multiplication with Word; for a two-digit multiplication, there are four smaller multiplications on single-digit numbers, which lends itself to recursion on the size of the number. However, this seems untidy, not to mention inefficient — as well as four multiplications, there are five additions (z_1 and z_2 being the carry of these additions).

Instead of straightforward multiplication, therefore, I implement multiplication *with accumulator*. The idea is that instead of the addition taking place at the top level, it is pushed through each recursive call as an accumulator with the actual addition only taking place in the base case. In this way, the four digits of the result can simply be read off, rather than calculated from addition of intermediate results. The general scheme (figure 5.11) is similar to that of long multiplication, but we see the addition in the intermediate computations. The type is as follows:

$$\underline{\text{let}} \quad \frac{w_1, w_2, z_1, z_2 \ : \ (\mathsf{Word}\ n)_0}{\mathbf{mult}\ w_1\ w_2\ z_1\ z_2 \ : \ (\mathsf{Word}\ n)_0 \times (\mathsf{Word}\ n)_0}$$

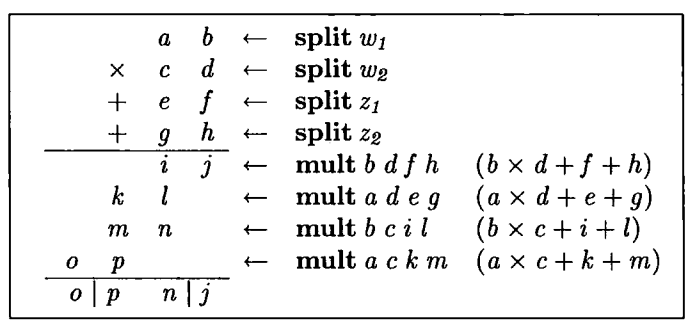


Figure 5.11: Scheme for multiplication with accumulator

This scheme is implemented by **split** on all four arguments, then recursion on the index

of the Word (figure 5.12). It should be noted that the zeroless representation allows us to take some shortcuts in this definition (although they are not presented here) since $n \times 0 = 0$.

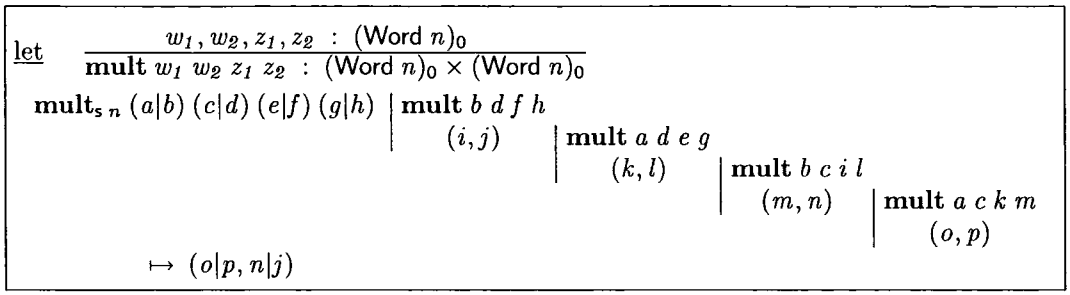


Figure 5.12: Recursive case of **mult**

This method of multiplication with accumulator does, however, still involve four sub multiplications. This is mainly as a result of the chosen representation; however, many more efficient algorithms exist for multiplication which reduce the number of multiplications on smaller digits to three, most notably Karatsuba multiplication [KO63]. Bernstein [Ber98] gives a survey of these techniques, attempting to present every technique known at the time of writing. While the implementation presented here is less sophisticated, it does give us some insight into how we might use type dependency to give more precise typing for complex operations.

5.2.6 Changing Bases

So far, we have taken the base $D = \{1\}$. What happens if we take some other base? Any type can be used as the base, provided that there is an implementation of **sucDigit**, **adcDigit** and **multDigit** for that type. To access these implementations, it becomes necessary to parametrise the type not only over D, but over the implementations of these base cases for arithmetic (hence separating out the definitions of **sucDigit**, **addDigit** and so on). This does not clutter the definitions of functions on Word, or the construction of values in Word, as may be expected, because parameters can be left implicit. An appropriate definition is shown in figure 5.13. The extra parameters, s, a and m are the base cases for successor, addition and multiplication on D respectively. Comparing this with the Haskell type class approach, we might consider digits to be a type class with successor, addition and multiplication defined as methods of that class. In Haskell, these methods would be passed around in a dictionary, in much the same way as they are passed as indices to the Word family here.

A natural choice for the base would be machine integers. We can imagine these to be simulated by an EPIGRAM data declaration as follows:

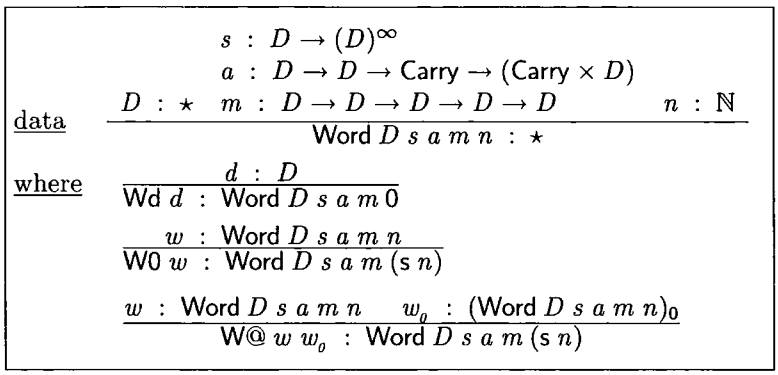


Figure 5.13: Word declaration, with base functions

data $\quad \mathsf{Int} : \star$

where $\quad 0 : \mathsf{Int} \quad 1 : \mathsf{Int} \quad 2 : \mathsf{Int} \quad \cdots \quad 4294967295 : \mathsf{Int}$

This definition would be accompanied by suitable definitions for accessing the low level implementations of successor, addition and multiplication. These functions would of course have to rely on features outside the core of EPIGRAM.

sucInt : $\mathsf{Int} \to (\mathsf{Int})^\infty$
adcInt : $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Carry} \to (\mathsf{Carry} \times \mathsf{Int})$
multInt : $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$

5.2.7 Building Big Numbers From Word

A problem with the Word n family as given is that it is still inherently bounded by $2^{2^n} - 1$. Thus it is not isomorphic to $\mathbb{N}$ and cannot be used invisibly as a drop-in replacement for $\mathbb{N}$ at runtime. One way to represent unbounded numbers based on Word is as a dependent pair:

bignum $\mapsto \Sigma n : \mathbb{N}.\ (\mathsf{Word}\ n)_0$

Unfortunately, this is still not isomorphic with $\mathbb{N}$; this can be seen by observing that, for example, while $(0, \mathsf{O})$ and $(\mathsf{s0}, \mathsf{O})$ are distinct **bignums**, they both represent the number zero. In many contexts, this is not a problem. However, it does make proving an elimination rule with behaviour corresponding to that of $\mathbb{N}$-**Elim** more difficult.

The difficulty is caused by the possibility of leading zeroes; an alternative representation of big numbers, built on top of Word, is to build a family BigNumber with constructors corresponding to Word, but without a leading zero constructor (figure 5.14). Zeroes are lifted with $(\,\cdot\,)_0$ as before.

There is a straightforward mapping between Word and BigNumber (figure 5.15) since the constructors are similar.

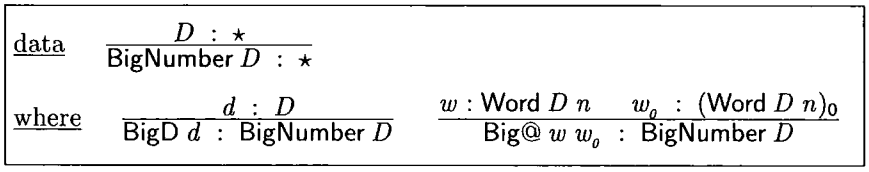


Figure 5.14: Big Number declaration

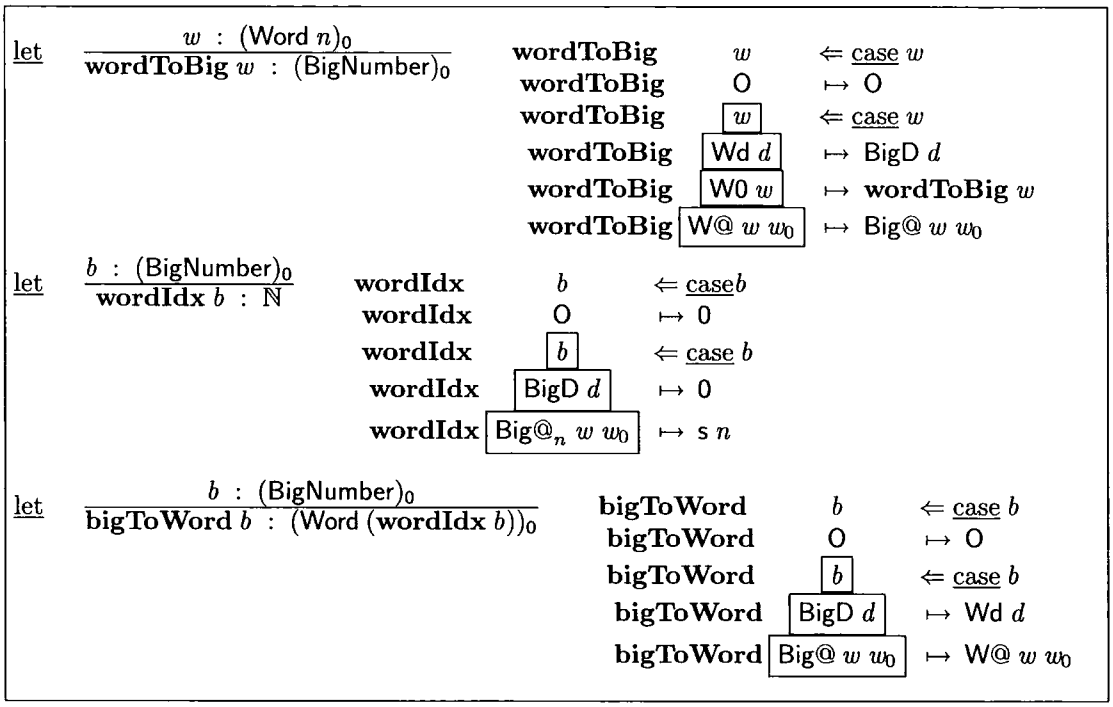


Figure 5.15: Mapping between BigNumber and Word

Arithmetic operations on BigNumber are implemented in terms of the Word functions. Some manipulation of indices is required so that both arguments are in the same branch of the Word family, and to deal with possible carry flags (in the case of addition) and accumulators (in the case of multiplication).

The basic pattern for arithmetic on two numbers x : (Word n)$_0$ and y : (Word m)$_0$ is to compare the indices m and n, and pad the smaller to the size of the larger, using the **pad** function (figure 5.16).

The type of **pad** (returning an index of **plus** p n) allows it to be used in conjunction with **max** (written by view **compare**) and **compare**. The definition of **max** is given in figure 5.17.

Then the helper function **adcBig**, which adds two differently sized Words, returning a Word of the larger size and a carry flag, can be defined by the same pattern of recursion, as

$$\underline{\text{let}}\quad \frac{w : (\mathsf{Word}\ n)_0 \qquad p : \mathbb{N}}{\mathbf{pad}\ w\ p\ :\ (\mathsf{Word}\ (\mathbf{plus}\ p\ n))_0}$$

$$\begin{array}{lll} \mathbf{pad}\ w & p & \Leftarrow \underline{\text{elim}}\ p \\ \mathbf{pad}\ w & 0 & \mapsto w \\ \mathbf{pad}\ w & (\mathsf{s}\ n) & \mapsto \mathsf{O}|(\mathbf{pad}\ w\ n) \end{array}$$

Figure 5.16: The **pad** function

$$\underline{\text{let}}\quad \frac{x, y : \mathbb{N}}{\mathbf{max}\ x\ y\ :\ \mathbb{N}}$$

$$\begin{array}{llll} \mathbf{max} & x & y & \Leftarrow \underline{\text{view}}\ \mathbf{compare}\ x\ y \\ \mathbf{max} & x & (\mathbf{plus}\ (\mathsf{s}\ y)\ x) & \mapsto \mathbf{plus}\ (\mathsf{s}\ y)\ x \\ \mathbf{max} & x & x & \mapsto x \\ \mathbf{max} & (\mathbf{plus}\ (\mathsf{s}\ x)\ y) & y & \mapsto \mathbf{plus}\ (\mathsf{s}\ x)\ y \end{array}$$

Figure 5.17: The **max** function

in figure 5.18.

$$\underline{\text{let}}\quad \frac{w_1 : (\mathsf{Word}\ n)_0 \qquad w_2 : (\mathsf{Word}\ m)_0 \qquad z_{in} : \mathsf{Carry}}{\mathbf{adcBig}\ :\ \mathsf{Carry} \times (\mathsf{Word}\ (\mathbf{max}\ n\ m))_0}$$

$$\begin{array}{llllll} \mathbf{adcBig} & n & w_1 & m & w_2\ z_{in} & \Leftarrow \underline{\text{view}}\ \mathbf{compare}\ n\ m \\ \mathbf{adcBig} & n & w_1\ (\mathbf{plus}\ (\mathsf{s}\ y)\ n) & & w_2\ z_{in} & \mapsto \mathbf{adc}\ (\mathbf{pad}\ w_1\ (\mathsf{s}\ y))\ w_2\ z_{in} \\ \mathbf{adcBig} & n & w_1 & n & w_2\ z_{in} & \mapsto \mathbf{adc}\ w_1\ w_2\ z_{in} \\ \mathbf{adcBig} & (\mathbf{plus}\ (\mathsf{s}\ x)\ n) & w_1 & m & w_2\ z_{in} & \mapsto \mathbf{adc}\ w_1\ (\mathbf{pad}\ w_2\ (\mathsf{s}\ x))\ z_{in} \end{array}$$

Figure 5.18: Adding two Words of different size

Finally, we write a function to convert the BigNumbers into Words, do the arithmetic, then convert back again. This function (in figure 5.19) also has to deal with any possible carry resulting from the addition and resize the BigNumber accordingly. **one** is a helper function which builds a Word n representing the number one, with appropriate index n.

5.2.8 Discussion

The BigNumber type gives us a method for computation with large numbers in type theory. Its advantages for arithmetic become more noticeable as numbers get larger; with small numbers there are overheads in constructing the data structure and the arithmetic operations are more complex than those of $\mathbb{N}$. We cannot replace $\mathbb{N}$ entirely with BigNumber however. Firstly, it is indexed over $\mathbb{N}$ so it does not make sense to remove $\mathbb{N}$ entirely. Secondly, the elimination rule for BigNumber does not give the same primitive recursion behaviour as that of $\mathbb{N}$. While it is possible to build such an induction principle, it relies on an isomorphism between BigNumber and $\mathbb{N}$ and conversion between the two structures; doing this means that we still have to use $\mathbb{N}$ as an intermediate structure and so the space advantages are lost. BigNumber is only really useful as an implementation of big number arithmetic.

$$
\begin{array}{l}
\underline{\text{let}} \quad \dfrac{a, b \ : \ (\mathsf{BigNumber})_0}{\mathbf{addBig} \ : \ (\mathsf{BigNumber})_0 \to (\mathsf{BigNumber})_0 \to (\mathsf{BigNumber})_0} \\
\quad \mathbf{addBig}\ a\ b \mid p \leftarrow \mathbf{adcBig}\ (\mathbf{bigToWord}\ a)\ (\mathbf{bigToWord}\ b) \ \Leftarrow \underline{\text{case}}\ p \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (z, w) \ \Leftarrow \underline{\text{case}}\ z \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{no}, w) \ \mapsto \mathbf{wordToBig}\ w \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{yes}, w) \ \mapsto \mathbf{one}|(\mathbf{wordToBig}\ w)
\end{array}
$$

Figure 5.19: Adding two BigNumbers

A further problem with BigNumber is the difficulty of implementing division. Addition and multiplication work well in the framework of two digit numbers, but there is no obvious way to implement division in terms of division on numbers with a smaller index, except by repeated subtraction.

5.3 External Implementation of $\mathbb{N}$

BigNumber's disadvantages are that it is less efficient than $\mathbb{N}$ for computation with small numbers and that there is no direct primitive recursion behaviour matching that of $\mathbb{N}$. We now consider an alternative approach to number representation, using an external library to implement unbounded numbers. GMP, the GNU Multi-precision arithmetic library [G+04] is one such library; some implementations of Haskell `Integers` (for example, in GHC and the Haskell B Compiler) use GMP. The issue with an external library is whether we can trust uncertified external code to be called from a certified core. We certainly have no reason *not* to trust GMP as a faithful implementation of unbounded numbers. In particular, as a well used library, a lot of software would fail if there were errors. Aside from this, there is research taking place into proving the correctness of features of GMP [BMZ02]. If we take GMP to be a trusted external oracle, what are the steps involved in compiling TT terms which use $\mathbb{N}$ into RunTT terms which use GMP?

To answer this question, consider how $\mathbb{N}$ is used. It has constructors and an elimination rule, so naturally these will need to be translated into the new setting. The ι-schemes of the elimination rule are implemented in terms of pattern matching on $\mathbb{N}$, so we will need to consider pattern matching on the new representation. Finally, in considering pattern matching, we should bear in mind that detagged and collapsible families may also have elimination rules implemented by pattern matching on $\mathbb{N}$s.

5.3.1 Construction of $\mathbb{N}$s

For constructing $\mathbb{N}$s in the RunTT setting, I introduce integer literals and an addition operator into ExTT and correspondingly into the supercombinator language. We introduce these at the intermediate level of ExTT rather than into RunTT because of the need to

pattern match on the new representation in the compilation of elimination rules; the rules for some detagged families in particular may need to match on $\mathbb{N}$. We add features to ExTT to construct $\mathbb{N}$s in this form, and to manipulate them, shown in figure 5.20.

$t ::=$	...	
	i	(Integer literal)
	$t\ op\ t$	(Arithmetic operator)
	$t\ cmp\ t$	(Comparison operator)
	$\underline{\text{if}}\ t\ \underline{\text{then}}\ t\ \underline{\text{else}}\ t$	(Integer testing)
$op ::=$	$+ \mid - \mid *$	
$cmp ::=$	$< \mid == \mid >$	

Figure 5.20: Additions to ExTT for external implementation of $\mathbb{N}$

This allows the following simple translation on ExTT terms:

$[\![\mathsf{0}]\!] \Longrightarrow 0$
$[\![\mathsf{s}\langle n\rangle]\!] \Longrightarrow [\![n]\!] + 1$

Any repeated successor applications (e.g., $\mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ k))$) results in multiple additions. A simple constant folding optimisation removes this. For example, $[\![\mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ k))]\!] \Longrightarrow k + 3$.

Note, however, that RunTT terms arising from elimination rules are, as usual, treated differently. This is necessary, since pattern matching on integers and pattern matching on inductive families are implemented in very different ways — a simple transformation on RunTT $\underline{\text{case}}$ expressions is not sufficient to cover this.

There is, perhaps, a worry about preserving type correctness here. Since the transformation occurs only on well typed terms, and all $\mathbb{N}$s are converted to integers (by observing that each constructor is mapped to an integer), we need not be concerned that correctness is compromised.

Boxing and Unboxing

In the current implementation, integers are given a **boxed** representation; i.e., they are stored on the heap as a reference to the integer, rather than the integer itself. This is because GMP uses a boxed representation; it is not the case that an arbitrary integer can fit into a single machine word.

Nevertheless, there is much to be gained from considering how to avoid boxing and unboxing where possible. GHC includes unboxed values as first class values [PL91a] which aids strictness analysis and allows unboxed values to be used as part of algebraic data structures.

While an advantage of boxing values is to give a uniform representation to data which aids in the compilation of polymorphic functions (in that only one version need be compiled, rather than separate version for instantiation with integers, characters, booleans etc), this

does mean that instantiating the function with a primitive type can be needlessly inefficient. In [HM95], Harper and Morrisett describe a technique for run-time type analysis which allows separate compilation of boxed and unboxed versions of polymorphic functions. To apply this technique in EPIGRAM would require the addition of a casetype operator at the RunTT level, and would necessitate the storing of some type information on the heap (rather than merely storing a TYPE node as we do currently), but the benefits from avoiding boxing may be enough to make this worthwhile.

5.3.2 Elimination and Pattern Matching

If we are to write the ι-schemes for the new implementation of $\mathbb{N}$, we will need additions to the pattern syntax which allow for matching on GMP integers. These extensions are shown in figure 5.21.

$p ::=$	$\ldots$	
	$\mid k$	(Integer literal pattern)
	$\mid x + k$	(Non zero variable)

Figure 5.21: Extensions to the pattern syntax for external implementation of $\mathbb{N}$

A similar transformation is applied to patterns as that which is applied to ExTT terms. Constant folding is applied on repeated applications of s so that the resulting pattern conforms to the syntax. The pattern transformation is as follows:

$[\![0]\!] \Longrightarrow 0$

$[\![\mathsf{s}\ n]\!] \Longrightarrow [\![n]\!] + 1$

To implement matching on these patterns, and so that we can ultimately take advantage of the GMP external implementation, I add further operations to RunTT which allow inspection and manipulation of integers, corresponding to the extensions to ExTT. The full extensions to RunTT are shown in figure 5.22.

$e ::=$	$\ldots$	
	$\mid i$	(Integer literal)
	$\mid e\ op\ e$	(Arithmetic operator)
	$\mid e\ cmp\ e$	(Comparison operator)
	$\mid$ if e then e else e	(Integer testing)
$op ::=$	$+ \mid - \mid *$	
$cmp ::=$	$< \mid == \mid >$	

Figure 5.22: Extensions to RunTT for external implementation of $\mathbb{N}$

The semantics of if are straightforward; if the expression being tested (a simple boolean comparison) is true, evaluate the then branch, otherwise evaluate the else branch.

The two new cases for the pattern syntax are handled by extra cases for the PROJECT' operation, shown in figure 5.23. Recall from section 4.4.2 on page 110 that PROJECT computes terms for projecting the values of arguments from patterns, with PROJECT' as a helper operation.

$$\begin{array}{|ll|} \hline \text{PROJECT'}(n, f, \quad k \quad) & = [] \\ \text{PROJECT'}(n, f, (x + k)) & = [x, (f\ n) - k] \\ \hline \end{array}$$

Figure 5.23: Extra cases to PROJECT'

For an $x + k$ pattern, if we know it matches, x is retrieved simply by subtracting k from the argument n.

To compile pattern matching in this form, two cases are added to the pattern matching compiler scheme $\mathcal{I}$. Recall (from page 111) that $\mathcal{I}$ examines the patterns $p_{11} \ldots p_{1n}$, which represent the patterns for the first argument e_1, to establish whether case distinction can be made on e_1. These additional cases are summarised in figure 5.24.

Case 5 Two possibilities, with $p_{a1} = 0$ and $p_{b1} = x + k$, where $a, b \in \{1, 2\}$

$$\mathcal{I}(e_1 \ldots e_i, \begin{Bmatrix} 0 & p_{a2} \ldots p_{ai} & \rightsquigarrow & x_a \\ (x+k) & p_{b2} \ldots p_{bi} & \rightsquigarrow & x_b \end{Bmatrix}) \Longrightarrow \underline{\text{if}}\ e_1 == 0\ \underline{\text{then}}\ x_a\ \underline{\text{else}}\ x_b$$

Case 6 $p_{i1} = 0$ for some i, and $p_{j1} = x + k$ for some j

Take P to be the smallest set of patterns such that $p_{i1} \in P$ if $p_{i1} = 0$ or $p_{i1} = x + k$ for some constant k. Then:

$$\mathcal{I}(e_1 \ldots e_i, \begin{Bmatrix} p_{11} \ldots p_{1i} & \rightsquigarrow & x_1 \\ \ldots & & \\ p_{n1} \ldots p_{ni} & \rightsquigarrow & x_n \end{Bmatrix}) \Longrightarrow$$

$$\underline{\text{if}}\ e_1 == 0 \quad \underline{\text{then}}\ \mathcal{I}(e_2 \ldots e_n, \begin{Bmatrix} p_{k2} \ldots p_{ki} & \rightsquigarrow & x_1 \\ \ldots & & \end{Bmatrix}) \quad [\forall k.\ p_{k1} \notin P \text{ or } p_{k1} = 0]$$

$$\underline{\text{else}}\ \mathcal{I}(e_2 \ldots e_n, \begin{Bmatrix} p_{k2} \ldots p_{ki} & \rightsquigarrow & x_1 \\ \ldots & & \end{Bmatrix}) \quad [\forall k.\ p_{k1} \neq 0]$$

Figure 5.24: Extra cases of $\mathcal{I}$

Case 5: $p_{a1} = 0$ and $p_{b1} = x + k$, where $a, b \in \{1, 2\}$ and $n = 2$

This is a special case for integers corresponding to case 2 for constructor patterns. Case distinction can be made on this argument alone. If $e_1 = 0$, we evaluate case a, otherwise we evaluate case b. The RunTT case expression is built as follows:

$$\mathcal{I}(e_1 \ldots e_i, \begin{Bmatrix} 0 & p_{a2} \ldots p_{ai} & \rightsquigarrow & x_a \\ (x+k) & p_{b2} \ldots p_{bi} & \rightsquigarrow & x_b \end{Bmatrix}) \Longrightarrow \underline{\text{if}}\ e_1 == 0\ \underline{\text{then}}\ x_a\ \underline{\text{else}}\ x_b$$

It is not necessary for the zero case to appear first; the cases can be in either order.

Case 6: $p_{i1} = 0$ for some i, and $p_{j1} = x + k$ for some j

This is a special case of the compiler for integers, corresponding to case 4, where two or more of $p_{11} \dots p_{n1}$ are headed by disjoint constructors. We take P to be the smallest set of patterns such that $p_{i1} \in P$ if $p_{i1} = 0$ or $p_{i1} = x + k$ for some constant k.

Then the RunTT expression is built as follows:

$$\mathcal{I}(e_1 \dots e_i, \left\{\begin{array}{lcl} p_{11} \cdots p_{1i} & \rightsquigarrow & x_1 \\ \dots & & \\ p_{n1} \cdots p_{ni} & \rightsquigarrow & x_n \end{array}\right\}) \Longrightarrow$$

$$\underline{\text{if}}\ e_1 == 0 \quad \underline{\text{then}}\ \mathcal{I}(e_2 \dots e_n, \left\{\begin{array}{lcl} p_{k2} \cdots p_{ki} & \rightsquigarrow & x_1 \\ \dots & & \end{array}\right\}) \quad [\forall k.\ p_{k1} \notin P \text{ or } p_{k1} = 0]$$

$$\underline{\text{else}}\ \mathcal{I}(e_2 \dots e_n, \left\{\begin{array}{lcl} p_{k2} \cdots p_{ki} & \rightsquigarrow & x_1 \\ \dots & & \end{array}\right\}) \quad [\forall k.\ p_{k1} \neq 0]$$

Example — ℕ-Elim

With this new representation, how is elimination of ℕs compiled into RunTT?

$$\mathbb{N}\textbf{-Elim}\ \ 0\ \ P\ m_0\ m_\mathsf{s} \ \rightsquigarrow\ m_0$$
$$\mathbb{N}\textbf{-Elim}\ (\mathsf{s}\ k)\ P\ m_0\ m_\mathsf{s} \ \rightsquigarrow\ m_\mathsf{s}\ k\ (\mathbb{N}\textbf{-Elim}\ k\ P\ m_0\ m_\mathsf{s})$$

The transformation to the integer representation gives us this rule to compile into RunTT:

$$\mathbb{N}\textbf{-Elim}\ \ 0\ \ P\ m_0\ m_\mathsf{s} \ \rightsquigarrow\ m_0$$
$$\mathbb{N}\textbf{-Elim}\ (k+1)\ P\ m_0\ m_\mathsf{s} \ \rightsquigarrow\ m_\mathsf{s}\ k\ (\mathbb{N}\textbf{-Elim}\ k\ P\ m_0\ m_\mathsf{s})$$

For the second case, applying PROJECT to the first argument (let us call this argument n) yields:

$$\text{PROJECT}(n, (k+1)) \Longrightarrow [(k, n-1)]$$

Examining the patterns for the first argument, we see that case 5 applies. The term in RunTT is therefore a straightforward if expression:

$$\mathbb{N}\textbf{-Elim} \ \mapsto\ \lambda n;\ P;\ m_0;\ m_\mathsf{s}.$$
$$\underline{\text{if}}\ n == 0\ \underline{\text{then}}\ m_0\ \underline{\text{else}}\ m_\mathsf{s}\ (n-1)\ (\mathbb{N}\textbf{-Elim}\ (n-1)\ P\ m_0\ m_{suc})$$

Aside — recursion and iteration

If reduction order does not matter, which it does not when termination is guaranteed, we might consider an alternative implementation of **ℕ-Elim** which is iterative rather than recursive. This relies on some additional notation for RunTT (for which I will not give a formal treatment); we add assignment to a mutable variable ($x := t$), explicit sequencing (indicated

by separating expressions with a semicolon) and a while loop for bounded iteration. With this additional notation, we can write an iterative version of ℕ-**Elim**:

$$
\begin{aligned}
&\mathbb{N}\text{-}\mathbf{Elim} \mapsto \lambda n;\ P;\ m_0;\ m_\mathsf{s}.\\
&\qquad\qquad acc := m_0;\\
&\qquad\qquad k := 0;\\
&\qquad\qquad \underline{\text{while}}\ k < n\\
&\qquad\qquad\quad acc := m_\mathsf{s}\ k\ acc;\\
&\qquad\qquad\quad k := k + 1\\
&\qquad\qquad \underline{\text{return}}\ acc
\end{aligned}
$$

This clearly has the same behaviour as the original ℕ-**Elim**, but without the overhead of building a thunk for the recursive call. The locally bound k and acc are reused, although this requires that m_s is evaluated eagerly — hence the requirement that termination is guaranteed. We also need to be careful in the case where the successor case does not make a recursive call. Unfortunately, this does not generalise; we can only do this because ℕ holds no data other than its own size. Nor can we build this function directly from the pattern matching representation of ℕ-**Elim** in ExTT. However, it may be worth hard-coding elimination rules such as this since optimising an elimination rule optimises those programs which are written in terms of it; a future research direction could potentially involve the identification of efficient (tail-recursive or iterative) elimination rules.

Example — between-**Elim**

Recall the between type from Chapter 4 which represents a proof that $m \leq n \leq p$:

$$
\underline{\text{data}}\quad \frac{m, n, p\ :\ \mathbb{N}}{\mathsf{between}\ m\ n\ p\ :\ \star}
$$

$$
\underline{\text{where}}\quad \frac{}{\mathsf{bO}\ :\ \mathsf{between}\ 0\ 0\ 0} \qquad \frac{b\ :\ \mathsf{between}\ 0\ 0\ m}{\mathsf{bOOs}\ b\ :\ \mathsf{between}\ 0\ 0\ (\mathsf{s}\ m)}
$$

$$
\frac{b\ :\ \mathsf{between}\ 0\ m\ n}{\mathsf{b0ss}\ b\ :\ \mathsf{between}\ 0\ (\mathsf{s}\ m)\ (\mathsf{s}\ n)} \qquad \frac{b\ :\ \mathsf{between}\ m\ n\ p}{\mathsf{bsss}\ b\ :\ \mathsf{between}\ (\mathsf{s}\ m)\ (\mathsf{s}\ n)\ (\mathsf{s}\ p)}
$$

The collapsing of this relation along with the translation to GMP integers gives ι-schemes as in figure 5.25. Applying the $\mathcal{I}$ compilation scheme, which repeatedly applies case 6, yields the supercombinator definition shown in figure 5.26. As before instances passed to the methods are replaced with the trivial canonical empty tuple, $\langle\rangle$.

5.3.3 Homomorphisms with ℕ

So far with this integer representation, we have managed to convert the TT representation into an efficient run-time representation. The only improvement we really have, though, is space compression — all functions are ultimately still written by structural induction over ℕ, using ℕ-**Elim**. In order to really take advantage of this efficient representation, we

$$
\begin{array}{lllllll}
\textsf{between-}\mathbf{Elim} & 0 & 0 & 0 & \{\textsf{bO}\} & P\ m_{\textsf{bO}}\ m_{\textsf{bOOs}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}} & \rightsquigarrow\ m_{\textsf{bO}} \\
\textsf{between-}\mathbf{Elim} & 0 & 0 & (\textsf{s}\ m) & \{\textsf{bOOs}\ m\ b\} & P\ m_{\textsf{bO}}\ m_{\textsf{bOOs}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}} & \\
\multicolumn{7}{l}{\quad \rightsquigarrow\ m_{\textsf{bOOs}}\ m\ (\{b\})\ (\textsf{between-}\mathbf{Elim}\ 0\ 0\ m\ \{b\}\ P\ m_{\textsf{bO}}\ m_{\textsf{bOOs}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}})} \\
\textsf{between-}\mathbf{Elim} & 0 & (\textsf{s}\ m) & (\textsf{s}\ n) & \{\textsf{b0ss}\ m\ n\ b\} & P\ m_{\textsf{bO}}\ m_{\textsf{bOOs}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}} & \\
\multicolumn{7}{l}{\quad \rightsquigarrow\ m_{\textsf{b0ss}}\ m\ n\ (\{b\})\ (\textsf{between-}\mathbf{Elim}\ 0\ m\ n\ \{b\}\ P\ m_{\textsf{bO}}\ m_{\textsf{bOOs}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}})} \\
\textsf{between-}\mathbf{Elim} & (\textsf{s}\ m) & (\textsf{s}\ n) & (\textsf{s}\ p) & \{\textsf{bsss}\ m\ n\ p\ b\} & P\ m_{\textsf{bO}}\ m_{\textsf{bOOs}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}} & \\
\multicolumn{7}{l}{\quad \rightsquigarrow\ m_{\textsf{bsss}}\ m\ n\ p\ (\{b\})\ (\textsf{between-}\mathbf{Elim}\ m\ n\ p\ \{b\}\ P\ m_{\textsf{bO}}\ m_{\textsf{bOOs}}\ m_{\textsf{b0ss}}\ m_{\textsf{bsss}})}
\end{array}
$$

Figure 5.25: ι-schemes for between-**Elim** with GMP

$$
\begin{array}{l}
\textsf{between-}\mathbf{Elim} \mapsto\ \lambda m;\ n;\ p;\ P;\ m_{\textsf{bO}};\ m_{\textsf{bOOs}};\ m_{\textsf{b0ss}};\ m_{\textsf{bsss}}. \\
\quad \underline{\text{if}}\ m == 0 \\
\quad\quad \underline{\text{then if}}\ n == 0 \\
\quad\quad\quad\quad \underline{\text{then if}}\ p == 0 \\
\quad\quad\quad\quad\quad\quad \underline{\text{then}}\ m_{\textsf{bO}} \\
\quad\quad\quad\quad\quad\quad \underline{\text{else}}\ m_{\textsf{bOOs}}\ (p-1)\ \langle\rangle\ \ldots \\
\quad\quad\quad\quad \underline{\text{else}}\ m_{\textsf{b0ss}}\ (n-1)\ (p-1)\ \langle\rangle\ \ldots \\
\quad\quad \underline{\text{else}}\ m_{\textsf{bsss}}\ (m-1)\ (n-1)\ (p-1)\ \langle\rangle\ \ldots
\end{array}
$$

Figure 5.26: Compiled ι-schemes for between-**Elim**

would like to use the arithmetic operations provided by the GMP library rather than the TT definitions.

I will consider three basic functions; **plus**, **mult** and **compare**. I consider **compare** to be an important function to optimise, if not a primitive, since it implements an ordering and subtraction on Ns at the same time. Not only this, but as **compare** n m has linear complexity for what is essentially subtraction, a more efficient implementation would be beneficial.

We write functions on GMP integers in ExTT corresponding to the N based definitions. **plus** and **mult** have corresponding implementations in ExTT defined using primitive operators as follows:

$$\mathbf{plusInt} \mapsto \lambda n;\ m.\ n + m$$
$$\mathbf{multInt} \mapsto \lambda n;\ m.\ n * m$$

To use these definitions in place of the TT definitions, the following transformations are applied during the transformation from TT to ExTT:

$$[\![\mathbf{plus}]\!] \Longrightarrow \mathbf{plusInt}$$
$$[\![\mathbf{mult}]\!] \Longrightarrow \mathbf{multInt}$$

As an additional optimisation, where these functions are fully applied the definitions can be unfolded. Hence **plusInt** x y becomes simply $x + y$. **compare** is slightly more difficult; for one thing, it must take into account the erasure of forced arguments in the Compare

family. As a result, the ExTT definition is similarly marked up. Its marked TT definition is as follows:

$$\begin{array}{l}\textbf{compareInt} \mapsto \lambda n;\ m. \\ \quad \underline{\text{if}}\ n < m\ \underline{\text{then}}\ \mathsf{lt}\ \{n\}\ (m - n - 1) \\ \quad\quad \underline{\text{else}}\ \underline{\text{if}}\ n == m\ \underline{\text{then}}\ \mathsf{eq}\ \{n\} \\ \quad\quad\quad \underline{\text{else}}\ \mathsf{gt}\ \{m\}\ (n - m - 1)\end{array}$$

Then a similar transformation as before is used in the translation phase from TT to ExTT. We do not automatically unfold **compareInt** as with **plusInt** and **multInt** as the definition is rather larger.

$$[\![\textbf{compare}]\!] \Longrightarrow \textbf{compareInt}$$

5.3.4 Typechecking the External Implementation

Since the GMP integers are added in ExTT, it is worth considering how the addition of GMP integers affects the typechecking algorithm. The conversion check, conceptually, involves the checking for syntactic equality of normal forms. For the external implementation of $\mathbb{N}$ we can use the equality defined by GMP to check the syntactic equality of the $\mathbb{N}$s:

$$\frac{x, y : \mathbb{N} \quad x =_{\text{GMP}} y}{x \equiv y}$$

Hence if two GMP-implemented $\mathbb{N}$s x and y are equal by a GMP equality test, then they are convertible. We are not attempting to reason about the conversion to GMP here — the use of this rule implies that we trust the correctness of GMP's implementation of equality.

