

Phase Distinctions In The Compilation Of EPIGRAM

James McKinna Edwin Brady

School of Computer Science, University of St Andrews
james.mckinna@st-andrews.ac.uk, eb@dcs.st-and.ac.uk

Abstract

We describe the execution of EPIGRAM on a stock architecture such as the G-machine, compiling via a core type theory and a super-combinator language. We show, via optimising transformations on the core type theory, that unused or duplicated values can be erased at run-time. Thus there exists a phase distinction, not between types and values, but between values which are used at compile-time only and values which are used at run-time. Through a simple example, lookup of a value in a sized vector, we show how our optimisations remove compile-time only values from terms and, furthermore, how we can use straightforward static analysis with our rich type information to avoid the need for any run-time bounds check when executing the `lookup` function.

1. Introduction

This paper describes the compilation techniques we have developed for the practical implementation of a dependently typed functional programming language, EPIGRAM [MM04, McB04], and summarises results obtained in the second author’s PhD thesis [Bra05].

A distinctive feature of EPIGRAM, by contrast with some other recent proposals for new programming languages, or extensions to the Hindley-Milner type system, which incorporate various restricted notions of type which can depend on values, is that it was designed with what we dub **full spectrum** type dependency in mind: values may depend on values, types on values, types on types, and values on types. Moreover, this dependency is not “faked” in any way by considering stratification of the type system of the language with run-time value types living in one layer, with compile-time copies at a kind layer to assist the typechecker, as for example in McBride’s *tour de force* in Haskell [McB02].

Type dependency allows the programmer to specify at compile time relationships intended to hold between values at run-time — such relationships may mean one value can be computed from another, so we might at run-time be able to choose not to 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 the code generated from that program.

The use of dependent types in programming leads to several implementation challenges on the one hand, and optimisation opportunities on the other. One difficulty is that having blurred the distinction between types and values, it appears less clear how to deal with types at run-time. Conversely, it is *necessary* to con-

sider evaluation during typechecking, not only at execution time, as the expressive power of type dependency derives from a notion of equality between types which transcends structural equality. Most work on implementation techniques for these languages has focused on improving compile-time evaluation, while delegating run-time evaluation to existing technology (for example Cayenne executes via LML [Aug98], while COQ extracts to OCaml [Let02, Coq01]).

It has been commonly believed since Cardelli’s early paper [Car88], continuing to the present day [SHL05], that full spectrum type dependency implies the lack of a strict phase distinction between types and values and thus prevents the erasure of type abstractions and type applications at run-time. One of the main technical contributions of this paper, by contrast, is to show that we *can*, in fact, elucidate a phase distinction — not between types and values, but between compile-time only values and run-time values. The blurring of the distinction between types and values means that the typechecker must do some evaluation at compile-time; correspondingly, it means that there are values which exist only to ensure type correctness. By identifying such values, we can recover an erasure semantics.

2. EPIGRAM

EPIGRAM is a platform for full spectrum dependently typed functional programming. It is based on a strongly normalising core type theory with **inductive families** [Dyb94], together with a sophisticated type-directed elaborator from source programs to the type theory, affording the programmer a concrete syntax considerably more terse than the type theory itself.

2.1 Inductive families and function definition in EPIGRAM

Inductive families are simultaneously defined collections of algebraic data types which can be indexed over values as well as types. For example, we can define a “lists with length” (or vector) type; to do this we first declare a type of natural numbers to represent such lengths, using the natural deduction style notation proposed in [MM04]:

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

Then we may make the following declaration of vectors; note that ϵ only targets vectors of length zero, and $x::xs$ only targets vectors of length greater than zero:

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

[copyright notice will appear here]

One advantage of indexing vectors over their length is that any type correct function written over a vector automatically satisfies length invariants specified in the type. For example we can write a function which looks up a value in the vector without the need for a bounds check. We begin by defining the family of finite sets, which we can use to represent bounded numbers:

$$\begin{array}{l} \text{data} \quad \frac{n : \mathbb{N}}{\text{Fin } n : \star} \\ \text{where} \quad \frac{}{f0 : \text{Fin } (s \ n)} \quad \frac{j : \text{Fin } n}{fs \ j : \text{Fin } (s \ n)} \end{array}$$

Reading the typing judgment $j : \text{Fin } n$ informally as ‘ $j < n$ ’, we can see that the family instance $\text{Fin } n$ represents the type of natural numbers bounded above by n . The constructors for Fin correspond to an inductive characterisation of the $<$ relation: $n < sn$ for any n , while if $j < n$, then certainly $j < sn$. Furthermore, we can see that $\text{Fin } 0$ is an *empty* type by examining the indices in the (types of the) constructors for Fin ; they ensure that it is not possible to create an element of $\text{Fin } 0$. Finally, and again obviously, the disjointness of the constructors ensures that $\text{Fin}(sn)$ contains one more element than $\text{Fin } n$, and thus that $\text{Fin}(sn)$ is indeed a type containing exactly n distinct values.

We can now specify the type of a bounds-safe **lookup** function:

$$\text{let} \quad \frac{i : \text{Fin } n \quad v : \text{Vect } A \ n}{\text{lookup } i \ v : A} \quad \dots$$

The dependencies on Fin and Vect give us invariants which must hold in the definition of the lookup function; the number represented by i must be no larger than the length n of the vector, so there is no possibility of looking outside the bounds of the vector. These invariants are verified at compile-time by the typechecker rather than at run time by the run-time system.

2.2 Elaboration

The first stage in the compilation of a programming language is translation to a core representation; e.g. 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 UTT [Luo94]. We call this core language TT.

Translation to TT is achieved by a series of **elaboration** rules, presented in [MM04]. These rules translate the high level program into a type theory term, with all of the implicit arguments and proofs of equalities made explicit. For example, elaboration of Vect leads to the introduction of the following constants into the context:

$$\begin{array}{l} \text{Vect} : \forall A : \star. \forall n : \mathbb{N}. \star \\ \epsilon : \forall A : \star. \text{Vect } A \ 0 \\ :: : \forall A : \star. \forall k : \mathbb{N}. \forall x : A. \forall xs : \text{Vect } A \ k. \text{Vect } A \ (s \ k) \end{array}$$

Note that the arguments A and k , together with their types, which are implicit in the original `data` declaration, have been made explicit here. This is necessary to be able to typecheck these terms; we cannot have A or k as part of the return type if we do not at some stage quantify over either! Similarly, implicit arguments are made explicit in the elaborated type for **lookup**:

$$\text{lookup} : \forall A : \star. \forall n : \mathbb{N}. \forall i : \text{Fin } n. \forall v : \text{Vect } A \ n. A$$

Since these arguments exist only for typechecking, we would like to remove them from the data structure at run-time. However, using a traditional types vs. values phase distinction is insufficient — it is unclear where such a distinction lies in a full spectrum language. Instead, we observe a phase distinction between values needed at compile-time (for typechecking and successful elaboration) and those genuinely needed at run-time; A and k are required at compile-time only, and hence can be removed at run-time. In this paper, we describe the techniques required to make this distinction.

2.3 Programming with Elimination Operators

When we declare an inductive family D such as Vect , we give the constructors which explain how to build objects in that family. Along with this, the machine generates an **elimination operator** $D\text{-Elim}$ (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]. For Vect , we obtain the following operator:

$$\begin{array}{l} \text{Vect-Elim} : \quad \forall A : \star. \forall n : \mathbb{N}. \forall v : \text{Vect } A \ n. \\ \quad \forall P : \forall n : \mathbb{N}. \forall v : \text{Vect } A \ n. \star. \\ \quad \forall m_\epsilon : P \ 0 \ (\epsilon \ A). \\ \quad \forall m_{::} : \forall k : \mathbb{N}. \forall x : A. \forall xs : \text{Vect } A \ k. \\ \quad \quad \forall ih : P \ k \ xs. P \ (s \ k) \ (A \ k \ x \ xs). \\ \quad P \ n \ v \\ \text{Vect-Elim } A \ 0 \ (\epsilon \ A) \ P \ m_\epsilon \ m_{::} : \rightsquigarrow m_\epsilon \\ \text{Vect-Elim } A \ (s \ k) \ (A \ k \ x \ xs) \ P \ m_\epsilon \ m_{::} : \\ \quad \rightsquigarrow m_{::} \ k \ x \ xs \ (\text{Vect-Elim } A \ k \ xs \ P \ m_\epsilon \ m_{::}) \end{array}$$

The arguments to the elimination operator are the **indices** (A and n here), the **target** (the object being eliminated; v here), the **motive** (a function which computes the return type of the elimination; P here) and the **methods** (which describe how to achieve the motive for each constructor form).