There is a problem, however, with typechecking the external GMP implementation of $\mathbb{N}$; namely that the conversion rules which previously held for **plus**, **mult** and **compare** do not hold for **plusInt**, **multInt** and **compareInt**. For example, the definition of **plus** gives two rules for the conversion checker (these rules arise from the direct reduction behaviour of **plus** when the first argument is in canonical form):

$$\begin{array}{l}\textbf{plus}\ \mathsf{0}\ m \simeq m \\ \textbf{plus}\ (\mathsf{s}\ k)\ m \simeq \mathsf{s}\ (\textbf{plus}\ k\ m)\end{array}$$

Similar rules do not hold for **plusInt** because the reduction behaviour of the + operator is defined externally. The solution adopted by [MB01] is to make these conversion rules explicit. To do this, we can define the following axioms describing the external behaviour of GMP $\mathbb{N}$s:

$$\begin{array}{l}\textbf{plus0} : \forall n{:}\mathbb{N}.\ \textbf{plusInt}\ \mathsf{0}\ m = m \\ \textbf{pluss} : \forall n, m{:}\mathbb{N}.\ \textbf{plusInt}\ (k+1)\ m = (\textbf{plusInt}\ k\ m) + 1\end{array}$$

These type isomorphisms (whose run-time implementations are effectively the identity function) are inserted by the typechecker where they transform a term's actual type into its

expected type, using the algorithm from [MB01]. In the current implementation, however, the typechecker uses the naïve representation of $\mathbb{N}$ for typechecking, only transforming for compilation. This is acceptable for many programs with limited type level computation, however a future implementation will also transform to an efficient implementation of $\mathbb{N}$ for compile-time execution.

5.3.5 Extensions to the G-machine

The new RunTT operations for manipulating GMP integers will clearly need to be translated to primitive operations in the G-machine. Johnsson's G-machine [Joh84] has a **value stack** of basic values for storing intermediate values in primitive types. Until now, we have considered the G-machine to be a 5-tuple (see section 3.5.2); now I add a value stack, so that the G-machine state is a 6-tuple $\langle C, S, V, G, E, D \rangle$. The primitive values are big numbers and boolean values (arising from comparisons on big numbers.) The idea behind the value stack is that it avoids building graphs from intermediate computations — a value is only transferred onto the main stack when a computation is complete. This is similar to the approach adopted by the STG machine.

There is a new graph node, BIGINT i, where i is an integer represented by GMP, and a graph node BOOL b where b is a boolean value. Also, I add new instructions for manipulation of big numbers and booleans on the value stack. Values on the value stack are either big integers or booleans, i or b. These instructions are defined as follows:

- PUSHBIG i constructs a graph BIGINT i and pushes it onto the stack S. There is no equivalent for booleans, since we do not have boolean literals in RunTT; they arise only from comparisons.
- PUSHINT i pushes the value i onto the value stack V.
- PUSHBOOL b pushes the value b onto the value stack V.
- GET retrieves the integer from the graph at the top of the stack (which must be a BIGINT i) and pushes i onto the value stack V.
- MKINT pushes the integer at the top of the value stack V onto the stack S (the opposite of GET.)
- MKBOOL pushes the boolean at the top of the value stack V onto the stack S.
- ADD, SUB and MULT apply the appropriate arithmetic operation to the top two values on the value stack V.
- LT, EQ and GT apply the appropriate boolean comparison to the top two values on the value stack V.

- JTRUE l examines the value on the top of the value stack and jumps to the label l if the value is a boolean "true".

The state transition rules for these instructions are given in figure 5.27.

$$
\begin{array}{lcl}
\langle \mathsf{PUSHBIG}\ i;\ c, S, V, G, E, D\rangle & \Longrightarrow & \langle c, n.S, V, G[n = \mathsf{BIGINT}\ i], E, D\rangle \\
\langle \mathsf{PUSHINT}\ i;\ c, S, V, G, E, D\rangle & \Longrightarrow & \langle c, S, i.V, G, E, D\rangle \\
\langle \mathsf{PUSHBOOL}\ b;\ c, S, V, G, E, D\rangle & \Longrightarrow & \langle c, S, b.V, G, E, D\rangle \\
\langle \mathsf{GET};\ c, n.S, V, G[n = \mathsf{BIGINT}\ i], E, D\rangle & \Longrightarrow & \langle c, S, i.V, G, E, D\rangle \\
\langle \mathsf{MKINT};\ c, S, i.V, G, E, D\rangle & \Longrightarrow & \langle c, n.S, V, G[n = \mathsf{BIGINT}\ i], E, D\rangle \\
\langle \mathsf{MKBOOL};\ c, S, b.V, G, E, D\rangle & \Longrightarrow & \langle c, n.S, V, G[n = \mathsf{BOOL}\ b], E, D\rangle \\
\langle \mathsf{ADD};\ c, S, x.y.V, G, E, D\rangle & \Longrightarrow & \langle c, S, x + y.V, G, E, D\rangle \\
\langle \mathsf{SUB};\ c, S, x.y.V, G, E, D\rangle & \Longrightarrow & \langle c, S, x - y.V, G, E, D\rangle \\
\langle \mathsf{MULT};\ c, S, x.y.V, G, E, D\rangle & \Longrightarrow & \langle c, S, x * y.V, G, E, D\rangle \\
\langle \mathsf{LT};\ c, S, x.y.V, G, E, D\rangle & \Longrightarrow & \langle c, S, x < y.V, G, E, D\rangle \\
\langle \mathsf{EQ};\ c, S, x.y.V, G, E, D\rangle & \Longrightarrow & \langle c, S, x = y.V, G, E, D\rangle \\
\langle \mathsf{GT};\ c, S, x.y.V, G, E, D\rangle & \Longrightarrow & \langle c, S, x > y.V, G, E, D\rangle \\
\langle \mathsf{JTRUE}\ l;\ c, S, \mathsf{true}.V, G, E, D\rangle & \Longrightarrow & \langle \mathsf{JUMP}\ l;\ c, S, V, G, E, D\rangle \\
\langle \mathsf{JTRUE}\ l;\ c, S, \mathsf{false}.V, G, E, D\rangle & \Longrightarrow & \langle c, S, V, G, E, D\rangle
\end{array}
$$

Figure 5.27: State transitions for computing basic values

5.3.6 Compilation Scheme

A new compilation scheme, $\mathcal{B}[\![\cdot]\!]$, compiles expressions of basic values. This scheme compiles code to put an expression on the value stack rather than the main stack. Since the value stack consists only of integers and booleans the resulting code will be more efficient than manipulations on a stack of graphs. Compiling with this scheme effectively implements the unboxing of integers and booleans for compilation of complex expressions, then boxing the result. The $\mathcal{B}[\![\cdot]\!]$ scheme is given in figure 5.28.

$$
\begin{array}{l}
\mathcal{B}[\![i]\!]\ r\ n \Longrightarrow \mathsf{PUSHINT}\ i \\
\mathcal{B}[\![e_1 + e_2]\!]\ r\ n \Longrightarrow \mathcal{B}[\![e_1]\!]\ r\ n;\ \mathcal{B}[\![e_2]\!]\ r\ n;\ \mathsf{ADD} \\
\mathcal{B}[\![e_1 - e_2]\!]\ r\ n \Longrightarrow \mathcal{B}[\![e_1]\!]\ r\ n;\ \mathcal{B}[\![e_2]\!]\ r\ n;\ \mathsf{SUB} \\
\mathcal{B}[\![e_1 * e_2]\!]\ r\ n \Longrightarrow \mathcal{B}[\![e_1]\!]\ r\ n;\ \mathcal{B}[\![e_2]\!]\ r\ n;\ \mathsf{MULT} \\
\mathcal{B}[\![e_1 < e_2]\!]\ r\ n \Longrightarrow \mathcal{B}[\![e_1]\!]\ r\ n;\ \mathcal{B}[\![e_2]\!]\ r\ n;\ \mathsf{LT} \\
\mathcal{B}[\![e_1 = e_2]\!]\ r\ n \Longrightarrow \mathcal{B}[\![e_1]\!]\ r\ n;\ \mathcal{B}[\![e_2]\!]\ r\ n;\ \mathsf{EQ} \\
\mathcal{B}[\![e_1 > e_2]\!]\ r\ n \Longrightarrow \mathcal{B}[\![e_1]\!]\ r\ n;\ \mathcal{B}[\![e_2]\!]\ r\ n;\ \mathsf{GT}
\end{array}
$$

Figure 5.28: The $\mathcal{B}[\![\cdot]\!]$ compilation scheme

I extend the $\mathcal{E}[\![\cdot]\!]$ scheme to handle basic values, as in figure 5.29; top level expressions are passed through to the $\mathcal{B}[\![\cdot]\!]$ scheme and the result placed on the stack. The $\mathcal{E}[\![\cdot]\!]$ scheme

also handles if expressions. The only addition to the $\mathcal{C}[\![\cdot]\!]$ scheme is to construct graphs of primitive values (shown in figure 5.30.

$$
\begin{array}{l}
\mathcal{E}[\![e_1 + e_2]\!]\ r\ n \implies \mathcal{B}[\![e_1 + e_2]\!]\ r\ n;\ \mathsf{MKINT} \\
\mathcal{E}[\![e_1 - e_2]\!]\ r\ n \implies \mathcal{B}[\![e_1 - e_2]\!]\ r\ n;\ \mathsf{MKINT} \\
\mathcal{E}[\![e_1 * e_2]\!]\ r\ n \implies \mathcal{B}[\![e_1 * e_2]\!]\ r\ n;\ \mathsf{MKINT} \\
\mathcal{E}[\![e_1 < e_2]\!]\ r\ n \implies \mathcal{B}[\![e_1 < e_2]\!]\ r\ n;\ \mathsf{MKBOOL} \\
\mathcal{E}[\![e_1 = e_2]\!]\ r\ n \implies \mathcal{B}[\![e_1 = e_2]\!]\ r\ n;\ \mathsf{MKBOOL} \\
\mathcal{E}[\![e_1 > e_2]\!]\ r\ n \implies \mathcal{B}[\![e_1 > e_2]\!]\ r\ n;\ \mathsf{MKBOOL} \\
\mathcal{E}[\![\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3]\!] \implies \mathcal{B}[\![e_1]\!]\ r\ n;\ \mathsf{JTRUE}\ l_T; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{E}[\![e_3]\!]\ r\ n;\ \mathsf{JUMP}\ l; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{LABEL}\ l_T; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \mathcal{E}[\![e_2]\!]\ r\ n; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{LABEL}\ l \\
\mathcal{E}[\![i]\!]\ r\ n \implies \mathsf{PUSHBIG}\ i
\end{array}
$$

Figure 5.29: Extensions to the $\mathcal{E}[\![\cdot]\!]$ compilation scheme

$$
\mathcal{C}[\![i]\!]\ r\ n \implies \mathsf{PUSHBIG}\ i
$$

Figure 5.30: Extension to the $\mathcal{C}[\![\cdot]\!]$ compilation scheme

5.3.7 Example — Factorial Computation

A common example of a recursive function over the natural numbers is the factorial function ($n!$). The simplest way to write this in EPIGRAM is as in figure 5.31, following the usual rules that $0! = 1$ and $n! = n * (n-1)!$:

$$
\begin{array}{l}
\underline{\text{let}}\quad \dfrac{n\ :\ \mathbb{N}}{\mathbf{fact}\ n\ :\ \mathbb{N}} \\
\quad \mathbf{fact}\ \ 0 \quad\ \mapsto \mathsf{s0} \\
\quad \mathbf{fact}\ (\mathsf{s}\ k)\ \mapsto \mathbf{mult}\ (\mathsf{s}\ k)\ (\mathbf{fact}\ k)
\end{array}
$$

Figure 5.31: Factorial Function

The problem with writing a function over a natural number n is that it it very likely to have complexity of at least $O(n)$. With factorial, the problem is even greater as the size of numbers involved grows very rapidly. This is as much a problem of storage as speed — the unary representation requires nearly four million cells to store 10!. Let us nevertheless examine the compilation of the **fact** function as defined in figure 5.31. The elaborator produces the following definition in TT:

$$
\mathbf{fact} \mapsto \lambda n{:}\mathbb{N}.\ \mathbf{natElim}\ n\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ (\mathsf{s0})\ (\lambda k, ih{:}\mathbb{N}.\ \mathbf{mult}\ (\mathsf{s}\ k)\ ih)
$$

Beginning with a naïve approach and taking the above definition as the ExTT version of **fact**, we get the following straightforward translation to RunTT supercombinators:

$$\textbf{fact} \mapsto \lambda n.\ \textbf{natElim}\ n\ \textbf{fact1}\ \mathsf{s}\langle 0\rangle\ \textbf{fact2}$$
$$\textbf{fact1} \mapsto \lambda n.\ \mathbb{N}$$
$$\textbf{fact2} \mapsto \lambda k;\ ih.\ \textbf{mult}\ \mathsf{s}\langle k\rangle\ ih$$

These supercombinators compile to G-code as follows:

$$\begin{aligned}\mathcal{S}[\![\textbf{fact}]\!] \Longrightarrow\ & \mathsf{PUSHFUN}\ \mathbb{N}\textbf{-Elim};\ \mathsf{PUSH}\ 1;\ \mathsf{MKAP};\ \mathsf{PUSHFUN}\ \textbf{fact1};\ \mathsf{MKAP};\\ & \mathsf{MKCON}\ 0\ 0;\ \mathsf{MKCON}\ \mathsf{s}\ 1;\ \mathsf{MKAP};\ \mathsf{PUSHFUN}\ \textbf{fact2};\ \mathsf{MKAP};\\ & \mathsf{UPDATE}\ 2;\ \mathsf{RET}\ 1\\ \mathcal{S}[\![\textbf{fact1}]\!] \Longrightarrow\ & \mathsf{MKTYPE};\ \mathsf{UPDATE}\ 2;\ \mathsf{RET}\ 1\\ \mathcal{S}[\![\textbf{fact2}]\!] \Longrightarrow\ & \mathsf{PUSHFUN}\ \textbf{mult};\ \mathsf{PUSH}\ 2;\mathsf{MKCON}\ \mathsf{s}\ 1;\ \mathsf{MKAP};\\ & \mathsf{PUSH}\ 1;\ \mathsf{MKAP};\ \mathsf{UPDATE}\ 3;\ \mathsf{RET}\ 2\end{aligned}$$

The inefficiency in this definition is caused by the use of the $O(n)$ function **mult** to do the multiplication. If, instead, we apply the transformations of this chapter to replace $\mathbb{N}$ with an external GMP based representation then we have access to a fast multiplication function. The translations yield the following ExTT definition:

$$\textbf{fact} \mapsto \lambda n{:}\mathbb{N}.\ \textbf{natElim}\ n\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ 1\ (\lambda k, ih{:}\mathbb{N}.\ \textbf{multInt}\ (k+1)\ ih)$$

We can unfold the definition of **multInt** to get the following simpler definition:

$$\textbf{fact} \mapsto \lambda n{:}\mathbb{N}.\ \textbf{natElim}\ n\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ 1\ (\lambda k, ih{:}\mathbb{N}.\ (k+1) * ih)$$

There is now the following translation to RunTT supercombinators:

$$\textbf{fact} \mapsto \lambda n.\ \textbf{natElim}\ n\ \textbf{fact1}\ 1\ \textbf{fact2}$$
$$\textbf{fact1} \mapsto \lambda n.\ \mathbb{N}$$
$$\textbf{fact2} \mapsto \lambda k;\ ih.\ (k+1) * ih$$

Finally, we get the following G-code for these supercombinators:

$$\begin{aligned}\mathcal{S}[\![\textbf{fact}]\!] \Longrightarrow\ & \mathsf{PUSHFUN}\ \mathbb{N}\textbf{-Elim};\ \mathsf{PUSH}\ 1;\ \mathsf{MKAP};\ \mathsf{PUSHFUN}\ \textbf{fact1};\ \mathsf{MKAP};\\ & \mathsf{PUSHBIG}\ 1;\ \mathsf{PUSHFUN}\ \textbf{fact2};\ \mathsf{MKAP};\mathsf{UPDATE}\ 2;\ \mathsf{RET}\ 1\\ \mathcal{S}[\![\textbf{fact1}]\!] \Longrightarrow\ & \mathsf{MKTYPE};\ \mathsf{UPDATE}\ 2;\ \mathsf{RET}\ 1\\ \mathcal{S}[\![\textbf{fact2}]\!] \Longrightarrow\ & \mathsf{PUSH}\ 1;\ \mathsf{GET}\ ;\mathsf{PUSHBASIC}\ 1;\ \mathsf{ADD};\ \mathsf{PUSH}\ 0;\ \mathsf{GET};\\ & \mathsf{MULT};\ \mathsf{MKINT};\ \mathsf{UPDATE}\ 3;\ \mathsf{RET}\ 2\end{aligned}$$

The main difference between the two definitions is simply that the user defined **mult** function has been replaced by an efficient external representation encoded as a single G-machine instruction. The effects of this simple transformation, even on the small application of **fact** sss0, are large, and shown in figure 5.32.

Note again that these results are based only on the optimisation of $\mathbb{N}$. Other overheads, including the extra layer of abstraction imposed by the use of elimination rules and the outputting of results (via a show function again defined by an elimination rule) are also present, and we will see some optimisations for removing these in Chapter 6.

Program	Version	Instructions	Thunks	Memory Accesses	Cells
fact sss0	Naïve	118931	55372	40863	7871
	Optimised	4304	1382	1098	182
	Change	**-96.38%**	**-97.50%**	**-97.31%**	**-97.69%**

Figure 5.32: Run-time costs of the factorial function

5.3.8 Extending to Other Primitives

The approach to primitives we have taken here is rather different from the approach taken by GHC in the Spineless Tagless G-machine [Pey92]. In that system, the philosophy is that the machinery for implementing user defined types should be efficient enough to be usable for primitives such as lists. We have taken a different approach to $\mathbb{N}$ for two reasons:

- $\mathbb{N}$ is inherently inefficient, being a unary representation of numbers. Despite this, the structure has advantages at compile-time, so it makes sense to use $\mathbb{N}$ and transform it to an efficient representation.
- In general, we can expect to get a big performance improvement by optimising common operations and datatypes (similar to the RISC approach to computer architecture, where the philosophy is to choose a small highly optimised set of common instructions). We should not see this as imposing a performance penalty on user defined types, but rather making a performance gain on common primitive types.

Having implemented $\mathbb{N}$ externally, we might consider whether other primitives can be implemented externally. In some ways this follows Landin [Lan66], who suggests a family of languages (ISWIM) parametrised over the set of primitives they choose; the choice of primitives is based on the problem domain. An EPIGRAM data type has the following features which an external implementation may provide:

Constructors. In the case of $\mathbb{N}$, 0 was mapped to 0 and $\mathsf{s}\ n$ to $n + 1$.

Elimination Rule. An implementation of **$\mathbb{N}$-Elim** was built by determining the correct pattern matching behaviour for 0 and $n + k$ from the pattern matching behaviour of 0 and s.

Primitive Operations. **plus**, **mult** and **compare** were mapped to **plusInt**, **multInt** and **compareInt** respectively.

Conversion Rule. A conversion rule for $\mathbb{N}$ constants was given in terms of the GMP equality test.

So providing an external implementation of a datatype means giving EPIGRAM types to externally implemented operations. Not all of these make sense for every data type;

in particular floating point numbers have no obvious primitive recursion behaviour, for example, and even a conversion rule is difficult since there is always an error bound in a floating point calculation. A floating point implementation would be treated as an abstract data type, only providing constructor functions and primitive arithmetic operations. Further investigation of such details should be in the context of a module system for EPIGRAM, primitive types being provided by an external module.

Integers

Having considered an efficient implementation of the natural numbers, we should also consider how integers ($\mathbb{Z}$) might be represented in EPIGRAM. The approach taken by LEGO [LP92] is to treat integers as a pair of natural numbers, including a positive and negative component. This is a simple representation, but has the disadvantage that a single integer can be represented in an infinite number of ways. A pair of a natural number and its sign is another possible representation. The current COQ implementation, on the other hand, uses a more sophisticated representation based on binary, as described in [MB01].

However integers are described as an EPIGRAM type, it will of course be possible to apply the same techniques we have applied to $\mathbb{N}$ to give an efficient internal representation.

Multiple Return Values

Numbers are not the only thing which it is useful to treat as a primitive. In EPIGRAM, and in programming with inductive families in general, we often find it useful to return pairs (or larger tuples) of values. This is because values carry around invariants; a Vect is paired with its length, for example. Hence, if a function can return a different length Vect depending on its input, it needs to return the length along with the Vect using a Σ type. For example, we can write the vector filter function as follows:

$$\underline{\text{let}} \quad \frac{f \;:\; A \to \mathsf{Bool} \qquad xs \;:\; \mathsf{Vect}\; A\; n}{\mathbf{vfilter}\; f\; xs \;:\; \Sigma\, \mathbb{N}\, (\mathsf{Vect}\; A)}$$

$$\begin{array}{lll}
\mathbf{vfilter}\; f \quad xs & \Leftarrow \underline{\text{elim}}\; xs & \\
\mathbf{vfilter}\; f \quad \epsilon & \mapsto (?, \mathsf{nil}) & \\
\mathbf{vfilter}\; f\; (x{::}xs) & \mid\; b \leftarrow f\; x \quad \Leftarrow \underline{\text{case}}\; b & \\
 & \quad\; \mathsf{true} \quad \mid\; p \leftarrow \mathbf{vfilter}\; f\; xs \;\Leftarrow \underline{\text{case}}\; p & \\
 & \qquad\qquad\; (_, xs') \qquad \mapsto (?, x{::}xs') & \\
 & \quad\; \mathsf{false} \quad \mid\; p \leftarrow \mathbf{vfilter}\; f\; xs \;\Leftarrow \underline{\text{case}}\; p & \\
 & \qquad\qquad\; (_, xs') \qquad \mapsto (?, xs') &
\end{array}$$

Returning values along with their dependencies is a common programming idiom with dependent types, as we saw in section 2.3.1. For this reason, it may be beneficial to implement techniques for dealing with multiple return values; doing so has already been investigated for Standard ML [Mit94] and similar techniques can apply to EPIGRAM. Using C-- as a back-end has an advantage here, as it supports multiple return values in machine registers.

5.4 Correctness of External Implementation

One of the advantages of programming in EPIGRAM is that proofs of correctness can be given in the language itself. When we use an external implementation of some language feature however, as we have with $\mathbb{N}$, we cannot do this. The closest we can get to a correctness proof of the GMP representation is to model GMP integers in EPIGRAM and check the correctness of the model. This is not quite the same as a full correctness proof; any errors in modelling are caught but not necessarily errors in the implementation. It is still worth doing, however, for the following reasons:

- We can at least check the correctness of the algorithms and memory allocation, as with a recent proof of GMP square root [BMZ02, Mag03].
- As a longer term goal, a precise specification of GMP numbers and their associated operations may lead to the extraction of a more efficient implementation.

I do not propose to give a full model of GMP numbers and their operators here; to do so would be a very large project and a possible direction for future research. However, let us briefly consider how we might represent their structure as an inductive family in EPIGRAM.

5.4.1 Representing GMP integers

The internal representation of an integer is as a C struct with three fields [G+04]. These fields are:

- An array of **limbs**. Limbs correspond to digits; the D parameter of the Word family gives a concrete representation of limbs.
- An integer representing the number of limbs in the number. This integer is negative when representing a negative number. We only consider positive numbers for $\mathbb{N}$, so we will only consider positive integers in this field. This integer corresponds to the n index of the Word family.
- An integer representing the space allocated for limbs. If any operation causes the number to outgrow the space allocated for it, more space is allocated and this field changed accordingly.

Leaving aside the memory allocation issue for the moment, we might model integers in a GMP style as follows:

$$\underline{\text{data}} \quad \frac{D : \star \quad n : \mathbb{N}}{\mathsf{GMP}\ D\ n\ :\ \star} \quad \underline{\text{where}} \quad \frac{ls\ :\ \mathsf{Vect}\ D\ n}{\mathsf{mkGMP}\ ls\ :\ \mathsf{GMP}\ D\ n}$$

GMP integers are modelled as a sized list, so we use Vect to keep the list of limbs and its size consistent (in fact, all we do here is add another level of constructor to keep the type distinct from Vect). n represents the amount of limbs, and ls the array of limbs. We can

use similar techniques to those in section 5.2.7 to resize the Vects accordingly for arithmetic operations between numbers of different sizes.

A refinement of this includes a representation of the memory allocated for the number, and carries a proof that there is enough memory to store the number:

$$\underline{\text{data}} \quad \frac{D : \star \quad n : \mathbb{N}}{\mathsf{GMP}\ D\ n\ :\ \star} \quad \underline{\text{where}} \quad \frac{a : \mathbb{N} \quad ls : \mathsf{Vect}\ D\ n \quad p : n \leq a}{\mathsf{mkGMP}\ a\ ls\ :\ \mathsf{GMP}\ D\ n}$$

The extra argument to mkGMP, a, represents the number of limbs available. The constructor also carries a proof p (represented by $\leq$, and therefore concretely collapsible) that there is enough space to store the limbs in this amount of memory. By writing the arithmetic functions on this representation, we can extract the following information:

- Where there is a possibility of overflow.
- Where and when memory allocations might be needed and where they are superfluous.

It is perhaps too much to hope that extraction of code for arithmetic on this representation would be more efficient than the highly tuned machine code implementation of GMP. However, modelling the properties of GMP data structures in this way can give us some insight into where safety checks are needed in the low level implementation.

5.4.2 Correctness of Behaviour

The representation we now have ensures that the memory allocated for limbs is always enough to hold the data. What it does not ensure is that arithmetic operations have the correct behaviour. This is another situation where inductive families can help; we can also index the GMP family by the $\mathbb{N}$ it represents. To do this, we write a function to convert the limb representation to its corresponding $\mathbb{N}$:

$$\underline{\text{let}} \quad \frac{ls : \mathsf{Vect}\ D\ n}{\mathbf{limbsToNat}\ ls\ :\ \mathbb{N}}$$

Then the return type of the mkGMP constructor also gives the $\mathbb{N}$ interpretation of the GMP number:

$$\underline{\text{data}} \quad \frac{D : \star \quad n, i : \mathbb{N}}{\mathsf{GMP}\ D\ n\ i\ :\ \star} \quad \underline{\text{where}} \quad \frac{a : \mathbb{N} \quad ls : \mathsf{Vect}\ D\ n \quad p : n \leq a}{\mathsf{mkGMP}\ a\ ls\ :\ \mathsf{GMP}\ D\ n\ (\mathbf{limbsToNat}\ ls)}$$

Then to implement, for example, addition on GMP numbers builds an implicit proof that the GMP addition is a homomorphism with the $\mathbb{N}$ addition. Since addition may overflow and therefore require more space to store the result, we return a dependent pair containing the length and the GMP value.

$$\underline{\text{let}} \quad \frac{x : \mathsf{GMP}\ D\ n\ i \quad y : \mathsf{GMP}\ D\ m\ j}{\mathbf{addGMP}\ x\ y\ :\ \Sigma\,\mathbb{N}\ (\lambda n{:}\mathbb{N}.\ \mathsf{GMP}\ D\ n\ (\mathbf{plus}\ i\ j))}$$

Any implementation of this function must be a correct implementation of addition; anything else would not typecheck.

5.5 Summary

We have seen three uses of natural numbers; two of these (bounded recursion and indexing of data structures) rely on the structure of $\mathbb{N}$, and the other (arithmetic) does not, as it provides implementations of abstract operations on numbers.

Since arithmetic operations are abstract operations on numbers, we can consider alternative representations of numbers to provide more efficient implementations of arithmetic. With the Word family in section 5.2, we saw an implementation of binary numbers purely in EPIGRAM, using size invariants to verify the structure of these numbers. However, even implementing binary numbers in this way is impractical when compared to a hardware implementation. One possibility to improve this is to parametrise Word over a base type which implements arithmetic in hardware. However, allowing access to a hardware implementation forces us to extend the core language of TT and consider the additional typing and conversion rules this entails. A more useful application of this kind of implementation is for the verification of hardware design — we could imagine using a dependently typed language to model the hardware and its properties and implement operations on the hardware in a type safe way.

When numbers are used primarily for their structural properties it is still good to consider an efficient representation. In section 5.3 we saw additions to ExTT and associated translation rules for using an efficient external implementation of $\mathbb{N}$ via the GMP library. The advantage of this approach is that no changes are required to the core language, although we do need to justify that the translation rules are valid. We can justify this using the same observation that we used in Chapter 4 to build efficient elimination rules; i.e., any representation can be used for a family provided that its elimination rule can discriminate between ι-schemes. Our translation scheme provides direct mappings from $\mathbb{N}$ to GMP, and corresponding additions to the $\mathcal{I}$ compilation scheme. The major difficulty is in verifying that the GMP implementation of arithmetic mirrors the EPIGRAM implementation — to do this directly in EPIGRAM is impossible, since GMP is an external library, but it is possible to model GMP integers in EPIGRAM. This verification is a large and difficult task, but we have seen one possible way to approach the problem. For all practical purposes, however, it would be unreasonable to assume that GMP is not a correct implementation of arithmetic, given its successful use in other programming language implementations (such as GHC and Python).

What we have not seen is how we might use EPIGRAM to implement heavily numerical programs. For this sort of application, we should think of numbers as abstract data, with abstract operations. Where possible, we would give these operations EPIGRAM types (e.g. Float, Double, Int, etc) and conversion rules (as we did with the GMP representation of $\mathbb{N}$). It would make sense to investigate this approach in the context of a module system, rather than as an addition to the core language.

The introduction of primitive types creates further implementation difficulties, not all

of which we have investigated yet. We have briefly considered boxing and unboxing of primitives, but further investigation is required as to how to handle unboxed values most efficiently, whether using the techniques of [PL91a], [HM95] or others. The casetype analysis of [HM95] may be particularly beneficial and relatively simple to implement since we already have type level programs.

Chapter 6

Additional Optimisations

Several other well known optimisation techniques can, of course, be applied to EPIGRAM terms arising from the optimisations already presented. This chapter presents some well-known optimisations and some which arise from the EPIGRAM type system and shows how these optimisations might interact with those already seen. The optimisations we present are to be applied after typechecking, and hence are run-time only.

The approach taken follows that of Santos, who exploits the advantages of Compilation by Transformation for Haskell in his thesis [San95]; the transformation based optimiser is also described in [PS98]. This approach to compilation uses a single intermediate representation during most of the compilation process. This has the advantages that it allows optimisations to be implemented in a simple way, and that transformations are easier to prove correct — each transformation can be implemented and verified independently. For EPIGRAM, we apply optimising transformations at two levels. Higher level transformations on ExTT terms are used to transform some of the more abstract features of the language into a form more suitable for efficient compilation — in particular, the transformation of recursion operators into direct recursive calls. We also apply optimising transformations at the RunTT level. We separate these optimisation passes for two reasons:

- Types are preserved in ExTT terms, which makes it easier to prove transformations correct. We would prefer to preserve types for as long as possible so we try to perform as many transformations at this level as we can. At this level, we can also take advantage of labelled types (see section 2.1.8) for optimisation.

- Further optimisations are available once all functions are transformed into their RunTT representation, since there is no longer the need to take care to maintain the separation between user defined functions and elimination rules. In particular, inlining of non-recursive elimination rules becomes available. More aggressive (and non type preserving) optimisations such as argument removal also become available.

We transform at the level of ExTT, rather than TT. This is firstly so that we do not interfere with the analysis of elimination rules which allows the deletion of redundant arguments (see Chapter 4). In addition, we can be more liberal with ExTT terms — we are already sure of their type correctness and termination properties, so we can concern ourselves more with their *meaning*, the main example being the replacement of elimination rules with the more efficient direct recursive calls.

6.1 Optimisations in ExTT

6.1.1 β-reduction

The most basic transformation which can be applied to terms in ExTT is β-reduction:

$$[\![(\lambda x : T.\ e)\, a]\!] \Longrightarrow e[a/x]$$

We must be careful not to apply this automatically, however. If x occurs more than once in e, we risk evaluating the same a more than once. In such a situation, it is safer to either not β-reduce, or to $\underline{\text{let}}$ bind the name before reducing, as follows:

$$[\![(\lambda x : T.\ e)\, a]\!] \Longrightarrow \underline{\text{let}}\ x \mapsto a\ \underline{\text{in}}\ e$$

β-reduction is always worth applying, in either of these forms, since it saves a reduction at run-time and, even more importantly, can expose the other transformations which we will discuss in this chapter.

6.1.2 Simplifying Non-recursive D-Elim

Writing a function over a family D by means of the elimination operator D-**Elim** gives the programmer access to a recursive call on any recursive arguments of D. But if a function written over D-**Elim** does not make any recursive calls (that is, it does not use the inductive hypothesis in a method call), it would be better written with the case operator D-**Case**. Both D-**Case** and D-**Elim** are generated automatically on elaboration of a family D, D-**Case** being constructed from D-**Elim** by discarding the inductive hypotheses.

Here is a trivial example:

$$\underline{\text{let}}\quad \frac{n\ :\ \mathbb{N}}{\mathbf{isZero}\ n\ :\ \mathsf{Bool}} \qquad \begin{aligned} \mathbf{isZero}\ \ n\ \ &\Leftarrow \underline{\text{elim}}\ n \\ \mathbf{isZero}\ \ 0\ \ &\mapsto \mathsf{true} \\ \mathbf{isZero}\ (\mathsf{s}\ k)\ &\mapsto \mathsf{false} \end{aligned}$$

This elaborates to the following term in ExTT:

$$\mathbf{isZero} \mapsto \lambda n{:}\mathbb{N}.\ \mathbb{N}\text{-}\mathbf{Elim}\ n\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ \mathsf{true}\ (\lambda k, ih{:}\mathbb{N}.\ \mathsf{false})$$

Since the inductive hypothesis ih is unused in the method for s, we can safely use $\mathbb{N}$-**Case** rather than $\mathbb{N}$-**Elim**:

$$\mathbf{isZero} \mapsto \lambda n{:}\mathbb{N}.\ \mathbb{N}\text{-}\mathbf{Case}\ n\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ \mathsf{true}\ (\lambda k{:}\mathbb{N}.\ \mathsf{false})$$

How do we tell which argument to the method is the inductive hypothesis? Due to the way we build elimination rules, an inductive hypothesis follows each recursive argument. More generally, however, we can make use of the labelling on types (see section 2.1.8). Until now, I have been suppressing labels; recall that elaborated EPIGRAM terms label recursive calls and inductive hypotheses, so that it is clear to the programmer (in the high level notation) what the meaning of the inductive hypothesis is and so that the elaborator can tell what the allowed recursive calls are. Elaboration of **isZero** with labels gives the following:

$$\begin{aligned}&\mathbf{isZero}\ :\ \forall n{:}\mathbb{N}.\ \langle \mathbf{isZero}\ n\ :\ \mathbb{N}\rangle\\ &\mathbf{isZero} \mapsto \lambda n{:}\mathbb{N}.\ \mathbb{N}\text{-}\mathbf{Elim}\ n\ (\lambda n{:}\mathbb{N}.\ \langle \mathbf{isZero}\ n\ :\ \mathbb{N}\rangle)\\ &\qquad\qquad (\underline{\mathsf{return}}\ \mathsf{true})\ (\lambda k{:}\mathbb{N}.\ \lambda ih{:}\langle \mathbf{isZero}\ k\ :\ \mathbb{N}\rangle.\ (\underline{\mathsf{return}}\ \mathsf{false}))\end{aligned}$$

It is clear which is the inductive hypothesis from the label on its type; since no inductive hypothesis is used, $\mathbb{N}$-**Elim** can be replaced with $\mathbb{N}$-**Case**. A similar transformation applies for D-**View** where there are no appeals to an inductive hypothesis.

This is an apparently trivial optimisation, which I call **elimination unfolding**, with not much obvious benefit in practical terms. However, it does open up the possibility for further optimisations which were previously inapplicable, as we will now see.

6.1.3 Rewriting labelled types

Where a recursive function over D is written by D-**Elim**, the elimination rule is one extra level of indirection. The purpose of the rule at compile-time is to ensure that all recursion is primitive and so recursive functions terminate. At run-time, however, we would like to remove this level of indirection since it has served its purpose and now merely causes a run-time overhead.

Let us examine the **plus** function again, defined recursively using $\mathbb{N}$-**Elim**. While we have better ways of optimising this function (using an external representation of $\mathbb{N}$), the simplicity of the data structures used in this definition allow us to focus attention on the rewriting of the recursive call.

$$\begin{aligned}&\underline{\mathsf{let}}\quad \frac{n, m\ :\ \mathbb{N}}{\mathbf{plus}\ n\ m\ :\ \mathbb{N}}\\ &\quad \mathbf{plus}\quad n\quad m\ \Leftarrow \underline{\mathsf{elim}}\ n\\ &\quad \mathbf{plus}\quad \mathsf{0}\quad m\ \mapsto m\\ &\quad \mathbf{plus}\ (\mathsf{s}\ k)\ m\ \mapsto \mathsf{s}\ (\mathbf{plus}\ k\ m)\end{aligned}$$

Elaboration, including the labelling, gives the following term in ExTT:

$$\begin{aligned}&\mathbf{plus}\ :\ \forall n, m{:}\mathbb{N}.\ \langle \mathbf{plus}\ n\ m\ :\ \mathbb{N}\rangle\\ &\mathbf{plus} \mapsto \lambda n, m{:}\mathbb{N}.\ \mathbb{N}\text{-}\mathbf{Elim}\ n\ (\lambda n{:}\mathbb{N}.\ \langle \mathbf{plus}\ n\ m\ :\ \mathbb{N}\rangle)\ (\underline{\mathsf{return}}\ m)\\ &\qquad\qquad (\lambda k{:}\mathbb{N}.\ \lambda ih{:}\langle \mathbf{plus}\ k\ m\ :\ \mathbb{N}\rangle.\ \underline{\mathsf{return}}\ \mathsf{s}(\underline{\mathsf{call}}\ \langle \mathbf{plus}\ k\ m\rangle\ ih))\end{aligned}$$

The labelling gives us the *meaning* of each inductive hypothesis; the call $\langle$**plus** k $m\rangle$ ih says that the use of ih represents a call of **plus** k m. If that is what it represents, why the indirection? Once termination (via a primitive recursive definition) has been established and the term typechecked, we can replace the appeal to the inductive hypothesis with its actual meaning. The transformation is simple, and shown in figure 6.1.

$$[\![\underline{\text{call}}\ \langle l\rangle\ t]\!] \Longrightarrow \underline{\text{call}}\ \langle l\rangle\ l$$

Figure 6.1: Rewriting a term with labelled type

The call and return are retained by this transformation to preserve type correctness, but we no longer use the inductive hypothesis. The elaborated **plus** becomes:

plus : $\forall n, m{:}\mathbb{N}.\ \langle$**plus** $n\ m$: $\mathbb{N}\rangle$
plus $\mapsto \lambda n, m{:}\mathbb{N}.$ $\mathbb{N}$-**Elim** n $(\lambda n{:}\mathbb{N}.\ \langle$**plus** $n\ m$: $\mathbb{N}\rangle)$ $(\underline{\text{return}}\ m)$
$(\lambda k{:}\mathbb{N}.\ \lambda ih{:}\langle$**plus** $k\ m$: $\mathbb{N}\rangle.\ \underline{\text{return}}\ \mathsf{s}(\underline{\text{call}}\ \langle$**plus** $k\ m\rangle$ (**plus** $k\ m$)))

Since there is now no use of the inductive hypothesis in this function, by the transformation of the previous section we can use $\mathbb{N}$-**Case** instead of $\mathbb{N}$-**Elim**:

plus : $\forall n, m{:}\mathbb{N}.\ \langle$**plus** $n\ m$: $\mathbb{N}\rangle$
plus $\mapsto \lambda n, m{:}\mathbb{N}.$ $\mathbb{N}$-**Case** n $(\lambda n{:}\mathbb{N}.\ \langle$**plus** $n\ m$: $\mathbb{N}\rangle)$ $(\underline{\text{return}}\ m)$
$(\lambda k{:}\mathbb{N}.\ \underline{\text{return}}\ \mathsf{s}(\underline{\text{call}}\ \langle$**plus** $k\ m\rangle$ (**plus** $k\ m$)))

The typing rules of the labelling operation are such that a well-typed term results from dropping the labelling annotations $\langle l$: $T\rangle$ and call and return expressions, by the substitutions in figure 6.2 (which must all be applied together to preserve type correctness):

$$\begin{array}{l}[\![\langle l\ :\ T\rangle]\!] \Longrightarrow T\\ [\![\underline{\text{call}}\ \langle l\rangle\ t]\!] \Longrightarrow t\\ [\![\underline{\text{return}}\ t]\!] \Longrightarrow t\end{array}$$

Figure 6.2: Removing labels

Applying these substitutions to the elaborated **plus** gives:

plus : $\forall n, m{:}\mathbb{N}.\ \mathbb{N}$
plus $\mapsto \lambda n, m{:}\mathbb{N}.$ $\mathbb{N}$-**Case** n $(\lambda n{:}\mathbb{N}.\ \mathbb{N})$ n
$(\lambda k{:}\mathbb{N}.\ \mathsf{s}($**plus** $k\ m))$

By doing these transformations we recover the pattern matching behaviour in the elaborated program which was specified in the original EPIGRAM program. In OLEG [McB00a], tactics exist to create a correct pattern matching program from the elaborated definition. This is a slightly different approach; here we recover the *compiled* pattern matching program directly. D-**Case** is after all merely a higher level abstraction of applying the case operator to an element of D $\vec{s}$. The approach we take here has two advantages:

- We still recover a compiled form for programs which do not have directly compilable pattern matching behaviour, such as those built by user defined elimination rules or views. We cannot, for example, compile the pattern matching form of **compare** on page 50 because the patterns on the left hand side are arbitrary terms rather than constructor forms.

- We do not have to take an additional step of compiling the resulting pattern matching definitions into simple case expressions; to do this would duplicate work, having compiled elimination rules and built the program in terms of those elimination rules.

With the current implementation of elimination rules, this transformation is always beneficial as it removes a layer of indirection. In future, however, we may need to be more careful if we choose an optimised iterative or tail-recursive implementation of an elimination rule (as suggested for $\mathbb{N}$-**Elim** on page 153) since this transformation would supersede the optimised elimination rule.

6.1.4 Optimising D-Rec

Recall from section 2.2.6 how structurally recursive functions are built by memoising recursive calls. The example given was the Fibonacci function:

$$\begin{array}{lll}
\underline{\text{let}} & \dfrac{n \;:\; \mathbb{N}}{\mathbf{fib}\; n \;:\; \mathbb{N}} & \\
\mathbf{fib} & n & \Leftarrow \underline{\text{rec}}\; n, \underline{\text{case}}\; n \\
\mathbf{fib} & 0 & \mapsto 0 \\
\mathbf{fib} & (\mathsf{s}\; k) & \Leftarrow \underline{\text{case}}\; k \\
\mathbf{fib} & (\mathsf{s}\; 0) & \mapsto \mathsf{s}\; 0 \\
\mathbf{fib}\; (\mathsf{s}\; (\mathsf{s}\; k')) & & \mapsto \mathbf{plus}\; (\mathbf{fib}\; k')\; (\mathbf{fib}\; (\mathsf{s}\; k'))
\end{array}$$

Elaboration of this gives rise to a frightening looking term, which is no less frightening (but perhaps more informative) for the insertion of labels into the types:

$$\begin{array}{l}
\mathbf{fib} \mapsto \lambda n{:}\mathbb{N}.\; \mathbb{N}\text{-}\mathbf{Rec}\; n\; (\lambda x{:}\mathbb{N}.\; \langle \mathbf{fib}\; x \;:\; \mathbb{N}\rangle) \\
\qquad (\lambda n'{:}\mathbb{N}.\; \mathbb{N}\text{-}\mathbf{Case}\; n'\; (\lambda x{:}\mathbb{N}.\; (\mathbb{N}\text{-}\mathbf{Memo}\; (\lambda y{:}\mathbb{N}.\; \langle \mathbf{fib}\; y \;:\; \mathbb{N}\rangle)\; x) \rightarrow \langle \mathbf{fib}\; x \;:\; \mathbb{N}\rangle) \\
\qquad\quad (\lambda u{:}\mathsf{True}.\; \underline{\text{return}}\; \mathsf{s0}) \\
\qquad\quad (\lambda k{:}\mathbb{N}.\; \mathbb{N}\text{-}\mathbf{Case}\; k\; (\lambda x{:}\mathbb{N}.\; (\mathbb{N}\text{-}\mathbf{Memo}\; (\lambda y{:}\mathbb{N}.\; \langle \mathbf{fib}\; y \;:\; \mathbb{N}\rangle)\; \mathsf{s}\; x) \rightarrow \langle \mathbf{fib}\; x \;:\; \mathbb{N}\rangle) \\
\qquad\qquad (\lambda x{:}(\mathbb{N}\text{-}\mathbf{Memo}\; (\lambda y{:}\mathbb{N}.\; \langle \mathbf{fib}\; y \;:\; \mathbb{N}\rangle)\; \mathsf{s0}).\; \underline{\text{return}}\; \mathsf{s0}) \\
\qquad\qquad (\lambda k{:}\mathbb{N}.\; \lambda M{:}(\mathbb{N}, (\mathbb{N}, \mathbb{N}\text{-}\mathbf{Memo}\; (\lambda x{:}\mathbb{N}.\; \langle \mathbf{fib}\; x \;:\; \mathbb{N}\rangle)\; k)). \\
\qquad\qquad\quad \underline{\text{return}}\; \mathbf{plus}\; (\underline{\text{call}}\; \langle \mathbf{fib}\; \mathsf{s}\; k\rangle\; (\mathbf{fst}\; M))\; (\underline{\text{call}}\; \langle \mathbf{fib}\; k\rangle\; (\mathbf{fst}\; (\mathbf{snd}\; M))))))
\end{array}$$

There are two questions to answer about optimisations of D-**Rec**:

1. Can we remove the outermost call to D-**Rec**?

2. Even if we *can*, do we *want* to?

D-**Rec** is used for memoising the results of recursive calls; the purpose of this is primarily to make recursive calls on structurally smaller values accessible. Given the labels on terms identifying the meaning of lookups in the memo structure (such as in the call on the last line of the **fib** function above) is the memo structure now necessary? Let us see what happens to **fib** after applying the call rewriting transformation and dropping labels:

$$
\begin{array}{l}
\textbf{fib} \mapsto \lambda n{:}\mathbb{N}.\ \mathbb{N}\textbf{-Rec}\ n\ (\lambda n{:}\mathbb{N}.\ \mathbb{N}) \\
\qquad (\lambda n{:}\mathbb{N}.\ \mathbb{N}\textbf{-Case}\ n\ (\lambda n{:}\mathbb{N}.\ (\mathbb{N}\textbf{-Memo}\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ n) \to \mathbb{N}) \\
\qquad\quad (\lambda u{:}\mathsf{True}.\ \mathsf{s0}) \\
\qquad\quad (\lambda k{:}\mathbb{N}.\ \mathbb{N}\textbf{-Case}\ k\ (\lambda n{:}\mathbb{N}.\ (\mathbb{N}\textbf{-Memo}\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ \mathsf{s}\ n) \to \mathbb{N}) \\
\qquad\qquad (\lambda x{:}(\mathbb{N}\textbf{-Memo}\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ \mathsf{s0}.\ \mathsf{s0})) \\
\qquad\qquad (\lambda k{:}\mathbb{N}.\ \lambda M{:}(\mathbb{N}, (\mathbb{N}, \mathbb{N}\textbf{-Memo}\ (\lambda n{:}\mathbb{N}.\ \mathbb{N})\ k)). \\
\qquad\qquad\quad \textbf{plus}\ (\textbf{fib}\ (\mathsf{s}\ k))\ (\textbf{fib}\ k))))
\end{array}
$$

The memo structure M is no longer used in the recursive case, which suggests that we can drop the outermost $\mathbb{N}$-**Rec** and indeed, if we do, the function behaves in the same way as our original elaborated definition:

$$
\begin{array}{l}
\textbf{fib} \mapsto \lambda n{:}\mathbb{N}.\ \mathbb{N}\textbf{-Case}\ n\ (\lambda n{:}\mathbb{N}.\ \mathbb{N}) \\
\qquad\quad (\mathsf{s0}) \\
\qquad\quad (\lambda k{:}\mathbb{N}.\ \mathbb{N}\textbf{-Case}\ k\ (\lambda n{:}\mathbb{N}.\ \mathbb{N}) \\
\qquad\qquad (\mathsf{s0}) \\
\qquad\qquad (\lambda k{:}\mathbb{N}. \\
\qquad\qquad\quad \textbf{plus}\ (\textbf{fib}\ (\mathsf{s}\ k))\ (\textbf{fib}\ k)))
\end{array}
$$

Unfortunately, doing this "optimisation" has in fact made the function less efficient; previously, the memoisation of recursive calls also ensured that no call to **fib** was computed twice. Here, however, in the recursive case, **fib** k is computed twice — once directly, and once as a recursive call of **fib** (s k).

The answer to the first question is yes — we can remove the outermost D-**Rec** by replacing lookups in the memo structure with the appropriate recursive call. In this case, the answer to the second question is no — it results in a less efficient definition. Perhaps, however, in cases where only one recursive call is made in each branch of the function, this transformation is worthwhile as the benefit of memoisation is limited to showing the function is structurally recursive.

We are now in a position to compile definitions efficiently into their RunTT representation. A further transformation phase is applied to RunTT, which is the subject of the next section.

6.2 Optimisations in RunTT

6.2.1 Inlining

The inlining transformation expands function definitions in-place; instead of calling the function at run-time, we replace the call with the body of the function at compile-time. We can not always be certain that inlining is an optimisation; [PM02] details many of the issues involved. Whether to inline a function depends on several factors such as the size of the function body (small functions are good candidates), the number of times it is applied (a function called few times is a good candidate) and whether we can be certain that inlining will not cause duplication of work.

Inlining of a suitable function **f** is given by the transformation in figure 6.3. Note that the instances of f which are inlined are fully applied; although this may rule out some useful optimisations, we do this because the RunTT syntax allows λ only at the top level.

$$\begin{array}{l} \mathbf{f} \mapsto \lambda\vec{x}.e \\ [\![\mathbf{f}\ \vec{a}]\!] \Longrightarrow e[\vec{a}/\vec{x}] \quad \text{where } \textsc{length}(\vec{a}) = \textsc{arity}(\mathbf{f}) \end{array}$$

Figure 6.3: The **inlining** transformation

Why do we do the inlining at the RunTT level rather than earlier, in ExTT? The reason is that inlining in RunTT allows inlining of D-**Case** operators. In ExTT, we cannot inline elimination rules since their form (direct definition of ι-reductions) is incompatible with the form of user defined functions.

Example — Inlining Case Operators

Following on from the rewriting of labelled types earlier, we observe that D-**Case** operators are good candidates for inlining — they are not recursive, and can often be syntactically smaller than the call. This removes the final layer of indirection introduced by using D-**Elim** operators in the first place.

After the transformations on ExTT which reduce **plus** to an application of ℕ-**Case** with direct recursion, the supercombinators generated for **plus** are:

$$\begin{array}{lll} \mathbf{plus} & \mapsto & \lambda n; m : \mathbb{N}.\ \mathbb{N}\text{-}\mathbf{Case}\ n\ \mathbf{plus1}\ m\ (\mathbf{plus2}\ n) \\ \mathbf{plus1} & \mapsto & \lambda n{:}\mathbb{N}.\ \mathbb{N} \\ \mathbf{plus2} & \mapsto & \lambda n; k{:}\mathbb{N}.\ \mathsf{s}\langle \mathbf{plus}\ n\ k\rangle \\ \mathbb{N}\text{-}\mathbf{Case} & \mapsto & \lambda n;\ P; m_0; m_\mathsf{s}.\ \underline{\text{case}}\ n\ \underline{\text{of}} \\ & & \quad \mathsf{0}\langle\rangle \rightsquigarrow m_0 \\ & & \quad \mathsf{s}\langle k\rangle \rightsquigarrow m_\mathsf{s}\ k \end{array}$$

plus1, **plus2** and ℕ-**Case** are all suitable candidates for inlining. At this stage, **plus1** and **plus2** are not fully applied in **plus** so they cannot be inlined, but ℕ-**Case** can, which

gives the following:

$$\begin{array}{lll}
\textbf{plus} \mapsto & \lambda n; m : \mathbb{N}.\ \underline{\text{case}}\ n\ \underline{\text{of}} & \\
& \quad \mathsf{0}\langle\rangle & \rightsquigarrow\ m \\
& \quad \mathsf{s}\langle k\rangle & \rightsquigarrow\ \textbf{plus2}\ k\ m \\
\textbf{plus1} \mapsto & \lambda n{:}\mathbb{N}.\ \mathbb{N} & \\
\textbf{plus2} \mapsto & \lambda n; k{:}\mathbb{N}.\ \mathsf{s}\langle\textbf{plus}\ n\ k\rangle &
\end{array}$$

Now, **plus2** is fully applied so can be inlined, and since P was not used by $\mathbb{N}$-**Case**, the use of **plus1** drops out of the program. This gives the following single definition for **plus**:

$$\begin{array}{lll}
\textbf{plus} \mapsto & \lambda n; m : \mathbb{N}.\ \underline{\text{case}}\ n\ \underline{\text{of}} & \\
& \quad \mathsf{0}\langle\rangle & \rightsquigarrow\ m \\
& \quad \mathsf{s}\langle k\rangle & \rightsquigarrow\ \mathsf{s}\langle\textbf{plus}\ k\ m\rangle
\end{array}$$

So **plus**, defined by an elimination rule, is ultimately transformed to a directly recursive function implemented by case analysis, which is the supercombinator definition we might expect if **plus** had been defined by pattern matching in the first place.

Another Example — flatten

plus is a straightforward example of this process, but perhaps an unrealistic one since we have better ways of optimising this function using the GMP transformation of Chapter 5. Let us examine how the same process applies to the **flatten** function over a user defined structure, a binary tree. Let us first look at the data structure, and the high level definition of the function.

We choose to define trees with values stored only at the leaves:

$$\underline{\text{data}} \quad \frac{A\ :\ \star}{\mathsf{Tree}\ A\ :\ \star}$$

$$\underline{\text{where}} \quad \frac{a\ :\ A}{\mathsf{Leaf}\ a\ :\ \mathsf{Tree}\ A} \qquad \frac{l\ :\ \mathsf{Tree}\ A \qquad r\ :\ \mathsf{Tree}\ A}{\mathsf{Node}\ l\ r\ :\ \mathsf{Tree}\ A}$$

Flattening a tree into a list involves creating a list of one item, in the Leaf case, and appending the result of recursive calls on the left and right trees in the Node case.

$$\underline{\text{let}} \quad \frac{t\ :\ \mathsf{Tree}\ A}{\textbf{flatten}\ t\ :\ \mathsf{List}\ A} \qquad \begin{array}{lll}
\textbf{flatten} & t & \Leftarrow \textsf{Tree-}\textbf{Elim}\ t \\
\textbf{flatten}\ (\mathsf{Leaf}\ a) & & \mapsto\ \mathsf{cons}\ a\ \mathsf{nil} \\
\textbf{flatten}\ (\mathsf{Node}\ l\ r) & & \mapsto\ \textbf{append}\ (\textbf{flatten}\ l)\ (\textbf{flatten}\ r)
\end{array}$$

This definition elaborates to the following, based on Tree-**Elim**, with the implicit argument A to **flatten** made explicit, and labels placed on the terms:

$$\begin{array}{l}
\textbf{flatten} \mapsto \lambda A{:}\star.\ \lambda t{:}\mathsf{Tree}\ A. \\
\qquad \textsf{Tree-}\textbf{Elim}\ t\ (\lambda t{:}\mathsf{Tree}\ A.\ \langle\textbf{flatten}\ A\ t\ :\ \mathsf{List}\ A\rangle) \\
\qquad\quad (\lambda a{:}A.\ \underline{\text{return}}\ (\mathsf{cons}\ a\ \mathsf{nil})) \\
\qquad\quad (\lambda l{:}\mathsf{Tree}\ A.\ \lambda \mathit{lih}{:}\langle\textbf{flatten}\ A\ l\ :\ \mathsf{List}\ A\rangle. \\
\qquad\quad \lambda r{:}\mathsf{Tree}\ A.\ \lambda \mathit{rih}{:}\langle\textbf{flatten}\ A\ r\ :\ \mathsf{List}\ A\rangle. \\
\qquad\qquad \underline{\text{return}}\ (\textbf{append}\ (\underline{\text{call}}\ \langle\textbf{flatten}\ A\ l\rangle\ \mathit{lih})\ (\underline{\text{call}}\ \langle\textbf{flatten}\ A\ r\rangle\ \mathit{rih})))
\end{array}$$

Rewriting in ExTT replaces the inductive hypotheses *lih* and *rih* with the recursive calls they represent, **flatten** $A\ l$ and **flatten** $A\ r$. Now the call to Tree-**Elim** can be replaced with a call to Tree-**Case**:

$$\begin{aligned}&\textbf{flatten} \mapsto \lambda A{:}\star.\ \lambda t{:}\mathsf{Tree}\ A.\\ &\qquad \mathsf{Tree\text{-}Case}\ t\ (\lambda t{:}\mathsf{Tree}\ A.\ \mathsf{List}\ A)\\ &\qquad\quad (\lambda a{:}A.\ \mathsf{cons}\ a\ \mathsf{nil})\\ &\qquad\quad (\lambda l{:}\mathsf{Tree}\ A.\ \lambda r{:}\mathsf{Tree}\ A.\\ &\qquad\qquad (\textbf{append}\ (\textbf{flatten}\ A\ l)\ (\textbf{flatten}\ A\ r)))\end{aligned}$$

In the translation to RunTT the call to Tree-**Case** can be inlined, resulting in the following supercombinator definition of **flatten** (note that the type labels, A, are removed from the structure by forcing):

$$\begin{aligned}&\textbf{flatten} \mapsto \lambda A; t.\ \underline{\mathsf{case}}\ t\ \underline{\mathsf{of}}\\ &\qquad\qquad \mathsf{Leaf}\langle a\rangle \rightsquigarrow \mathsf{cons}\langle a, \mathsf{nil}\langle\rangle\rangle\\ &\qquad\qquad \mathsf{Node}\langle l, r\rangle \rightsquigarrow \textbf{append}\ (\textbf{flatten}\ A\ l)\ (\textbf{flatten}\ A\ r)\end{aligned}$$

6.2.2 Unused Argument Removal

An apparently trivial but nevertheless important optimisation is the removal of arguments which are unused in a supercombinator body, by examining their syntactic occurrence. In simply typed languages, such a transformation is unlikely to have much of an effect, since all arguments are there because the programmer has put them there. With EPIGRAM's implicit arguments, sometimes arguments are inserted into elaborated terms only for typechecking. This is particularly likely when programming with inductive families — an index to a family must also be passed as an (often implicit) argument to a function over the family, whether needed or not by that function.

An example is the **flatten** function in the previous section. The A argument is unused, although it must be there for the function to typecheck. At run-time, this argument can clearly be discarded.

The Argument Removal Transformation

Consider a supercombinator $\mathbf{f}$ with arguments $\vec{x}$.

$$\mathbf{f} \mapsto \lambda\vec{x}.\ e$$

The body of $\mathbf{f}$ may make any number of direct recursive calls to $\mathbf{f}$, with arguments $\vec{y}_j$. Then each argument in $\vec{x}$ may be classified as follows:

- **Passive**: Either x_i does not appear in e at all, or it is used *only* as (or as part of) the ith argument to any fully applied recursive call of $\mathbf{f}$.
- **Active**: x_i appears anywhere else in e — either not as an argument to a recursive call of $\mathbf{f}$, or as part of the jth argument, where $j \neq i$.

A *passive* argument need not be passed to **f**, for obvious reasons — **f** will never examine it. In a base case, a passive argument x_i is unused since it appears only as the ith argument to recursive calls. In a recursive case, x_i is unused if it is unused in the recursive call; it must be unused, by induction.

So, splitting the arguments $\vec{x}$ into $\vec{x}_a$ (the active arguments) and $\vec{x}_p$ (the passive arguments), we make the removal optimisation as in figure 6.4. Note that the substitution of $\mathbf{f}'$ for **f** is also made in the body of **f** so that unused arguments to recursive calls are also substituted.

$$\begin{array}{|l} \mathbf{f}' \mapsto \lambda \vec{x}_a.\ e \\ \mathbf{f} \mapsto \lambda \vec{x}_a; \vec{x}_p.\ \mathbf{f}'\ \vec{x}_a \\ [\![\mathbf{f}\ \vec{x}_a\ \vec{x}_p]\!] \Longrightarrow \mathbf{f}'\ \vec{x}_a \end{array}$$

Figure 6.4: The **argument removal** transformation

Why build a new $\mathbf{f}'$ rather than simply modifying **f**? The problem is that a higher order function may call **f**, and we have changed the type of **f** by removing arguments. Such a function cannot know until run-time which function to call and therefore it cannot know which arguments have been dropped from the function. Therefore, only fully applied instances of **f** are transformed. This is a technique also used by [PL91a] to exploit strictness analysis by changing boxed values to unboxed values without changing the type of a function. **f** is a **wrapper** function for $\mathbf{f}'$, which is the **worker**.

Since the argument removal optimisation removes some type information from definitions, it is one of the last transformations to be applied — after all type preserving optimisations have been applied.

Nested Unused Arguments

If an argument a_i is used in **f** only in a call to **g**, but is unused in **g**, what happens to a_i in **f**? The design of the system is that a declaration gets elaborated, optimised and translated to G-code immediately — as a result, when **f** is declared, we already have the substitutions generated from the declaration of **g**. Applying these substitutions to **f** before looking for new transformations means that the argument a_i is also identified as unused in **f**.

Argument Removal and Detagging

This optimisation does raise a question about the detagging transformation from Chapter 4. The compilation of the Vect append function illustrates the problem:

$$\underline{\text{let}} \quad \frac{xs : \mathsf{Vect}\ A\ n \qquad ys : \mathsf{Vect}\ A\ m}{\mathbf{append}\ xs\ ys : \mathsf{Vect}\ A\ (\mathbf{plus}\ n\ m)}$$

$$\begin{array}{lll} \mathbf{append} & \epsilon & ys \mapsto ys \\ \mathbf{append}\ (x{::}xs) & ys & \mapsto x{::}(\mathbf{append}\ xs\ ys) \end{array}$$

Although not explicitly stated, or used, n and m must be arguments to the elaborated **append** for the term to typecheck. At run-time, they are never explicitly used — this causes meaningless extra stack push and pop instructions in the G-code, so can these arguments simply be dropped? Unfortunately, it is not so simple. Consider the elaborated **append**, after the detagging optimisation:

$$\begin{aligned}
\textbf{append} \mapsto\ & \lambda A{:}\star.\ \lambda n, m{:}\mathbb{N}.\ \lambda xs{:}\mathsf{Vect}\ A\ n.\ \lambda ys{:}\mathsf{Vect}\ A\ m. \\
& \quad \textsf{Vect-}\textbf{Elim}\ n\ xs \\
& \quad\quad (\lambda n{:}\mathbb{N}.\ \lambda xs{:}\mathsf{Vect}\ A\ n.\ \langle \textbf{append}\ A\ n\ m\ xs\ ys\ :\ \mathsf{Vect}\ A\ (\textbf{plus}\ n\ m)\rangle) \\
& \quad\quad \underline{\text{return}}\ ys \\
& \quad\quad (\lambda k{:}\mathbb{N}.\ \lambda x{:}A.\ \lambda xs{:}\mathsf{Vect}\ A\ k. \\
& \quad\quad\quad \lambda ih{:}\langle \textbf{append}\ A\ k\ m\ xs\ ys\ :\ \mathsf{Vect}\ A\ (\textbf{plus}\ k\ m)\rangle. \\
& \quad\quad\quad \underline{\text{return}}\ \{:\}\ \{A\}\ \{k\}\ x\ (\underline{\text{call}}\ \langle \textbf{append}\ A\ k\ m\ xs\ ys\rangle\ ih))
\end{aligned}$$

After the transformations of the previous section (making recursion direct, and inlining) as well as detagging, we get the following supercombinator for **append**:

$$\begin{aligned}
\textbf{append} \mapsto\ & \lambda A; n; m; xs; ys.\ \underline{\text{case}}\ n\ \underline{\text{of}} \\
& \quad 0 \rightsquigarrow ys \\
& \quad \mathsf{s}\langle k\rangle \rightsquigarrow {::}\langle (xs!0), (\textbf{append}\ A\ k\ m\ (xs!1)\ ys)\rangle
\end{aligned}$$

We need to keep n, as we establish which constructor of Vect was used by examining n. We can still drop two arguments from **append** however (A and m are both passive, since they are used only in the recursive call), and get the following substitution:

$$\begin{aligned}
\textbf{append}' \mapsto\ & \lambda n; xs; ys.\ \underline{\text{case}}\ n\ \underline{\text{of}} \\
& \quad 0 \rightsquigarrow ys \\
& \quad \mathsf{s}\langle k\rangle \rightsquigarrow {::}\langle (xs!0), (\textbf{append}'\ k\ (xs!1)\ ys)\rangle \\
[\![\textbf{append}\ A\ n\ m\ xs\ ys]\!] \Longrightarrow\ & \textbf{append}'\ n\ xs\ ys
\end{aligned}$$

If we do not allow detagging, and only apply the forcing optimisation, we get the following supercombinator which discriminates on the constructors of Vect:

$$\begin{aligned}
\textbf{append} \mapsto\ & \lambda A; n; m; xs; ys.\ \underline{\text{case}}\ xs\ \underline{\text{of}} \\
& \quad \epsilon \rightsquigarrow ys \\
& \quad {::}\langle a, v\rangle \rightsquigarrow {::}\langle a, (\textbf{append}\ A\ (n!0)\ m\ v\ ys)\rangle
\end{aligned}$$

Here, clearly A, n and m are passive as they are only used in the recursive call so we can build **append'** by dropping these unused arguments and apply the following substitution:

$$\begin{aligned}
\textbf{append}' \mapsto\ & \lambda xs; ys.\ \underline{\text{case}}\ xs\ \underline{\text{of}} \\
& \quad \epsilon \rightsquigarrow ys \\
& \quad {::}\langle a, v\rangle \rightsquigarrow {::}\langle a, (\textbf{append}'\ v\ ys)\rangle \\
[\![\textbf{append}\ A\ n\ m\ xs\ ys]\!] \Longrightarrow\ & \textbf{append}'\ xs\ ys
\end{aligned}$$

This is an example of the tradeoffs which have to be made when optimising. Detagging reduces the storage requirement of Vect, but we lose this run-time speed optimisation. Since

the space optimisation of detagging is small, this is one reason why we restrict detagging to those families which are also concretely collapsible.

Types and Proofs

A common application of this transformation is to polymorphic functions, that is, those which depend on a type. Since there is no casetype construct at present, any argument $A : \star$ cannot be used. Such arguments will ultimately be identified as unused and hence dropped from run-time code.

Another consideration is what happens to proofs which cannot be collapsed. Dependently typed programs may carry proofs of properties which are verified by the typechecker, but never used at run-time. These are the kind of object which would be in the Prop family in Coq and hence dropped by program extraction. In our system, we identify such objects as unused and drop them. This definition of $\leq$, for example, is *not* concretely collapsible, because it is not detaggable:

$$\underline{\text{data}} \quad \frac{m, n : \mathbb{N}}{m \leq n : \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{leN}\ n : n \leq n} \quad \frac{p : n \leq m}{\mathsf{leR}\ p : n \leq \mathsf{s}\ m}$$

We may use this relation to verify properties of functions. We can define **minus** by induction over the numbers, using the proof to ensure that no invalid call of **minus** can be made:

$$\underline{\text{let}} \quad \frac{n, m : \mathbb{N} \quad p : m \leq n}{\mathbf{minus}\ n\ m\ p : \mathbb{N}} \qquad \begin{array}{lll} \mathbf{minus}\ n & 0 & p \mapsto n \\ \mathbf{minus}\ n\ (\mathsf{s}\ m)\ p & & \mapsto \mathsf{s}\ (\mathbf{minus}\ n\ m\ (\mathbf{le\_trans\_S}\ p)) \end{array}$$

A small amount of theorem proving is necessary to create the third argument to the recursive call of **minus**. **le_trans_S** is a lemma which proves $\mathsf{s}\ n \leq m \rightarrow n \leq m$. When elaborated and compiled to an optimised supercombinator, we get the following:

$$\begin{array}{lll} \mathbf{minus} \mapsto & \lambda n; m; p.\ \underline{\text{case}}\ m\ \underline{\text{of}} & \\ & 0 & \rightsquigarrow\ n \\ & \mathsf{s}\langle k \rangle & \rightsquigarrow\ \mathsf{s}\langle \mathbf{minus}\ n\ k\ (\mathbf{le\_trans\_S}\ p) \rangle \end{array}$$

The 3rd argument, p, is *passive*; it only appears as part of the 3rd argument to the recursive call. As such, it can be removed.

Higher Order Functions

A limitation of this transformation can arise with the use of higher order functions. For example, consider the following function **vmap** which maps a function across a vector.

$$\underline{\text{let}} \quad \frac{f : A \rightarrow B \quad xs : \mathsf{Vect}\ A\ n}{\mathbf{vmap}\ f\ xs : \mathsf{Vect}\ B\ n} \qquad \begin{array}{lll} \mathbf{vmap}\ f & xs & \Leftarrow \underline{\text{elim}}\ xs \\ \mathbf{vmap}\ f & \epsilon & \mapsto \epsilon \\ \mathbf{vmap}\ f\ (x{::}xs) & & \mapsto (f\ x){::}(\mathbf{vmap}\ f\ xs) \end{array}$$

We may wish to map a function with unused arguments across this vector, such as the following function **mkPair** which pairs a value of any type with itself. In its elaborated form, there is an unused argument A giving the type of the value.

$$\underline{\text{let}} \quad \frac{a \ : \ A}{\mathbf{mkPair}\ a \ : \ (A \times A)} \qquad \mathbf{mkPair}\ a \ \mapsto \ (a, a)$$

$$\mathbf{mkPair} \mapsto \lambda A; a.\ (a, a)_A$$

When passed to the higher order function **vmap**, for example to map across a vector xs : $\mathsf{Vect}\,\mathbb{N}\ n$, the elaborated ExTT is as follows:

$$\mathbf{vmap}\ \mathbb{N}\ (\mathbb{N} \times \mathbb{N})\ n\ (\mathbf{mkPair}\ \mathbb{N})\ xs$$

Since **mkPair** is not fully applied here, we cannot apply the argument removal transformation, and must instead call the wrapper function. One possible way to avoid this is to η-expand the application and λ-lift the resulting abstraction. It is not obvious whether this additional step, creating an extra supercombinator, is beneficial, and further experimentation with real EPIGRAM programs will be necessary to provide useful data for comparison.

6.2.3 Identifying No-Operations

Consider the following function, which weakens an element of a finite set by lifting it into the next biggest set:

$$\underline{\text{let}} \quad \frac{i \ : \ \mathsf{Fin}\ n}{\mathbf{weaken}\ i \ : \ \mathsf{Fin}\ (\mathsf{s}\ n)} \qquad \begin{array}{l} \mathbf{weaken}\ \ \mathsf{f0} \ \mapsto \mathsf{f0} \\ \mathbf{weaken}\ (\mathsf{fs}\ i) \ \mapsto \mathsf{fs}\ (\mathbf{weaken}\ i) \end{array}$$

This function doesn't appear to be doing much — in fact, most of the activity is implicit; in the f0 case the return value is in the next higher set but we do not see this due to the implicit argument. To clarify what is happening, let us look at the elaboration of **weaken** to TT:

$$\begin{array}{rl} \mathbf{weaken} \mapsto & \lambda n{:}\mathbb{N}.\ \lambda i{:}\mathsf{Fin}\ n.\ \mathsf{Fin}\text{-}\mathbf{Elim}\ i\ (\lambda n{:}\mathbb{N}.\ \lambda i{:}\mathsf{Fin}\ n.\ \langle \mathbf{weaken}\ n\ i \ : \ \mathsf{Fin}\ (\mathsf{s}\ n)\rangle) \\ & (\lambda n{:}\mathbb{N}.\ \underline{\text{return}}\ (\mathsf{f0}\ (\mathsf{s}\ n))) \\ & (\lambda n{:}\mathbb{N}.\ \lambda i{:}\mathsf{Fin}\ n.\ \lambda ih{:}\langle \mathbf{weaken}\ n\ i \ : \ \mathsf{Fin}\ (\mathsf{s}\ n)\rangle. \\ & \quad \underline{\text{return}}\ \mathsf{fs}\ (\mathsf{s}\ n)\ (\underline{\text{call}}\ \langle \mathbf{weaken}\ n\ i\rangle\ ih)) \end{array}$$

We now see explicitly that the function increments the index of the finite set. Fin has forceable arguments, however, so after the forcing optimisation, we have the following ExTT definition:

$$\begin{array}{rl} \mathbf{weaken} \mapsto & \lambda n{:}\mathbb{N}.\ \lambda i{:}\mathsf{Fin}\ n.\ \mathsf{Fin}\text{-}\mathbf{Elim}\ i\ (\lambda n{:}\mathbb{N}.\ \lambda i{:}\mathsf{Fin}\ n.\ \langle \mathbf{weaken}\ n\ i \ : \ \mathsf{Fin}\ (\mathsf{s}\ n)\rangle) \\ & (\lambda n{:}\mathbb{N}.\ \underline{\text{return}}\ (\mathsf{f0}\ \{\mathsf{s}\ n\})) \\ & (\lambda n{:}\mathbb{N}.\ \lambda i{:}\mathsf{Fin}\ n.\ \lambda ih{:}\langle \mathbf{weaken}\ n\ i \ : \ \mathsf{Fin}\ (\mathsf{s}\ n)\rangle. \\ & \quad \underline{\text{return}}\ \mathsf{fs}\ \{\mathsf{s}\ n\}\ (\underline{\text{call}}\ \langle \mathbf{weaken}\ n\ i\rangle\ ih)) \end{array}$$

The indices which are incremented (which is, of course, the purpose of this function) have been deleted! We might begin to get suspicious of the purpose and run-time cost

of this function, even more so when we examine the supercombinator which results after elimination unfolding, inlining and argument removal:

$$\begin{aligned}\textbf{weaken} \mapsto\ & \lambda i.\ \underline{\text{case}}\ i\ \underline{\text{of}} \\ & \quad \mathsf{f0}\langle\rangle \rightsquigarrow \mathsf{f0}\langle\rangle \\ & \quad \mathsf{fs}\langle i'\rangle \rightsquigarrow \mathsf{fs}\langle \textbf{weaken}\ i'\rangle\end{aligned}$$

We have a function which does nothing, recursively. Clearly, we would like to avoid running this function as it has existed only to manage indices for typechecking. The RunTT transformation we would like is:

$$[\![\textbf{weaken}]\!] \Longrightarrow \textbf{id}$$

(where $\textbf{id} \mapsto \lambda x.\ x$ and can itself be inlined.)