Like [McB00a] we give ι -schemes in pattern matching form. These functions should not be confused with EPIGRAM definitions; we use the \rightsquigarrow arrow to indicate that these are machine generated ι -reductions.

For an inductive family D , as well as the basic elimination operator $D\text{-Elim}$, EPIGRAM generates an operator, $D\text{-Case}$, which gives case analysis on D , but no recursion. Although this can be defined in terms of $D\text{-Elim}$, it is more efficient to implement $D\text{-Case}$ reductions directly.

Other elimination operators are canonically associated with an inductive family D , but as they are not germane to this paper, we simply refer the interested reader for further details to [MM04].

EPIGRAM programs are implemented at the TT level in terms of these elimination operators. We note in particular that the basic elimination operator $D\text{-Elim}$ is the only function which has direct access to D values. This has important consequences:

- $D\text{-Elim}$ is a total function; by implementing all functions in terms of $D\text{-Elim}$ and prohibiting general recursion, we can ensure that all functions are total.
- Since $D\text{-Elim}$ is the only means we have for inspecting data in D , we are free to choose any representation for D .

It is not essential to prohibit general recursion, but rather a design choice (see for example the discussion in [AMM05]). The salient point is to ensure a distinguished total fragment of the type theory TT. Compile-time values which are definable in this fragment provide strong static guarantees (and maintain decidability of typechecking), and there are practical benefits for optimisation; in particular, evaluation order in this fragment is irrelevant. An optimiser can also use the knowledge that \perp is not a value to presuppose the forms of values.

2.4 Evaluation at compile-time and run-time

By contrast with evaluation in ordinary functional languages, compile-time evaluation must operate on open terms (or equivalently, go under binders), and its characteristics are quite different to run-time evaluation: while we work hard to ensure good properties, such as termination (and hence decidability of typechecking), we have to accept a wider notion of value. There is not space to

here to elucidate the similarities and contrasts between compile-time and run-time evaluation in EPIGRAM; the interested reader is referred to [Bra05] for the complete picture. It suffices to note that at run-time, we make extensive use of the following **adequacy** property of EPIGRAM: at run-time, a well-typed term t in an inductive family must reduce to some constructor form.

$$\begin{array}{l} \text{if } \vdash t : D \vec{s} \\ \text{then } \text{WHNF}(t) = c \vec{t} \text{ for some } \vec{t} \end{array}$$

Figure 1. Adequacy of TT

2.5 Defining the lookup function

The **lookup** function is defined *interactively*; instead of typing the whole definition, the programmer indicates how the function should be written and EPIGRAM’s elaborator gives appropriate patterns. This is achieved by means of the **by rule**, \Leftarrow .

To indicate that the definition of **lookup** should proceed first by elimination (primitive recursion) over i followed by case analysis on v , the programmer supplies the notation $\Leftarrow \text{elim } i \Leftarrow \text{case } v$ to the elaborator.

Elaborating such notation employs the techniques first described in [McB00b]. The resulting instances of **lookup** to be defined arise from the patterns computed from **Fin-Elim** and **Vect-Case**:

$$\begin{array}{l} \text{lookup } i \quad v \quad \Leftarrow \text{elim } i \Leftarrow \text{case } v \\ \text{lookup } f0 \quad (x :: xs) \mapsto \dots \\ \text{lookup } (fs \ j) \quad (x :: xs) \mapsto \dots \end{array}$$

Note that the elaborator does not give patterns for the empty vector; this is because in a type correct application, this case cannot happen and so there is no need to write code for it. Correspondingly, it is reasonable to expect that there will be no code generated for this case.

The programmer, having employed a more sophisticated type to specify **lookup**, and exploited the power of EPIGRAM’s elaborator, may now finish the definition of **lookup** by returning the head of the vector in the $f0$ case, while making a recursive call in the fs case. This leaves the completed, typecheckable definition as follows:

$$\begin{array}{l} \text{let } \frac{i : \text{Fin } n \quad v : \text{Vect } A \ n}{\text{lookup } i \ v : A} \\ \text{lookup } i \quad v \quad \Leftarrow \text{elim } i \Leftarrow \text{case } v \\ \text{lookup } f0 \quad (x :: xs) \mapsto x \\ \text{lookup } (fs \ j) \quad (x :: xs) \mapsto \text{lookup } j \ xs \end{array}$$

The challenge in compiling EPIGRAM, or indeed any language with full spectrum type dependency, is to ensure at run-time the safety guarantees predicted by the types, while avoiding as far as possible any additional computational overhead arising from the elaboration of such terse high-level source programs.

3. Compiling EPIGRAM Programs

3.1 The Core Language TT

TT is based on Luo’s UTT with definitions, inductive families and equality. The syntax of TT is shown in figure 2. 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].

The type inference rules for TT are given in Appendix A. We note in particular the presence of **labelled types**, introduced in [MM04]. Labelled types are an extension to the core type theory

$$\begin{array}{l} t ::= \star_i \quad (\text{type universes}) \\ | x \quad (\text{variable}) \\ | D \quad (\text{inductive family}) \\ | c \quad (\text{constructor}) \\ | \text{D-Elim} \quad (\text{elimination operator}) \\ | t \ t \quad (\text{application}) \\ | \forall x : t. t \quad (\text{function space}) \\ | \lambda x : t. t \quad (\text{abstraction}) \\ | \text{let } x \mapsto t : t \text{ in } t \quad (\text{let binding}) \\ | \langle c : t \rangle \quad (\text{computation type}) \\ | \text{call } \langle c \rangle \ t \quad (\text{call a computation}) \\ | \text{return } t \quad (\text{return from a computation}) \\ \\ c ::= x \ \vec{t} \quad (\text{computation}) \end{array}$$

Figure 2. The core language, TT

which allow terms to be “labelled” by another term which describes its meaning. The typing and contraction rules for labelled types are given in figure 3.

$$\begin{array}{l} \frac{\Gamma \vdash T : \star_n}{\Gamma \vdash \langle l : T \rangle : \star_n} \quad \text{Label} \\ \frac{\Gamma \vdash t : T}{\Gamma \vdash \text{return } t : \langle l : T \rangle} \quad \text{Return} \\ \frac{\Gamma \vdash t : \langle l : T \rangle}{\Gamma \vdash \text{call } \langle l \rangle \ t : T} \quad \text{Call} \\ \frac{}{\Gamma \vdash \text{call } \langle l \rangle \ (\text{return } t) \rightsquigarrow t} \quad \rho\text{-contraction} \end{array}$$

Figure 3. Typing and contraction rules for labelled types

EPIGRAM programs are defined interactively, with metavariables standing for parts of programs which have not yet been written, and their type. Labelled types allow the types of metavariables to be more informative; the system implicitly inserts a label into the return type of a function. Inserting the label into the type of **lookup** gives:

$$\text{lookup} : \forall A : \star. \forall n : \mathbb{N}. \forall i : \text{Fin } n. \forall v : \text{Vect } A \ n. \langle \text{lookup } A \ n \ i \ v : A \rangle$$

We can read $\langle \text{lookup } A \ n \ i \ v : A \rangle$ as “**lookup** $A \ n \ i \ v$ with type A is computable”. The elaborated definition of **lookup**, being defined in terms of the elimination operator **Fin-Elim**, will contain subexpressions which correspond to computing recursive calls on **lookup** in the $(:: A \ k \ x \ xs)$ case; the use of labelled types allows the types of such expressions to be correlated with the corresponding recursive call **lookup** $A \ k \ j \ xs$. Labels thus provide useful annotation for the programmer, as to the allowable recursive calls of functions, and for the compiler: at run-time we can dispense with the explicit guarantee of terminating recursion obtained via some higher-order elimination operator, in favour of making a direct recursive call, derived from the appropriate label.

Elaborated programs are marked up with these labelled types and include proofs of equational constraints which explain the pattern matching behaviour of the high level program. In this sense, we can think of the core type theory not so much as a primitive programming language, but as a language for explaining why the EPIGRAM programs work. Rather than compiling the core code directly, we ought to be able to use these detailed explanations to

generate efficient code. Figure 4 shows the elaborated **lookup**, including the labelled types and proofs of equational constraints. Let us consider how a naïve approach to compiling this might proceed.

The equational constraints which explain why the ϵ case cannot happen at run-time result in a much larger program than the high level program might suggest. These constraints are necessary in the core code — it is these constraints which allow the program to typecheck in the first place — but we should expect not to have to execute them.

3.2 The Run-Time Language RunTT

Compilation of TT to abstract machine code consists of two high level steps; first we translate to an intermediate representation RunTT, then from RunTT to abstract machine code. RunTT is a language of supercombinators, i.e. higher order functions with no free variables. Each supercombinator sequence is then compiled to a code sequence which, when executed, builds the supercombinator body.

The syntax of RunTT is presented in figure 5. The main features which distinguish RunTT from the core language are:

- λ bindings appear only at the top level of terms; there are no inner λ s and no free variables.
- All constructor applications (including type constructors) are fully applied.
- There is a **case** construct — in TT case analysis is performed by elimination rules; definition of elimination rules in RunTT is via this **case** construct, which arise by compilation of the ι -schemes..

Type information, although it is not executable, is retained as a potential aid to optimisation; we will generally suppress the type label on λ s since at this stage such labels serve no computational purpose.

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

Figure 5. The supercombinator language, RunTT

Translating function definitions to RunTT is by means of lambda lifting; we omit the details, which are standard [PL91]. We make sure all data and type constructors are fully applied, by η -expansion. We can translate pattern matching elimination rules into a compilable form in a straightforward manner, 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.

3.3 Abstract Machine Code

We have based the abstract machine on the G-machine, since it is a standard, well-understood and well-documented approach for implementing run-time systems for lazy languages [Joh84]. We may later consider a more sophisticated implementation based on, for example, the Spineless Tagless G-machine [Pey92, MP04] as

used in GHC, but it is not an essential feature of EPIGRAM, TT 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. However, it is interesting to note that a traditional G-machine architecture can be used as a back end with only a minor modification, being the addition of a single heap node to stand for all types.

3.3.1 Compiling Supercombinators

Each supercombinator is compiled to a series of abstract machine instructions which, when executed, construct an instance of the supercombinator body. Full details of this compilation are presented in [Bra05]; here we simply note that we add a single graph node to the traditional presentation (see for example [Joh84]):

- **TYPE**, into which all types are compiled. 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.

Correspondingly, we add an instruction:

- **MKTYPE**, which creates a reference to the graph **TYPE**.

These are the only significant modifications we have made to the traditional presentation of the G-machine. There is only one **TYPE** node; all references to it are shared. We could, however, imagine further 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].

EPIGRAM and TT 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 this naïve implementation, therefore, we do not remove types. Later, in the optimised compilation path, we will see methods for removing types which can be shown never to be examined.

3.3.2 Variable arity: types which appear to depend on values at run-time

One issue we should additionally consider when implementing a compiler for a dependently typed language is how to deal with functions of varying arity. Consider the following definition of a variadic addition function:

```

let  $\frac{n : \mathbb{N}}{\text{adderType } n : \star}$ 
    adderType  $n \Leftarrow \text{elim } n$ 
    adderType  $0 \mapsto \mathbb{N}$ 
    adderType  $(s\ k) \mapsto \forall n : \mathbb{N}. \text{adderType } k$ 

let  $\frac{n, a : \mathbb{N}}{\text{adder } n\ a : \text{adderType } n}$ 
    adder  $n\ a \Leftarrow \text{elim } n$ 
    adder  $0\ a \mapsto a$ 
    adder  $(s\ k)\ a \mapsto \lambda n : \mathbb{N}. \text{adder } k\ (\text{plus } a\ n)$ 

```

The arity of **adder** 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):

```

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

```

$$\begin{aligned}
\mathbf{dMotive} & : \quad \forall n:\mathbb{N}. \star \\
\mathbf{dMotive} & \mapsto \quad \lambda n:\mathbb{N}. \mathbf{N-Case} \ n \ (\forall n:\mathbb{N}. \star) \ \mathbf{False} \ (\lambda k:\mathbb{N}. \mathbf{True}) \\
\mathbf{discriminate} & : \quad \forall n:\mathbb{N}. \forall p:s \ n = 0. \ \mathbf{False} \\
\mathbf{discriminate} & \mapsto \quad \lambda n:\mathbb{N}. \lambda p:s \ n = 0. \\
& \quad = \mathbf{-elim} \ \mathbb{N} \ (s \ n) \ p \ \mathbf{dMotive} \ () \\
\mathbf{fzCase} & : \quad \forall A:\star. \forall n, k:\mathbb{N}. \forall v:\mathbf{Vect} \ A \ k. \forall p:(s \ n) = k. \langle \mathbf{lookup} \ A \ k \ (f0 \ k) \ v : A \rangle \\
\mathbf{fzCase} & \mapsto \quad \lambda A:\star. \lambda n, k:\mathbb{N}. \lambda v:\mathbf{Vect} \ A \ k. \lambda p:(s \ n) = k. \\
& \quad (\mathbf{Vect-Case} \ k \ v \ (\lambda k:\mathbb{N}. \lambda v':\mathbf{Vect} \ A \ k. \forall p:(s \ n) = k). \langle \mathbf{lookup} \ A \ k \ (f0 \ k) \ v' : A \rangle) \\
& \quad (\lambda p:(s \ n) = 0). \mathbf{False-Elim} \ (\mathbf{lookup} \ A \ k \ (f0 \ k) \ (\epsilon \ A) : A) \ (\mathbf{discriminate} \ n \ p)) \\
& \quad (\lambda k:\mathbb{N}. \lambda a:A. \lambda v':\mathbf{Vect} \ A \ k. \lambda p:(s \ n) = s \ k). \mathbf{return} \ a)) \ p \\
\mathbf{fsCase} & : \quad \forall A:\star. \forall n, k:\mathbb{N}. \forall i:\mathbf{Fin} \ n. \forall i':\mathbf{Fin} \ k. \forall v:\mathbf{Vect} \ A \ k. \\
& \quad \forall ih:\forall v':\mathbf{Vect} \ A \ n. \langle \mathbf{lookup} \ A \ n \ i \ v' : A \rangle. \forall p:(s \ n) = k. \\
& \quad \langle \mathbf{lookup} \ A \ k \ i' \ v : A \rangle \\
\mathbf{fsCase} & \mapsto \quad \lambda A:\star. \lambda n, k:\mathbb{N}. \forall i:\mathbf{Fin} \ n. \lambda i':\mathbf{Fin} \ k. \lambda v:\mathbf{Vect} \ A \ k. \\
& \quad \lambda ih:\forall v':\mathbf{Vect} \ A \ n. \langle \mathbf{lookup} \ A \ n \ i \ v' : A \rangle. \lambda p:(s \ n) = k. \\
& \quad (\mathbf{Vect-Case} \ k \ v \ (\lambda k:\mathbb{N}. \lambda v':\mathbf{Vect} \ A \ k. \forall i':\mathbf{Fin} \ k. \forall p:(s \ n) = k). \langle \mathbf{lookup} \ A \ k \ i' \ v' : A \rangle) \\
& \quad (\lambda i':\mathbf{Fin} \ 0. \lambda p:(s \ n) = 0). \mathbf{False-Elim} \ (\mathbf{lookup} \ A \ k \ i' \ (\epsilon \ A) : A) \ (\mathbf{discriminate} \ n \ p)) \\
& \quad (\lambda k:\mathbb{N}. \lambda a:A. \lambda v:\mathbf{Vect} \ A \ k. \lambda i':\mathbf{Fin} \ (s \ k). \lambda p:(s \ n) = s \ k). \\
& \quad \mathbf{call} \ (\mathbf{lookup} \ A \ n \ i \ v) \ ih \ (= \mathbf{-elim} \ (\mathbf{S_inj} \ n \ k \ p) \ (\lambda n:\mathbb{N}. \mathbf{Vect} \ A \ n)) \ v)) \ i' \ p \\
\mathbf{lookup} & : \quad \forall A:\star. \forall n:\mathbb{N}. \forall i:\mathbf{Fin} \ n. \forall v:\mathbf{Vect} \ A \ n. \langle \mathbf{lookup} \ A \ n \ i \ v : A \rangle \\
\mathbf{lookup} & \mapsto \quad \lambda A:\star. \lambda n:\mathbb{N}. \lambda i:\mathbf{Fin} \ n. \\
& \quad \mathbf{Fin-Elim} \ n \ i \ (\lambda n':\mathbb{N}. \lambda i':\mathbf{Fin} \ n'. \forall v:\mathbf{Vect} \ A \ n'. \langle \mathbf{lookup} \ A \ n' \ i' \ v : A \rangle) \\
& \quad (\lambda n:\mathbb{N}. \lambda v:\mathbf{Vect} \ A \ (s \ n). \mathbf{fzCase} \ A \ n \ (s \ n) \ v \ (\mathbf{refl} \ (s \ n))) \\
& \quad (\lambda n:\mathbb{N}. \lambda i':\mathbf{Fin} \ n. \lambda ih:\forall v':\mathbf{Vect} \ A \ n. \langle \mathbf{lookup} \ A \ n \ i' \ v' : A \rangle. \\
& \quad \lambda v:\mathbf{Vect} \ A \ (s \ n). \mathbf{fsCase} \ A \ n \ (s \ n) \ i' \ (fs \ n \ i') \ v \ (\mathbf{refl} \ (s \ n)))
\end{aligned}$$