How can we establish systematically whether a RunTT function is equivalent to the identity function? A function $\mathbf{f}$ of a family D with n constructors $\mathsf{c}_i\ \vec{a}_i\ \vec{y}_i$ where $\vec{a}$ are non-recursive arguments and $\vec{y}$ are recursive arguments is effectively a no-operation if it has the following form:

$$\begin{aligned}\mathbf{f} \mapsto\ & \lambda x.\ \underline{\text{case}}\ x\ \underline{\text{of}} \\ & \quad \mathsf{c}_1\langle\vec{a},\vec{y}\rangle \rightsquigarrow \mathsf{c}_1\langle\vec{a},\mathbf{f}\ y_{11},\ldots,\mathbf{f}\ y_{1m}\rangle \\ & \quad \ldots \\ & \quad \mathsf{c}_n\langle\vec{a},\vec{y}\rangle \rightsquigarrow \mathsf{c}_n\langle\vec{a},\mathbf{f}\ y_{n1},\ldots,\mathbf{f}\ y_{nm}\rangle\end{aligned}$$

The property that a function of this form is effectively the identity function can be shown as follows. Where the input to $\mathbf{f}$ is a constructor c_i with no recursive arguments, then $\mathbf{f}\ \mathsf{c}_i\langle\vec{a}\rangle = \mathsf{c}_i\langle\vec{a}\rangle$, and so for all base cases, the function is equivalent to the identity function. Now, where the input is a constructor c_i with recursive arguments, we have

$$\mathbf{f}\ \mathsf{c}_i\langle\vec{a}_i,\vec{y}_i\rangle \mapsto \mathsf{c}_i\langle\vec{a},\mathbf{f}\ y_{i1}\ldots\mathbf{f}\ y_{im}\rangle$$

We must show that

$$\mathsf{c}_i\langle\vec{a},\mathbf{f}\ y_{i1}\ldots\mathbf{f}\ y_{im}\rangle = \mathsf{c}_i\langle\vec{a}_i,\vec{y}_i\rangle$$

To show this, it suffices to show that $\mathbf{f}\ y_{ij} = y_{ij}$ for all i,j. This is exactly what we get from the inductive hypothesis, so $\mathbf{f}\ x = x$ for all x.

This is another important optimisation in a dependently typed setting which we would not expect to have to deal with in a simply typed setting. Such "invariant management" functions may often be used in typechecking and it is clearly desirable that we do not get a corresponding run-time effect of taking structures apart only to put them back together again immediately. Note that this transformation does not consider the possibility of mutually recursive no-operations. In such a case, a more sophisticated analysis is required.

6.2.4 Rewriting of False-Elim

So far we have been examining transformations which exist primarily to catch up with the position where we can start optimising simply typed programs. This in itself is a good thing;

several well understood optimisations are now open to us such as strictness analysis, tail recursion optimisation, deforestation, several lower level code transformations from [San95] and so on. But now, on top of these, we finally see an example of how dependent types can *further* optimise programs.

The empty type False, is declared as follows:

$$\underline{\text{data}} \quad \frac{}{\mathsf{False} : \star} \quad \underline{\text{where}}$$

There are no constructors (i.e., no canonical forms) and, correspondingly, the elimination rule has no ι-schemes and hence no reduction behaviour. At run-time, where elimination rules are only executed when applied to canonical forms, we can be sure that **False-Elim** will *never* be executed, because False has no canonical forms. What are the consequences of this?

Recall that at run-time all arguments to functions must be reducible to a canonical form. Since False has no canonical forms, we can be sure that *any* function taking an argument of type False (or indeed any type with no constructors) will never be executed. Also, a function which returns an instance of False can never produce such an instance so it, too, will never be executed. We introduce a new constant, Impossible into RunTT to indicate that an expression cannot be evaluated. Compilation of this constant produces code which places a dummy value on the stack; this value cannot be examined since it has no canonical forms, nor can we generate any code which attempts to examine it.

$$\begin{array}{|l|}\hline \mathcal{E}[\![\underline{\text{Impossible}}]\!]\ r\ n \implies \mathsf{ALLOC} \\ \mathcal{C}[\![\underline{\text{Impossible}}]\!]\ r\ n \implies \mathsf{ALLOC} \\ \hline \end{array}$$

Figure 6.5: Compilation of Impossible

The ALLOC instruction (figure 6.6) pushes a dummy value, HOLE, onto the stack. In practice, we never expect to build such a value in a lazy setting. ALLOC was used in Johnsson's original G-machine to allocate space for the results of letrec expressions.

$$\boxed{\langle \mathsf{ALLOC};\ c, S, V, G, E, D\rangle \implies \langle c, n'.S, V, G[n' = \mathsf{HOLE}], E, D\rangle}$$

Figure 6.6: Operational semantics of ALLOC

Given a function

$$\mathbf{f} \mapsto \lambda \vec{a}.\ e$$

If a_i : False for any i, or f : $\forall \vec{a} : \vec{A}$. False, then we modify the definition of **f** to

$$\mathbf{f} \mapsto \lambda \vec{a}.\ \underline{\text{Impossible}}$$

This transformation, impossible expression elimination, is defined as in figure 6.7. The condition that T is a type with no elements may not be decidable, therefore in practice we take as the condition that T is a family with no constructors.

Such functions obviously have all arguments in $\vec{a}$ unused, and can be inlined. This is one example of a situation where retaining the types on supercombinators is convenient. It is preferable to introduce Impossible as a keyword in RunTT rather than ExTT since introducing it in ExTT would cause some terms not to be typecheckable.

$[\![\lambda \vec{a}.\ e]\!] \Longrightarrow \lambda \vec{a}.\ \underline{\text{Impossible}}$
if $\exists\, i$ such that $a_i\ :\ T$
where T is a type with no elements.

Figure 6.7: Impossible expression elimination

An obvious example of a function which takes an argument of type False is False-**Elim**. Recall from Section 4.4.2 the difficulty of compiling the ι-schemes for False-**Elim** — by this transformation however, the supercombinator built for False-**Elim** is straightforward:

False-**Elim** $\mapsto \lambda x; P.\ \underline{\text{Impossible}}$

When combined with other transformations, particularly inlining, this is a powerful optimisation technique. We will see shortly in section 6.2.6 how marking of impossible to execute functions leads to the removal of impossible case branches.

This transformation relies on the strong normalisation property of TT; there is no $\bot$ in EPIGRAM. If we were to remove the strong normalisation restriction from TT and allow either or both of general recursion and partial function definitions, we would be able to create a value of type False, which would invalidate this transformation. One way to do this is by defining a general recursive function **absurd** as follows:

$$\underline{\text{let}}\quad \frac{}{\mathbf{absurd}\ :\ \mathsf{False}}\quad \mathbf{absurd} \mapsto \mathbf{absurd}$$

Another way is to use a partial function definition, and take the head of an empty list:

$$\underline{\text{let}}\quad \frac{l\ :\ \mathsf{List}\ A}{\mathbf{head}\ l\ :\ A}\quad \mathbf{head}\ (\mathsf{cons}\ x\ xs) \mapsto x$$

$$\underline{\text{let}}\quad \frac{}{\mathbf{absurd}\ :\ \mathsf{False}}\quad \mathbf{absurd} \mapsto \mathbf{head}\ \mathsf{nil}$$

Since TT terms are strongly normalising, however, we need not worry about this and can be certain that an element of False cannot be constructed.

6.2.5 Distributing Applications of case

If the result of a case expression is applied to some expression x, then x will be applied to whatever the result of the case application is. In this situation, we move the application to each case branch (figure 6.8). The advantage of this transformation is that it opens up

possible inlining in each case branch by ensuring that each branch is as fully applied as possible. Inlining is important here since it can help to explicitly identify impossible cases.

$$
\begin{array}{|ll|}
\hline
[\![(\underline{\text{case}}\ e\ \underline{\text{of}} & \Longrightarrow\ \underline{\text{case}}\ e\ \underline{\text{of}} \\
\quad \mathsf{c}_1\ \vec{a}_1\ \vec{y}_1\ \rightsquigarrow\ e_1 & \quad \mathsf{c}_1\ \vec{a}_1\ \vec{y}_1\ \rightsquigarrow\ e_1\ x \\
\quad \ldots & \quad \ldots \\
\quad \mathsf{c}_n\ \vec{a}_n\ \vec{y}_n\ \rightsquigarrow\ e_n)\ x]\!] & \quad \mathsf{c}_n\ \vec{a}_n\ \vec{y}_n\ \rightsquigarrow\ e_n\ x \\
\hline
\end{array}
$$

Figure 6.8: Distributing Applications

6.2.6 Impossible Case Deletion

If a case branch leads to impossible to execute code, that case branch can be deleted; there is no point in generating code for an error condition since we know error conditions can never be reached. Figure 6.9 shows the transformation.

$$
\begin{array}{|ll|}
\hline
[\![\underline{\text{case}}\ e\ \underline{\text{of}} & \Longrightarrow\ \underline{\text{case}}\ e\ \underline{\text{of}} \\
\quad \mathsf{c}_1\ \vec{a}_1\ \vec{y}_1\ \rightsquigarrow\ e_1 & \quad \mathsf{c}_1\ \vec{a}_1\ \vec{y}_1\ \rightsquigarrow\ e_1 \\
\quad \ldots & \quad \ldots \\
\quad \mathsf{c}_i\ \vec{a}_i\ \vec{y}_i\ \rightsquigarrow\ \underline{\text{Impossible}} & \quad \mathsf{c}_n\ \vec{a}_n\ \vec{y}_n\ \rightsquigarrow\ e_n \\
\quad \ldots & \\
\quad \mathsf{c}_n\ \vec{a}_n\ \vec{y}_n\ \rightsquigarrow\ e_n]\!] & \\
\hline
\end{array}
$$

Figure 6.9: Impossible Case Deletion

The possibility of this transformation arises after inlining of functions which cannot be executed either due to returning False or taking an argument of type False.

case Collapsing

Santos [San95] lists several transformations which eliminate case expressions. These are:

- **case reduction**, where the case expression scrutinises a constructor application, a variable which has already been scrutinised, or a variable let bound to a constructor application.
- **case elimination**, which eliminates case expressions on already evaluated unboxed values. Such case expressions exist to evaluate the unboxed value.
- **case merging**, which combines two case expressions which scrutinise the same variable.
- **case error**, which reduces case $\bot$ of ... to $\bot$.

- **default binding elimination**, which removes a default binding if the variable it binds is unused.
- **dead alternative elimination**, which removes alternatives which cannot apply, given previous case expressions. This is similar to our impossible case deletion, but is based on analysis of code rather than types.

Many of these also apply in RunTT, but we have another possibility, which arises from the fact that $\perp$ is not a value in EPIGRAM. If deletion of impossible cases and dead alternative elimination leave only one option, then we no longer need to examine the scrutinee — we already know what its value must be! This is only possible in the absence of $\perp$; otherwise, $\perp$ is an element of every type and can always be a possibility during case analysis.

Example – vTail

Consider again **vTail**, where the following simple definition hides an elaborate proof that the empty vector case is impossible:

$$\begin{array}{l}\underline{\text{let}} \quad \dfrac{v : \mathsf{Vect}\ A\ (\mathsf{s}\ n)}{\mathbf{vTail}\ v\ :\ \mathsf{Vect}\ A\ n} \\ \quad \mathbf{vTail}\quad v \quad \Leftarrow \mathsf{Vect\text{-}Case}\ v \\ \quad \mathbf{vTail}\ (a{::}\ v) \ \mapsto\ v\end{array}$$

It takes a number of elaboration and transformation steps before we are in a position to apply any impossible case deletion at the RunTT level. The details of the elaboration and compilation are given in appendix A; here we will only consider the final stages, in two settings — firstly, where Vect has had the forcing optimisation only applied (figure 6.10), and secondly, where it has had the detagging optimisation applied (figure 6.11).

$$\begin{array}{|l|}\hline \mathbf{vTail} \mapsto \lambda A; n; v.\ \ \underline{\text{case}}\ v\ \underline{\text{of}} \\ \qquad\qquad\qquad\qquad\qquad\qquad \epsilon\langle\rangle \ \rightsquigarrow\ \underline{\text{Impossible}} \\ \qquad\qquad\qquad\qquad\qquad\qquad ::\langle x, xs\rangle \ \rightsquigarrow\ (v!1) \\ \hline\end{array}$$

Figure 6.10: **vTail** supercombinator for forced Vect

In the case of forced vectors, the ϵ case branch has been explicitly marked as impossible to reach. By impossible case deletion, we can remove the Impossible branch, which results in a case expression with only one possibility:

$$\begin{array}{l}\mathbf{vTail} \mapsto \lambda A; n; v.\ \ \underline{\text{case}}\ v\ \underline{\text{of}} \\ \qquad\qquad\qquad\qquad ::\langle x, xs\rangle \ \rightsquigarrow\ (v!1)\end{array}$$

This results in a case expression which can be collapsed. We know that the only possibility is a :: constructor, so we do not even need to check. Note that if we had xs rather than $(v!1)$ as the right hand side, we would not be able to collapse the case; we would need

to use the case expression to bind xs to the tail of the vector. We only map argument projections back to the name of the argument they project if we still have the name bound by a case expression after all possible case eliminations have been applied. The resulting supercombinators (after removal of unused arguments) are:

$\mathbf{vTail'} \mapsto \lambda v.\, (v!1)$
$\mathbf{vTail} \mapsto \lambda A; n; v.\, \mathbf{vTail'}\ v$

$$\mathbf{vTail} \mapsto \lambda A; n; v.\quad \underline{\text{case}}\ \mathsf{s}\langle n\rangle\ \underline{\text{of}}$$
$$\mathsf{0}\langle\rangle \rightsquigarrow \underline{\text{Impossible}}$$
$$\mathsf{s}\langle k\rangle \rightsquigarrow \underline{(v!1)}$$

Figure 6.11: **vTail** supercombinator for detagged Vect

In the case of detagged vectors (figure 6.11), the scrutinee of the case expression is already a canonical form, so case selection can be made at compile-time rather than run-time before even examining the contents of each case branch. Again, we end up with:

$\mathbf{vTail'} \mapsto \lambda v.\, (v!1)$
$\mathbf{vTail} \mapsto \lambda A; n; v.\, \mathbf{vTail'}\ v$

6.2.7 Interaction Between Optimisations

Several of the optimisations presented here depend on each other — in particular, inlining is important to all of them and applying more transformations opens up more possibility for inlining. On definition of a function, after its elaboration to TT, we apply transformations in the following order:

Transformations in ExTT

- Translate to ExTT from TT applying the forcing, detagging and collapsing markings from Chapter 4 and the GMP transformation from Chapter 5.
- Apply β-reductions. We do this simple optimisation first as it can arise from optimisations already made (forcing in particular) and can open up the possibility of further optimisations.
- Rewrite labelled types and drop labelling annotations.
- Unfold D-**Elim** rules to D-**Case**. This is made possible by the removal of induction hypotheses in the previous stage.
- Translate to RunTT.

Transformations in RunTT

- Inline **D-Case** operators. Later optimisations are applied to case expressions directly and have a different effect depending on the function which uses the case operator, so inlining these early gives the most benefit.
- Apply other inlining transformations generated from previous function definitions.
- Distribute case applications, to create opportunities for inlining and identification of impossible cases.
- Identify expressions which cannot be evaluated due to taking or returning a value in a type with no constructors.
- Apply inlining again, since the last stage may open up more inlinable applications.
- Delete impossible cases which may have arisen from the previous inlining stage.
- Collapse case expressions where possible. These may arise from the previous stage where all but one case branches are impossible.
- Apply inlining again, since new inlining rules may have been generated and new opportunities may have arisen from case collapsing.
- Remove unused arguments.
- Identify No-operations. This can be applied at any time but it makes sense to wait until as many removals as possible have been made since more no-operations may arise as a result.
- Apply a final inlining phase, in particular to inline any applications of **id** generated by the previous stage.

It is not clear, however, what the optimal ordering of transformations is, or even if such an ordering exists. Since many transformations can expose possibilities for further transformations, it may even be preferable to apply groups of transformations iteratively, as is the case with GHC's rewrite rules.

6.3 More Examples

Finally, let us look at how these transformations affect the compilation of some functions we have already seen in a higher level form. First, we look at the RunTT code for the collapsing of the domain predicate for **quicksort**. We then look at projection; firstly, projection of a value from a vector, then the corresponding projection from a value environment indexed over a vector as in the interpreter example in section 4.6.

6.3.1 Collapsing a domain predicate — quicksort

We saw in Chapter 4 how domain predicates for proving termination, such as that for **quicksort**, are collapsible, so that the resulting code has the intended operational semantics of the original program. **quicksort** is defined by means of a view, hence by induction over a proof of the qsAcc predicate:

$$\underline{\text{data}} \quad \frac{l \ : \ \mathsf{List}\,\mathbb{N}}{\mathsf{qsAcc}\ l \ : \ \star}$$

$$\underline{\text{where}} \quad \frac{}{\mathsf{qsNil} \ : \ \mathsf{qsAcc\ nil}}$$

$$\frac{qsl \ : \ \mathsf{qsAcc}\ (\mathbf{filter}\ (<\ x)\ xs) \qquad qsr \ : \ \mathsf{qsAcc}\ (\mathbf{filter}\ (\geq\ x)\ xs)}{\mathsf{qsCons}\ qsl\ qsr \ : \ \mathsf{qsAcc}\ (\mathsf{cons}\ x\ xs)}$$

$$\begin{array}{lll} \mathbf{quicksort} & xs & \Leftarrow \underline{\text{view}}\ \mathbf{allQsAcc}\ xs \\ \mathbf{quicksort} & \mathsf{nil} & \mapsto \mathsf{nil} \end{array}$$

$$\mathbf{quicksort}\ (\mathsf{cons}\ x\ xs)$$
$$\mapsto \mathbf{quicksort}\ (\mathbf{filter}\ (<\ x)\ xs) +\!\!+ \mathsf{cons}\ x\ (\mathbf{quicksort}\ (\mathbf{filter}\ (\geq\ x)\ xs))$$

Since qsAcc is collapsible (by being indexed over the List being sorted), its elimination rule is defined by case analysis on the List. The RunTT supercombinator for **quicksort** is simplified by the following transformations:

- Collapsing of qsAcc and forcing of List (which removes the element type from the structure).
- Unfolding the view rule qsAcc-**View** to qsAcc-**Case**.
- Inlining qsAcc-**Case** to a direct case expression on the List index.

The resulting supercombinator is given in figure 6.12. This version of **quicksort** that we compile is therefore exactly the version we would write in a high level language if we did not have to show termination. The advantage is that we know this general recursive definition must terminate.

$$\begin{array}{lll} \mathbf{quicksort} \mapsto \lambda l.\ \underline{\text{case}}\ l\ \underline{\text{of}} & & \\ \qquad \mathsf{nil}\langle\rangle & \rightsquigarrow & \mathsf{nil}\langle\rangle \\ \qquad \mathsf{cons}\langle x, xs\rangle & \rightsquigarrow & \mathbf{quicksort}\ (\mathbf{filter}\ (<\ x)\ xs) +\!\!+ \\ & & \qquad \mathsf{cons}\langle x, \mathbf{quicksort}\ (\mathbf{filter}\ (\geq\ x)\ xs)\rangle \end{array}$$

Figure 6.12: Supercombinator definition of **quicksort**

6.3.2 Projection from a vector — lookup

The **lookup** function projects the ith value from a vector. It does not need to check for an empty vector, because its type specifies that the input cannot be an empty vector and there can be no overflow. This should be reflected in the compiled code.

$$\underline{\text{let}} \quad \frac{i : \mathsf{Fin}\ n \qquad v : \mathsf{Vect}\ A\ n}{\mathbf{lookup}\ i\ v\ :\ A} \qquad \begin{array}{l} \mathbf{lookup}\ \ \mathsf{f0}\ \ (a :: v) \ \mapsto\ a \\ \mathbf{lookup}\ (\mathsf{fs}\ i)\ (a :: v) \ \mapsto\ \mathbf{lookup}\ i\ v \end{array}$$

The RunTT supercombinator for **lookup** is simplified by the following transformations:

- Detagging of Vect and forcing of Fin.
- Unfolding of the elimination rule Fin-**Elim** to Fin-**Case**.
- Inlining Fin-**Case**.
- Elimination of the impossible case of ϵ.
- Dropping the unused arguments representing the indices of the Fin and Vect.

The resulting supercombinator is as follows:

$$\begin{array}{l} \mathbf{lookup} \mapsto \lambda i; v.\ \underline{\text{case}}\ i\ \underline{\text{of}} \\ \qquad\qquad\qquad\qquad \mathsf{f0}\langle\rangle \ \rightsquigarrow\ v!0 \\ \qquad\qquad\qquad\qquad \mathsf{fs}\langle x\rangle \ \rightsquigarrow\ \mathbf{lookup}\ x\ (v!1) \end{array}$$

Figure 6.13: Supercombinator definition of **lookup**

This supercombinator reflects the fact that no run-time testing is required on the length of the vector — to project values out and make the recursive calls, we simply assume that the vector must be non-empty and project out the relevant argument. Figure 6.14 shows the compiled G-code for the **lookup** function.

```
S[[λi; v. case i of
          f0⟨⟩ ⇝ v!0
          fs⟨x⟩ ⇝ lookup x (v!1)]]
  ⟹ PUSH 1; EVAL;
     CASEJUMP (f0, l1) (fs, l2);
     LABEL l1;
        PUSH 2; EVAL; PROJ 0; MOVE 1; DISCARD 1; JUMP l;
     LABEL l1;
        SPLIT 1; PUSHFUN lookup; PUSH 0; MKAP;
        PUSH 2; PROJ 1; MKAP; EVAL;
        MOVE 2; DISCARD 2;
     LABEL l;
```

Figure 6.14: Compiled G-code for **lookup**

6.3.3 Projection from an environment — envLookup

In the implementation of the well-typed interpreter in section 4.6, we used the **envLookup** function to project a value from the environment. This function was defined as follows:

$$\underline{\text{let}} \quad \frac{v \ : \ \mathsf{Var}\ e\ i\ T \qquad ve \ : \ \mathsf{ValEnv}\ e}{\mathbf{envLookup}\ v\ ve \ : \ T}$$

$$\mathbf{envLookup}\ \ \mathsf{stop}\ \ (\mathsf{extend}\ t\ r) \ \mapsto\ t$$
$$\mathbf{envLookup}\ (\mathsf{pop}\ v)\ (\mathsf{extend}\ t\ r) \ \mapsto\ \mathbf{envLookup}\ v\ r$$

Recall that the value environment is indexed over a type environment (represented by a Vect of types) to ensure that values of the correct type are projected out of the value environment. Again, the type specifies that the environment cannot be empty.

The RunTT supercombinator for **envLookup** is simplified by the following transformations:

- Detagging of ValEnv, forcing of Fin and collapsing of Var such that constructor choice is made by the constructor of Fin.
- Unfolding of the elimination rule Var-**Elim** to Var-**Case**.
- Inlining of Var-**Case** to a direct case expression on its Fin index.
- Elimination of the impossible case of empty environments.
- Dropping the unused arguments representing the indices of the Var, Fin and type and value environments.

The resulting supercombinator is given in figure 6.15.

$$\begin{array}{l}\mathbf{envLookup} \mapsto \lambda i; ve.\ \underline{\text{case}}\ i\ \underline{\text{of}} \\ \qquad\qquad\qquad\qquad \mathsf{f0}\langle\rangle \ \rightsquigarrow\ ve!0 \\ \qquad\qquad\qquad\qquad \mathsf{fs}\langle x\rangle \ \rightsquigarrow\ \mathbf{envLookup}\ x\ (ve!1)\end{array}$$

Figure 6.15: Supercombinator definition of **envLookup**

Perhaps unsurprisingly, the resulting code has the same shape as the code for **lookup**; the only difference in the high level definition is the introduction of several invariants to check that the environments are synchronised. Removal of the invariants leads to code of the same form.

Correspondingly, the G-code for **envLookup** is almost identical to that for **lookup**; the only difference is in the recursive call (which is to **envLookup** rather than **lookup**). Figure 6.16 shows the G-code for **envLookup**.

```
S[[λi; ve. case i of
        f0⟨⟩  ⇝  ve!0
        fs⟨x⟩  ⇝  envLookup x (ve!1)]]
  ⟹ PUSH 1; EVAL;
     CASEJUMP (f0, l1) (fs, l2);
     LABEL l1;
        PUSH 2; EVAL; PROJ 0; MOVE 1; DISCARD 1; JUMP l;
     LABEL l1;
        SPLIT 1; PUSHFUN envLookup; PUSH 0; MKAP;
        PUSH 2; PROJ 1; MKAP; EVAL;
        MOVE 2; DISCARD 2;
     LABEL l;
```

Figure 6.16: Compiled G-code for **envLookup**

6.4 Summary

In EPIGRAM, we build function definitions by elimination rules. This has several advantages — it gives a uniform way to build functions, ensures that functions are terminating by abstracting recursive calls and we have also seen how we can use elimination rules to remove duplicated data. However, they do add an extra level of abstraction; when we have finished with the elimination rule, we would like to remove that level of abstraction and do recursion directly. In this chapter, we have seen a technique for doing so, using labelled types to replace inductive hypotheses with direct recursive calls.

Having removed this level of abstraction, we are now in a position to apply other well-known optimisation techniques. Two very simple techniques are β-reduction and inlining. While they do not in themselves produce a great improvement, their main purpose is to expose other optimisation opportunities. We have seen several examples in this chapter, in particular exposing impossible cases for removal and in extreme cases, such as with **vTail** to remove the case expression completely since only one branch is possible. This ability to do case collapsing when only one case is valid arises from the type system, because canonical values cannot be $\bot$.

Some other optimisations are necessitated by using dependent types, such as the removal of identity functions like **weaken**. Dropping unused arguments is also more important here than in a simply typed language, since several arguments may be added to functions implicitly as the indices of a family. Inductive families are tied to their indices in that they are always passed around with their indices. If a particular function does not use the indices, we would like to avoid passing them to that function, but D-**Case** and D-**Elim** need the indices to pass through to their methods. By removing D-**Elim** and inlining D-**Case**, we can establish which of the indices are unused and remove them.

After applying the optimisations in this chapter, we are in a better position than we would be with simple types — we can now apply several more well known optimisations

for tail recursion, strictness analysis, and so on, but we have already applied additional optimisations based on the dependent type system.

Chapter 7

Conclusions

7.1 Contributions

In this thesis, we have seen several techniques for compiling dependently typed functional programs. The style of programming involves extensive use of indices on inductive families to maintain invariant properties of programs. In the course of developing an implementation of the core language, TT, we have made the following observations:

- Well understood methods, with some minor extensions and modifications, can be used to compile a dependently typed programming language based on inductive families. We get a compiled implementation of TT by translating to supercombinators and G-code. The additional considerations for EPIGRAM are as follows:
  - We need to take account of functions which accept varying numbers of arguments. This is dealt with simply, without needing to modify the lambda lifting process, by a supercombinator of fixed arity returning a value of function type if more arguments are expected.
  - The compilation scheme needs to take account of type constructors. Since we have no means to analyse type constructors at run-time, we need only add one node to the heap to represent all types.
- Implementing TT involves the compilation of pattern matching elimination rules. We have observed that to compile such rules involves dealing with repeated arguments, arbitrary terms, and the presence of presupposed constructor symbols in patterns. Far from making pattern matching more difficult to compile (since we might expect to have to perform a run-time conversion check), we can exploit the fact that all elimination rules are well-defined and respectful — we do not have to test repeated arguments for convertibility since we know by typechecking that such arguments *must* be convertible. This analysis of the patterns in an elimination rule leads to three optimisations; forcing,

detagging and collapsing. Forcing and detagging remove parts of structures which represent information duplicated elsewhere whereas collapsing removes entire data structures, meaning that a program can be defined by induction over a proof without that proof having to be stored at run-time (domain predicates being an important example of this).

- We have considered the implementation of a numerical data type in EPIGRAM. To do so in core EPIGRAM is possible, however in a practical implementation we would like to make use of the arithmetic operations available on the underlying CPU. To this end, we have seen transformation rules inserted into the compilation process to translate the unary definition of $\mathbb{N}$ into a GMP based implementation of big numbers.

- Impossible case removal is an optimisation which requires complex static analysis in a simply typed language, but is easily implementable in our setting by analysis of types and application of a set of fairly obvious program transformations. We can remove impossible cases by observing that an element of the empty type, False, cannot be constructed.

While there are obvious overheads in a naïve implementation of TT, by a series of remarkably straightforward transformations we can remove these overheads and even achieve optimisations which are not obviously available in equivalent simply typed programs.

7.2 Conclusions

Programming in EPIGRAM is based on using elimination rules to implement the pattern matching behaviour of functions. While the high level notation involves writing functions in pattern matching form, the elaboration of these definitions into TT gives a definition in terms of elimination rules derived from data declarations. Effectively, these elimination rule based definitions correspond to compilation to simple case expressions. An elimination rule D-**Elim** and its variants D-**Case** and D-**View** are high level abstractions of case expressions and can be translated into case expressions by a simple unfolding and inlining transformation, as seen in Chapter 6.

Elimination rules implement pattern matching, and are always used down to the RunTT level (at least until D-**Case** operators are inlined) to abstract pattern matching. Only elimination rules have access to the actual data; as such, we are free to choose any concrete representation for a data type provided that:

- The implementation of its elimination rule knows how to choose the appropriate ι-scheme based either on its own representation, or the representation of its indices.

- All other elimination rules know how to discriminate on its representation, if necessary for detagging.

- If its elimination rule cannot discriminate on its own representation (as is the case with detagged families) then no other elimination rule will attempt to discriminate on its representation.

While pattern matching and fixpoint equations are often considered better as they are more readable for programmers [Coq92], elimination rules have advantages for implementation purposes, and so EPIGRAM translates pattern matching definitions into elimination rule based definitions. A further advantage is that implementation by elimination rules provides an optimisation opportunity; moving all case analysis on a datatype into one place means that it is easier to change the representation of that datatype, as we did with the forcing, detagging and collapsing optimisations in Chapter 4.

This is also why we can choose a GMP implementation of $\mathbb{N}$ — only $\mathbb{N}$-**Elim** and elimination rules which discriminate on $\mathbb{N}$s need to know how to discriminate between 0 and $n + 1$. It is conceivable that we could implement other datatypes externally in the same way — an implementation of Vect may, for example, simply be an appropriate sized block of memory. As long as the elimination rule knows how to discriminate between empty and non-empty Vects (which it can do on length, as we know from detagging) and can extract the head and tail of non-empty Vects, then we can choose this implementation. Optimisations of data structures arise from analysis of elimination rules; forcing, detagging, collapsing, and the transformation of $\mathbb{N}$ to a GMP representation all arise by such an analysis (forcing, detagging and collapsing are automatic, the $\mathbb{N}$ transformation is not).

Ultimately, compilation of TT to an executable form is by using standard techniques with small modifications. The modifications we made to the G-machine were simply a graph node for holding types and argument projection for data structures for use in forced and detagged elimination rules. We also have a modified pattern matching compilation scheme for ι-schemes; this does not need to be as general as a scheme for pattern matching definitions in a simply typed language because of the restriction that ι-schemes must be respectful and well-defined. In particular, we have no need to check for unmatched patterns; there can be no error case. Given the minimal modification made to the G-machine, we can expect the same modifications to be applicable to more sophisticated and efficient run-time systems, such as GRIN [BJ96, Boq99] or the STG-machine [Pey92].

The removal of domain predicates (such as in showing termination of **quicksort**) is an important application of the collapsing technique. Bove points out [Bov02a] that if we suppress the proofs of the domain predicate, we get almost exactly the original algorithm. This is certainly true for the purposes of display and understanding, but the usual method for suppressing proofs *at run-time* (by making them part of the logical Prop universe and not allowing computation over them) does not work; we need to be able to write the function by induction over these proofs. Collapsing provides a method for actually removing proofs of the domain predicate at run-time.

Some of the techniques we have seen can also apply to program extraction, particularly as

implemented in COQ [PM89, Let02]. The difficulty is in translating to the `Case/Fix` setting, although implementing case as an operator **D-Case**, abstracting away the case analysis as with elimination rules, is a possible approach. The current COQ extraction system does not remove forced arguments from inductive families; it is primarily designed for extracting programs built from a specification which pairs a result with proofs of properties of that result. The forcing optimisation would improve code extracted from indexed inductive families. Collapsing would also be beneficial; extraction aims to remove logical parts from proofs and retain computational parts. A collapsible data structure describes some other computation (such as the domain predicate for **quicksort**) and as such is not itself a computational part; removing such a structure would be a valuable optimisation for extracted code.

In imperative and simply typed functional languages, sophisticated techniques are necessary to apply dead code elimination. In DML, Xi shows how constraint checking can be used to eliminate unreachable case branches [Xi99a]. In our setting, with full inductive families, the compile-time approach is even simpler — any function which takes an argument of a type with no constructors (e.g. False), or returns a value in a type with no constructors, can be replaced by the constant Impossible, leading to obvious transformations on RunTT case expressions. Values of type False arise from the equational reasoning performed by the elaborator on the indices of a family; it is the use of inductive families which allows impossible cases to be deleted easily.

Array bounds check elimination is an optimisation which arises from Xi's work [XP98] with DML, where expressing constraints on function types results in the removal of bounds checking code at run-time. The **lookup** function over the Vect family demonstrates a similar optimisation in an inductive family based language. We never check the vector is empty because the type proves that it cannot be. Again, with **lookup**, the impossible cases of the empty vector are eliminated. This kind of optimisation is likely to come up often in practice where a function's domain type covers only part of a family — we see examples in the interpreter at the end of Chapter 4 and the implementation of big number arithmetic in Chapter 5.

The techniques described here depend on the knowledge gained from the type system. However, many of them also depend on terms being strongly normalising. Without strong normalisation, we can build a value of type False (although, obviously, not a canonical value since there are none). If we can build an element of the empty type, we have arguments which we can pass to **False-Elim**, which does not have ι-schemes. It does, however, allow us to build non-canonical but type correct terms which prove something that ought to be unprovable. For example, we can use a function **absurd** : False to build the proof of $0 = sn$ which makes an application of **vTail** to an empty list type correct. Without strong normalisation, we must introduce checks into the run-time system which make sure a term is canonical before it is reduced; as soon as the possibility of non-canonical terms at run-time is introduced, we lose the possibility of collapsing and impossible case branch elimination. Forcing and detagging are still applicable however. Strong normalisation is

even more important than we first thought — not only do we need it to ensure decidability of typechecking, we also need it to make full use of types in optimisation.

The real question is whether the inductive family based programming paradigm can compete with more mainstream programming paradigms. It is clear that programs based on inductive families are safer, in that their type specifies more precisely what the program does and hence gives the compiler more possibility of identifying errors at compile-time. However, does this lead to slower, more memory-intensive performance at run-time? At this stage, our implementation is not mature enough to give solid results for comparison, nor is there a sufficient body of EPIGRAM programs to get real world examples. However, the nature of the code which is generated (both at the RunTT level and the compiled G-machine code) with several run-time checks eliminated and no obvious redundant data or arguments suggests that dependently typed programs can have at least as efficient a run-time performance as simply typed programs; when run-time checks which would otherwise be present are eliminated due to the richness of the type system (for example in **lookup**, **vTail**) this suggests that dependently typed programs can ultimately be more efficient than simply typed programs.

The techniques described in this thesis show that the style of programming implemented by EPIGRAM is a feasible approach to generating safe and efficient code at run-time — apparent overheads are removable by remarkably straightforward analysis of elimination rules and further optimisations arise directly from typing constraints. I believe that dependent types will lead to programs which are faster and more easily shown correct than their simply typed counterparts.

7.3 Further Work

Programming with inductive families as in EPIGRAM is an innovative approach to programming; until [McB00a] and [MM04b] the main focus of research into dependent types was for theorem proving and program verification using systems such as COQ and LEGO. This thesis has presented a first implementation of compilation techniques specifically designed with dependently typed programming in mind. As such, it raises several questions and suggests many possible directions for future work. We will finish by examining these questions and considering how further work may proceed.

At the time of writing, the EPIGRAM front end is still in development — an early version has recently been released, but the majority of the work presented in this thesis was carried out in a theorem prover based on TT, using small example programs which were elaborated either by hand or with the help of other theorem proving systems. When the system is stable, we need to write programs both to demonstrate the advantages of dependently typed programming for ensuring program correctness and to have a more realistic body of programs with which to test the run-time efficiency. Many of the examples of this thesis could be extended or adapted; in particular it would be interesting to develop a compiler for a subset

of Haskell in EPIGRAM.

The compilation techniques discussed in this thesis are geared towards compilation of code for run-time execution (although we have seen that the forcing and detagging optimisations in particular are also applicable at compile-time). The execution strategy we have examined, via RunTT and G-machine code, is designed for run-time execution only. However, it is also worth considering building abstract machine code for compile-time execution by the typechecker, as in [GL02]. This work improves the speed of typechecking in COQ substantially for theorems involving a large amount of computation, although for the COQ standard library the speed is close to the original implementation. Checking the standard library requires little computation; we might expect more in programs which use inductive families heavily and so this approach is worth considering. Grégoire and Leroy implement strict evaluation, whereas we have used lazy evaluation for EPIGRAM — their techniques are nevertheless adaptable to lazy evaluation by adding a new heap node type to the G-machine for free variables.

EPIGRAM is based on a strongly normalising dependent type theory. The strong normalisation property presents several possibilities for optimisation although many of these have not yet been investigated. In a strongly normalising language choice of reduction order is less important — whatever happens, the program will terminate, although choice of redex can determine how quickly reduction reaches a normal form. If terms are not strongly normalising, we have to be careful with optimisation due to the undecidability of the Halting Problem; in a Turing complete language we cannot evaluate arbitrary subexpressions at compile-time since they might not terminate. A lot of effort can be spent in a compiler for a lazy language on finding which subexpressions can be evaluated strictly without causing a program to loop forever due to the evaluation of an infinite structure, e.g. [CP85]. However, since we have strong normalisation for EPIGRAM, we can safely choose to evaluate *any* subterm strictly. We originally chose lazy evaluation because of the number of values (implicit arguments to both functions and constructors in particular) which exist only for typechecking and which never need to be evaluated at run-time. In the presence of our optimisations, perhaps we should reconsider this choice. There are still many problems where lazy evaluation is a more attractive choice — search problems are an example, where we build a search tree for the whole search space, but only evaluate a small part of this tree — perhaps we should default to strict evaluation and limit lazy evaluation to such problems. Robert Ennals, in his thesis on adaptive evaluation strategies [Enn03], reaches the conclusion that it is better to default to strict evaluation and annotate programs where laziness is required. Further investigation is required on the benefits of each evaluation strategy in a strongly normalising language.

Many of our optimisations are based on changing the implementation of a family's elimination rules so that the family can be stored in a more efficient way. Optimising the elimination rule has the consequence of optimising programs which elaborate in terms of it. Hence, we might not only consider implementations which allow more efficient storage of data, but

also implementations which traverse data structures in a more efficient manner. We briefly considered an iterative implementation of $\mathbb{N}$-**Elim** in Chapter 5; traversal of Lists and Vects is also an iterative process (since the structures are linear) so the recursive elimination rules we generate are perhaps not the best implementation. Making functions tail-recursive is well known as an important optimisation in functional programming [Ste77, LS00]; we ought to look for such an optimisation in compiler generated elimination rules, since these rules form the basis of all computation in EPIGRAM. There are several things to consider in making elimination rules for Lists and Vects iterative — it may involve changing the order of traversal (right to left, rather than left to right) or even changing the internal representation of the data structure.