Figure 4. Elaboration of lookup

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

3.3.3 Run-Time Considerations

While we can certainly implement EPIGRAM using standard techniques with minor modifications, this naïve approach ignores certain 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 see in the vector **lookup** example. Some overheads which we ought to pay close attention to in the design of an optimised run-time system for TT are:

Invariants of Inductive Families As we noted in section 2.2, implicit arguments in the **data** declaration are made explicit in TT. In a naïve implementation, these values are stored on the heap along with the rest of the structure. An efficient implementation must consider methods for removing implicit information, whether it be inferable at run-time (like the length of a **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.

Proofs Dependently typed functional programs typically include proofs of equations both as additional checks on invariants and in order to assist the type checker. 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 type family which witnesses some computationally irrelevant property.

Dead Code In Impossible Cases The machinery required to prove that the empty vector cases in **lookup** are impossible is quite complex and leaves a lot of computationally redundant information in the term. It is reasonable, however, for the programmer to expect a target machine version of **lookup** which implements the high level analysis directly.

How might we compile TT terms to take these considerations into account? The following sections describe static analyses on TT which address these issues.

4. Optimising TT Terms

The elimination operator **D-Elim** is the basic means TT provides for inspecting data in the inductive family **D**. 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 operator has direct access to the arguments of **D**'s constructors. In this section, based on work presented in [BMM04, Bra05], we give a method for removing redundant information from data structures.

For the case of the **Vect** family, recall that the ι -schemes are as follows:

$$\begin{aligned}
\mathbf{Vect-Elim} \ A \ 0 \quad (\epsilon \ A) \ P \ m_\epsilon \ m_\cdot & : \rightsquigarrow m_\epsilon \\
\mathbf{Vect-Elim} \ A \ (s \ k) \ (:\!:\! A \ k \ x \ xs) \ P \ m_\epsilon \ m_\cdot & : \\
& \rightsquigarrow m_\cdot \ k \ x \ xs \ (\mathbf{Vect-Elim} \ A \ k \ xs \ P \ m_\epsilon \ m_\cdot)
\end{aligned}$$

We can make two observations about this pattern matching definition:

1. 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. However, an 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] and has important consequences; effectively it tells us that a naïve implementation of **Vect-Elim** is passed duplicate information — surely we can erase all but one instance of each repeated argument?
2. 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. Indeed, we can see this as a more general instance of observation (1).

4.1 Presupposed Arguments in Pattern Matching

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

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

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. We annotate 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 6. 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 above, (1) that only one occurrence of a repeated argument need be matched, and (2) 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 operator must be well typed. We define an operation $|p|$ which strips these presupposition marks from a pattern.

$$p ::= \begin{array}{l} x \quad (\text{pattern variable}) \\ | \mathbf{c} \vec{p} \quad (\text{constructor pattern}) \\ | [t] \quad (\text{presupposed term}) \\ | [\mathbf{c}] \vec{p} \quad (\text{presupposed constructor pattern}) \end{array}$$

Figure 6. Pattern Syntax

The meta-operation **MATCH** (figure 7) 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).

The first two lines of **MATCH** test constructors and bind pattern variables as is usual in implementations of pattern matching [Aug85, Pey87]. The remaining two lines, however, presuppose the successful outcome of testing.

To be an acceptable implementation of an elimination operator, we require the following two properties:

- The implementation must be **respectful** of well-typed instances. This condition states that if a set of patterns with presupposition marks matches an argument sequence \vec{t} , yielding

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

Figure 7. Pattern matching semantics

substitutions σ , then applying those substitutions to the unmarked patterns, $|\vec{p}_i|$, yields the original argument sequence \vec{t} .

- The implementation must be **well-defined**. Well-definedness states that in a set of ι -schemes, there is *exactly one* scheme which matches when the operator is fully applied and the target is constructor headed; the elimination operator is implemented by a total function with non-overlapping patterns.

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

In the **standard implementation** of an operator **D-Elim**, 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 $\mathbf{D} : \forall \vec{i} : \vec{I}. \star$, with typical constructor

$$\mathbf{c} : \forall \vec{a} : \vec{A}. \forall d_1 : \mathbf{D} \vec{r}_1. \dots \forall d_j : \mathbf{D} \vec{r}_j. \mathbf{D} \vec{s},$$

our typical ι -scheme has the following standard implementation:

$$\text{For typical } \mathbf{c} : \forall \vec{a} : \vec{A}. \forall d_1 : \mathbf{D} \vec{r}_1. \dots \forall d_j : \mathbf{D} \vec{r}_j. \mathbf{D} \vec{s}$$

$$\mathbf{D}\text{-Elim } [\vec{s}] (\mathbf{c} \vec{a} \vec{y}) P \vec{m} \rightsquigarrow m_{\epsilon} \vec{a} \vec{y} (\mathbf{D}\text{-Elim } \vec{r}_1 y_1 P \vec{m}) \dots (\mathbf{D}\text{-Elim } \vec{r}_j y_j P \vec{m})$$

The standard implementation is well-defined — we have exactly one scheme explicitly matching each of \mathbf{D} 's constructors — and respectful, by inversion of the typing rules. For example, figure 8 shows the standard implementation of **Vect-Elim**; the parameter type A and the length may be presupposed.

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

Figure 8. Standard implementation of **Vect-Elim**

In the case of vectors, alternative implementations are possible which remain well-defined and respectful. We may either examine the constructor of the vector, as in the first implementation in figure 9 or instead privilege index length over vector contents, as in the second implementation.

These implementations suggest two alternative representations of vectors; the first suggests a representation of a list along with its length, the second suggests a Cayenne style representation of length with iterated projection from a tuple [Aug98].

4.2 EXT

The presupposition of arguments in patterns means that those arguments are not needed by the elimination operator. Since only the elimination operator has direct access to the target, this leads naturally to space optimisations where we do not merely “comment

1. **Vect-Elim** $A [0] (\epsilon [A]) P m_\epsilon m_{::} \rightsquigarrow m_\epsilon$
Vect-Elim $A ([s] k) (:: [A] [k] a v) P m_\epsilon m_{::}$
 $\rightsquigarrow m_{::} k a v (\mathbf{Vect-Elim} A k v P m_\epsilon m_{::})$
2. **Vect-Elim** $A 0 ([\epsilon] [A]) P m_\epsilon m_{::} \rightsquigarrow m_\epsilon$
Vect-Elim $A (s k) (\{::\} [A] [k] a v) P m_\epsilon m_{::}$
 $\rightsquigarrow m_{::} k a v (\mathbf{Vect-Elim} A k v P m_\epsilon m_{::})$

Figure 9. Alternative implementations of **Vect-Elim**

out” unnecessary data from patterns — we delete them entirely from the representation of datatypes. **ExTT** is an extension of **TT** which augments the syntax with **marked** terms $\{t\}$ and its operational semantics with corresponding marked patterns (figure 10). The intention of marked 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 operator in order to ι -reduce. Marked patterns match only marked arguments, and yield the identity substitution.

$$\begin{aligned}
 p ::= & \dots \\
 & | \{t\} \quad (\text{marked term}) \\
 & | \{c\} \bar{p} \quad (\text{marked constructor pattern}) \\
 t ::= & \dots \\
 & | \{t\} \quad (\text{marked term}) \\
 & | \forall \{x:t\}. t \quad (\text{marked function}) \\
 \text{MATCH}(\{p\}, \{t\}) \implies & \text{ID}
 \end{aligned}$$

Figure 10. Extensions to **TT** in **ExTT**

We define forgetful mapping operations $|\cdot|$ which remove the deletion marks from **ExTT** patterns and terms, giving a **TT** term; $|p|$ removes the deletion marks from patterns and $|t|$ removes deletion marks from terms.

We will consider a variety of optimisations of inductive families and their elimination operators. 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.

4.3 Forcing — Eliding Redundant Constructor Arguments

The first alternative implementation of **Vect-Elim** in figure 9 above matches A and k in the indices rather than the target. In general, when can we comment out an argument of a constructor, as with ϵ and $::$ in this implementation?

If we have two constructor headed terms, $c \bar{a}$, $c \bar{b}$ in the same type $D \bar{s}$ and the value of the i th argument of c is uniquely determined by (or forced by) the indices \bar{s} , such that $a_i \simeq b_i$, we say that the i th argument of c is **forceable** (figure 11). For example, the A argument to ϵ is forceable since if ϵa , $\epsilon b : \mathbf{Vect} A 0$ then clearly $a \simeq b \simeq A$; no other value would be well typed. For $::$, A and k are forceable in the same way.

Looking at the marking of the alternative implementation we take the presupposition marks in the target and mark those arguments for deletion. We have called this the **forcing** optimisation; it relies on the *uniqueness* of presupposed arguments. The results of applying the forcing optimisation to **Vect** are shown in figure 12.

The i th argument of a constructor c is **forceable** if $\vdash c \bar{a}$, $c \bar{b} : D \bar{s}$ implies $\vdash a_i \simeq b_i$

Figure 11. Forceable arguments

ure 12. The general case of the forcing optimisation is detailed in [BMM04] and given for reference in Appendix B.

$$\begin{aligned}
 [\epsilon] & \implies \lambda A. \epsilon \{A\} \\
 [::] & \implies \lambda A; k; a; v. :: \{A\} \{k\} a v \\
 \mathbf{Vect-Elim} A [0] (\epsilon \{A\}) P m_\epsilon m_{::} & \rightsquigarrow m_\epsilon \\
 \mathbf{Vect-Elim} A ([s] k) (:: \{A\} \{k\} a v) P m_\epsilon m_{::} & \rightsquigarrow m_{::} k a v (\mathbf{Vect-Elim} A k v P m_\epsilon m_{::})
 \end{aligned}$$

Figure 12. Forcing for **Vect**

Note in the **Vect-Elim** operator that the constructor tags 0 and s are presupposed (but not marked for deletion) to indicate that they are not inspected.

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.

4.4 Detagging — Eliding Redundant Constructor Tags

In the second alternative implementation of **Vect-Elim** in figure 9, case selection is by analysis of the length index rather than the target itself.

If we have two constructor headed terms $c \bar{a}$, $c' \bar{b}$ in a type $D \bar{s}$, and the constructor choice is uniquely determined by (or forced by) the indices \bar{s} , such that $c \equiv c'$ we say that the family D is **detaggable** (figure 13) i.e. given \bar{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 $s k$).

A family D is **detaggable** if $\vdash c \bar{a}$, $c' \bar{b} : D \bar{s}$ implies $c \equiv c'$

Figure 13. Detaggable families

The detagging optimisation subsumes the forcing optimisation, and additionally marks constructor tags for deletion. The results of applying the detagging optimisation to **Vect** are shown in figure 14, and the general scheme detailed in [BMM04] and given in Appendix B.

$$\begin{aligned}
 [\epsilon] & \implies \lambda A. \{c\} \{A\} \\
 [::] & \implies \lambda A; k; a; v. \{::\} \{A\} \{k\} a v \\
 \mathbf{Vect-Elim} A 0 (\{c\} \{A\}) P m_\epsilon m_{::} & \rightsquigarrow m_\epsilon \\
 \mathbf{Vect-Elim} A (s k) (\{::\} \{A\} \{k\} a v) P m_\epsilon m_{::} & \rightsquigarrow m_{::} k a v (\mathbf{Vect-Elim} A k v P m_\epsilon m_{::})
 \end{aligned}$$

Figure 14. Detagging for **Vect**

4.5 Collapsing Content Free Families

If we have two terms a, b in a family $D \vec{s}$, and the *values* of a and b are determined entirely by \vec{s} , such that there is at most one canonical element of $D \vec{s}$, then we say D is **collapsible** (figure 15).

A family D is **collapsible**
 if $\vdash x, y : D \vec{s}$ implies $\vdash x \simeq y$

Figure 15. Collapsible families

A central example is the heterogeneous definition of equality (figure 16), which is built in to the type theory and used in the equational constraints generated by the elaborator. Applying the detagging optimisation to this structure yields an elimination operator which does not examine the target at all. This is not surprising — we know that any value of type $x = x$ must be $\text{refl } x$, and not \perp . At run-time, we can exploit the adequacy property of TT and assume that the target is in canonical form. Thus equality reasoning can be removed completely at run-time.

$$\begin{array}{c} \frac{a : A \quad b : B}{a = b : \star} \quad \frac{A : \star \quad a : A}{\text{refl } a : a = a} \\ = \mathbf{-elim} : \forall A : \star. \forall a : A. \forall b : A. \\ \quad \forall x : a = b. \forall P : \forall p : a = b. \star. \\ \quad \forall m_{\text{refl}} : P (\text{refl } A a). P x \\ = \mathbf{-elim} A a a (\text{refl } A a) P m_{\text{refl}} \rightsquigarrow m_{\text{refl}} A a \end{array}$$

Figure 16. Heterogeneous Equality

Other kinds of “propositional” information may also be represented in types in such a way that they may be removed completely at run-time. Consider the less than or equal relation, declared and elaborated as follows:

$$\begin{array}{c} \mathbf{data} \quad \frac{x, y : \mathbb{N}}{x \leq y : \star} \\ \mathbf{where} \quad \frac{}{\text{leO} : 0 \leq y} \quad \frac{p : x \leq y}{\text{leS } p : s x \leq s y} \\ \leq : \forall x : \mathbb{N}. \forall y : \mathbb{N}. \star \\ \text{leO} : \forall y : \mathbb{N}. \leq 0 y \\ \text{leS} : \forall x, y : \mathbb{N}. \forall p : \leq x y. \leq (s x) (s y) \end{array}$$

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 17.

$$\begin{array}{c} \llbracket \text{leO} \rrbracket \implies \lambda y. (\{\text{leO}\} \{y\}) \\ \llbracket \text{leS} \rrbracket \implies \lambda x; y; p. (\{\text{leS}\} \{x\} \{y\} p) \\ \leq \mathbf{-Elim} 0 \quad y \quad (\{\text{leO}\} \{y\}) \quad P \quad m_{\text{leO}} \quad m_{\text{leS}} \rightsquigarrow m_{\text{leO}} y \\ \leq \mathbf{-Elim} (s x) (s y) (\{\text{leS}\} \{x\} \{y\} p) \quad P \quad m_{\text{leO}} \quad m_{\text{leS}} \\ \rightsquigarrow m_{\text{leS}} x y p (\leq \mathbf{-Elim} x y p P m_{\text{leO}} m_{\text{leS}}) \end{array}$$

Figure 17. Detagging 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 be examined directly by $\leq \mathbf{-Elim}$ 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).

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 18, and the general scheme again detailed in [BMM04] and presented in Appendix B.

$$\begin{array}{c} \llbracket \text{leO} \rrbracket \implies \lambda y. (\{\text{leO}\} \{y\}) \\ \llbracket \text{leS} \rrbracket \implies \lambda x; y; p. (\{\text{leS}\} \{x\} \{y\} p) \\ \leq \mathbf{-Elim} 0 \quad y \quad (\{\text{leO}\} \{y\}) \quad P \quad m_{\text{leO}} \quad m_{\text{leS}} \rightsquigarrow m_{\text{leO}} y \\ \leq \mathbf{-Elim} (s x) (s y) (\{\text{leS}\} \{x\} \{y\} p) \quad P \quad m_{\text{leO}} \quad m_{\text{leS}} \\ \rightsquigarrow m_{\text{leS}} x y (\{p\}) (\leq \mathbf{-Elim} x y \{p\} P m_{\text{leO}} m_{\text{leS}}) \end{array}$$

Figure 18. Collapsing 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.