There are some limits to the forcing optimisation as implemented in Chapter 4. Not all forceable arguments are *concretely* forceable, as forcing relies on identifying the inverse of injective functions for which we do not have a decision procedure in general. This means that, potentially, we are storing duplicate values without being able to tell they are duplicates. For example, we could index a binary tree over the number of items stored at the leaves:

$$\underline{\text{data}} \quad \frac{A : \star \quad n : \mathbb{N}}{\mathsf{Tree}\ A\ n\ :\ \star}$$

$$\underline{\text{where}} \quad \frac{a\ :\ A}{\mathsf{Leaf}\ a\ :\ \mathsf{Tree}\ A\ (\mathsf{s0})} \qquad \frac{l\ :\ \mathsf{Tree}\ A\ n \quad r\ :\ \mathsf{Tree}\ A\ m}{\mathsf{Node}\ l\ r\ :\ \mathsf{Tree}\ A\ (\mathbf{plus}\ n\ m)}$$

We cannot drop both n and m from the arguments to the Node constructor, but in theory we can work out one from the other. In practice, however, the forcing optimisation keeps both. Possible solutions involve allowing the user to specify how to compute a value which is forceable, but not concretely forceable, or even allowing the user to specify that a value is unused (and therefore deletable) at run-time and then checking that it really is unused. Similar problems apply to the detagging and collapsing optimisations, where a value may be detaggable or collapsible, but not concretely so. Many views are collapsible, for example — Compare, however, is a view which is collapsible but not concretely collapsible.

In Chapter 5 we saw how an external implementation of $\mathbb{N}$ could be used to optimise arithmetic. We could imagine extending this to give low level implementations of other common data structures, List and Vect being obvious examples. To do this would be to adopt an opposing philosophy to that adopted in the design of the STG machine; a design philosophy of the STG machine is that user defined types should be efficient enough that the same technology can apply to built in types and standard library types (such as lists). However, where efficient external implementations exist it makes sense to make use of them, particularly when applying the optimisation is a simple matter of replacing the constructors and elimination rule with appropriate alternatives. Introducing primitives also encourages us to think about unboxing representations; to implement unboxing in polymorphic functions, we can consider introducing a type level case construct for run-time type analysis as in [HM95]. The overheads of this approach, namely that types (in many cases) need to be stored at run-time are potentially outweighed by the benefits of unboxing. Of course, in any case

where types remain unused they can still be deleted by the optimisations of Chapter 6. We can also use external types in another way, by defining abstract datatypes and an interface — e.g., a mathematical program may wish to make use of an external implementation of floating point values and associated operations. In this setting, the abstract datatype has no constructors or elimination rule, but simply a set of functions (with EPIGRAM types). This would require us to think about the structure of a module system for EPIGRAM, perhaps following some of the ideas of the recently introduced COQ module system [Chr04].

An effect of the forcing optimisation is that it changes the *shape* of a data type [Jay96]. The shape of a data type refers to its structure and the "holes" where data can be inserted. The forcing optimisation changes Fin to a type with a constructor of no arguments, and a constructor with a recursive argument. This resulting shape of Fin is the same as that of $\mathbb{N}$; it follows that optimisations which apply to $\mathbb{N}$ ought to apply to Fin too — we could, for example, reasonably store Fin as a GMP integer. We might even be able to go further with Fin, since its upper bound is known from the type, and store it as a machine integer. Note also that the value environments in Chapter 4's interpreter have the same shape as Vect after forcing and detagging. If we have a low level implementation of Vect (for example as a block of memory), a low level implementation of value environments follows. We saw in Chapter 6 that this also leads to projection functions for Vect and ValEnv having the same G-code. Low level implementation of a lookup function on Vect (for example, by directly inspecting the ith location in a block of memory) ought therefore to lead to a low level implementation of a lookup function on value environments. This kind of optimisation should take place at the RunTT level; if constructors are represented not by their names, but by an index into a jump table of ι-reductions, such optimisations become easier to identify.

The implementation described in this thesis uses well understood technology, but with known limitations. According to Santos, an implementation based on λ-lifting suffers a run-time penalty compared with one which can deal with free variables [San95]. The G-machine is perhaps not abstract enough; there are too many low level details, such as the use of a stack for local variables, which may not map as directly as we might hope onto a real CPU. Such limitations are dealt with in recent implementations of GHC [Pey92, PMR99, SMG+99] and can be adapted to a dependently typed language in a similar way to the adaptation of the G-machine in this thesis. While the results of this thesis show that a dependently typed programming language is feasible to implement, we would ultimately like to have a complete implementation giving us a real execution platform for comparison with other languages.

Appendix A

Compiling vTail

The **vTail** function, which returns the tail of a non-empty vector, has a simple definition which hides a complex elaboration:

$$\begin{array}{ll} \underline{\text{let}} & \dfrac{v : \mathsf{Vect}\ A\ (\mathsf{s}\ n)}{\mathbf{vTail}\ v\ :\ \mathsf{Vect}\ A\ n} \\ \quad \mathbf{vTail} & v \quad \Leftarrow \underline{\text{case}}\ v \\ \quad \mathbf{vTail}\ (a{::}v) & \mapsto\ v \end{array}$$

By examining the input type $\mathsf{Vect}\ A\ (\mathsf{s}\ n)$ we see that ϵ is an impossible case, since it has the type $\mathsf{Vect}\ A\ 0$ which does not convert with the input type. This much is clear to see, but how does the elaboration mechanism know that $\mathbf{vTail}\ (a{::}v)$ is the only case and how does it produce a valid term in TT?

A.1 vTail elaboration – a first attempt

A first attempt at elaborating the definition of **vTail** into TT meets with some difficulties. Applying **Vect-Case** to v immediately and filling in the case for $a{::}v$, the resulting term, still leaving out the implicit arguments A and n:

$$\begin{array}{lll} \mathbf{vTail} \mapsto & \lambda A{:}\star.\ \lambda n{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ (\mathsf{s}\ n). & \\ & \mathbf{Vect\text{-}Case}\ v & \text{(Target)} \\ & (\lambda k{:}\mathbb{N}.\lambda v{:}\mathsf{Vect}\ A\ k.\mathsf{Vect}\ A\ n) & \text{(Motive)} \\ & (\Box\ :\ \mathsf{Vect}\ A\ n) & \text{(Method for } \epsilon) \\ & (\lambda k{:}\mathbb{N}.\lambda A{:}\star.\lambda v{:}\mathsf{Vect}\ A\ k.v) & \text{(Method for ::)} \end{array}$$

This attempt runs into trouble with the case for ϵ. The metavariable to fill in is the method for this case and we have neither a value of type $\mathsf{Vect}\ A\ n$ nor a means of making one. Somehow the information that this case is impossible has been lost — the simple reason for this is that the motive of the elimination is not expressive enough. If we include this information in the motive then we retain enough information to fill in the case.

To do this, we construct proofs of equalities which must hold and pass them into the motive; this is the basis of the **elimination with a motive** technique [McB00b].

A.2 vTail elaboration – second attempt

The first step in elaborating **vTail** is not to apply the elimination rule, but rather to modify the input to be a vector of length k along with a proof that $k = \mathsf{s}\,n$. This proof then becomes part of the motive which results in the case for the empty vector being passed a proof that $0 = \mathsf{s}\,n$. This is clearly a contradiction from which we can construct an element of the empty type. From here, we can prove anything, including an impossible case.

To introduce the equality proof we wrap the body of the definition inside a λ abstraction applied to the proof:

$$
\begin{aligned}
\textbf{vTail} \mapsto\ & \lambda A{:}\star.\, \lambda n{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ (\mathsf{s}\ n).\\
& (\lambda k{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ k.\\
& \lambda P{:}\forall k{:}\mathbb{N}.\, \forall v{:}\mathsf{Vect}\ A\ k.\, (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n.\\
& P\ (\mathsf{s}\ n)\ v\ (\mathsf{refl}\ (\mathsf{s}\ n)))\\
& n\ v\ (\Box\ :\ \forall k{:}\mathbb{N}.\, \forall v{:}\mathsf{Vect}\ A\ k.\, (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n)
\end{aligned}
$$

The goal we are left with on applying this proof, $\forall k{:}\mathbb{N}.\, \forall v{:}\mathsf{Vect}\ A\ k.\, (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n$, has the same meaning as the type of **vTail**, namely that any vectors to which it applies must be of non zero length, with the difference that the proof is passed explicitly, rather than implicit in the type.

We now define the helper function, **vTailAux**.

$$
\begin{aligned}
\textbf{vTailAux}\ :\ & \forall n{:}\mathbb{N}.\, \forall A{:}\star.\, \forall k{:}\mathbb{N}.\, \forall v{:}\mathsf{Vect}\ A\ k.\, (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n\\
\textbf{vTailAux} \mapsto\ & \lambda n{:}\mathbb{N}.\, \lambda A{:}\star.\, \lambda k{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ k.\\
& \textbf{Vect-Case}\ v\\
& (\lambda k{:}\mathbb{N}.\lambda v{:}\mathsf{Vect}\ A\ k.(\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n)\\
& (\Box\ :\ (\mathsf{s}\ n\ =\ 0) \to \mathsf{Vect}\ A\ n)\\
& (\lambda k{:}\mathbb{N}.\lambda a{:}A.\lambda v{:}\mathsf{Vect}\ A\ k.\Box\ :\ (\mathsf{s}\ n\ =\ \mathsf{s}\ k) \to \mathsf{Vect}\ A\ n)
\end{aligned}
$$

The motive now holds the equality proof which means that we have enough information to eliminate the ϵ case. We write an auxiliary function to show that $\mathsf{s}\ n\ =\ 0$ gives us an element of the empty type.

$$
\begin{aligned}
\textbf{dMotive}\ :\ & \forall n{:}\mathbb{N}.\, \star\\
\textbf{dMotive} \mapsto\ & \lambda n{:}\mathbb{N}.\mathbb{N}\textbf{-Case}\ n\ (\forall n{:}\mathbb{N}.\star)\ \mathsf{False}\ (\lambda k{:}\mathbb{N}.\mathsf{True})\\
\textbf{discriminate}\ :\ & \forall n{:}\mathbb{N}.\, \forall p{:}\mathsf{s}\ n\ =\ 0.\ \mathsf{False}\\
\textbf{discriminate} \mapsto\ & \lambda n{:}\mathbb{N}.\, \lambda p{:}\mathsf{s}\ n\ =\ 0.\\
& =\textbf{-elim}\ \mathbb{N}\ (\mathsf{s}\ n)\ p\ \textbf{dMotive}\ ()
\end{aligned}
$$

dMotive computes the return type for **discriminate**. If the second item in the equality is 0, then we return an element of the empty type, otherwise we return an element of the unit

type. Since $=$ **-elim** only requires a method for when the second item is equal to the first, this means we only need provide an element of True to complete the proof – but since we know the second element is zero, the empty type is returned.

Now that we have an element of the empty type we can prove anything, so it is trivial to construct the element of Vect $A\ n$ required in the ϵ case:

$$\begin{aligned}
&\textbf{emptyCase} : && \forall A{:}\star.\, \forall n{:}\mathbb{N}.\, (\mathsf{s}\ n\ =\ 0) \to \mathsf{Vect}\ A\ n \\
&\textbf{emptyCase} \mapsto && \lambda A{:}\star.\, \lambda n{:}\mathbb{N}.\, \lambda p{:}\mathsf{s}\ n\ =\ 0. \\
& && \mathsf{False}\textbf{-Elim}\ (\textbf{discriminate}\ n\ p)\ (\mathsf{Vect}\ A\ n)
\end{aligned}$$

Filling in the hole for the ϵ case, this leaves us with the :: case:

$$\begin{aligned}
&\textbf{vTailAux} : && \forall n{:}\mathbb{N}.\, \forall A{:}\star.\, \forall k{:}\mathbb{N}.\, \forall v{:}\mathsf{Vect}\ A\ k.\, (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n \\
&\textbf{vTailAux} \mapsto && \lambda n{:}\mathbb{N}.\, \lambda A{:}\star.\, \lambda k{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ k. \\
& && \mathsf{Vect}\textbf{-Case}\ v \\
& && (\lambda k{:}\mathbb{N}.\lambda v{:}\mathsf{Vect}\ A\ k.(\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n) \\
& && (\textbf{emptyCase}\ A\ n) \\
& && (\lambda k{:}\mathbb{N}.\lambda a{:}A.\lambda v{:}\mathsf{Vect}\ A\ k.\Box\ :\ (\mathsf{s}\ n\ =\ \mathsf{s}\ k) \to \mathsf{Vect}\ A\ n)
\end{aligned}$$

The :: case requires a function from $\mathsf{s}\ n\ = \mathsf{s}\ k \to k\ =\ n$ and rewriting with $=$ **-elim** to complete the proof. We use **S_inj** : $\forall n, m{:}\mathbb{N}.\, \mathsf{s}\ n\ = \mathsf{s}\ m \to n\ =\ m$ and **eq_sym** : $\forall A{:}\star.\, \forall x, y : A.\ x\ =\ y \to y\ =\ x$ to rewrite the equality, and define the case for :: with the following function, **consCase**:

$$\begin{aligned}
&\textbf{consCase} : && \forall A{:}\star.\, \forall n{:}\mathbb{N}.\, \forall k{:}\mathbb{N}.\, \mathsf{Vect}\ A\ k \to (\mathsf{s}\ n\ =\ \mathsf{s}\ k) \to \mathsf{Vect}\ A\ n \\
&\textbf{consCase} \mapsto && \lambda A{:}\star.\, \lambda n{:}\mathbb{N}.\, \lambda k{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ k.\, \lambda p{:}k\ =\ n. \\
& && =\textbf{-elim}\ \mathbb{N}\ k\ n\ (\textbf{S_inj}\ k\ n\ (\textbf{eq_sym}\ \mathbb{N}\ n\ k\ p))\ (\lambda n{:}\mathbb{N}.\, \mathsf{Vect}\ A\ n)\ v
\end{aligned}$$

Then we can complete the definition of **vTailAux**:

$$\begin{aligned}
&\textbf{vTailAux} : && \forall n{:}\mathbb{N}.\, \forall A{:}\star.\, \forall k{:}\mathbb{N}.\, \forall v{:}\mathsf{Vect}\ A\ k.\, (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n \\
&\textbf{vTailAux} \mapsto && \lambda n{:}\mathbb{N}.\, \lambda A{:}\star.\, \lambda k{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ k. \\
& && \mathsf{Vect}\textbf{-Case}\ A\ k\ v \\
& && (\lambda k{:}\mathbb{N}.\lambda v{:}\mathsf{Vect}\ A\ k.(\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n) \\
& && (\textbf{emptyCase}\ n) \\
& && (\lambda k{:}\mathbb{N}.\, \lambda a{:}A.\, \lambda v{:}\mathsf{Vect}\ A\ k.\, \textbf{consCase}\ A\ n\ k)
\end{aligned}$$

Finally, we can use this helper function to fill in the hole in **vTail**:

$$\begin{aligned}
&\textbf{vTail} \mapsto && \lambda A{:}\star.\, \lambda n{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ (\mathsf{s}\ n). \\
& && (\lambda k{:}\mathbb{N}.\, \lambda v{:}\mathsf{Vect}\ A\ k. \\
& && \lambda P{:}\forall k{:}\mathbb{N}.\, \forall v{:}\mathsf{Vect}\ A\ k.\, (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n. \\
& && P\ (\mathsf{s}\ n)\ v\ (\mathsf{refl}\ (\mathsf{s}\ n))) \\
& && n\ v\ (\textbf{vTailAux}\ n\ A)
\end{aligned}$$

The complete definition of **vTail** is rather complex although the machinery required to produce this is added automatically by the compiler. The user need not worry about

the details of this machinery but it is important for ensuring function totality that this machinery is there.

A.3 Building Supercombinators

For quick reference, the full elaborated definition of **vTail** is given in figure A.1, with the functions in the order in which they will be compiled. We will take each one in turn and consider its compilation into optimised supercombinators, in the presence of forcing of the Vect (but not detagging, or collapsing of equality proofs, since I would like to focus attention on the optimisation of **vTail** in isolation).

$$
\begin{array}{ll}
\textbf{dMotive} : & \forall n{:}\mathbb{N}.\ \star \\
\textbf{dMotive} \mapsto & \lambda n{:}\mathbb{N}.\ \mathbb{N}\textbf{-Case}\ n\ (\forall n{:}\mathbb{N}.\ \star)\ \mathsf{False}\ (\lambda k{:}\mathbb{N}.\ \mathsf{True}) \\
\textbf{discriminate} : & \forall n{:}\mathbb{N}.\ \forall p{:}\mathsf{s}\ n\ =\ 0.\ \mathsf{False} \\
\textbf{discriminate} \mapsto & \lambda n{:}\mathbb{N}.\ \lambda p{:}\mathsf{s}\ n\ =\ 0. \\
 & \quad =\textbf{-elim}\ \mathbb{N}\ (\mathsf{s}\ n)\ p\ \textbf{dMotive}\ () \\
\textbf{emptyCase} : & \forall A{:}\star.\ \forall n{:}\mathbb{N}.\ (\mathsf{s}\ n\ =\ 0) \to \mathsf{Vect}\ A\ n \\
\textbf{emptyCase} \mapsto & \lambda A{:}\star.\ \lambda n{:}\mathbb{N}.\ \lambda p{:}\mathsf{s}\ n\ =\ 0. \\
 & \quad \mathsf{False}\textbf{-Elim}\ (\textbf{discriminate}\ n\ p)\ (\mathsf{Vect}\ A\ n) \\
\textbf{consCase} : & \forall A{:}\star.\ \forall n{:}\mathbb{N}.\ \forall k{:}\mathbb{N}.\ \mathsf{Vect}\ A\ k \to (\mathsf{s}\ n\ =\ \mathsf{s}\ k) \to \mathsf{Vect}\ A\ n \\
\textbf{consCase} \mapsto & \lambda A{:}\star.\ \lambda n{:}\mathbb{N}.\ \lambda k{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ k.\ \lambda p{:}k\ =\ n. \\
 & \quad =\textbf{-elim}\ \mathbb{N}\ k\ n\ (\textbf{S\_inj}\ k\ n\ (\textbf{eq\_sym}\ \mathbb{N}\ n\ k\ p))\ (\lambda n{:}\mathbb{N}.\ \mathsf{Vect}\ A\ n)\ v \\
\textbf{vTailAux} : & \forall n{:}\mathbb{N}.\ \forall A{:}\star.\ \forall k{:}\mathbb{N}.\ \forall v{:}\mathsf{Vect}\ A\ k.\ (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n \\
\textbf{vTailAux} \mapsto & \lambda n{:}\mathbb{N}.\ \lambda A{:}\star.\ \lambda k{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ k. \\
 & \quad \mathsf{Vect}\textbf{-Case}\ A\ k\ v \\
 & \quad (\lambda k{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ k.\ (\mathsf{s}\ n\ =k) \to \mathsf{Vect}\ A\ n) \\
 & \quad (\textbf{emptyCase}\ A\ n) \\
 & \quad (\lambda k{:}\mathbb{N}.\ \lambda a{:}A.\ \lambda v{:}\mathsf{Vect}\ A\ k.\ \textbf{consCase}\ A\ n\ k\ v) \\
\textbf{vTail} \mapsto & \lambda A{:}\star.\ \lambda n{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ (\mathsf{s}\ n). \\
 & (\lambda k{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ k. \\
 & \lambda P{:}\forall k{:}\mathbb{N}.\ \forall v{:}\mathsf{Vect}\ A\ k.\ (\mathsf{s}\ n\ =\ k) \to \mathsf{Vect}\ A\ n. \\
 & P\ (\mathsf{s}\ n)\ v\ (\mathsf{refl}\ (\mathsf{s}\ n))) \\
 & n\ v\ (\textbf{vTailAux}\ n\ A)
\end{array}
$$

Figure A.1: Elaborated **vTail**

A.3.1 dMotive and discriminate

$$
\begin{array}{ll}
\textbf{dMotive} \mapsto & \lambda n{:}\mathbb{N}.\ \mathbb{N}\textbf{-Case}\ n\ (\forall n{:}\mathbb{N}.\ \star)\ \mathsf{False}\ (\lambda k{:}\mathbb{N}.\ \mathsf{True}) \\
\textbf{discriminate} \mapsto & \lambda n{:}\mathbb{N}.\ \lambda p{:}\mathsf{s}\ n\ =\ 0. \\
 & \quad =\textbf{-elim}\ \mathbb{N}\ (\mathsf{s}\ n)\ p\ \textbf{dMotive}\ ()
\end{array}
$$

dMotive is the motive for the elimination applied by **discriminate**, so we will take these two functions together.

dMotive initially compiles to the following set of supercombinators:

$$\textbf{dMotive} \mapsto \lambda n.\, \mathbb{N}\textbf{-Case}\ n\ \textbf{dMotive1}\ \mathsf{False}\ \textbf{dMotive2}$$
$$\textbf{dMotive1} \mapsto \lambda n.\, \star$$
$$\textbf{dMotive2} \mapsto \lambda k.\, \mathsf{True}$$

Inlining of $\mathbb{N}$-**Case** results in the following single supercombinator:

$$\begin{aligned}\textbf{dMotive} \mapsto\ & \lambda n.\, \underline{\text{case}}\ n\ \underline{\text{of}} \\ & \quad \mathsf{0}\langle\rangle \rightsquigarrow \mathsf{False} \\ & \quad \mathsf{s}\langle k\rangle \rightsquigarrow \mathsf{True}\end{aligned}$$

discriminate is straightforward, given **dMotive**:

$$\textbf{discriminate} \mapsto \lambda n; p.\ =\textbf{-elim}\ \mathbb{N}\ \mathsf{s}\langle n\rangle\ p\ \textbf{dMotive}\ ()$$

However, we observe that **discriminate** returns an element of the empty type; this is clearly impossible. The function therefore collapses as follows:

$$\textbf{discriminate} \mapsto \lambda n; p.\, \underline{\text{Impossible}}$$

A further transformation is applied to remove the two arguments which are unused in the body of **discriminate**. The supercombinators we generate are summarised in figure A.2, and the substitution rules in figure A.3.

$$\begin{aligned}\textbf{dMotive} \mapsto\ & \lambda n.\, \underline{\text{case}}\ n\ \underline{\text{of}} \\ & \quad \mathsf{0}\langle\rangle \rightsquigarrow \mathsf{False} \\ & \quad \mathsf{s}\langle k\rangle \rightsquigarrow \mathsf{True} \\ \textbf{discriminate}' \mapsto\ & \underline{\text{Impossible}} \\ \textbf{discriminate} \mapsto\ & \lambda n; p.\, \textbf{discriminate}'\end{aligned}$$

Figure A.2: Compilation of **discriminate** and **dMotive**

$$\begin{aligned}[\![\textbf{discriminate}\ n\ p]\!] &\Longrightarrow \textbf{discriminate}' \\ [\![\textbf{discriminate}']\!] &\Longrightarrow \underline{\text{Impossible}}\end{aligned}$$

Figure A.3: Substitution rules for **discriminate** and **dMotive**

A.3.2 emptyCase

emptyCase is defined in ExTT as follows:

$$\begin{aligned}\textbf{emptyCase}\ :\ & \forall A{:}\star.\, \forall n{:}\mathbb{N}.\, (\mathsf{s}\ n\ =\ \mathsf{0}) \rightarrow \mathsf{Vect}\ A\ n \\ \textbf{emptyCase} \mapsto\ & \lambda A{:}\star.\, \lambda n{:}\mathbb{N}.\, \lambda p{:}\mathsf{s}\ n\ =\ \mathsf{0}. \\ & \quad \mathsf{False}\textbf{-Elim}\ (\textbf{discriminate}\ n\ p)\ (\mathsf{Vect}\ A\ n)\end{aligned}$$

Building a supercombinator definition for **emptyCase** initially gives:

$$\textbf{emptyCase} \mapsto \lambda A; n; p.\ \text{False-}\textbf{Elim}\ (\textbf{discriminate}\ n\ p)\ (\text{Vect}\ A\ n)$$

False-**Elim** expects an argument in the empty type; therefore it can never be evaluated, so this function can never be evaluated. We therefore collapse it to the supercombinator definitions in figure A.4, with substitutions as in figure A.5.

$$\begin{array}{|l} \textbf{emptyCase}' \mapsto \underline{\text{Impossible}} \\ \textbf{emptyCase} \mapsto \lambda A; n; p.\ \textbf{emptyCase}' \end{array}$$

Figure A.4: Compilation of **emptyCase**

$$\begin{array}{|l} [\![\textbf{emptyCase}\ A\ n\ p]\!] \Longrightarrow \textbf{emptyCase}' \\ [\![\textbf{emptyCase}']\!] \Longrightarrow \underline{\text{Impossible}} \end{array}$$

Figure A.5: Substitution rules for **emptyCase**

A.3.3 consCase

consCase is defined in ExTT as follows:

$$\begin{aligned} \textbf{consCase} \mapsto\ & \lambda A{:}\star.\ \lambda n{:}\mathbb{N}.\ \lambda k{:}\mathbb{N}.\ \lambda v{:}\text{Vect}\ A\ k.\ \lambda p{:}k = n. \\ & = \textbf{-elim}\ \mathbb{N}\ k\ n\ (\textbf{S\_inj}\ k\ n\ (\textbf{eq\_sym}\ \mathbb{N}\ n\ k\ p))\ (\lambda n{:}\mathbb{N}.\ \text{Vect}\ A\ n)\ v \end{aligned}$$

Building a supercombinator definition for **consCase** initially gives:

$$\begin{aligned} \textbf{consCase} \mapsto\ & \lambda A; n; k; v; p. \\ & = \textbf{-elim}\ \mathbb{N}\ k\ n\ (\textbf{S\_inj}\ k\ n\ (\textbf{eq\_sym}\ \mathbb{N}\ n\ k\ p))\ (\textbf{consCase1}\ A)\ v \\ \textbf{consCase1} \mapsto\ & \lambda A; n.\ \text{Vect}\ A\ n \end{aligned}$$

After elimination unfolding and inlining, we get:

$$\begin{aligned} \textbf{consCase} \mapsto\ & \lambda A; n; k; v; p.\ \underline{\text{case}}\ p\ \underline{\text{of}} \\ & \quad \text{refl}\langle n' \rangle \rightsquigarrow v \end{aligned}$$

Since case expressions with only one branch can be trivially reduced to that branch (since the scrutinee will always be in canonical form), this definition reduces to that in figure A.6, with the obvious inlining and argument removal substitutions in figure A.7

$$\boxed{\textbf{consCase} \mapsto \lambda A; n; k; v; p.\ v}$$

Figure A.6: Compilation of **consCase**

$$\begin{array}{l}[\![\textbf{consCase}\ A\ n\ k\ v\ p]\!] \Longrightarrow \textbf{consCase}'\ v \\ [\![\textbf{consCase}'\ v]\!] \Longrightarrow\ v\end{array}$$

Figure A.7: Substitution rules for **consCase**

A.3.4 vTailAux

vTailAux is where most of the work is done; it is defined in ExTT as follows:

$$\begin{array}{lll}\textbf{vTailAux} & : & \forall n{:}\mathbb{N}.\,\forall A{:}\star.\,\forall k{:}\mathbb{N}.\,\forall v{:}\mathsf{Vect}\ A\ k.\ (\mathsf{s}\ n\ =\ k) \rightarrow \mathsf{Vect}\ A\ n \\ \textbf{vTailAux} & \mapsto & \lambda n{:}\mathbb{N}.\,\lambda A{:}\star.\,\lambda k{:}\mathbb{N}.\,\lambda v{:}\mathsf{Vect}\ A\ k. \\ & & \mathsf{Vect}\textbf{-Case}\ A\ k\ v \\ & & (\lambda k{:}\mathbb{N}.\,\lambda v{:}\mathsf{Vect}\ A\ k.\ (\mathsf{s}\ n\ =k) \rightarrow \mathsf{Vect}\ A\ n) \\ & & (\textbf{emptyCase}\ A\ n) \\ & & (\lambda k{:}\mathbb{N}.\,\lambda a{:}A.\,\lambda v{:}\mathsf{Vect}\ A\ k.\ \textbf{consCase}\ A\ n\ k\ v)\end{array}$$

Building supercombinator definitions for **vTailAux** initially gives:

$$\begin{array}{lll}\textbf{vTailAux} & \mapsto & \lambda n; A; k; v.\ \mathsf{Vect}\textbf{-Case}\ A\ k\ v\ (\textbf{vTailAux1}\ A\ n) \\ & & (\textbf{emptyCase}\ A\ n)\ (\textbf{vTailAux2}\ A\ \ n) \\ \textbf{vTailAux1} & \mapsto & \lambda A; n; k; v; p.\ \mathsf{Vect}\ A\ n \\ \textbf{vTailAux2} & \mapsto & \lambda A; n; k; a; v.\ \textbf{consCase}\ A\ n\ k\ v\end{array}$$

After inlining of Vect-**Case** we get the following:

$$\begin{array}{lll}\textbf{vTailAux} & \mapsto & \lambda n; A; k; v.\ \underline{\text{case}}\ v\ \underline{\text{of}} \\ & & \quad \epsilon\langle\rangle\ \rightsquigarrow\ \textbf{emptyCase}\ A\ n \\ & & \quad ::\langle x, xs\rangle\ \rightsquigarrow\ \textbf{vTailAux2}\ A\ n\ (n!0)\ (v!0)\ (v!1)\end{array}$$

vTailAux2 is now inlinable as it is fully applied, and a small definition. There is now no more which can be done to transform this function; it returns a function which expects an equality proof, which is to be passed through to **emptyCase** and **consCase**. We do consider **vTailAux** a good candidate for inlining, however. The definition and substitutions are given in figures A.8 and A.9.

$$\begin{array}{lll}\textbf{vTailAux} & \mapsto & \lambda n; A; k; v.\ \underline{\text{case}}\ v\ \underline{\text{of}} \\ & & \quad \epsilon\langle\rangle\ \rightsquigarrow\ \textbf{emptyCase}\ A\ n \\ & & \quad ::\langle x, xs\rangle\ \rightsquigarrow\ \textbf{consCase}\ A\ n\ (n!0)(v!1)\end{array}$$

Figure A.8: Compilation of **vTailAux**

$$[\![\mathbf{vTailAux}\ n\ A\ k\ v]\!] \Longrightarrow \underline{\text{case}}\ v\ \underline{\text{of}}$$
$$\epsilon\langle\rangle \rightsquigarrow \mathbf{emptyCase}\ A\ n$$
$$::\langle x, xs\rangle \rightsquigarrow \mathbf{consCase}\ A\ n\ (n!0)\ (v!1)$$

Figure A.9: Substitution rules for **vTailAux**

A.3.5 vTail

vTail, the top level function, is defined in ExTT as follows:

$$\begin{aligned}\mathbf{vTail} \mapsto\ & \lambda A{:}\star.\ \lambda n{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ (\mathsf{s}\ n).\\ & (\lambda k{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ k.\\ & \lambda P{:}\forall n{:}\mathbb{N}.\ \forall v{:}\mathsf{Vect}\ A\ k.\ (\mathsf{s}\ n\ =\ k) \rightarrow \mathsf{Vect}\ A\ n.\\ & P\ (\mathsf{s}\ n)\ v\ (\mathsf{refl}\ (\mathsf{s}\ n)))\\ & n\ v\ (\mathbf{vTailAux}\ n\ A)\end{aligned}$$

Before we start compiling to supercombinators, we notice that this function β-reduces to the following:

$$\begin{aligned}\mathbf{vTail} \mapsto\ & \lambda A{:}\star.\ \lambda n{:}\mathbb{N}.\ \lambda v{:}\mathsf{Vect}\ A\ (\mathsf{s}\ n).\\ & (\mathbf{vTailAux}\ n\ A\ (\mathsf{s}\ n)\ v\ (\mathsf{refl}\ (\mathsf{s}\ n)))\end{aligned}$$

Building supercombinator definitions for **vTail** from this simplified definition gives:

$$\mathbf{vTail} \mapsto \lambda A; n; v.\ \mathbf{vTailAux}\ n\ A\ \mathsf{s}\langle n\rangle\ v\ \mathsf{refl}\langle\mathsf{s}\langle n\rangle\rangle$$

The substitution rules built from **vTailAux** tell us that **vTailAux** is inlinable. Applying this, we get:

$$\begin{aligned}\mathbf{vTail} \mapsto \lambda A; n; v.\ & (\underline{\text{case}}\ v\ \underline{\text{of}}\\ & \quad \epsilon\langle\rangle \rightsquigarrow \mathbf{emptyCase}\ A\ n\\ & \quad ::\langle x, xs\rangle \rightsquigarrow \mathbf{consCase}\ A\ n\ (n!0)\ (v!1))\\ & \mathsf{refl}\langle\mathsf{s}\langle n\rangle\rangle\end{aligned}$$

That is, the result of the case is applied to the equality proof. The proof can be lifted into each branch of the case expression — this is to make each branch as fully applied as possible. We get:

$$\begin{aligned}\mathbf{vTail} \mapsto \lambda A; n; v.\ & \underline{\text{case}}\ v\ \underline{\text{of}}\\ & \quad \epsilon\langle\rangle \rightsquigarrow \mathbf{emptyCase}\ A\ n\ \mathsf{refl}\langle\mathsf{s}\langle n\rangle\rangle\\ & \quad ::\langle x, xs\rangle \rightsquigarrow \mathbf{consCase}\ A\ n\ (n!0)\ (v!1)\ \mathsf{refl}\langle\mathsf{s}\langle n\rangle\rangle\end{aligned}$$

Now we have inlining available on each branch. Applying the inlining substitution for **emptycase** (section A.3.2) and for **consCase** (section A.3.3) gives:

$$\begin{aligned}\mathbf{vTail} \mapsto \lambda A; n; v.\ & \underline{\text{case}}\ v\ \underline{\text{of}}\\ & \quad \epsilon\langle\rangle \rightsquigarrow \underline{\text{Impossible}}\\ & \quad ::\langle x, xs\rangle \rightsquigarrow (v!1)\end{aligned}$$

This is what we expected all along! The ϵ case branch has been explicitly marked as impossible to reach. We can go even further, and remove the Impossible branch, which results in a case expression with only one possibility.

$$\textbf{vTail} \mapsto \lambda A; n; v.\ \ \underline{\text{case}}\ v\ \underline{\text{of}}$$
$$::\langle x, xs\rangle \ \rightsquigarrow\ (v!1)$$

Having only one possibility, there is no need to test what v is — we already know! As it is a total function, there is no possibility of an error case and the type specifies which is the only case that can apply. Figure A.10 gives the final supercombinator for **vTail** — effectively, all it does is move a pointer to the next cell, just as we would have hoped. There are, incidentally, also two unused arguments which can be dropped in a fully applied call to **vTail**.

$$\textbf{vTail}' \mapsto \lambda v.\ (v!1)$$
$$\textbf{vTail} \mapsto \lambda A; n; v.\ \textbf{vTail}'\ v$$

Figure A.10: Compilation of **vTail**

A.4 G-code

Given this definition of **vTail**, the resulting G-code is extremely simple. **vTail′** is inlined, making the RunTT definition of **vTail** the following:

$$\textbf{vTail} \mapsto \lambda A; n; v.\ (v!1)$$

Compilation to G-code of **vTail** and **vTail′** is given in figure A.11. We see that execution of this function consists of evaluating the argument to canonical form, projecting out the first argument and then evaluating that argument to canonical form. In practice, the inlining of **vTail′** and analysis of the G-code sequences produced will often mean that many of the evaluations are not necessary, since the variable is already in canonical form.

vTail :
$\mathcal{S}[\![\lambda v.\ (v!1)]\!] \Longrightarrow$ PUSH 0; EVAL; PROJ 1; EVAL; UPDATE 2; RET 1
vTail′ :
$\mathcal{S}[\![\lambda A; n; v.\ (v!1)]\!] \Longrightarrow$ PUSH 0; EVAL; PROJ 1; EVAL; UPDATE 4; RET 3

Figure A.11: G-code for **vTail** and **vTail′**

Appendix B

Typechecking ExTT

In this appendix, I give proofs that typechecking for ExTT terms built from TT by the forcing and detagging optimisations is equivalent to typechecking the original TT terms. In the presentation that follows, we will distinguish between TT and ExTT judgments by annotating the turnstile. Where there is no ambiguity, I will omit the annotation.

B.1 Typechecking Algorithms

It is standard [Hue89, Coq96] to implement checking the judgment $\Gamma \overset{\mathsf{TT}}{\vdash} a : A$ by checking that A is a type, inferring a type B for the term a and testing by conversion whether it matches the proposed type A. i.e. we check the following:

- $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\Longrightarrow} X \twoheadrightarrow \star_n$
- $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} B$
- $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} B$

Since we assume Church Rosser holds for TT, conversion can be implemented as follows:

$\Gamma \overset{\mathsf{TT}}{\vdash} a \simeq b$ if $\Gamma \overset{\mathsf{TT}}{\vdash} a \rhd c$ and $\Gamma \overset{\mathsf{TT}}{\vdash} b \rhd d$ and $\Gamma \overset{\mathsf{TT}}{\vdash} c \equiv d$

In practice, we take c and d as the normal forms of a and b respectively.

Figure B.1 gives a type synthesis algorithm for TT. It is standard that the TT inference algorithm is sound and complete for TT.