4.6 Phase Distinctions Revisited

The distinction between compile-time evaluation and run-time evaluation is an important one in a dependently typed programming language. The purpose of compile-time evaluation is to check the conversion relation during typechecking; i.e. to check that two terms have a common reduct. This requires strong reduction, reducing under binders. At run-time, however, we perform weak reduction, never reducing under a binder. It is therefore reasonable to consider different optimisations in these two settings; in fact forcing and detagging are applicable at compile-time and mark terms which need never be reduced. Collapsing is applicable at run-time only.

ExTT introduces an extra stage into the compilation process which identifies the distinction between compile-time only and run-time values, marking those which are required at compile-time only for deletion. Having marked the terms, they can be deleted entirely as part of the supercombinator lifting algorithm. It is during the translation to ExTT that we identify and apply the phase distinction — as in implementations of simply typed languages, we erase static information at run-time, but unlike such languages, we require a more sophisticated analysis to identify the distinction.

5. Extending RunTT

5.1 Translating from ExTT to RunTT

We modify the supercombinator lifting algorithm to take account of the 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 marked arguments are matched only by marked patterns and therefore both can safely be removed.

We extend RunTT to deal with the markings which arise from the presuppositions we make in ExTT. The extension we make to RunTT are given in figure 19. These are:

- Argument projection, $e!i$. Where a term t in RunTT is known to have the form $c\langle e_0, e_1, \dots, e_n \rangle$, $t!i$ projects the i th argument out of the tuple.
- Untagged constructors, $\langle \vec{e} \rangle$, for the representation of detagged families. case analysis on such forms is not allowed; instead, argument projection is used to retrieve arguments.
- Impossible to execute code, Impossible. This exists to mark code which can never be executed due to the fact that it requires an instance of an empty type; we will see in section 5.4 how this is used to optimise RunTT terms.

$s ::=$	$\lambda \vec{a} : \vec{e}. e$	(supercombinator)
$e ::=$	\dots	
	$e!i$	(argument projection)
	$\langle \vec{e} \rangle$	(untagged constructor)
	<u>Impossible</u>	(impossible to execute code)

Figure 19. Extensions to RunTT

Compiling elimination operators into RunTT is no longer so straightforward as compilation to a case on the target. In the naïve approach to compiling elimination operators presented in section 3.2 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. As a result, we use argument projection to project subterms out of these terms without checking their form.

An algorithm for transating the optimised ι -schemes into RunTT is given in [Bra05], which takes advantage of the well-definedness and respectfulness properties of elimination operators. It is an important feature of dependently typed pattern matching that testing one argument can tell us the form of another — as we have seen in **lookup** — so it is important that the compilation scheme knows how to deal with this. For example, this algorithm produces the following RunTT function for the detagged elimination operator for Vect:

$$\text{Vect-Elim} \mapsto \lambda A; n; x; P; m_\epsilon; m_{\cdot}. \text{case } n \text{ of}$$

$$\begin{array}{l} 0 \rightsquigarrow m_\epsilon \\ (s\langle k \rangle) \rightsquigarrow m_{\cdot} (n!0) (x!0) (x!1) \\ \quad (\text{Vect-Elim } A (n!0) (x!1) P m_\epsilon m_{\cdot}) \end{array}$$

Here matching on n has told us enough about the form of the x that we do not need to analyse its form, and we can simply project out the arguments where appropriate. Now that we have removed the redundant and duplicated (i.e. compile-time only) information from data structures, we are in a position to apply transformations to the RunTT terms.

5.2 Unfolding Elimination Operators

Where a recursive function over D is written by **D-Elim**, the elimination operator is one extra level of indirection. The purpose of the operator 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.

In section 3.1 we introduced labelled types, which allow expressions to be annotated with their corresponding recursive call. The labelling gives us the *meaning* of each inductive hypothesis; the

term $\text{call } \langle \text{lookup } A k i v \rangle ih$ says that the use of ih represents a call of **lookup** $A k i v$. If that is what it represents, there is no need for 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 20.

$$\llbracket \text{call } \langle f \vec{s} \rangle (t) \rrbracket \implies \text{call } \langle f \vec{s} \rangle (f \vec{s})$$

Figure 20. Rewriting a term with labelled type to introduce recursive calls

The effect of this in ExTT is that the inductive hypotheses are no longer used, and hence **D-Elim** can be replaced directly by **D-Case**. In itself, this does not have a great effect, but when combined with inlining the effect is amplified, as we shall see.

5.3 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.

Inlining is a particularly powerful optimisation technique when combined with other optimisations. Combined with the unfolding of elimination operators, inlining of **D-Case** leads to a direct pattern matching definition of a function. This leads immediately to the removal of the elimination operator's motive, which exists to compute the type of an elimination.

5.4 Avoiding Code Generation in Impossible Cases

The elaborator introduces equational constraints into terms to show cases which can never be evaluated. Where a constraint is impossible to satisfy (due to disjointness of constructors) this gives a function which returns the empty type. The empty type **False**, is declared as follows:

$$\text{data } \overline{\text{False}} : \star \quad \text{where}$$

There are no constructors and correspondingly the elimination operator has no ι -schemes and hence no reduction behaviour. At run-time, where elimination operators 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.

Since **False** has no canonical forms, we can be sure that *any* function taking an argument of type **False** will never be executed. Also, a function whose return type is **False** can never construct a value so it, too, will never be executed. We introduce a new constant, Impossible into RunTT to stand for terms which will never be evaluated. Any function which takes an argument of type **False** or returns **False** has its body replaced with this constant. Such functions can obviously be inlined.

Case branches which are marked as impossible can clearly be pruned. For example, consider the **vTail** function which takes the tail of a vector and hides an intricate proof that the empty vector case is impossible:

$$\text{let } \frac{xs : \text{Vect } A (s \ n)}{\text{vTail } xs : \text{Vect } A \ n}$$

$$\text{vTail } xs \Leftarrow \text{Vect-Case } xs$$

$$\text{vTail } (a::v) \mapsto xs$$

After application of forcing, compilation to RunTT and the marking of impossible to execute terms, we are left with:

$$\mathbf{vTail} \mapsto \lambda A; n; v. \frac{\text{case } v \text{ of}}{\begin{array}{l} \epsilon \langle \rangle \rightsquigarrow \text{Impossible} \\ :: \langle x, xs \rangle \rightsquigarrow (v!1) \end{array}}$$

Since there is only one possible case remaining, we can collapse the case expression entirely, and compile \mathbf{vTail} to a function which simply returns a pointer to the tail, as we would expect from the original definition.

$$\mathbf{vTail} \mapsto \lambda A; n; v. (v!1)$$

The type of \mathbf{vTail} is informative, stating that a vector of length zero is not a well-typed argument. The elaborator accepts the obvious definition with no need to write code for the empty vector, and there is no inefficiency or loss of safety in the generated code.

5.5 Unused Arguments

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.

We partition the arguments into the **active** arguments (which are used in the function body) and the **passive** arguments (which are either unused, or used only in the same argument position in a recursive call). A *passive* argument need not be passed to a function, for obvious reasons — the function will never examine it. Some care is required; this optimisation is only valid if a function is fully applied. For these cases, we retain a wrapper function with no arguments removed.

5.6 Optimising lookup

Returning to our example, we can consider the simplification of the RunTT supercombinator for **lookup**. The following transformations are applied:

- Detagging of Vect and forcing of Fin.
- Unfolding of the elimination operator **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 shown in figure 21. 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.

$$\begin{array}{l} \mathbf{lookup} \mapsto \lambda A; k; i; v. \mathbf{lookup}' i v \\ \mathbf{lookup}' \mapsto \lambda i; v. \text{case } i \text{ of} \\ \quad \mathbf{f0} \langle \rangle \rightsquigarrow v!0 \\ \quad \mathbf{fs} \langle j \rangle \rightsquigarrow \mathbf{lookup}' j (v!1) \end{array}$$

Figure 21. RunTT definition of **lookup**

We arrive at this definition not through detailed analysis of constraints or program flow, but by application of a series of simple, well-understood optimisations to the transformed ExTT terms.

Through the use of dependent types, and a strongly normalising core language, we get a vector lookup function with no run-time bounds check. Some of the intermediate steps required to arrive at this definition are shown in Appendix C.

6. Results

We have applied the optimisations to several programs, compiled for our prototype G-machine implementation. 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.

We have executed programs to do the following:

- Look up an element of a vector
- Compute the greatest common divisor of two \mathbb{N} s.
- Quicksort. This is implemented using Bove and Capretta's domain predicate for quicksort [BC03], which, like all such domain predicates, is collapsible [BMM04, Bra05].
- Total a DList, a list with no repeated elements.
- Typecheck a term, using the typechecker implementation presented in [MM04]
- Evaluate a small program on a well-typed interpreter similar to that implemented in [AC99].

The results are summarised in figure 22. In each case there is a significant reduction in the number of cell allocations made on the heap. 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.

7. Related Work

There have been several experiments in introducing dependent types into practical programming languages. DML [Xi98] is a conservative extension of ML which allows types to be predicated on integers, separating indexing expressions from programs. DML exploits dependent types to catch more errors at compile-time, and also for optimisations, including dead code elimination [Xi99] and

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 ($\lambda x : \iota. x$) ι	Naïve	23232	13746	6910	1620
	Optimised	20891	11820	6722	1136
	Change	-10.08%	-14.01%	-2.72%	-29.88%
Interpreting mult 2 3	Naïve	199832	113916	54669	27766
	Optimised	187473	112620	53637	16919
	Change	-6.18%	-1.14%	-1.89%	-39.07%

Figure 22. Results of marking optimisation

array bounds check elimination [XP98]. Cayenne [Aug98] on the other hand does not separate types and values; despite this, Augustsson shows that, for the type system used in Cayenne, types may be erased at run-time.

For EPIGRAM, we have also chosen a point in the design space where types and values are indistinguishable. More recent experiments with forms of dependent types, such as Generalised Algebraic Data Types [PWW04] and Ω mega [SHL05], prefer to maintain the separation between types and values. [SHL05] suggests that the design choice we have made for EPIGRAM means the “loss of the opportunity to use an erasure semantics, and the ensuing increase in the amount of explicit type annotation required.” However, our work here shows that we *can* use an erasure semantics, using instead a distinction between compile-time and run-time values. Moreover, the type annotations required for typechecking do not carry a price; they are implicit in the high level analysis, inserted by the elaborator, and erased by the compiler.

There are other approaches to run-time erasure. The extraction mechanism of the theorem prover COQ [PM89, Let02] relies on a distinction between a universe of sets (computational values) and propositions (computationally irrelevant values), removing propositions from extracted code. The aim here is slightly different; extraction aims to produce a simply typed program from a specification, rather than compilation of a dependently typed program, and as such does not identify implicit arguments to constructors for erasure. Letouzey and Spitters [LS05] introduce an approach to erasure of such implicit arguments using monads.

Since evaluation takes place at compile-time in order to type-check EPIGRAM terms, we should also consider efficient strong reduction. The current implementation uses normalisation by evaluation [BS91], but we can also consider Grégoire and Leroy’s compiled strong reduction technique [GL02], as applied in COQ. The forcing and detagging optimisations also work at compile time and can be applied *before* type synthesis — the marked values are marked because they are duplicated, and hence have already been typechecked. [Bra05] describes this optimisation further, including a proof that typechecking in this setting is sound and complete.

There is active research into programming with EPIGRAM; [AMM05] describes the rationale and gives an example of programming with dependent types; [MMA05] gives an example of generic programming in EPIGRAM.

8. Conclusions and Further Work

We have seen several techniques for compiling dependently typed programming in this paper. The style of programming encouraged by EPIGRAM 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:

- EPIGRAM is executable on a stock architecture for execution of lazy functional languages with only minor modifications, specifically the introduction of a single heap node for all types.
- There is an identifiable distinction between run-time values and compile-time only values. There is some work to do to establish this distinction but once it is established, we can erase compile-time only values.
- The extra type information, specifically the indices on inductive families which describe properties of values in the family, leads to the possibility of further optimisation. In this paper we have seen how, through static analysis of the elimination rule for Vect and application of straightforward and well-known optimisations, we can remove the bounds check on vector lookup.
- The elimination operator is the only construct which can scrutinise a constructor application. This means we can choose any representation for which a well-defined and respectful operator can be implemented. This can lead to further optimisation, for example a GMP implementation of \mathbb{N} , or perhaps an optimised implementation of Vect as a single block of memory of known size.

It is reasonable to wonder why we compile via the full ExTT terms, rather than compiling EPIGRAM programs directly; after all, the elaboration process gives us a high level pattern matching program. However, the high level pattern matching functions are more general than simply constructor matching. It is possible to create alternative views of data [Wad87, MM04] which give rise to higher level pattern matching, not necessarily based on constructors.

Programming with full spectrum type dependency as in EPIGRAM is an innovative approach to programming; as such the results presented here are the fruits of some of the first investigations into efficient compilation techniques for this point in the design space. This work shows that a dependent type theory such as TT is

indeed an effective base on which to build a feasible programming language, but there is much work still to be done. In particular, we are developing a compiler for EPIGRAM alongside a suite of non-trivial example programs, in order to investigate the effectiveness of these compilation techniques in a more realistic setting. We expect that a more advanced account of the phase distinction and erasure semantics will give rise to many more optimisation opportunities.

Acknowledgments

This paper owes much to our long-standing collaboration with Conor McBride, for which all thanks. Lennart Augustsson proved a considerate but exacting examiner of the second author's PhD.

References

- [AC99] Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter, 1999.
- [AMM05] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter, 2005.
- [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.
- [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.
- [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.
- [Bra05] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.
- [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.
- [Car88] Luca Cardelli. Phase distinctions in type theory. Manuscript, 1988.
- [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.
- [Coq01] Coq Development Team. The Coq proof assistant — reference manual. <http://coq.inria.fr/>, 2001.
- [Dyb94] Peter Dybjer. Inductive families. *Formal Aspects Of Computing*, 6:440–465, 1994.
- [GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming*, pages 235–246, 2002.
- [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.
- [HP91] Robert Harper and Randy Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136, 1991.
- [Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, June 1984.
- [Let02] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for proofs and programs*, LNCS. Springer, 2002.
- [LS05] Pierre Letouzey and Bas Spitters. Implicit and non-computational arguments using monads, 2005.
- [Luo94] Zhaohui Luo. *Computation and Reasoning – A Type Theory for Computer Science*. International Series of Monographs on Computer Science. OUP, 1994.
- [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.
- [MM04] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
- [MMA05] Peter Morris, Conor McBride, and Thorsten Altenkirch. Exploring the regular tree types. In *Types for Proofs and Programs 2004*, 2005.
- [MP04] Simon Marlow and Simon Peyton Jones. How to make a fast curry: push/enter vs eval/apply. In *International Conference on Functional Programming, Snowbird, Sept 2004*, pages 4–15, 2004.
- [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.
- [PL91] Simon Peyton Jones and David Lester. A modular fully lazy lambda lifter in Haskell. *Software Practice and Experience*, 21(5):479–506, May 1991.
- [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.
- [PWW04] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types, 2004. Submitted to POPL 2005.
- [SHL05] Tim Sheard, James Hook, and Nathan Linger. GADTs + extensible kinds = dependent programming, 2005.
- [TT01] Andrew Tolmach and The GHC Team. An external representation for the GHC core language, September 2001.
- [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.
- [Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, December 1998.
- [Xi99] 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.
- [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.