We seek to show that the corresponding methods of inference for ExTT may be used to solve the checking problem in TT, as follows:

- $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\Longrightarrow} X \overset{\mathsf{Ex}}{\twoheadrightarrow} \star_n$
- $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$

$$\frac{\Gamma \vdash \underline{\text{valid}}}{\Gamma \vdash \star_n \Longrightarrow \star_{n+1}}$$

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad x \,:\, S \in \Gamma}{\Gamma \vdash x \Longrightarrow S}$$

(Similarly for c, D, D-**Elim**)

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad x \,:\, S \mapsto s \in \Gamma}{\Gamma \vdash x \Longrightarrow S}$$

$$\frac{\Gamma \vdash f \Longrightarrow X \twoheadrightarrow \forall x{:}S.\, T \quad \Gamma \vdash s \Longrightarrow S' \quad \Gamma \vdash S \simeq S'}{\Gamma \vdash f\ s \Longrightarrow \underline{\text{let}}\ x \,:\, S' \mapsto s\ \underline{\text{in}}\ T}$$

$$\frac{\Gamma; x \,:\, S \vdash e \Longrightarrow T \quad \Gamma \vdash \forall x{:}S.\, T \Longrightarrow X \twoheadrightarrow \star_n}{\Gamma \vdash \lambda x{:}S.\, e \Longrightarrow \forall x{:}S.\, T}$$

$$\frac{\Gamma; x \,:\, S \vdash T \Longrightarrow X \twoheadrightarrow \star_n \quad \Gamma \vdash S \Longrightarrow X' \twoheadrightarrow \star_n}{\Gamma \vdash \forall x{:}S.\, T \Longrightarrow \star_n}$$

$$\frac{\begin{array}{c}\Gamma \vdash S \Longrightarrow X \twoheadrightarrow \star_n \quad \Gamma \vdash e_1 \Longrightarrow S' \quad \Gamma \vdash S \simeq S' \\ \Gamma; x \,:\, S \mapsto e_1 \vdash e_2 \Longrightarrow T \quad \Gamma; x \,:\, S \mapsto e_1 \vdash T \Longrightarrow X' \twoheadrightarrow \star_n\end{array}}{\Gamma \vdash \underline{\text{let}}\ x \,:\, S \mapsto e_1\ \underline{\text{in}}\ e_2 \Longrightarrow \underline{\text{let}}\ x \,:\, S \mapsto e_1\ \underline{\text{in}}\ T}$$

Figure B.1: Type synthesis for TT

- $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\simeq} B$

Figure B.2 gives a type synthesis algorithm for ExTT.

An optimisation $[\![\cdot]\!]$ from TT to ExTT is admissible at compile-time if it satisfies the following three properties:

Property 1. *If* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$ *then* $\exists A.\, \Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ *and* $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$

Property 2. *If* $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ *then* $\exists B.$
$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$ *and*
$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![A]\!]$ *and*
$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$

Property 3. *If* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\simeq} B$ *then* $\Gamma \overset{\mathsf{TT}}{\vdash} A \simeq |B|$

These properties state that if an optimised term is well-typed in ExTT, then the original term must also be well-typed in TT such that its TT type converts with the unmarked ExTT type. Therefore if these properties hold, we never have to typecheck TT terms and can rely on typechecking the marked terms.

Assuming that these properties hold for an optimisation, we can show the soundness and completeness of the ExTT typechecking algorithm by the following theorems (Note that we use $\Gamma \vdash x \Longrightarrow A \simeq A'$ as a shorthand for $\Gamma \vdash x \Longrightarrow A, \Gamma \vdash A \simeq A'$).

$$\frac{\Gamma \vdash \underline{\text{valid}}}{\Gamma \vdash \star_n \Longrightarrow \star_{n+1}}$$

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad x \,:\, S \in \Gamma}{\Gamma \vdash x \Longrightarrow S}$$

(Similarly for c, D, D-**Elim**)

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad x \,:\, S \mapsto s \in \Gamma}{\Gamma \vdash x \Longrightarrow S}$$

$$\frac{\Gamma \vdash f \Longrightarrow X \twoheadrightarrow \forall x{:}S.\, T \quad \Gamma \vdash s \Longrightarrow S' \quad \Gamma \vdash S \simeq S'}{\Gamma \vdash f\ s \Longrightarrow \underline{\text{let}}\ x \,:\, S' \mapsto s\ \underline{\text{in}}\ T}$$

$$\frac{\Gamma \vdash f \Longrightarrow X \twoheadrightarrow \forall \{x{:}S\}.\, T \quad \Gamma \vdash s \Longrightarrow S' \quad \Gamma \vdash S \simeq S'}{\Gamma \vdash f\ \{s\} \Longrightarrow \underline{\text{let}}\ x \,:\, S' \mapsto s\ \underline{\text{in}}\ T}$$

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad \{f\} \,:\, \forall x{:}S.\, T \in \Gamma \quad \Gamma \vdash s \Longrightarrow S' \quad \Gamma \vdash S \simeq S'}{\Gamma \vdash \{f\}\ s \Longrightarrow \underline{\text{let}}\ x \,:\, S' \mapsto s\ \underline{\text{in}}\ T}$$

$$\frac{\Gamma \vdash \underline{\text{valid}} \quad \{f\} \,:\, \forall \{x{:}S\}.\, T \in \Gamma \quad \Gamma \vdash s \Longrightarrow S' \quad \Gamma \vdash S \simeq S'}{\Gamma \vdash \{f\}\ \{s\} \Longrightarrow \underline{\text{let}}\ x \,:\, S' \mapsto s\ \underline{\text{in}}\ T}$$

$$\frac{\Gamma; x \,:\, S \vdash e \Longrightarrow T \quad \Gamma \vdash \forall x{:}S.\, T \Longrightarrow \star_n}{\Gamma \vdash \lambda x{:}S.\, e \Longrightarrow \forall x{:}S.\, T}$$

$$\frac{\Gamma; x \,:\, S \vdash T \Longrightarrow X \twoheadrightarrow \star_n \quad \Gamma \vdash S \Longrightarrow X' \twoheadrightarrow \star_n}{\Gamma \vdash \forall x{:}S.\, T \Longrightarrow \star_n}$$

$$\frac{\begin{array}{c}\Gamma \vdash S \Longrightarrow X \twoheadrightarrow \star_n \quad \Gamma \vdash e_1 \Longrightarrow S' \quad \Gamma \vdash S \simeq S' \\ \Gamma; x \,:\, S \mapsto e_1 \vdash e_2 \Longrightarrow T \quad \Gamma; x \,:\, S \mapsto e_1 \vdash T \Longrightarrow X' \twoheadrightarrow \star_n\end{array}}{\Gamma \vdash \underline{\text{let}}\ x \,:\, S \mapsto e_1\ \underline{\text{in}}\ e_2 \Longrightarrow \underline{\text{let}}\ x \,:\, S \mapsto e_1\ \underline{\text{in}}\ T}$$

Figure B.2: Type synthesis for ExTT

Theorem B.1 (Soundness of ExTT for typechecking TT). *If* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$ *and* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$ *and* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\simeq} B$ *then* $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A' \overset{\mathsf{TT}}{\simeq} A$ *and* $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\Longrightarrow} X' \twoheadrightarrow \star_n$.

Proof. $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$ shows that $\exists A'.\, \Gamma \vdash a \overset{\mathsf{TT}}{\Longrightarrow} A'$ and $\Gamma \vdash A' \overset{\mathsf{TT}}{\simeq} |B|$, by Property 1.

Also, by Property 3, $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\simeq} B$ shows that $\Gamma \vdash A \overset{\mathsf{TT}}{\simeq} |B|$.

Hence, $\Gamma \vdash A \overset{\mathsf{TT}}{\simeq} A'$, so $\Gamma \vdash a \overset{\mathsf{TT}}{\Longrightarrow} A' \overset{\mathsf{TT}}{\simeq} A$.

$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$ shows that $\exists X'.\, \Gamma \vdash A \overset{\mathsf{TT}}{\Longrightarrow} X'$ and $\Gamma \vdash X' \overset{\mathsf{TT}}{\simeq} |X|$, by Property 1, and since $|\star_n| = \star_n$ then $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\Longrightarrow} X' \twoheadrightarrow \star_n$.

□

Theorem B.2 (Completeness of ExTT for typechecking TT). *If* $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ *then* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$ *and* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$ *and* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\simeq} B$.

Proof. By Property 2. □

We show in this appendix that the forcing and detagging optimisations of Chapter 4 satisfy Properties 1 to 3, and hence that typechecking an ExTT term produced by these

optimisations gives a sound and complete typechecking algorithm for TT.

B.2 The Forcing Optimisation

Forcing is given in full in figure B.3. The translations are on *well-scoped* terms, i.e. all variables are declared or defined in the context:

$$
\begin{array}{l}
[\![\star_n]\!] \Longrightarrow \star_n \\
[\![x]\!] \Longrightarrow x \\
[\![\mathsf{D}]\!] \Longrightarrow \mathsf{D} \\
[\![\textbf{D-Elim}]\!] \Longrightarrow \textbf{D-Elim} \\
[\![f\ s]\!] \Longrightarrow [\![f]\!]\ [\![s]\!] \\
[\![\forall x : S.\ T]\!] \Longrightarrow \forall x : [\![S]\!].\ [\![T]\!] \\
[\![\lambda x : S.\ e]\!] \Longrightarrow \lambda x : [\![S]\!].\ [\![e]\!] \\
[\![\underline{\text{let}}\ x\ :\ S \mapsto v\ \underline{\text{in}}\ e]\!] \Longrightarrow \underline{\text{let}}\ q\ :\ [\![S]\!] \mapsto [\![v]\!]\ \underline{\text{in}}\ [\![e]\!] \\
[\![\mathsf{c}]\!] \Longrightarrow \lambda \vec{a} : [\![\vec{A}]\!].\ \lambda \vec{y} : \mathsf{D}\ [\![\vec{i}]\!].\ \mathsf{c}\ \vec{a}^{\{V\}}\ \vec{y} \\
\quad \underline{\text{where}}\ V \text{ is the set of concretely forceable variables in } \vec{a} \\
\qquad a^{\{V\}} \Longrightarrow \{a\}\ \underline{\text{if}}\ a \in V \\
\qquad a^{\{V\}} \Longrightarrow a\ \underline{\text{otherwise}}
\end{array}
$$

Figure B.3: The forcing optimisation

Correspondingly, the types of c and D-**Elim** are modified in the forced context so that marked arguments are expected in forced argument position. Forcing of a context is given in figure B.4:

$$
\begin{array}{l}
[\![\mathcal{E}]\!] \Longrightarrow \mathcal{E} \\
[\![\Gamma; x\ :\ S]\!] \Longrightarrow [\![\Gamma]\!]; x\ :\ [\![S]\!] \\
[\![\Gamma; \mathsf{c}\ :\ \forall \vec{a} : \vec{A}.\ \forall \vec{y} : \vec{Y}.\ \mathsf{D}\ \vec{s}]\!] \Longrightarrow [\![\Gamma]\!]; \mathsf{c}\ :\ \forall \vec{a} : \vec{A}^{\{V\}}.\ \forall \vec{y} : \vec{Y}.\ \mathsf{D}\ [\![\vec{s}]\!] \\
\quad \underline{\text{where}}\ V \text{ is the set of concretely forceable variables in } \vec{a} \\
\qquad \forall a : A^{\{V\}} \Longrightarrow \forall \{a : A\}\ \underline{\text{if}}\ a \in V \\
\qquad \forall a : A^{\{V\}} \Longrightarrow \forall a : A\ \underline{\text{otherwise}} \\
[\![\Gamma; e\ :\ S \mapsto s]\!] \Longrightarrow [\![\Gamma]\!]; x\ :\ [\![S]\!] \mapsto [\![s]\!]
\end{array}
$$

Figure B.4: Forcing a context

B.2.1 Equivalence of Reduction

The following theorem shows that a reduction step in TT either maps to a reduction step in ExTT, or does nothing (since the reduction takes place inside a marked term).

Lemma B.3. *If* $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\triangleright}_1 b$ *then either* $\Gamma \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\equiv} [\![b]\!]$ *or* $\exists c.\ [\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\triangleright} c$ *and* $c \overset{\mathsf{Ex}}{\equiv} [\![b]\!]$.

Proof Sketch. By the structure of a, and the definition of the forcing optimisation. The contractions available in all cases are the same for $[\![a]\!]$ and a, except in the case of constructor application, where some arguments may be marked. In this case, either there is no reduction in ExTT (because it takes place inside a marked argument) or (after β-reductions of arguments) there is an equivalent reduction in ExTT inside another argument. Since the reduction rules for ExTT correspond to those for TT, this reduction must be equivalent to the TT reduction. □

Due to this property, if a term a has a normal form b in TT, then $[\![a]\!]$ has a normal form c in ExTT such that $[\![b]\!] \overset{\mathsf{Ex}}{\equiv} c$.

Corollary B.4. *If* $\Gamma \overset{\mathsf{TT}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} T$ *then* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![S]\!] \overset{\mathsf{Ex}}{\simeq} [\![T]\!]$.

Proof. Trivial, by Lemma B.3. □

B.2.2 Equivalence of Typechecking for Forcing

Lemma B.5. $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\simeq} |[\![a]\!]|$

Proof. Trivial, since we have η-conversion. The proof is by induction on the typing judgment.

- Case $a = \mathsf{c}$. Then $[\![\mathsf{c}]\!] \Longrightarrow \lambda \vec{a} : [\![\vec{A}]\!].\ \lambda \vec{y} : \mathsf{D}\ [\![\vec{i}]\!].\ \mathsf{c}\ \vec{a}^{\{\mathcal{V}\}}\ \vec{y}$.
  Removing the marks yields $\lambda \vec{a} : \left|[\![\vec{A}]\!]\right| .\ \lambda \vec{y} : \mathsf{D}\ \left|[\![\vec{i}]\!]\right| .\ \mathsf{c}\ \vec{a}\ \vec{y}$, which is η-convertible with c.
- All other cases are provable trivially by induction.

□

Lemma B.6. *If* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} T$ *then* $\Gamma \overset{\mathsf{TT}}{\vdash} |S| \overset{\mathsf{TT}}{\simeq} |T|$.

Proof. By induction on the typing judgement for normal forms of S and T. Take $a = \mathrm{NF}(S)$ and $b = \mathrm{NF}(T)$ (where NF gives a normal form in ExTT).

The possible normal forms are:

- $\star_n$
- x
- $\lambda x : S.\ e$, where S, e are normal forms.
- $\forall x : S.\ T$, where S, T are normal forms.
- $\mathsf{c}\ \vec{a}\ \vec{y}$, where $\vec{a}, \vec{y}$ are normal forms or of the form $\{t\}$, and t is any term.

If the outermost constructors differ, conversion does not hold, so we consider the cases where a and b are of the same form. In each case, we can assume that $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} a \overset{\mathsf{Ex}}{\simeq} b$.

- Case $a = \star_i$ and $b = \star_j$. Then $|a| = \star_i$ and $|b| = \star_j$, so conversion holds in both systems if $i = j$.
- Case $a = x$ and $b = x'$. Then $|a| = x$ and $|b| = x'$. If $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} x \overset{\mathsf{Ex}}{\equiv} x'$ then $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} x \overset{\mathsf{TT}}{\equiv} x'$ so conversion holds.
- Case $a = \lambda x{:}S.\ e$ and $b = \lambda x{:}S'.\ e'$. Then $|a| = \lambda x{:}|S|.\ |e|$ and $|b| = \lambda x{:}|S'|.\ |e'|$. By induction, if $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} S'$ then $\Gamma \overset{\mathsf{TT}}{\vdash} |S| \overset{\mathsf{TT}}{\simeq} |S'|$ and if $[\![\Gamma; x : S]\!] \overset{\mathsf{Ex}}{\vdash} e \overset{\mathsf{Ex}}{\simeq} e'$ then $\Gamma; x : S \overset{\mathsf{TT}}{\vdash} |e| \overset{\mathsf{TT}}{\simeq} |e'|$, so conversion holds in TT if it holds in ExTT.
- Case $a = \forall x{:}S.\ T$ and $b = \forall x{:}S'.\ T'$. Then $|a| = \forall x{:}|S|.\ |T|$ and $|b| = \forall x{:}|S'|.\ |T'|$. By induction, if $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} S'$ then $\Gamma \overset{\mathsf{TT}}{\vdash} |S| \overset{\mathsf{TT}}{\simeq} |S'|$ and if $[\![\Gamma; x : S]\!] \overset{\mathsf{Ex}}{\vdash} T \overset{\mathsf{Ex}}{\simeq} T'$ then $\Gamma; x : S \overset{\mathsf{TT}}{\vdash} |T| \overset{\mathsf{TT}}{\simeq} |T'|$, so conversion holds in TT if it holds in ExTT.
- Case $a = \mathsf{c}\ \vec{a}\ \vec{y}$ and $b = \mathsf{c}\ \vec{b}\ \vec{z}$. Then $|a| = \mathsf{c}\ \vec{a}'\ \vec{y}'$ and $|b| = \mathsf{c}'\ \vec{b}'\ \vec{z}'$, where, as before:
  - $a_i = [\![a_i']\!]$ if a_i' is not concretely forceable.
  - $a_i = \{a_i'\}$ if a_i' is concretely forceable.
  - $b_i = [\![b_i']\!]$ if b_i' is not concretely forceable.
  - $b_i = \{b_i'\}$ if b_i' is concretely forceable.
  - $y_i = [\![y_i']\!]$.
  - $z_i = [\![z_i']\!]$.

  Synthesising types of a and b we get:

  - $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} a \overset{\mathsf{Ex}}{\Longrightarrow} \mathsf{D}\ \vec{s}$ for some $\vec{s}$.
  - $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} \mathsf{D}'\ \vec{t}$ for some $\vec{t}$.

  $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} a \overset{\mathsf{Ex}}{\simeq} b$ if and only if the constructors are identical, i.e. $\mathsf{c} \overset{\mathsf{Ex}}{\equiv} \mathsf{c}'$, and corresponding arguments are convertible. For each argument, due to the forcing optimisation, either both a_i and b_i will be marked, or neither.

  - If neither are marked, then if $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} a_i \overset{\mathsf{Ex}}{\simeq} b_i$ then $\Gamma \overset{\mathsf{TT}}{\vdash} |a_i| \overset{\mathsf{TT}}{\simeq} |b_i|$ by induction. As $\Gamma \overset{\mathsf{TT}}{\vdash} |a_i| \overset{\mathsf{TT}}{\simeq} a_i'$ and $\Gamma \overset{\mathsf{TT}}{\vdash} |b_i| \overset{\mathsf{TT}}{\simeq} b_i'$, then $\Gamma \overset{\mathsf{TT}}{\vdash} a_i' \overset{\mathsf{TT}}{\simeq} b_i'$.
  - If both are marked, then it is because they are concretely forceable arguments. Hence, by Lemma 4.2, they are also forceable. By the definition of forceable, this means the arguments are determined by their indices, which are already in the context since the terms are well-scoped. Hence the conversion check has already been made.

□

Theorem B.7 (Property 1 for forcing). *If* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$ *then* $\exists A.\, \Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ *and* $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$

Proof. By induction on the typing judgment, $\Delta \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} B$, where $\forall \Gamma, a,\ \Delta = [\![\Gamma]\!]$ and $b = [\![a]\!]$. In each case, we synthesise B and find that there is appropriate A such that $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$.

- Case $b = \star_n$. Then $B = \star_{n+1}$. We must have $a = \star_n$, so take $A = \star_{n+1}$.
- Case $b = x$. Then $B = S$ if:
  - $x : S \in \Delta$ or
  - $x : S \mapsto e \in \Delta$

  So if $\Delta = [\![\Gamma]\!]$ we must have
  - $x : S' \in \Gamma$ or
  - $x : S' \mapsto e \in \Gamma$

  where $S = [\![S']\!]$.

  So take $A = S'$, and by Lemma B.5, $S' \overset{\mathsf{TT}}{\simeq} |S|$, so $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$.
- Cases $b = \mathsf{D}$, $b = \mathsf{D}$-**Elim**, $b = \mathsf{c}$ similarly to $b = x$.
- Case $b = \forall x{:}S.\ T$. Then $a = \forall x{:}S'.\ T'$ where $S = [\![S']\!]$ and $T = [\![T']\!]$.

  If $\Delta \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$ and $\Delta; x : S \overset{\mathsf{Ex}}{\vdash} T \overset{\mathsf{Ex}}{\Longrightarrow} Y \twoheadrightarrow \star_n$ then $B = \star_n$.

  Then by induction:
  - $\Delta \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$ gives i.h.
    $\forall \Gamma, a.\ \Delta = [\![\Gamma]\!], S = [\![a]\!], \exists A.\ \Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ and $\Gamma \overset{\mathsf{TT}}{\vdash} A \simeq |X|$
  - $\Delta; x : S \overset{\mathsf{Ex}}{\vdash} T \overset{\mathsf{Ex}}{\Longrightarrow} Y \twoheadrightarrow \star_n$ gives i.h.
    $\forall \Gamma, b.\ \Delta; x : S = [\![\Gamma]\!], T = [\![b]\!]$,
    $\exists B.\ \Gamma \overset{\mathsf{TT}}{\vdash} b \overset{\mathsf{TT}}{\Longrightarrow} B$ and $\Gamma \overset{\mathsf{TT}}{\vdash} B \overset{\mathsf{TT}}{\simeq} |Y|$

  So $\Gamma \overset{\mathsf{TT}}{\vdash} S' \overset{\mathsf{TT}}{\Longrightarrow} A \simeq \star_n$ and $\Gamma; x : |S|' \overset{\mathsf{TT}}{\vdash} T' \overset{\mathsf{TT}}{\Longrightarrow} B \overset{\mathsf{TT}}{\simeq} \star_n$. So $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$.
- Case $b = \underline{\mathsf{let}}\ x : S \mapsto v\ \underline{\mathsf{in}}\ e$. Then $a = \underline{\mathsf{let}}\ x : S' \mapsto v'\ \underline{\mathsf{in}}\ e'$ where $S = [\![S']\!]$, $v = [\![v']\!]$ and $e = [\![e']\!]$.

  If $\Delta \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$, $\Delta \overset{\mathsf{Ex}}{\vdash} v \overset{\mathsf{Ex}}{\Longrightarrow} S''$, $\Delta \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} S''$, $\Delta; x : S \mapsto v \overset{\mathsf{Ex}}{\vdash} e \overset{\mathsf{Ex}}{\Longrightarrow} T$ and $\Delta; x : S \mapsto v \overset{\mathsf{Ex}}{\vdash} T \overset{\mathsf{Ex}}{\Longrightarrow} X' \twoheadrightarrow \star_n$ then $B = \underline{\mathsf{let}}\ x : S \mapsto v\ \underline{\mathsf{in}}\ T$.

  Then by induction:

- $\Delta \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$ gives i.h.
  $\forall \Gamma, a.\, \Delta = [\![\Gamma]\!], S = [\![a]\!], \exists A.\, \Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ and $\Gamma \overset{\mathsf{TT}}{\vdash} A \simeq |X|$.
- $\Delta \overset{\mathsf{Ex}}{\vdash} v \overset{\mathsf{Ex}}{\Longrightarrow} S''$ gives i.h.
  $\forall \Gamma, b.\, \Delta = [\![\Gamma]\!], v = [\![b]\!], \exists B.\, \Gamma \overset{\mathsf{TT}}{\vdash} b \overset{\mathsf{TT}}{\Longrightarrow} B$ and $\Gamma \overset{\mathsf{TT}}{\vdash} B \simeq |S''|$.
- $\Delta; x : S \mapsto v \overset{\mathsf{Ex}}{\vdash} e \overset{\mathsf{Ex}}{\Longrightarrow} T$ gives i.h.
  $\forall \Gamma, c.\, \Delta; x : S \mapsto v = [\![\Gamma]\!], e = [\![c]\!]$,
  $\exists C.\, \Gamma \overset{\mathsf{TT}}{\vdash} c \overset{\mathsf{TT}}{\Longrightarrow} C$ and $\Gamma \overset{\mathsf{TT}}{\vdash} C \overset{\mathsf{TT}}{\simeq} |T|$.
  So $\Gamma \overset{\mathsf{TT}}{\vdash} S'' \overset{\mathsf{TT}}{\Longrightarrow} |X|$, $\Gamma \overset{\mathsf{TT}}{\vdash} v' \overset{\mathsf{TT}}{\Longrightarrow} |S''|$ and $\Gamma; x : |S| \overset{\mathsf{TT}}{\vdash} e' \overset{\mathsf{TT}}{\Longrightarrow} |T|$. Then $A = \underline{\mathsf{let}}\, x : |S''| \mapsto v' \,\underline{\mathsf{in}}\, |T|$ if $\Gamma \overset{\mathsf{TT}}{\vdash} |v'| \overset{\mathsf{TT}}{\simeq} v$ (which holds by lemma B.5) and $\Gamma \vdash |S''| \overset{\mathsf{TT}}{\simeq} |S|$ (which holds by lemma B.6).

- Case $b = f\ s$. Then $a = f'\ s'$ where $f = [\![f']\!]$ and $s = [\![s']\!]$.
  If $\Delta \overset{\mathsf{Ex}}{\vdash} f \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \forall x{:}S.\, T$ and $\Delta \overset{\mathsf{Ex}}{\vdash} s \overset{\mathsf{Ex}}{\Longrightarrow} S'$ and $\Delta \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} S'$ then
  $B = \underline{\mathsf{let}}\, x : S' \mapsto [\![s']\!] \,\underline{\mathsf{in}}\, T$.

  Then by induction:

  - $\Delta \overset{\mathsf{Ex}}{\vdash} f \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \forall x{:}S.\, T$ gives i.h.
    $\forall \Gamma, a.\, \Delta = [\![\Gamma]\!], f = [\![a]\!], \exists A.\, \Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ and $A \overset{\mathsf{TT}}{\simeq} |X|$
  - $\Delta \overset{\mathsf{Ex}}{\vdash} s \overset{\mathsf{Ex}}{\Longrightarrow} S'$ gives i,h.
    $\forall \Gamma, b.\, \Delta = [\![\Gamma]\!], s = [\![b]\!], \exists B.\, \Gamma \overset{\mathsf{TT}}{\vdash} b \overset{\mathsf{TT}}{\Longrightarrow} B$ and $B \overset{\mathsf{TT}}{\simeq} |S'|$

  So $\Gamma \overset{\mathsf{TT}}{\vdash} f' \overset{\mathsf{TT}}{\Longrightarrow} A \overset{\mathsf{TT}}{\simeq} |X| \twoheadrightarrow \forall x{:}|S|\,.\, |T|$ and $\Gamma \overset{\mathsf{TT}}{\vdash} s' \overset{\mathsf{TT}}{\Longrightarrow} B \overset{\mathsf{TT}}{\simeq} |S'|$.
  Then $A = \underline{\mathsf{let}}\, x : |S'| \mapsto s' \,\underline{\mathsf{in}}\, |T|$, if $|S| \overset{\mathsf{TT}}{\simeq} |S'|$.
  By Lemma B.6, if $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} S'$ then $\Gamma \overset{\mathsf{TT}}{\vdash} |S| \overset{\mathsf{TT}}{\simeq} |S'|$.
  So $A = \underline{\mathsf{let}}\, x : |S'| \mapsto s' \,\underline{\mathsf{in}}\, |T|$,
  $B = \underline{\mathsf{let}}\, x : S' \mapsto [\![s']\!] \,\underline{\mathsf{in}}\, T$,
  and therefore $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$, by Lemma B.5.

- Case $b = f\ \{s\}$. Then $a = f'\ x$ where $f = [\![f']\!]$, $x = s$ and $x \in \Gamma$, since the forcing optimisation only places variable names in $\{\}$.
  If $\Delta \overset{\mathsf{Ex}}{\vdash} f \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \forall x{:}S.\, T$ and $\Delta \overset{\mathsf{Ex}}{\vdash} s \overset{\mathsf{Ex}}{\Longrightarrow} S'$ and $\Delta \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} S'$ then
  $B = \underline{\mathsf{let}}\, x : S' \mapsto s \,\underline{\mathsf{in}}\, T$.

  Then by induction:

  - $\Delta \overset{\mathsf{Ex}}{\vdash} f \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \forall x{:}S.\, T$ gives i.h.
    $\forall \Gamma, a.\, \Delta = [\![\Gamma]\!], f = [\![a]\!], \exists A.\, \Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ and $A \overset{\mathsf{TT}}{\simeq} |X|$

So $\Gamma \overset{\mathsf{TT}}{\vdash} f' \overset{\mathsf{TT}}{\Longrightarrow} A \overset{\mathsf{TT}}{\simeq} |X| \twoheadrightarrow \forall x{:}|S|\,.\,|T|$.

If $x : S' \in \Delta$, then $x : |S'| \in \Gamma$.

Then $A = \underline{\mathsf{let}}\ x : |S'| \mapsto s\ \underline{\mathsf{in}}\ |T|$, and $A \overset{\mathsf{TT}}{\simeq} |B|$.

- Case $b = \{f\}\ s$ does not arise by forcing.
- Case $b = \{f\}\ \{s\}$ does not arise by forcing.
- Case $b = \lambda x{:}S.\ e$. If $\Delta; x : S \overset{\mathsf{Ex}}{\vdash} e \overset{\mathsf{Ex}}{\Longrightarrow} T$ and $\Delta \overset{\mathsf{Ex}}{\vdash} \forall x{:}S.\ T \overset{\mathsf{Ex}}{\Longrightarrow} \star_n$ then $B = \forall x{:}S.\ T$.

  Then either:

  - $a = \lambda x{:}S'.\ e'$ where $S = [\![S']\!]$ and $e = [\![e']\!]$.

    Then by induction:

    * $\Delta; x : S \overset{\mathsf{Ex}}{\vdash} e \overset{\mathsf{Ex}}{\Longrightarrow} T$ gives i.h.

      $\forall \Gamma, a.\ \Delta; x : S = [\![\Gamma]\!], e = [\![a]\!], \exists A.\ \Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ and $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |T|$

    * $\Delta \overset{\mathsf{Ex}}{\vdash} \forall x{:}S.\ T \Longrightarrow \star_n$ gives i.h.

      $\forall \Gamma, b.\ \Delta = [\![\Gamma]\!], \forall x{:}S.\ T = [\![b]\!], \exists B.\ \Gamma \overset{\mathsf{TT}}{\vdash} b \overset{\mathsf{TT}}{\Longrightarrow} B$ and $\Gamma \overset{\mathsf{TT}}{\vdash} B \overset{\mathsf{TT}}{\simeq} |\star_n|$.

    So $\Gamma; x : S' \overset{\mathsf{TT}}{\vdash} e' \overset{\mathsf{TT}}{\Longrightarrow} A \overset{\mathsf{TT}}{\simeq} |T|$ and $\Gamma \overset{\mathsf{TT}}{\vdash} \lambda x{:}S'.\ e' \overset{\mathsf{TT}}{\Longrightarrow} \forall x{:}S'.\ |T|$.

    $|S| \overset{\mathsf{TT}}{\simeq} S'$, so take $A = \forall x{:}|S|\,.\,|T|$, and $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$.

  - $a = \mathsf{c}$ if $b = \lambda \vec{a}{:}\vec{A}.\ \lambda \vec{y}{:}\vec{Y}.\ \mathsf{c}\ \vec{a}^{\{V\}}\ \vec{y}$.

    Then $\Delta \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} \forall \vec{a}{:}\vec{A}.\ \forall \vec{y}{:}\vec{Y}.\ \underline{\mathsf{let}}\ \vec{a} : \vec{A} \mapsto \vec{a}\ \underline{\mathsf{in}}\ \underline{\mathsf{let}}\ \vec{y} : \vec{Y} \mapsto \vec{y}\ \underline{\mathsf{in}}\ \mathsf{D}\ \vec{s}$
    and $\Gamma \vdash a \overset{\mathsf{TT}}{\Longrightarrow} \forall \vec{a}{:}\vec{A}'.\ \forall \vec{y}{:}\vec{Y}'.\ \mathsf{D}\ \vec{s}'$, by lookup in Γ.

    Since $\Delta = [\![\Gamma]\!]$, $\vec{A} = [\![\vec{A}']\!]$ and $\vec{Y} = [\![\vec{Y}']\!]$.

    By Lemma B.5, $\left|\vec{A}\right| = \vec{A}'$, $\left|\vec{Y}\right| = \vec{Y}'$ and $|\vec{s}| = \vec{s}'$, so

    $\Gamma \overset{\mathsf{TT}}{\vdash} \left|\forall \vec{a}{:}\vec{A}.\ \forall \vec{y}{:}\vec{Y}.\ \underline{\mathsf{let}}\ \vec{a} : \vec{A} \mapsto \vec{a}\ \underline{\mathsf{in}}\ \underline{\mathsf{let}}\ \vec{y} : \vec{Y} \mapsto \vec{y}\ \underline{\mathsf{in}}\ \mathsf{D}\ \vec{s}\right| \overset{\mathsf{TT}}{\simeq} \forall \vec{a}{:}\vec{A}'.\ \forall \vec{y}{:}\vec{Y}'.\ \mathsf{D}\ \vec{s}'$
    and take $A = \forall \vec{a}{:}\vec{A}'.\ \forall \vec{y}{:}\vec{Y}'.\ \mathsf{D}\ \vec{s}'$ and
    $B = \forall \vec{a}{:}\vec{A}.\ \forall \vec{y}{:}\vec{Y}.\ \underline{\mathsf{let}}\ \vec{a} : \vec{A} \mapsto \vec{a}\ \underline{\mathsf{in}}\ \underline{\mathsf{let}}\ \vec{y} : \vec{Y} \mapsto \vec{y}\ \underline{\mathsf{in}}\ \mathsf{D}\ \vec{s}$

□

Theorem B.8 (Property 2 for forcing). *If* $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ *then* $\exists B.$
$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$ *and*
$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![A]\!]$ *and*
$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$

Proof. By induction on the TT typing judgement, $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$. In each case, we synthesise A and find appropriate B such that $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![A]\!]$.

- Case $a = \star_n$. Then $[\![a]\!] = \star_n$. So take $A = \star_{n+1}$ and $B = \star_{n+1}$.
- Case $a = x$. Then $[\![a]\!] = x$.

  Then $A = S$ if:

  - $x : S \in \Gamma$ or
  - $x : S \mapsto e \in \Gamma$

  So if $\Delta = [\![\Gamma]\!]$ we must have

  - $x : S' \in \Delta$ or
  - $x : S' \mapsto e \in \Delta$

  where $S' = [\![S]\!]$.

  Take $A = S$ and $B = S'$, so by definition $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![A]\!]$.
- Case $a = \mathsf{D}, a = \mathsf{D}$-**Elim**. Similarly to $a = x$.
- Case $a = \mathsf{c}$. Then $[\![a]\!] = \lambda \vec{a} : \vec{A}.\, \lambda \vec{y} : \vec{Y}.\, \mathsf{c}\ \vec{a}\{V\}\ \vec{y}$.

  $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} \forall \vec{a} : \vec{A}.\, \forall \vec{y} : \vec{Y}.\, \mathsf{D}\ \vec{s}$, by lookup of c in Γ.
  If $\Delta = [\![\Gamma]\!]$ then $\Delta \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} \forall \vec{a} : [\![\vec{A}]\!].\, \forall \vec{y} : [\![\vec{Y}]\!].\, \underline{\mathsf{let}}\, \vec{a} \ :\ \vec{A} \mapsto \vec{a}\, \underline{\mathsf{in}}\, \underline{\mathsf{let}}\, \vec{y} \ :\ \vec{Y} \mapsto \vec{y}\, \underline{\mathsf{in}}\, \mathsf{D}\, [\![\vec{s}']\!]$.

  Take $A = \forall \vec{a} : \vec{A}.\, \forall \vec{y} : \vec{Y}.\, \mathsf{D}\ \vec{s}$ and
  $B = \forall \vec{a} : [\![\vec{A}]\!].\, \forall \vec{y} : [\![\vec{Y}]\!].\, \underline{\mathsf{let}}\, \vec{a} \ :\ \vec{A} \mapsto \vec{a}\, \underline{\mathsf{in}}\, \underline{\mathsf{let}}\, \vec{y} \ :\ \vec{Y} \mapsto \vec{y}\, \underline{\mathsf{in}}\, \mathsf{D}\, [\![\vec{s}']\!]$
  so by definition $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![A]\!]$.
- Case $a = f\ s$. Then $[\![a]\!] = [\![f]\!]\ [\![s]\!]$.

  If $\Gamma \overset{\mathsf{TT}}{\vdash} f \overset{\mathsf{TT}}{\Longrightarrow} X \twoheadrightarrow \forall x : S.\, T$
  and $\Gamma \overset{\mathsf{TT}}{\vdash} s \overset{\mathsf{TT}}{\Longrightarrow} S'$ and $\Gamma \overset{\mathsf{TT}}{\vdash} S \overset{\mathsf{TT}}{\simeq} S'$
  then $\Gamma \overset{\mathsf{TT}}{\vdash} f\ s \overset{\mathsf{TT}}{\Longrightarrow} \underline{\mathsf{let}}\ x \ :\ S' \mapsto s\, \underline{\mathsf{in}}\, T$.

  By induction:

  - $\Gamma \overset{\mathsf{TT}}{\vdash} f \overset{\mathsf{TT}}{\Longrightarrow} X \twoheadrightarrow \forall x : S.\, T$ gives i.h.
    $\forall \Gamma, b.\, \Delta = [\![\Gamma]\!], b = [\![f]\!], \exists B.\, \Delta \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} B$ and $\Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![\forall x : S.\, T]\!]$ and
    $\Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \Downarrow \star_n$
  - $\Gamma \overset{\mathsf{TT}}{\vdash} s \overset{\mathsf{TT}}{\Longrightarrow} S'$ gives i.h.
    $\forall \Gamma, b.\Delta = [\![\Gamma]\!], b = [\![s]\!], \exists B.\Delta \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} B \text{and} \Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![S']\!] \text{and} \Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \Downarrow \star_n$

So $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![f]\!] \overset{\mathsf{Ex}}{\Longrightarrow} X \overset{\mathsf{Ex}}{\simeq} [\![\forall x{:}S.\ T]\!]$.
$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![s]\!] \overset{\mathsf{Ex}}{\Longrightarrow} Y \overset{\mathsf{Ex}}{\simeq} [\![S']\!]$.
If $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![S]\!] \overset{\mathsf{Ex}}{\simeq} [\![S']\!]$ then $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![f]\!]\ [\![s]\!] \overset{\mathsf{Ex}}{\Longrightarrow} \underline{\mathsf{let}}\ x\ :\ [\![S']\!] \mapsto [\![s]\!]\ \underline{\mathsf{in}}\ [\![T]\!]$.
$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![S]\!] \overset{\mathsf{Ex}}{\simeq} [\![S']\!]$ holds by Corollary B.4, so take
$A = \underline{\mathsf{let}}\ x\ :\ S' \mapsto s\ \underline{\mathsf{in}}\ T$ and
$B = \underline{\mathsf{let}}\ x\ :\ [\![S']\!] \mapsto [\![s]\!]\ \underline{\mathsf{in}}\ [\![T]\!]$, hence $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![A]\!]$.

- Case $a = \forall x{:}S.\ T$. Then $[\![a]\!] = \forall x{:}[\![S]\!].\ [\![T]\!]$.
  If $\Gamma \overset{\mathsf{TT}}{\vdash} S \overset{\mathsf{TT}}{\Longrightarrow} \star_n$ and $\Gamma; x\ :\ S \overset{\mathsf{TT}}{\vdash} T \overset{\mathsf{TT}}{\Longrightarrow} \star_n$ then
  $\Gamma \overset{\mathsf{TT}}{\vdash} \forall x{:}S.\ T \overset{\mathsf{TT}}{\Longrightarrow} \star_n$.

  By induction:

  - $\Gamma \overset{\mathsf{TT}}{\vdash} S \overset{\mathsf{TT}}{\Longrightarrow} \star_n$ gives i.h.
    $\forall \Gamma, b.\Delta = [\![\Gamma]\!], b = [\![S]\!], \exists B.\Delta \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} B \text{and} \Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} \star_n \text{and} \Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$.
  - $\Gamma; x\ :\ S \overset{\mathsf{TT}}{\vdash} T \overset{\mathsf{TT}}{\Longrightarrow} \star_n$ gives i.h.
    $\forall \Gamma, b.\ \Delta = [\![\Gamma]\!]; x\ :\ [\![S]\!], b = [\![T]\!], \exists B.\ \Delta \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} B$ and $\Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} \star_n$ and
    $\Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$.

  So $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} \forall x{:}[\![S]\!].\ [\![T]\!] \overset{\mathsf{Ex}}{\Longrightarrow} \star_n$.

  So we take $A = \star_n$ and $B = \star_n$.

- Case $a = \underline{\mathsf{let}}\ x\ :\ S \mapsto v\ \underline{\mathsf{in}}\ e$. Then $[\![a]\!] = \underline{\mathsf{let}}\ x\ :\ [\![S]\!] \mapsto [\![v]\!]\ \underline{\mathsf{in}}\ [\![e]\!]$.
  If $\Gamma \overset{\mathsf{TT}}{\vdash} S \overset{\mathsf{TT}}{\Longrightarrow} X \twoheadrightarrow \star_n$ and $\Gamma \overset{\mathsf{TT}}{\vdash} v \overset{\mathsf{TT}}{\Longrightarrow} S'$ and $\Gamma \overset{\mathsf{TT}}{\vdash} S \overset{\mathsf{TT}}{\simeq} S'$ and
  $\Gamma; x\ :\ S \mapsto v \overset{\mathsf{TT}}{\vdash} e \overset{\mathsf{TT}}{\Longrightarrow} T$ and $\Gamma; x\ :\ S \mapsto v \overset{\mathsf{TT}}{\vdash} T \overset{\mathsf{TT}}{\Longrightarrow} X' \twoheadrightarrow \star_n$ then
  $A = \underline{\mathsf{let}}\ x\ :\ S \mapsto v\ \underline{\mathsf{in}}\ T$.

  By induction:

  - $\Gamma \overset{\mathsf{TT}}{\vdash} S \overset{\mathsf{TT}}{\Longrightarrow} \star_n$ gives i.h.
    $\forall \Gamma, b.\Delta = [\![\Gamma]\!], b = [\![S]\!], \exists B.\Delta \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} B \text{and} \Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} \star_n \text{and} \Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$.
  - $\Gamma \overset{\mathsf{TT}}{\vdash} v \overset{\mathsf{TT}}{\Longrightarrow} S'$ gives i.h.
    $\forall \Gamma, b.\ \Delta = [\![\Gamma]\!], b = [\![v]\!], \exists B.\ \Delta \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} B$ and $\Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![S']\!]$ and
    $\Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$.
  - $\Gamma; x\ :\ S \mapsto v \overset{\mathsf{TT}}{\vdash} e \overset{\mathsf{TT}}{\Longrightarrow} T$ gives i.h.
    $\forall \Gamma, b.\ \Delta = [\![\Gamma]\!]; x\ :\ [\![S]\!] \mapsto [\![v]\!], b = [\![e]\!], \exists B.\ \Delta \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} B$ and
    $\Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![T]\!]$ and $\Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$.

So $B = \underline{\text{let}}\ x\ :\ [\![S]\!] \mapsto [\![v]\!]\ \underline{\text{in}}\ [\![T]\!]$ if $\Gamma \overset{\mathsf{TT}}{\vdash} [\![S]\!] \overset{\mathsf{TT}}{\simeq} [\![S']\!]$ (which holds by Corollary B.4).

- Case $a = \lambda x{:}S.\ e$. Then $[\![a]\!] = \lambda x{:}[\![S]\!].\ [\![e]\!]$.

  If $\Gamma; x\ :\ S \overset{\mathsf{TT}}{\vdash} e \overset{\mathsf{TT}}{\Longrightarrow} T$ and $\Gamma \overset{\mathsf{TT}}{\vdash} \forall x{:}S.\ T \overset{\mathsf{TT}}{\Longrightarrow} X \twoheadrightarrow \star_n$ then $\Gamma \overset{\mathsf{TT}}{\vdash} \lambda x{:}S.\ e \overset{\mathsf{TT}}{\Longrightarrow} \forall x{:}S.\ T$.

  By induction:

  - $\Gamma; x\ :\ S \overset{\mathsf{TT}}{\vdash} e \overset{\mathsf{TT}}{\Longrightarrow} T$ gives i.h.

    $\forall \Gamma, b.\ \Delta = [\![\Gamma]\!]; x\ :\ [\![S]\!], b = [\![e]\!], \exists B.\ \Delta \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} B$ and $\Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\cong} [\![T]\!]$ and $\Delta \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$.

  So $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} \lambda x{:}[\![S]\!].\ [\![e]\!] \overset{\mathsf{Ex}}{\Longrightarrow} \forall x{:}[\![S]\!].\ [\![T]\!]$, if $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} \forall x{:}[\![S]\!].\ [\![T]\!] \overset{\mathsf{Ex}}{\Longrightarrow} \star_n$.

  $[\![\Gamma]\!]; x\ :\ [\![S]\!] \overset{\mathsf{Ex}}{\vdash} [\![T]\!] \overset{\mathsf{Ex}}{\Longrightarrow} \star_n$, by i.h. and $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![S]\!] \overset{\mathsf{Ex}}{\Longrightarrow} \star_n$ holds if $[\![\Gamma]\!]; x\ :\ [\![S]\!]$ $\underline{\text{valid}}$.

  So take $A = \forall x{:}S.\ T$ and $B = \forall x{:}[\![S]\!].\ [\![T]\!]$.

□

Theorem B.9 (Property 3 for forcing). *If* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\simeq} B$ *then* $\Gamma \overset{\mathsf{TT}}{\vdash} A \simeq |B|$

Proof. By Lemma B.6, $\Gamma \overset{\mathsf{TT}}{\vdash} |[\![A]\!]| \overset{\mathsf{TT}}{\simeq} |B|$.

Then by Lemma B.5, $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$. □

B.3 The Detagging Optimisation

Detagging is given in full in figure B.5. The translations are on *well-scoped* terms, i.e. all variables are declared or defined in the context:

$[\![\star_n]\!] \Longrightarrow \star_n$
$[\![x]\!] \Longrightarrow x$
$[\![\mathsf{D}]\!] \Longrightarrow \mathsf{D}$
$[\![\textbf{D-Elim}]\!] \Longrightarrow \textbf{D-Elim}$
$[\![f\ s]\!] \Longrightarrow [\![f]\!]\ [\![s]\!]$
$[\![\forall x{:}S.\ T]\!] \Longrightarrow \forall x{:}[\![S]\!].\ [\![T]\!]$
$[\![\lambda x{:}S.\ e]\!] \Longrightarrow \lambda x{:}[\![S]\!].\ [\![e]\!]$
$[\![\underline{\text{let}}\ x\ :\ S \mapsto v\ \underline{\text{in}}\ e]\!] \Longrightarrow \underline{\text{let}}\ q\ :\ [\![S]\!] \mapsto [\![v]\!]\ \underline{\text{in}}\ [\![e]\!]$
$[\![\mathsf{c}]\!] \Longrightarrow \lambda \vec{a}{:}[\![\vec{A}]\!].\ \lambda \vec{y}{:}\mathsf{D}\ [\![\vec{i}]\!].\ \{\mathsf{c}\}\ \vec{a}^{\{V\}}\ \vec{y}$ $\underline{\text{if}}$ D is concretely detaggable.
$[\![\mathsf{c}]\!] \Longrightarrow \lambda \vec{a}{:}[\![\vec{A}]\!].\ \lambda \vec{y}{:}\mathsf{D}\ [\![\vec{i}]\!].\ \mathsf{c}\ \vec{a}^{\{V\}}\ \vec{y}$ $\underline{\text{otherwise}}$.
where V is the set of concretely forceable variables in $\vec{a}$
$a^{\{V\}} \Longrightarrow \{a\}$ $\underline{\text{if}}$ $a \in V$
$a^{\{V\}} \Longrightarrow a$ $\underline{\text{otherwise}}$

Figure B.5: The detagging optimisationg

As with forcing, detagging is applied across a context, with the types of c and D-**Elim** modified accordingly. Detagging of a context is given in figure B.6:

$$
\begin{array}{l}
[\![\mathcal{E}]\!] \Longrightarrow \mathcal{E} \\
[\![\Gamma; \mathsf{c} \ : \ \forall \vec{a}{:}\vec{A}.\, \forall \vec{y}{:}\vec{Y}.\, \mathsf{D}\ \vec{s}]\!] \Longrightarrow [\![\Gamma]\!]; \{\mathsf{c}\} \ : \ \forall \vec{a}{:}\vec{A}^{\{V\}}.\, \forall \vec{y}{:}\vec{Y}.\, \mathsf{D}\ [\![\vec{s}]\!] \text{ if } \mathsf{D} \text{ is concretely detaggable} \\
[\![\Gamma; \mathsf{c} \ : \ \forall \vec{a}{:}\vec{A}.\, \forall \vec{y}{:}\vec{Y}.\, \mathsf{D}\ \vec{s}]\!] \Longrightarrow [\![\Gamma]\!]; \mathsf{c} \ : \ \forall \vec{a}{:}\vec{A}^{\{V\}}.\, \forall \vec{y}{:}\vec{Y}.\, \mathsf{D}\ [\![\vec{s}]\!] \text{ otherwise} \\
\quad \text{where } V \text{ is the set of concretely forceable variables in } \vec{a} \\
\qquad\quad \forall a{:}A^{\{V\}} \Longrightarrow \forall \{a{:}[\![A]\!]\} \text{ if } a \in V \\
\qquad\quad \forall a{:}A^{\{V\}} \Longrightarrow \forall a{:}[\![A]\!] \text{ otherwise} \\
[\![\Gamma; x \ : \ S]\!] \Longrightarrow [\![\Gamma]\!]; x \ : \ [\![S]\!] \\
[\![\Gamma; e \ : \ S \mapsto s]\!] \Longrightarrow [\![\Gamma]\!]; x \ : \ [\![S]\!] \mapsto [\![s]\!]
\end{array}
$$

Figure B.6: Detagging a context

B.3.1 Equivalence of Typechecking for Detagging

Lemma B.10. $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\simeq} |[\![a]\!]|$

Proof. Trivial, since we have η-conversion. The proof is by induction on the typing judgment.

- Case $a = \mathsf{c}$. Then $[\![\mathsf{c}]\!] \Longrightarrow \lambda \vec{a}{:}[\![\vec{A}]\!].\, \lambda \vec{y}{:}\mathsf{D}\ [\![\vec{i}]\!].\, \mathsf{c}\ \vec{a}^{\{V\}}\ \vec{y}$ or
  $[\![\mathsf{c}]\!] \Longrightarrow \lambda \vec{a}{:}[\![\vec{A}]\!].\, \lambda \vec{y}{:}\mathsf{D}\ [\![\vec{i}]\!].\, \{\mathsf{c}\}\ \vec{a}^{\{V\}}\ \vec{y}$, depending whether c's type is detaggable.

  Either way, removing the marks yields $\lambda \vec{a}{:}\vec{A}.\, \lambda \vec{y}{:}\mathsf{D}\ \vec{i}.\, \mathsf{c}\ \vec{a}\ \vec{y}$, which is η-convertible with c.

- All other cases are provable trivially by induction.

□

Lemma B.11. *If* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} S, T \ : \ V$ *and* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} T$ *then* $\Gamma \overset{\mathsf{TT}}{\vdash} |S| \overset{\mathsf{TT}}{\simeq} |T|$.

Proof. Similarly to Lemma B.6, except that there is an additional normal form possible, and hence an additional case:

- $\{\mathsf{c}\}\ \vec{a}\ \vec{y}$, where $\vec{a}, \vec{y}$ are normal forms or of the form $\{t\}$, and t is any term.

- Case $a = \{\mathsf{c}\}\ \vec{a}\ \vec{y}$ and $b = \{\mathsf{c}'\}\ \vec{b}\ \vec{z}$. Then $|a| = \mathsf{c}\ \vec{a}'\ \vec{y}'$ and $|b| = \mathsf{c}'\ \vec{b}'\ \vec{z}'$, where:
  - $a_i = [\![a_i']\!]$ if a_i' is not concretely forceable.
  - $a_i = \{a_i'\}$ if a_i' is concretely forceable.
  - $b_i = [\![b_i']\!]$ if b_i' is not concretely forceable.
  - $b_i = \{b_i'\}$ if b_i' is concretely forceable.
  - $y_i = [\![y_i']\!]$.

– $z_i = [\![z_i']\!]$.

Synthesising types of a and b we get:

– $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} a \overset{\mathsf{Ex}}{\Longrightarrow} \mathsf{D}\ \vec{s}$ for some $\vec{s}$.

– $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} \mathsf{D}'\ \vec{t}$ for some $\vec{t}$.

Since $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} a, b\ :\ V$, we know $\mathsf{D} \overset{\mathsf{Ex}}{\equiv} \mathsf{D}'$.

$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} a \overset{\mathsf{Ex}}{\simeq} b$ if and only if the constructors are identical, i.e. $\mathsf{c} \overset{\mathsf{Ex}}{\equiv} \mathsf{c}'$, and corresponding arguments are convertible. Conversion holds for marked arguments as in Lemma B.6. So we now show that if we are comparing marked constructors, they must be the same constructor.

By the definition of detaggable, marked constructors are determined by their indices, which are already in the context since the terms are well-scoped. Hence an equivalent conversion check has already been made.

□

Theorem B.12 (Property 1 for detagging). *If* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$ *then* $\exists A.\, \Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ *and* $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$

Proof. By induction on the typing judgment, $\Delta \overset{\mathsf{Ex}}{\vdash} b \overset{\mathsf{Ex}}{\Longrightarrow} B$, where $\forall \Gamma, a,\ \Delta = [\![\Gamma]\!]$ and $b = [\![a]\!]$. In each case, we find appropriate A and B such that $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$. Cases are as Theorem B.7 except:

- Case $b = \{\!|f|\!\}\ s$. Then $a = f'\ s'$ where $s = [\![s']\!]$ and $f' \in \Gamma$, since the detagging optimisation only marks constructor names in function position.

  If $\{\!|f|\!\}\ :\ \forall x{:}S.\ T \in \Delta$ and $\Delta \overset{\mathsf{Ex}}{\vdash} s \overset{\mathsf{Ex}}{\Longrightarrow} S'$ and $\Delta \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} S'$ then
  $\Delta \overset{\mathsf{Ex}}{\vdash} \{\!|f|\!\}\ s \overset{\mathsf{Ex}}{\Longrightarrow} \underline{\mathsf{let}}\ x\ :\ S' \mapsto s\ \underline{\mathsf{in}}\ T$, so $B = \underline{\mathsf{let}}\ x\ :\ S' \mapsto s\ \underline{\mathsf{in}}\ T$.

  By induction:

  – $\Delta \overset{\mathsf{Ex}}{\vdash} s \overset{\mathsf{Ex}}{\Longrightarrow} S'$ gives i.h.
    $\forall \Gamma, a.\ \Delta = [\![\Gamma]\!], s = [\![a]\!], \exists A.\ \Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ and $A \overset{\mathsf{TT}}{\simeq} |S'|$

  So $\Gamma \overset{\mathsf{TT}}{\vdash} s' \overset{\mathsf{TT}}{\Longrightarrow} A \overset{\mathsf{TT}}{\simeq} |S'|$ and if $\{\!|f|\!\}\ :\ \forall x{:}S.\ T \in \Delta$ then $f\ :\ \forall x{:}|S|.\ |T| \in \Gamma$.

  So $A = \underline{\mathsf{let}}\ x\ :\ |S'| \mapsto s'\ \underline{\mathsf{in}}\ |T|$ if $\Gamma \overset{\mathsf{TT}}{\vdash} |S| \overset{\mathsf{TT}}{\simeq} |S'|$.

  Since $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} \underline{\mathsf{valid}},\ S, S'\ :\ \star_n$. Then by Lemma B.11, if $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} S, S'\ :\ \star_n$ and $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} S'$ then $\Gamma \overset{\mathsf{TT}}{\vdash} |S| \overset{\mathsf{TT}}{\simeq} |S'|$, and so we now have $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$.

- Case $b = \{f\}\ \{s\}$. Then $a = f'\ x$ where $s = x$, $x \in \Gamma$ and $f' \in \Gamma$.

  If $\{f\} \ : \ \forall\{x{:}S\}.\ T \in \Delta$ and $\Delta \overset{\mathsf{Ex}}{\vdash} s \overset{\mathsf{Ex}}{\Longrightarrow} S'$ and $\Delta \overset{\mathsf{Ex}}{\vdash} S \overset{\mathsf{Ex}}{\simeq} S'$ then
  $\Delta \overset{\mathsf{Ex}}{\vdash} \{f\}\ \{s\} \overset{\mathsf{Ex}}{\Longrightarrow} \underline{\mathsf{let}}\ x \ : \ S' \mapsto s\ \underline{\mathsf{in}}\ T$, so $B = \underline{\mathsf{let}}\ x \ : \ S' \mapsto s\ \underline{\mathsf{in}}\ T$.

  If $\{f\} \ : \ \forall x{:}S.\ T \in \Delta$ then $f \ : \ \forall x{:}|S|\ .\ |T| \in \Gamma$.

  If $x \ : \ S' \in \Delta$, then $x \ : \ |S'| \in \Gamma$.

  Then $A = \underline{\mathsf{let}}\, x \ : \ |S'| \mapsto s\,\underline{\mathsf{in}}\,|T|$, if $\Gamma \overset{\mathsf{TT}}{\vdash} |S| \overset{\mathsf{TT}}{\simeq} |S'|$ (by Lemma B.11) so $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$.

□

Theorem B.13 (Property 2 for detagging). *If* $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} A$ *then* $\exists B.$
$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} B$ *and*
$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![A]\!]$ *and*
$[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\Longrightarrow} X \twoheadrightarrow \star_n$

Proof. By induction on the TT typing judgment. Cases are as for Theorem B.8, except:

- Case $a = \mathsf{c}$. Then either
  - $[\![a]\!] = \lambda\vec{a}{:}\vec{A}.\ \lambda\vec{y}{:}\vec{Y}.\ \mathsf{c}\ \vec{a}^{\{V\}}\ \vec{y}$ or
  - $[\![a]\!] = \lambda\vec{a}{:}\vec{A}.\ \lambda\vec{y}{:}\vec{Y}.\ \{\mathsf{c}\}\ \vec{a}^{\{V\}}\ \vec{y}$

  $\Gamma \overset{\mathsf{TT}}{\vdash} a \overset{\mathsf{TT}}{\Longrightarrow} \forall\vec{a}{:}\vec{A}.\ \forall\vec{y}{:}\vec{Y}.\ \mathsf{D}\ \vec{s}$, by lookup of c in Γ.
  If $\Delta = [\![\Gamma]\!]$ then $\Delta \overset{\mathsf{Ex}}{\vdash} [\![a]\!] \overset{\mathsf{Ex}}{\Longrightarrow} \forall\vec{a}{:}[\![\vec{A}]\!].\ \forall\vec{y}{:}[\![\vec{Y}]\!].\ \underline{\mathsf{let}}\,\vec{a} \ : \ \vec{A} \mapsto \vec{a}\,\underline{\mathsf{in}}\,\underline{\mathsf{let}}\,\vec{y} \ : \ \vec{Y} \mapsto \vec{y}\,\underline{\mathsf{in}}\,\mathsf{D}\,[\![\vec{s'}]\!]$,
  whether or not c is marked.

  Take $A = \forall\vec{a}{:}\vec{A}.\ \forall\vec{y}{:}\vec{Y}.\ \mathsf{D}\ \vec{s}$ and
  $B = \forall\vec{a}{:}[\![\vec{A}]\!].\ \forall\vec{y}{:}[\![\vec{Y}]\!].\ \underline{\mathsf{let}}\,\vec{a} \ : \ \vec{A} \mapsto \vec{a}\,\underline{\mathsf{in}}\,\underline{\mathsf{let}}\,\vec{y} \ : \ \vec{Y} \mapsto \vec{y}\,\underline{\mathsf{in}}\,\mathsf{D}\,[\![\vec{s'}]\!]$
  so by definition $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} B \overset{\mathsf{Ex}}{\simeq} [\![A]\!]$.

□

Theorem B.14 (Property 3 for detagging). *If* $[\![\Gamma]\!] \overset{\mathsf{Ex}}{\vdash} [\![A]\!] \overset{\mathsf{Ex}}{\simeq} B$ *then* $\Gamma \overset{\mathsf{TT}}{\vdash} A \simeq |B|$

Proof. By Lemma B.11, $\Gamma \overset{\mathsf{TT}}{\vdash} |[\![A]\!]| \overset{\mathsf{TT}}{\simeq} |B|$.
Then by Lemma B.10, $\Gamma \overset{\mathsf{TT}}{\vdash} A \overset{\mathsf{TT}}{\simeq} |B|$. □

Appendix C

An Implementation of Normalisation By Evaluation

In this appendix I give an implementation in Haskell of normalisation by evaluation for ExTT, first with the core terms then adding inductive families and ι-schemes. Since ExTT has η-conversion, we will be producing η-long normal forms; that is, all names are fully applied. This implementation is based on ideas of Filinski [Fil01] and discussion with Conor McBride.

Recall that the technique of normalisation by evaluation (figure C.1) is to build a meta-level representation of the term, then reify it back to an object level representation of normal forms and finally to revert to the representation of terms.

C.1 Representation of terms

C.1.1 Representing Well Typed Terms

Leaving aside the representation of inductive families and ι-schemes for the moment, we can represent well-typed terms in ExTT with the Haskell data structure in figure C.2, called `Term`. The scope of a binding is represented explicitly in `Term`, using `Scope`. The purpose of this is to be able to distinguish by type between closed terms (of type `Term`) and terms with free variables (of type `Scope Term`). Local variables are de Bruijn indexed [dB72]; there is no explicit name bound in the term. The index represents the number of bindings since the variable was bound — zero represents the most recently bound variable.

Remark: `String` is not necessarily the best choice for representing variable names, although it is adequate for our purposes here. It may be better to use a representation which distinguishes scope, for example, or distinguishes between machine generated names and user supplied names. [MM04a] details the issues involved.

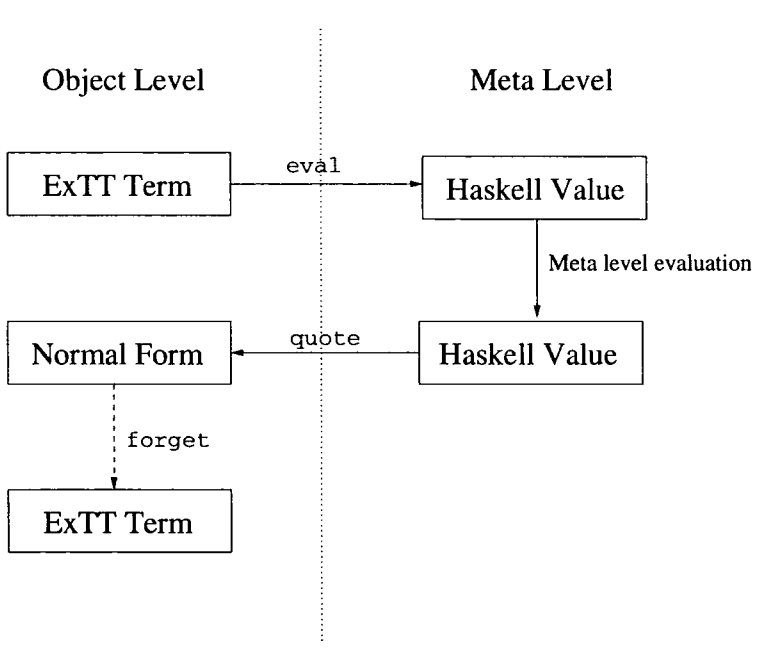


Figure C.1: Normalisation By Evaluation

C.1.2 Representing Normal Forms

There are two stages to the normalisation by evaluation process; translating from the object level to the meta-level, then translating back again. First, we build a model of the term in the meta-language using a function called `eval`. Then we reify the meta-level representation as a syntactic representation of the object language using a function called `quote`.

An important structure in this process is the representation of normal forms. These can be represented both semantically (i.e., the representation in the meta-language) and syntactically (i.e., the representation of normal forms in the object language). We declare a datatype `Model`, given in figure C.3, which is parametrised over a **scope former**, of kind `* -> *`. This parametrisation means that the scope of a binding can be represented in several ways, allowing us to build semantic as well as syntactic representations of values in the same framework. We build a semantic representation by using a function rather than a data constructor as a scope former.

Normal forms are split into two possible cases, the **ready** (or canonical) terms which are already in normal form and the **blocked** terms which could possibly be reduced further if given additional arguments. Blocked terms consist of a head term (with its type, which will be used to direct η-expansion) and a **spine** which holds the arguments applied to the head term. The data type which represents the spine is simply a list where new items are added to the end, rather then the beginning. We implement `fmap`, `splength` and `append` functions for spines as in figure C.4, corresponding to `map`, `length` and `++` on ordinary lists..

```
data Term = V Int                          -- de Bruijn indexed variable
          | P Name                         -- Global name
          | App Term Term                  -- Function application
          | Lam Term (Scope Term)          -- Lambda binding
          | Let Term Term (Scope Term)     -- Let binding
          | Pi Term (Scope Term)           -- Pi binding
          | Const Const                    -- Constant

newtype Scope x = Sc x
data Const = Type Int
type Name = String
```

Figure C.2: Representation of ExTT terms

```
infix 1 :::
data thing ::: type = thing ::: type

data Model s = R (Ready s)
             | B (Blocked ::: (Model s)) (Spine (Model s))

data Ready s = RLam (Model s) (s (Model s))
             | RPi (Model s) (s (Model s))
             | RConst Const

data Blocked = BV Int

data Spine x = Empty | Snoc (Spine x) x
```

Figure C.3: Representation of normal forms

Representing normal forms in this way prevents us from inadvertently creating a term which is not in normal form, for example we cannot construct a λ-binding applied to an argument as a normal form since only blocked terms can have arguments applied to them. That is, we use the type system to help us avoid errors by creating a more precise representation for normal forms.

C.1.3 Representing Scope

`Model` is parametrised over s which, when instantiated, indicates how to represent scope. As well as `Scope`, I introduce a second representation for the scope of a binding, `Kripke` (figure C.5).

`Scope` represents the scope of a binding syntactically; a `Model Scope` is therefore a syntactic representation of normal forms with an obvious forgetful map back to `Term`.

`Kripke`, on the other hand, is a Kripke-style semantic representation of values (possible

```
instance Functor Spine where
  fmap f Empty = Empty
  fmap f (Snoc xs x) = Snoc (fmap f xs) (f x)

splength :: Spine a -> Int
splength Empty = 0
splength (Snoc s v) = 1+splength s

append :: Spine a -> Spine a -> Spine a
append xs Empty = xs
append xs (Snoc ys y) = Snoc (append xs ys) y
```

Figure C.4: Utility functions for `Spine`

```
newtype Kripke x = Kr (Weakening -> x -> x, Weakening)
newtype Weakening = Wk Int
```

Figure C.5: `Kripke` declaration

world semantics — there are many possible values but given more information, i.e. the function argument, we can decide which value applies). This scope former takes a Haskell function to evaluate the body of the scope when passed a `Weakening` and the value to substitute in the body. The weakening is an integer which is used to handle de Bruijn indices correctly — when we go under a binder the index 0 refers to the most recently introduced variable, and all variables above the binder are weakened by 1.

Remark: Using de Bruijn levels, rather than de Bruijn indices, would eliminate the need for the weakening [Fil99, Fil01]. However, this makes it harder to manipulate terms directly and so we prefer to use de Bruijn indices.

We now have two representations for normal forms, called **values** in the semantic case (because they represent a value in the meta-language) and **normals** in the syntactic case (because they represent normal forms directly), with type synonyms declared for convenience, as in figure C.6.

```
type Value = Model Kripke
type Normal = Model Scope
```

Figure C.6: Normal form type synonyms

To implement the normalisation function we have three operations to define — an **evaluation** function to convert a term to its meta-language representation, a **quotation** function which converts the meta-language's semantic representation to a syntactic representation (reification) and finally, it is often useful to have a **forgetful map** which converts the syntactic normal form back to the original representation of well-typed terms (it is a forgetful

map because it forgets the additional information that the term is in normal form).

The evaluation function keeps a local context, `Env`, to keep track of variable bindings. This is represented as a list of values. The global context for the normalisation operation, `Ctxt`, is a lookup table from names to typed `Values` (i.e. `Value:::Value`) — all global definitions are stored as a normal form, built from their TT definitions. Initially, we have a table of definitions, `Defs`, which is a lookup table from names to typed `Terms`. I will take as an invariant of the normalisation operation that all names which are used are guaranteed to be defined in the context. Where the terms are well-typed and there are no names which are not bound to terms (for example, axioms) this will always be the case. `Env` and `Ctxt` are declared as in figure C.7. Figure C.8 shows the declarations of the evaluation, quotation and forgetful map functions.

```
type Defs = [(Name,(Term ::: Term))]
newtype Env = Env [Value]
type Ctxt = [(Name,(Value ::: Value)]
```

Figure C.7: Environment and global context

```
eval :: Ctxt -> Env -> Term -> Value
quote :: Value -> Normal
forget :: Normal -> Term
```

Figure C.8: Normalisation functions

The context, `Ctxt`, is built from the TT definitions, `Defs`, using the `mkCtxt` function, which creates the `Value` representing each definition from the original `Term`:

```
mkctxt :: Defs -> Ctxt -> Ctxt
mkctxt [] acc = acc
mkctxt ((n,v:::t):xs) acc
   = mkctxt xs ((n, (eval acc (Env []) v) ::: eval acc (Env []) t):acc)
```

C.2 The evaluation function "eval"

We write `eval`, the function which translates from well-typed terms to a semantic representation of terms, by case analysis on the input term. The complete definition, for TT but without inductive families and ι-reduction, is shown in figure C.9.

Two cases are straightforward, these being the evaluation of constants and de Bruijn indexed variables. Evaluation of constants is a direct mapping to normal form and evaluation of variables involves looking up the value in the context. This implements δ-reduction:

```
eval :: Ctxt -> Env -> Term -> Value
eval ctxt g (Const c) = R (RConst c)
eval ctxt (Env g) (V n) = g!!n
eval ctxt g (P x) = case lookup x ctxt of
                      (Just (v ::: t)) -> v
eval ctxt g (Lam t (Sc b)) = R (RLam (eval ctxt g t)
                                 (Kr (\w x -> eval (x:weaken w g) b,Wk 0)))
eval ctxt g (Pi t (Sc b)) = R (RPi (eval ctxt g t)
                                 (Kr (\w x -> eval (x:weaken w g) b,Wk 0)))
eval ctxt (Env g) (Let v t (Sc b))
                                = eval ctxt (Env (eval ctxt (Env g) v):g) b
eval ctxt g (App f a) = apply (eval ctxt g f) (eval ctxt g a)

apply :: Value -> Value -> Value
apply (R (RLam t (Kr (f,w)))) v = f w v
apply (B b s) v = B b (Snoc s v)
```

Figure C.9: eval definition for TT without inductive families

```
eval ctxt g (Const c) = R (RConst c)
eval ctxt (Env g) (V n) = g!!n
```

Evaluation of global names involves looking them up in the global context `Ctxt`, substituting the body for the name. Since the context stores normal forms, no further work is required to produce a `Value`:

```
eval ctxt g (P x) = case lookup x ctxt of
                      (Just (v ::: t)) -> v
```

β-reduction lies at the heart of the normalisation algorithm and so the `Lam` and `Pi` cases are where the real work takes place. These cases involve building up an appropriate semantic representation of the scope of the normal form, so that we use the meta-language's implementation of substitution.

```
eval ctxt g (Lam t (Sc b)) = R (RLam (eval ctxt g t)
                                 (Kr (\w x -> eval (x:weaken w g) b,Wk 0)))
eval ctxt g (Pi t (Sc b)) = R (RPi (eval ctxt g t)
                                 (Kr (\w x -> eval (x:weaken w g) b,Wk 0)))
```

In each of these cases the scope of the binding is a function which adds the argument to the local context, weakening the values already in the context by the given weakening, then evaluates the body of the lambda binding in this new context. Thanks to Haskell's lazy evaluation semantics, this function is not executed yet and will not be until requested by the `quote` function. In this way we rely on Haskell's substitution mechanism to perform the

substitution rather than implementing it ourselves. Implementation of the `weaken` function is by recursion over `Value`, incrementing any variables in the value by the weakening.

Evaluation of a let binding is similar, except that we already know the value which is to be added to the environment. We continue by evaluating the scope, with the bound value added to the environment. This implements the contextual closure rule:

```
eval ctxt (Env g) (Let v t (Sc b))
                              = eval ctxt (Env (eval ctxt (Env g) v):g) b
```

Finally, we have the function application case. This evaluates the function and its argument and uses a helper function to perform the application.

```
eval ctxt g (App f a) = apply (eval ctxt g f) (eval ctxt g a)
```

The `apply` function checks whether the function is a lambda binding, and if so applies the function in its scope. If the function is a blocked application, we simply add the argument to the spine of that blocked application. When adding ι-schemes this will become more important, since adding an extra argument may make the blocked term reducible, specifically in the cases where it makes a constructor or elimination rule fully applied.

```
apply :: Value -> Value -> Value
apply (R (RLam t (Kr (f,w)))) v = f w v
apply (B b s) v = B b (Snoc s v)
```

C.3 The quotation function "quote"

The `quote` function takes a semantic representation of a term and translates it back to a syntactic representation. In the process, the meta-language reduces the semantic representation to normal form, hence the `quote` function produces syntactic normal forms, `Normal`, from their semantic representations, `Value`. This involves traversing the term and evaluating any unevaluated scope functions with an appropriate weakening and argument. We define a type class, in figure C.10, for quotable terms. While this is a very general class definition, there are two advantages to using a class rather than simply defining a function:

- It allows the name `quote` to be overloaded for each part of the `Model` structure.
- We may at some stage wish to extend the class definition to include extra features such as name generation (say, to map de Bruijn indexed local variables back to the user's chosen name).

This class definition relies on multi parameter type classes and functional dependencies, non-standard features of Haskell available in the Glasgow Haskell Compiler and some other

```
class Quote x y | x -> y where
   quote :: x -> y
```

Figure C.10: Type class for quote

```
instance Quote Value Normal where
  quote (R r) =  R (quote r)
  quote (B (b ::: t) s) = B (b ::: quote t) (fmap quote s)

instance Quote (Ready Kripke) (Ready Scope) where
  quote (RConst c) = RConst c
  quote (RLam t s) = RLam (quote t) (syntactify t s)
  quote (RPi t s) = RPi (quote t) (syntactify t s)

syntactify :: Value -> Kripke Value -> Scope Normal
syntactify t (Kr (f,w))
   = (Sc (quote (f (weaken w (Wk 1)) (B ((BV 0) ::: t) Empty))))
```

Figure C.11: The quote operation on Value

implementations. The instance definition which converts values into syntactic normal forms is given in figure C.11.

The function syntactify is a helper function for this operation which applies the function representing the scope to an appropriate value. Since we don't know what the argument is as the function is not fully applied, the appropriate argument is naturally the de Bruijn index 0, standing for the most recently bound variable. The function f evaluates the scope of a binding passed to syntactify. Since this function evaluates under a binder, the context in which the scope is evaluated is weakened by 1, meaning that variables which were bound on a higher level are referred to correctly.

In this case, no further work is required to produce an η-long normal form, since there are no blocked names to expand.

C.4 The forgetful map "forget"

Having used a more precise data structure to create the normal form of a term, it is often helpful to be able to return the normal form as a Term itself. This is the purpose of the forget function which maps a syntactic normal form to the equivalent Term.

Again forget is defined using a type class, in figure C.12, which allows the name to be overloaded for each part of the Model structure.

forget is a straightforward traversal of normal forms, the only difficulty being that functions are applied to only one argument, rather than an entire spine. To deal with this we use a helper function makeApp. The instance definitions which create a Term from a

```
class Forget x y | x -> y where
  forget :: x -> y
```

Figure C.12: Type class for `forget`

`Normal` are given in figure C.13.

```
instance Forget Normal Term where
  forget (B (b ::: t) s) = makeApp (forget b) (fmap forget s)
  forget (R r) = forget r

instance Forget Blocked Term where
  forget (BV i) = V i

instance Forget (Ready Scope) Term where
  forget (RLam t (Sc s)) = Lam (forget t) (Sc (forget sc))
  forget (RPi t (Sc s)) = Pi (forget t) (Sc (forget sc))
  forget (RConst c) = Const c

makeApp f Empty = f
makeApp f (Snoc xs x) = App (makeApp f xs) x
```

Figure C.13: The forgetful map from `Normal` to `Term`

C.5 Adding ι-schemes

The normalisation function presented so far gives the basic details of normalisation by evaluation. To make the system useful however, we would like data structures and some way of choosing between different code branches. Chapter 2 described inductive families and their elimination rules — in this section we will see how elimination rules can be implemented in a normalisation by evaluation setting. This requires adding a representation of constructor forms and elimination rules to the term language.

C.5.1 Constructors

A constructor form is simply a global name applied to some arguments; we can already represent this in the term language. Unlike function names, however, they do not map to a definition, but rather are used to direct ι-reduction. We therefore modify the definition of `Defs`. A name maps to either a function definition (`Fun Term`), a construtor with its arity (`Con Int`) or a type constructor with its arity (`TyCon Int`). The new definiition of `Defs` is shown in figure C.14.

```
data NameDef = Fun Term
             | Con Int
             | TyCon Int
type Defs = [(Name,(NameDef ::: Term))]
```

Figure C.14: Adding constructor definitions

An elimination rule can only be reduced when given a fully applied constructor. We therefore add constructor names and type constructors to the blocked normal forms (for constructors which are not fully applied) and to the ready normal forms (for those which are fully applied). These additions are shown in figure C.15.

```
data Blocked = ...
             | BCon Name Int
             | BTyCon Name Int

data Ready s = ...
             | RCon Name (Spine (Model s))
             | RTyCon Name (Spine (Model s))
```

Figure C.15: Adding constructors to normal forms

The fully applied constructors also store the values to which they are applied; this is convenient for implementing elimination rules which access the arguments of a constructor.

There are situations where it might be useful to add further information to constructor names. An integer tag on the constructor can act as a reference into a lookup table of ι-reductions. This is the representation chosen by several graph reduction systems to speed up choice of reduction, including early implementations of the G-machine [Pey87, PL92]. Instead of a tag, a function pointer can be used to directly point to the code for the ι-scheme, which is the approach taken by the STG machine [Pey92]. We choose the straightforward representation of the name and arguments here because of the approach we take to implementing elimination rules.

As before, we use a function `mkctxt` to build a context from the list of definitions. Function definitions map to `Values` as before. Constructor names also map to `Values`; a constructor of zero arity is fully applied so we build a ready form, otherwise we build a blocked form.

C.5.2 Elimination Rules

Elimination rules are generated from a user defined data type, rather than by the user directly. Syntactically, an elimination rule is simply a name; however, semantically, there must be an implementation of its pattern matching behaviour. For this reason, we do not

```
mkctxt :: Defs -> Ctxt -> Ctxt
mkctxt [] acc = acc
mkctxt ((n,(Fun v):::t):xs) acc
   = mkctxt xs ((n, (eval acc (Env []) v) ::: eval acc (Env []) t):acc)
mkctxt ((n,(Con 0):::t):xs) acc
   = mkctxt xs ((n, R (RCon n Empty) ::: eval acc (Env []) t):acc)
mkctxt ((n,(Con i):::t):xs) acc
   = mkctxt xs ((n, B ((BCon n i) ::: ty) Empty ::: ty):acc)
  where ty = eval acc (Env []) t
mkctxt ((n,(TyCon 0):::t):xs) acc
   = mkctxt xs ((n, R (RTyCon n Empty) ::: eval acc (Env []) t):acc)
mkctxt ((n,(TyCon i):::t):xs) acc
   = mkctxt xs ((n, B ((BTyCon n i) ::: ty) Empty ::: ty):acc)
  where ty = eval acc (Env []) t
```

Figure C.16: Building a context of `Values` with constructors

represent ι-schemes directly as terms, but rather as a function implementing that rule's behaviour.

An elimination rule takes a number of arguments and if it is possible to apply the rule to those arguments, returns the `Value` representing the result of elimination. If not, evaluation cannot proceed. The Haskell type describing this behaviour is:

```
type ElimRule = Spine Value -> Maybe Value
```

For each data type, there is a function of type `ElimRule` which defines its ι-schemes. We do not add elimination rules to the term language; they are represented by their names, which are bound in the global context to a `Value`.

In the language of normal forms, it may be that we have an elimination operator which cannot be applied, either because it has too few arguments or because its target is not in canonical form. A value is in canonical form if it cannot be reduced further; that is, it is ready rather than blocked. For data types, this means that the target is a fully applied constructor. For this reason, we add elimination rules to the blocked normal forms, as in figure C.17. We keep the name of the elimination rule as well as its implementation, so that we can implement the forgetful map back to `Terms` if the elimination rule cannot be reduced.

```
type ElimRule = Spine Value -> Maybe Value

data Blocked = ...
             | BElim (ElimRule,Name)
```

Figure C.17: Adding elimination rules to normal forms

We extend `NameDef` to map a name to an implementation of an elimination rules, and extend `mkctxt` accordingly, as in figure C.18. Elimination rules are represented in pattern

matching form (`Patterns`) and compiled to an implementation (`ElimRule`) using `mkelim`, although we will not go into the details of this representation here.

```
data NameDef = ...
             | Elim Patterns

mkctxt ((n,(Elim p):::t):xs) acc
   = mkctxt xs ((n, B ((BElim (mkelim p,n)) ::: ty) Empty ::: ty):acc)
  where ty = eval acc (Env []) t
```

Figure C.18: Adding elimination rules to `Defs`

C.5.3 Evaluation of Elimination Operators

It now remains to define cases of `eval`, `quote` and `forget` for constructors and elimination operators. The complete definition of `eval` is given in figure C.19. The only addition is in the `apply` helper function, since constructor and elimination rules are evaluated by looking their values up in the context. When applying a blocked constructor to an argument, we check whether the constructor is now fully applied; if so, we create a ready term, otherwise we simply add the argument to the spine.

```
apply (B ((BCon n i) ::: ty) s) v
    | splength (Snoc s v) = i = R (RCon n (Snoc s v))
    | otherwise = B ((BTyCon n i) ::: ty) (Snoc s v)
apply (B ((BTyCon n i) ::: ty) s) v
    | splength (Snoc s v) = i = R (RTyCon n (Snoc s v))
    | otherwise = B ((BTyCon n i) ::: ty) (Snoc s v)
```

Whenever an argument is added to a blocked elimination rule, we try to apply the elimination rule to its arguments by applying its `ElimRule` function. If this produces a value we continue, otherwise we retain the blocked term.

```
apply (B ((BElim (e,x)) ::: ty) s) v
       = case e (Snoc s v) of
             Nothing -> (B ((BElim (e,x)) ::: ty) (Snoc s v))
             Just v -> v
```

For quotation, no extra work is required for blocked constructors since they are not parametric in their scope. The complete definition, with constructors and ι-reduction, is shown in figure C.20. The addition to the previous definition is for fully applied constructors, which are quoted as follows:

```
quote (RCon n s) = RCon n (fmap quote s)
quote (RTyCon n s) = RTyCon n (fmap quote s)
```

```
eval :: Ctxt -> Env -> Term -> Value
eval ctxt g (Const c) = R (RConst c)
eval ctxt (Env g) (V n) = g!!n
eval ctxt g (P x) = case lookup x ctxt of
                        (Just (v ::: t)) -> v
eval ctxt g (Lam t (Sc b)) = R (RLam (eval ctxt g t)
                                     (Kr (\w x -> eval (x:weaken w g) b,Wk 0)))
eval ctxt g (Pi t (Sc b)) = R (RPi (eval ctxt g t)
                                   (Kr (\w x -> eval (x:weaken w g) b,Wk 0)))
eval ctxt (Env g) (Let v t (Sc b))
                                  = eval ctxt (Env (eval ctxt (Env g) v):g) b
eval ctxt g (App f a) = apply (eval ctxt g f) (eval ctxt g a)

apply :: Ctxt -> Value -> Value -> Value
apply (R (RLam t (Kr (f,w)))) v = f w v
apply (B ((BCon n i) ::: ty) s) v
    | splength (Snoc s v) = i = R (RCon n (Snoc s v))
    | otherwise = B ((BTyCon n i) ::: ty) (Snoc s v)
apply (B ((BTyCon n i) ::: ty) s) v
    | splength (Snoc s v) = i = R (RTyCon n (Snoc s v))
    | otherwise = B ((BTyCon n i) ::: ty) (Snoc s v)
apply (B ((BElim (e,x)) ::: ty) s) v
    = case e (Snoc s v) of
        Nothing -> (B ((BElim (e,x)) ::: ty) (Snoc s v))
        Just v -> v
apply (B b s) v = B b (Snoc s v)
```

Figure C.19: Complete `eval` definition, with ι-reduction

The forget operation is also relatively straightforward since most of the work, dealing with the spine, has already been done. The complete definition is shown in figure C.21. Forgetting blocked constructors is a straightforward map to the `Term` constructors:

```
forget (BCon n i) = P n
forget (BTyCon n i) = P n
```

When we forget an elimination rule which could not be applied, we get back the name of the rule, rather than its implementation:

```
forget (BElim (e,x)) = P x
```

Forgetting fully applied constructors deals with application of the spine in a similar way to the spine of blocked applications, using `makeApp`:

```
forget (RCon n s) = makeApp (Con n (splength s)) (fmap forget s)
forget (RTyCon n s) = makeApp (TyCon n (splength s)) (fmap forget s)
```

```
instance Quote Value Normal where
  quote (R r) =  R (quote r)
  quote (B (b ::: t) s) = B (b ::: quote t) (fmap quote s)

instance Quote (Ready Kripke) (Ready Scope) where
  quote (RConst c) = RConst c
  quote (RLam t s) = RLam (quote t) (syntactify t s)
  quote (RPi t s) = RPi (quote t) (syntactify t s)
  quote (RCon n s) = RCon n (fmap quote s)
  quote (RTyCon n s) = RTyCon n (fmap quote s)

syntactify :: Value -> Kripke Value -> Scope Normal
syntactify t (Kr (f,w))
   = (Sc (quote (f (weaken w (Wk 1)) (B ((BV 0) ::: t) Empty))))
```

Figure C.20: The quote operation on Value, with constructors

C.5.4 Quotation to η-long normal form

Now that we have added constructors, the final step is quotation to η-long normal form, implementated as in figure C.22. This is type directed; we quote a term/type pair, and η-expand all elements of function type. At the top level it is fairly straightforward; if we have a function type, we make sure the term is a λ form:

```
quote (v ::: (R (RPi ty (Kr (f,w)))))
   = (R (RLam (quote ty) (Sc (quote ((apply v v0) ::: f w v0)))))
  where v0 = (B ((BV 0) ::: ty) Empty)
```

If we have a blocked application, we have the type of the head symbol, which we use to direct the quotation of the arguments in the spine. The spine holds the arguments backwards, which is slightly inconvenient, but not difficult to deal with:

```
quote ((B (bl ::: ty) sp) ::: _)
   = B (bl ::: quote ty) (fst (qspine sp))
       where qspine Empty = (Empty, ty)
             qspine (Snoc sp v) | (sp', R (RPi t (Kr (f,w)))) <- qspine sp
                 = (Snoc sp' (quote (v ::: t)), f w (v0 t))
             v0 t = (B ((BV 0) ::: t) Empty)
```

C.5.5 Example — Natural Numbers

The natural number data type and its ι-scheme were defined as below in Chapter 2:

```
instance Forget Normal Term where
  forget (B (b ::: t) s) = makeApp (forget b) (fmap forget s)
  forget (R r) = forget r

instance Forget Blocked Term where
  forget (BV i) = V i
  forget (BCon n i) = P n
  forget (BTyCon n i) = P n
  forget (BElim (e,x)) = P x

instance Forget (Ready Scope) Term where
  forget (RLam t (Sc s)) = Lam (forget t) (Sc (forget sc))
  forget (RPi t (Sc s)) = Pi (forget t) (Sc (forget sc))
  forget (RConst c) = Const c
  forget (RCon n s) = makeApp (Con n (splength s)) (fmap forget s)
  forget (RTyCon n s) = makeApp (TyCon n (splength s)) (fmap forget s)

makeApp f Empty = f
makeApp f (Snoc xs x) = App (makeApp f xs) x
```

Figure C.21: The forgetful map from `Normal` to `Term`, with constructors and ι-reduction

$$\underline{\text{data}} \quad \frac{}{\mathbb{N} : \star} \quad \underline{\text{where}} \quad \frac{}{0 : \mathbb{N}} \quad \frac{n : \mathbb{N}}{\mathsf{s}\, n : \mathbb{N}}$$

$$\mathbb{N}\text{-}\mathbf{Elim}\ \ 0\ \ P\ m_0\ m_\mathsf{s} \rightsquigarrow m_0$$

$$\mathbb{N}\text{-}\mathbf{Elim}\ (\mathsf{s}\, k)\ P\ m_0\ m_\mathsf{s} \rightsquigarrow m_\mathsf{s}\ k\ (\mathbb{N}\text{-}\mathbf{Elim}\ k\ P\ m_0\ m_\mathsf{s})$$

The constructor names are represented in `Defs` as follows:

```
[("0",Con 0),("s",Con 1)]
```

For the elimination operator, we define a function of type `ElimRule` which takes a spine of the arguments and returns a value if reduction is possible. Reduction is possible when the spine contains the correct number of arguments (four in the case of $\mathbb{N}$-**Elim**) and the argument in the target position is in canonical form.

For $\mathbb{N}$-**Elim**, we can define such a function by hand, as below. There are two cases in which the function can produce a value. These are when the target matches a fully applied instance of either constructor and the other arguments, P, m_0 and m_s are present. In any other case, no reduction is possible.

```
natelim (Snoc (Snoc (Snoc (Snoc Empty x) P) mZ) mS) = case x of
        (R (RCon "0" Empty)) -> return mZ
        (R (RCon "s" (Snoc Empty n))) ->
             return (apply (apply mS n) (B (BElim (natelim,"natelim"))
                                  (Snoc (Snoc (Snoc (Snoc Empty n) P) mZ) mS)))
```

```
instance Quote (Value ::: Value) Normal where
    quote (v ::: (R (RPi ty (Kr (f,w)))))
       = (R (RLam (quote ty) (Sc (quote ((apply v v0) ::: f w v0)))))
      where v0 = (B ((BV 0) ::: ty) Empty)
    quote ((B (bl ::: ty) sp) ::: _)
       = B (bl ::: quote ty) (fst (qspine sp))
           where qspine Empty = (Empty, ty)
                 qspine (Snoc sp v) | (sp', R (RPi t (Kr (f,w)))) <- qspine sp
                     = (Snoc sp' (quote (v ::: t)), f w (v0 t))
                 v0 t = (B ((BV 0) ::: t) Empty)
    quote (v ::: t) = quote v
```

Figure C.22: Quotation to η-long normal form

```
        _ -> Nothing
natelim _ _ = Nothing
```

C.6 Building Elimination Rules

Of course, we cannot hard code elimination rules for all inductive families — although doing so may be an optimisation for some commonly used families like $\mathbb{N}$, we would like a more general way of evaluating eliminations. For an inductive family D, we would like a general method of constructing a function of type `Spine Value -> Maybe Value` representing its elimination rule D-**Elim**, defined by the following general scheme in pattern matching style:

$$\text{D-}\mathbf{Elim}\ \vec{s}\ (\mathsf{c}_1\ \vec{a}_1\ \vec{y}_1)\ P\ \vec{m} \rightsquigarrow \iota_1$$
$$\ldots$$
$$\text{D-}\mathbf{Elim}\ \vec{s}\ (\mathsf{c}_n\ \vec{a}_n\ \vec{y}_n)\ P\ \vec{m} \rightsquigarrow \iota_n$$

An elimination rule is reducible if the target is in canonical form (that is, there is an `RCon` at the head) and it has been passed the right number of arguments — that is, the length of the spine equals the arity of the elimination rule.

So, given an arity, the location of the target on the spine, and a list of reductions (mapping from constructor name to a function which produces a `Value`, given a local context), we can build a generic implementation of an elimination rule, shown in figure C.23.
The function checks that the spine it is given is the correct length; if not, it cannot proceed:

```
genElim a c rs sp
   | splength sp < a = Nothing
```

If the spine has the appropriate number of arguments, we try to apply the appropriate ι-scheme. `reduce` is a helper operation which takes the target, and the spine with the constructor removed:

```
genElim :: Int -> Int -> [(Name,(Gamma -> Value))] ->
           Spine Value -> Maybe Value
genElim a c rs sp
   | splength sp < a = Nothing
   | otherwise = reduce (sp!!c) (remove c sp rs)

reduce (R (RCon n as)) sp rs
   = do v <- lookup n as
        return v (Env (as 'append' sp))
reduce _ sp rs = Nothing
```

Figure C.23: Complete definition of `genElim`

```
genElim a c rs sp
   | otherwise = reduce (sp!!c) (remove c sp rs)

reduce (R (RCon n as)) sp rs
   = do v <- lookup n as
        return v (Env (as 'append' sp))
reduce _ sp rs = Nothing
```

How does this help? We can use `genElim` to build any reduction rule from its arity, target and ι-schemes. For each constructor $\mathtt{c_i}$ of D, such that $\mathtt{c_i}\ \vec{a}\ \vec{y}\ :\ \mathsf{D}\ \vec{s}$, we build a function representing the ι-scheme for that constructor, with motive P and methods $\vec{m}$, following the ideas of [CL99]:

$\lambda \vec{s}; \vec{a}; \vec{y}; P; \vec{m}.\ \iota_i$

The arity of the elimination rule a is calculated from the number of indices of the type (s) and the number of constructors (n); $a = s + n + 2$, the extra 2 accounting for the target and motive. The position of the constructor in the argument list, c, is given by the number of indices; $c = s$.

The reductions, rs, are given by constructing a map such that $\mathtt{c_i}$ maps to ι_i; ι_i is typechecked so that local variables are represented by the appropriate de Bruijn index. Then the function implementing the elimination rule for D is given by:

D-Elim = `genElim` $a\ c\ rs$

We can build `natelim` in this way as follows:

```
natZ g = eval g (V 0)
natS g = eval g (App (App (V 0) (V 3))
                           (App (App (App (App
                           (P "natelim") (V 3)) (V 2)) (V 1)) (V 0)))
natelim = genElim 4 0 [("0",natZ), ("S",natS)]
```

The de Bruijn indices in the reductions of this rule refer to the `Values` passed through in the local context `g`. Hence, there are no lambdas; the variables have already been bound.

C.7 Conversion Using Normalisation by Evaluation

We have implemented normalisation by evaluation to support the conversion test, which is required for typechecking in an implementation of TT. It is useful to be able to run arbitrary terms within the system during program development (as with COQ's `Eval` tactic) but we are interested in efficient evaluation primarily to speed up the conversion check.

Usually, we would reduce to weak head-normal form when typechecking, as in Coquand's algorithm [Coq96], because this is more efficient in the case where terms differ at the head; it does not require the evaluation to finish when we already know that two terms do not convert. We can however take advantage of Haskell's lazy evaluation to ensure that normalisation does not continue longer than necessary and write the conversion check on `Normals` rather than weak head-normal forms. This conversion check is simply a check for syntactic equality since the terms in question cannot be reduced further.

The efficiency of performing the conversion check this way relies on lazy evaluation. The equality check between two terms in normal form proceeds by checking the head of each term and if there are differences, returning false immediately. Only if the heads are the same does the body need to be evaluated and the `quote` function run on the scope.

The main disadvantage to this approach is that normalisation expands definitions by δ-reduction. For example, we know that **plus** $\simeq$ **plus** because the function names match. Using normalisation by evaluation for the conversion check, we expand the names and check whether the definitions match. With small definitions, this does not appear to be a big overhead and we would expect it in general to be outweighed by the efficiency of normalisation by evaluation as compared to other approaches such as the Krivine machine. As we begin to implement larger programs in EPIGRAM, however, it may be wise to rethink this strategy and implement a more efficient technique, perhaps based on Grégoire and Leroy's compiled strong reduction [GL02].

Note that there is more work to be done on both the theory and practice of normalisation by evaluation for dependent type systems. We would like to do some experiments to determine how efficient normalisation by evaluation is compared with other approaches such as the Krivine Machine and compiled strong reduction. Also, we do not yet have a proof of correctness of normalisation by evaluation for a dependent type system.

Appendix D

G-Machine Implementation Details

The G-machine is written in C++, using the Boehm and Demers garbage collector [BDXH01]. It is not implemented with efficiency as a primary concern (in particular because more recent abstract machine designs such as the STG machine [Pey92] are more efficient) but rather with clarity and ease of results generation. This appendix gives an outline of the implementation.

D.1 Heap Nodes

Heap nodes are represented with a C++ class `Value`, with subclasses for each node type. `Value` itself is derived from `gc_cleanup` which allows garbage collection, and causes the destructor to be called when the structure is no longer accessible. The only interface function we require is `canonical` which returns whether a node can not be reduced further, although in practice we add functions for display and debugging purposes.

```
class Value : public gc_cleanup {
public:
    virtual bool canonical() = 0;
    ...
};
```

Application nodes contain a pointer to the function and its argument.

```
class AppNode : public Value
{
public:
```

```
    AppNode(Value* f, Value* a);
    virtual bool canonical() { return false; }
    ...
private:
    Value *m_f, *m_a;
};
```

Function nodes cannot be reduced further. They contain a pointer to the code implementing the function. Each function itself returns a code pointer, which is used to implement tail recursion (by returning the address of the function to call next).

```
typedef void*(*func)();

class FunNode : public Value
{
public:
    FunNode(func f);
    virtual bool canonical() { return true; }
    ...
private:
    func m_fun;
    int m_arity;
};
```

Constructor nodes contain a tag and an array of their arguments. On construction, the arguments are taken from the stack. We know, from the design of ExTT, that constructors are always fully applied, so there is no need to take into account arguments which may be added later.

```
class ConNode : public Value
{
public:
    ConNode(int m_tag, int arity);
    virtual bool canonical() { return true; }
    ....
private:
    int m_tag;
    Value **m_args;
};
```

Tuple nodes represent detagged constructors, and as such are implemented like `ConNode`, but without a tag.

```
class TupNode : public Value
{
public:
    TupNode(int arity);
    virtual bool canonical() { return true; }
    ....
private:
    Value **m_args;
};
```

D.2 Machine State

The G-machine state is a tuple $\langle C, S, G, E, D \rangle$, holding the code, stack, heap, environment and dump respectively. Each of these components are implemented as follows:

- Code is simply C++ code. References to code within the G-machine are implemented by function pointers.
- The stack is represented as an array of `Value` pointers, together with pointers to the base and the top of the stack.
- The heap is managed by the Boehm-Demers garbage collector, with local variables and the stack holding pointers to `Values` in the garbage collected heap.
- The environment is handled by the C++ compiler; each supercombinator becomes a C++ function, so mapping functions to code can be achieved by means of a function pointer.
- The dump is effectively a call stack, and can therefore be managed by the C++ call stack. Nevertheless, we also need to remember the stack state at the time the call was made, via the following `DumpItem` structure:

  ```
  struct DumpItem {
    Value** stack_base;
    Value** stack_ptr;
  };
  ```

  We therefore maintain a stack of `DumpItem` pointers, and whenever a heap node is evaluated, record the current stack state.

D.3 Evaluation

Each G-machine instruction is implemented by a C++ function, and so each supercombinator is translated to a sequence of C++ function calls. Of these, most are straightforward direct implementations of the operational semantics. The main difficulty is with tail calls, which are implemented using a trick similar to the "tiny interpreter" described in [Pey92]. Each function returns a pointer to the code block to which it would like to jump, rather than calling it.

```
void run(func cont) {
    while(cont!=NULL) {
        cont = (*cont)();
    }
}
```

In a real implementation designed to get the most out of the target machine, we might prefer to use a portable assembly language, such as C-- [PRR99] as the target language, rather than C or C++, reserving C or C++ for some of the higher level details of the runtime system. C-- in particular has useful features such as a lightweight calling convention, tail recursion and multiple return values, giving low level control without having to worry about the details of different architectures.

Bibliography

[AC99] Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. `http://www.cs.chalmers.se/~augustss/cayenne/`, 1999.

[Acz77] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic.* North Holland, 1977.

[AHS95] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, volume 953 of *LNCS*, pages 182–199, 1995.

[AJ89] Lennart Augustsson and Thomas Johnsson. Parallel graph reduction with the $\langle \nu, g \rangle$-machine. In *Functional Programming Languages and Computer Architecture.* ACM Press, 1989.

[Alt93] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, University of Edinburgh, November 1993.

[App92] Andrew W. Appel. *Compiling With Continuations.* Cambridge University Press, 1992.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers — Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Aug84] Lennart Augustsson. A compiler for Lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 218–227, August 1984.

[Aug85] Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 368–381. Springer-Verlag, September 1985.

[Aug98] Lennart Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

[Bar84] Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, 1984.

[BC01] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In *Theorem Proving In Higher Order Logics: 14th International Conferences, TPHOLS 2001*, volume 2152 of *LNCS*, pages 121–135. Springer–Verlag, September 2001.

[BC03] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory, February 2003. Under consideration for publication in Math. Struct. in Comp. Science. Draft, DCS, CTH — INRIA, Sophia Antipolis, France.

[BDXH01] Hans-J. Boehm, Alan J. Demers, Xerox Corporation Silicon Graphic, and Hewlett-Packard Company. A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`, 2001.

[Ber96] Stefano Berardi. Pruning simply typed lambda terms. *Journal of Logic and Computation*, 6(5):663–681, 1996.

[Ber98] Daniel J. Bernstein. Multidigit multiplication for mathematicians. *Advances in Applied Mathematics*, 1998.

[BES98] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In *Prospects for Hardware Foundations 1998*, LNCS, pages 117–137, 1998.

[BJ96] Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Implementation of Functional Languages*, pages 58–84, 1996.

[BMM04] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs 2003*, volume 3085, pages 115–129. Springer, 2004.

[BMZ02] Yves Bertot, Nicolas Magaud, and Paul Zimmerman. A proof of GMP square root. *Journal of Automated Reasoning*, 29:225–252, 2002.

[Boq99] Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, April 1999.

[Bov02a] Ana Bove. *General Recursion in Type Theory.* PhD thesis, Chalmers University of Technology, November 2002.

[Bov02b] Ana Bove. Mutual general recursion in type theory. Technical report, Department of Computing Science, Chalmers University of Technology, May 2002.

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, 1991.

[Bur69] Rod Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System.* Prentice-Hall, NJ, 1986.

[Cap02] Venanzio Capretta. *Abstraction and Computation.* PhD thesis, Katholieke Universiteit Nijmegen, 2002.

[Car88] Luca Cardelli. Phase distinctions in type theory. Manuscript, 1988.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic, volume 1.* North Holland, 1958.

[CH00] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, 2000.

[Chr04] Jacek Chrzaszcz. Modules in Coq are and will be correct. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs 2003*, volume 3085. Springer, 2004.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[CL99] Paul Callaghan and Zhaohui Luo. Implementation techniques for inductive types in Plastic. In Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, volume 1956 of *LNCS*, pages 94–113. Springer-Verlag, 1999.

[CL01] Paul Callaghan and Zhaohui Luo. An implementation of LF with coercive subtyping and universes. *Journal of Automated Reasoning*, 27(1):3–27, 2001.

[CL02] Dave Clarke and Andres Löh. Generic haskell, specifically. In Jeremy Gibbons and Johan Jeuring, editors, *Proceedings of the IFIP TC2 Working Conference on Generic Programming*, pages 21–48. Kluwer Academic Publishers, 2002.

[CO01] Olga Caprotti and Martijn Oostdijk. How to formally and efficiently prove prime(2999). *Symbolic Computation and Automated Reasoning*, pages 114–125, 2001.

[Coq92] Thierry Coquand. Pattern matching with dependent types. Available from `http://www.cs.chalmers.se/~coquand/type.html`, 1992.

[Coq96] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, 1996.

[Coq01] Coq Development Team. The Coq proof assistant — reference manual. `http://coq.inria.fr/`, 2001.

[CP85] Chris Clack and Simon Peyton Jones. Strictness analysis - a practical approach. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Hardware*, pages 35–49. Springer-Verlag, September 1985.

[Dan98] Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

[dB91] N.G. de Bruijn. Telescoping mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204, April 1991.

[Dyb94] Peter Dybjer. Inductive families. *Formal Aspects Of Computing*, 6:440–465, 1994.

[Enn03] Robert Ennals. *Adaptive Evaluation of Non-Strict Programs.* PhD thesis, King's College, University of Cambridge, December 2003.

[EP00] Martin Erwig and Simon Peyton Jones. Pattern guards and transformational patterns. Haskell Workshop, 2000.

[EP03] Robert Ennals and Simon Peyton Jones. Optimistic evaluation — an adaptive evaluation strategy for non-strict programs. In *International Conference on Functional Programming*, pages 287–298, March 2003.

[FI00] Daniel Fridlender and Mia Indrika. Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, 2000.

[Fil99] Andrzej Filinski. A semantic account of type-directed partial evaluation. In G. Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *LNCS*, pages 378–395. Springer-Verlag, 1999.

[Fil01] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In *Typed Lambda Calculi and Applications: 5th International Conference, TLCA 2001*, volume 2044 of *LNCS*, pages 151–165. Springer-Verlag, May 2001.

[FM01] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *International Conference on Functional Programming*, pages 26–37, 2001.

[FW87] John Fairbairn and Stuart Wray. TIM – a simple lazy abstract machine to execute supercombinators. In *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 34–45. Springer-Verlag, 1987.

[G+04] Torbjörn Granlund et al. The GNU Multiple Precision arithmetic library 4.1.3 — manual. Available from `http://www.swox.com/gmp/manual/`, 2004.

[Geu93] Herman Geuvers. *Logic and Type Systems.* PhD thesis, Katholieke Universiteit Nijmegen, 1993.

[GHC03] The GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 6.0*, 2003.

[Gim94] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *Proceedings of TYPES 1994*, pages 39–59, 1994.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming*, pages 235–246, 2002.

[Gog94] Healfdene Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh, 1994.

[Gol00] Mayer Goldberg. An adequate and efficient left associated binary numeral system in the λ-calculus. *Journal of Functional Programming*, 10(6), 2000.

[Gra03] Paul Graham. The hundred year language. Available from `http://www.paulgraham.com/`, April 2003. Keynote address at PyCon 2003.

[Hal01] Thomas Hallgren. Alfa users' guide. Available from `http://www.cs.chalmers.se/~hallgren/Alfa/`, 2001.

[Hin03] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun Of Programming*, Cornerstones of Computing, pages 245–262. Palgrave, March 2003.

[HJJ82] Peter Henderson, Geraint Jones, and Simon Jones. The LispKit manual. Oxford University Computing Laboratory, 1982.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130 – 141, 1995.

[HMP96] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional back-ends within the lambda-sigma calculus. Technical report, INRIA, November 1996.

[Hoa62] C.A.R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[How80] William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B.Curry: Essays on combinatory logic, lambda calculus and formalism.* Academic Press, 1980. A reprint of an unpublished manuscript from 1969.

[HP91] Robert Harper and Randy Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136, 1991.

[Hue89] Gérard Huet. The constructive engine. In R. Narasimhan, editor, *A Perspective in Theoreticak Computer Science*, pages 38–69. World Scientific Publishing, 1989. Commemorative Volume for Gift Siromoney.

[Hug84] John Hughes. *The design and implementation of programming languages.* PhD thesis, Programming Research Group, Oxford, September 1984.

[Hug91] John Hughes, editor. *Functional programming Languages and Computer Architecture*, volume 523 of *LNCS*. Springer-Verlag, 1991.

[IBM54] IBM Applied Science Division. Specifications for the IBM mathematical formula translating system, FORTRAN, November 1954.

[Jay96] Barry Jay. Shape in computing. *ACM Computing Surveys*, 28(2):355–357, 1996.

[JG95] Barry Jay and Neil Ghani. The virtues of eta-expansion. *Journal of Functional Programming*, 5(2):135–154, 1995.

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, June 1984.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag, September 1985.

[Jon94] Mark P. Jones. The implementation of the gofer functional programming system. Technical Report YALEU/DCS/RR-1030, Yale University, May 1994.

[Knu69] Donald E. Knuth. *The Art of Computer Programming - Seminumerical Algorithms*, volume 2. Addison Wesley, 1969.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers by automata. *Soviet Physics-Doklady*, 7:595–596, 1963.

[Lan64] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Lan66] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3), March 1966.

[Ler02] Xavier Leroy. The Objective Caml system release 3.06. `http://caml.inria.fr/ocaml/htmlman/`, August 2002.

[Let02] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for proofs and programs*, LNCS. Springer, 2002.

[LP92] Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical report, Department of Computer Science, University of Edinburgh, 1992.

[LS00] Yanhong A. Liu and Scott D. Stoller. From recursion to iteration: What are the optimizations? In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 73–82, 2000.

[Luo94] Zhaohui Luo. *Computation and Reasoning – A Type Theory for Computer Science.* International Series of Monographs on Computer Science. OUP, 1994.

[Mag94] Lena Magnusson. *The implementation of ALF – A Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitutions.* PhD thesis, Chalmers University of Technology, Göteborg, 1994.

[Mag03] Nicolas Magaud. *Changement de Representation des données dans le Calcul de Constructions.* PhD thesis, Université de Nice - Sophia Antipolis, October 2003.

[MB01] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types For Proofs And Programs 2000*, pages 181–196. Springer, 2001.

[McB00a] Conor McBride. *Dependently Typed Functional Programs and their proofs.* PhD thesis, University of Edinburgh, May 2000.

[McB00b] Conor McBride. Elimination with a motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs*, pages 197–216. Springer, 2000.

[McB02] Conor McBride. Faking it – simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4+5):375–392, 2002.

[McB04] Conor McBride. Epigram: Practical programming with dependent types. Lecture Notes, International Summer School on Advanced Functional Programming, 2004.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, 1960.

[Mit94] Kevin Mitchell. Multiple values in Standard ML. Technical Report 94-312, LFCS, Dept of Computer Science, University of Edinburgh, 1994.

[Mit03] John C. Mitchell. *Concepts in Programming Languages.* Cambridge University Press, 2003.

[ML71] Per Martin-Löf. An intuitionistic theory of types, 1971.

[ML75] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73.* North-Holland, 1975.

[ML85] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1985.

[MM04a] Conor McBride and James McKinna. I am not a number, I am a free variable. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, 2004.

[MM04b] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[MSD01] David R. Musser, Atul Saini, and Gillmer J. Derge. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison Wesley, 2001.

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

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A proof assistant for higher order logic*, volume 2283 of *LNCS.* Springer-Verlag, March 2002.

[Oka99] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, May 1999.

[P+02] Simon Peyton Jones et al. Haskell 98 language and libraries — the revised report. Available from `http://www.haskell.org/`, December 2002.

[Par92] Will Partain. The nofib benchmark suite of Haskell programs. In J. Launchbury and P.L. Sansom, editors, *Functional Programming*, Workshops in Computing. Springer Verlag, 1992.

[Pey87] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Pey92] Simon Peyton Jones. Implementing lazy functional languages on stock hardware – the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PL91a] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In Hughes [Hug91], pages 636–666.

[PL91b] Simon Peyton Jones and David Lester. A modular fully lazy lambda lifter in Haskell. *Software Practice and Experience*, 21(5):479–506, May 1991.

[PL92] Simon Peyton Jones and David Lester. *Implementing Functional Languages - A Tutorial.* Prentice Hall International, 1992.

[PM89] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* PhD thesis, Paris 7, 1989.

[PM02] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, September 2002.

[PMR99] Simon Peyton Jones, Simon Marlow, and Alastair Reid. The STG runtime system (revised). Available from `http://www.haskell.org/ghc/documentation.html`, February 1999.

[PNO97] Simon Peyton Jones, Thomas Nordin, and Dino Oliva. C–: A portable assembly language. In C Clack, editor, *Workshop on Implementing Functional Languages, St Andrews.* Springer-Verlag, 1997.

[PRR99] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C–: a portable assembly language that supports garbage collection, 1999. Invited talk at PPDP'99.

[PS98] Simon Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32:3–47, 1998.

[PWW04] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types, 2004. Submitted to POPL 2005.

[Röj95] Niklas Röjemo. Highlights from nhc: A space efficient Haskell compiler. In *Functional Programming Languages and Computer Architecture*, pages 282–292, 1995.

[San95] André Luís de Medeiros Santos. *Compilation By Transformation In Non-Strict Functional Languages.* PhD thesis, University of Glasgow, 1995.

[SMG+99] Julian Seward, Simon Marlow, Andy Gill, Sigbjorn Finne, and Simon Peyton Jones. Architecture of the Haskell execution platform. Available from `http://www.haskell.org/ghc/documentation.html`, July 1999. Version 6.

[SNvP91] Sjaak Smetsers, Eric Nöcker, John van Groningen, and Rinus Plasmeijer. Generating efficient code for lazy functional languages. In Hughes [Hug91], pages 592–617.

[SR00] Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter? Technical Report CS-2000-13, Department of Computer Science, University of Virginia, May 2000.

[Ste77] Guy L. Steele Jr. Lambda : The ultimate goto, 1977. MIT AI Memo 443.

[TT01] Andrew Tolmach and The GHC Team. An external representation for the GHC core language, September 2001.

[Tur79] David Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, 9:31–49, 1979.

[Tur96] David Turner. Elementary strong functional programming. In *First International Symposium on Functional Programming Languages in Education, Nijmegen, Netherlands, December 1995.*, number 1022 in LNCS, pages 1–13. Springer, 1996.

[Wad84] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 45–52, 1984.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Steve Munchnik, editor, *Proceedings, 14th Symposium on Principles of Programming Languages*, pages 307–312. Association for Computing Machinery, 1987.

[Wad90] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[WBBL99] Jon Whittle, Alan Bundy, Richard J. Boulton, and Helen Lowe. An ML editor based on proofs-as-programs. In *Automated Software Engineering*, pages 166–173, 1999.

[WF03] Mitchell Wand and Daniel P. Friedman. On the correctness and efficiency of the Krivine machine. Submitted for publication, October 2003.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming.* PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, December 1998.

[Xi99a] Hongwei Xi. Dead code elimination through dependent types. In *The First International Workshop on Practical Aspects of Declarative Languages*, pages 228–242, San Antonio, January 1999.

[Xi99b] Hongwei Xi. Dependently Typed Data Structures. In *Proceedings of Workshop of Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)*, pages 17–32, Paris, September 1999.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

Index