A. Typing rules for TT

$$\begin{array}{c}
\frac{\Gamma \vdash \text{valid}}{\Gamma \vdash \star_n : \star_{n+1}} \text{ Type} \\
\frac{\Gamma; x : S; \Gamma' \vdash \text{valid}}{\Gamma; x : S; \Gamma' \vdash x : S} \text{ Var (Similarly for c, D, D-Elim)} \\
\frac{\Gamma; x \mapsto s : S; \Gamma' \vdash \text{valid}}{\Gamma; x \mapsto s : S; \Gamma' \vdash x : S} \text{ Val} \\
\frac{\Gamma \vdash f : \forall x : S. T \quad \Gamma \vdash s : S}{\Gamma \vdash f s : \text{let } x : S \mapsto s \text{ in } T} \text{ App} \\
\frac{\Gamma; x : S \vdash e : T \quad \Gamma \vdash \forall x : S. T : \star_n}{\Gamma \vdash \lambda x : S. e : \forall x : S. T} \text{ Lam} \\
\frac{\Gamma; x : S \vdash T : \star_n \quad \Gamma \vdash S : \star_n}{\Gamma \vdash \forall x : S. T : \star_n} \text{ Forall} \\
\frac{\Gamma \vdash e_1 : S \quad \Gamma; x \mapsto e_1 : S \vdash e_2 : T}{\Gamma \vdash S : \star_n \quad \Gamma; x \mapsto e_1 : S \vdash T : \star_n} \text{ Let} \\
\frac{\Gamma \vdash T : \star_n}{\Gamma \vdash \langle l : T \rangle : \star_n} \text{ Label} \\
\frac{\Gamma \vdash t : T}{\Gamma \vdash \text{return } t : \langle l : T \rangle} \text{ Return} \\
\frac{\Gamma \vdash t : \langle l : T \rangle}{\Gamma \vdash \text{call } \langle l \rangle t : T} \text{ Call} \\
\frac{\Gamma \vdash x : A \quad \Gamma \vdash A' : \star_n \quad \Gamma \vdash A \simeq A'}{\Gamma \vdash x : A'} \text{ Conv}
\end{array}$$

B. Optimisation Details

B.1 The Forcing Optimisation

Consider a typical constructor, fully applied to variables, $c \vec{a} \vec{y} : D \vec{s}$. 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 23) since they can be retrieved in constant time by pattern matching on the indices.

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

Figure 23. Concretely forceable arguments

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 24. 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.

The general scheme for the forcing optimisation is given in figure 25. Note that types are elided in the λ -bindings; this is to avoid distracting attention from the optimisation itself — $\lambda a; b; \dots e$ is used here as a shorthand for $\lambda a : A. \lambda b : B. \dots e$.

$$\begin{array}{l}
\text{PAT } (V, x) \implies (x \cup V, x) \text{ if } x \notin V \\
\text{PAT } (V, c \vec{t}) \implies (V', \text{LAZY}(c, \vec{p})) \\
\quad \text{if } \text{PATS } (V, \vec{t}) \implies (V', \vec{p}) \\
\text{PAT } (V, t) \implies (V, [t]) \\
\text{PATS } (V, \text{nil}) \implies (V, \text{nil}) \\
\text{PATS } (V, t \vec{t}) \implies (V'', p \vec{p}) \\
\quad \text{if } \text{PAT } (V, t) \implies (V', p) \\
\quad \text{and } \text{PATS } (V', \vec{t}) \implies (V'', \vec{p}) \\
\text{LAZY}(c, [\vec{p}]) \implies [c \vec{p}] \\
\text{LAZY}(c, \vec{p}) \implies [c] \vec{p} \text{ otherwise}
\end{array}$$

Figure 24. Extracting patterns from a constructor's indices

For each $c : \forall \vec{a} : \vec{A}. \forall d_1 : D \vec{r}_1. \dots \forall d_j : D \vec{r}_j. D \vec{s}$
 where $\text{PATS } (\emptyset, \vec{s}) \implies (V, \vec{p})$
 take $[[c]] \implies \lambda \vec{a}; \vec{y}. c \vec{a}^{(V)} \vec{y}$
 D-**Elim** $\vec{p} (c \vec{a}^{(V)} \vec{y}) P \vec{m}$
 $\sim m_c \vec{a} \vec{y} (\text{D-Elim } \vec{r}_1 y_1 P \vec{m}) \dots (\text{D-Elim } \vec{r}_j y_j P \vec{m})$
 where $a^{(V)} \implies \{a\}$ if $a \in V$
 $a^{(V)} \implies a$ otherwise

Figure 25. The Forcing Optimisation

B.2 The Detagging Optimisation

For the detagging optimisation, we match on the indices of an elimination operator. To do this we must actually examine their constructors, so the previous lazy definition of PATS is not sufficient. We compute the patterns we need for the detagging optimisation with EPATS — the same as PATS but with LAZY replaced by EAGER:

$$\text{EAGER}(c, \vec{p}) \implies c \vec{p}$$

EAGER generates patterns without commented out constructors, to indicate to the pattern matching compiler that it may inspect these tags.

For each constructor $c_i : \forall \vec{x} : \vec{X}_i. D \vec{s}_i$, 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 26).

For a family D with i constructors of the form

$$c_i : \forall \vec{x} : \vec{X}_i. D \vec{s}_i$$

Where for each i , EPATS $(\emptyset, \vec{s}_i) \implies (V_i, \vec{p}_i)$

if $\forall i \neq j. \exists k. \text{DISJOINT}(p_{ik}, p_{jk})$ then D is **concretely detaggable**

Figure 26. Concretely detaggable families

The pattern sets are mutually exclusive if the following property holds:

$$\forall i \neq j. \exists k. \text{DISJOINT}(p_{ik}, p_{jk}) \implies \text{true}$$

$$\begin{aligned} \llbracket c_i \rrbracket &\Longrightarrow \lambda \vec{x}. \{c_i\} \vec{x}^{\{V\}} \\ \text{D-Elim } \vec{p}_i (\{c_i\} \vec{x}^{\{V\}}) P \vec{m} &\rightsquigarrow m_c \vec{a} \vec{y} (\text{D-Elim } \vec{r}_1 y_1 P \vec{m}) \dots (\text{D-Elim } \vec{r}_j y_j P \vec{m}) \end{aligned}$$

Figure 27. The detagging optimisation

B.3 The Collapsing Optimisation

We say a family is **concretely collapsible** (figure 28) if it is concretely detaggable and for each constructor $c : \forall \vec{a} : \vec{A}. \forall d_1 : D \vec{r}_1. \dots \forall d_j : D \vec{r}_j. \vec{s}$, 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

$$c_i : \forall \vec{a} : \vec{A}_i. \forall d_1 : D \vec{r}_{i1}. \dots \forall d_j : D \vec{r}_{ij}. D \vec{s}$$

If for each i , EPATS $(\emptyset, \vec{s}) \Longrightarrow (\vec{a}, \vec{p})$

then D is **concretely collapsible**

Figure 28. Concretely collapsible families

The general case for the collapsing optimisation is given in figure 29. 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, $\langle \rangle$.

$$\begin{aligned} \text{D-Elim } \vec{p} \{c \vec{a} \vec{y}\} P \vec{m} &\rightsquigarrow m_c \vec{a} (\{y_1\}) \dots (\{y_n\}) \\ &\quad (\text{D-Elim } \vec{r}_1 \{y_1\} P \vec{m}) \dots (\text{D-Elim } \vec{r}_n \{y_n\} P \vec{m}) \\ \llbracket c \rrbracket &\Longrightarrow \lambda \vec{a}; \vec{y}. \{c \vec{a} \vec{y}\} \\ \llbracket \text{D-Elim} \rrbracket &\Longrightarrow \lambda \vec{v}; x; P; \vec{m}. \text{D-Elim } \vec{v} \{x\} P \vec{m} \end{aligned}$$

Figure 29. The collapsing optimisation

C. Compilation of lookup

After applying the forcing optimisation, the elaborated **lookup** translates to RunTT as follows:

- **dMotive** computes a type, which cannot be analysed at runtime so compilation is straightforward.
- **discriminate** computes a value of type `False`, so can never be executed and compiles to `Impossible`. This function is inlined throughout.
- **fzCase**, after inlining and `case` collapsing, produces the following RunTT code:

$$\text{fzCase} \mapsto \lambda A; n; k; v; p. v!0$$

The empty vector case includes an application of **False-Elim** and so is removed.

- **fsCase**, after inlining and `case` collapsing, produces the following RunTT code:

$$\text{fsCase} \mapsto \lambda A; n; k; i; i'; v; ih; p. \text{lookup } A \ n \ i \ v$$

Again, the empty vector case is removed as it can never be executed.

- The top level **lookup** function initially compiles to:

$$\begin{aligned} \text{lookup} &\mapsto \lambda A; n; i; v. \\ \text{case } i \text{ of} & \\ \text{f0} \langle \rangle &\rightsquigarrow \text{fzCase } A \ n \ (\text{s } n) \ v \ \langle \rangle \\ \text{fs} \langle j \rangle &\rightsquigarrow \text{fsCase } A \ n \ (\text{s } n) \ i' \ (\text{fs} \langle i' \rangle) \ (v!1) \ \langle \rangle \end{aligned}$$

Clearly, it is beneficial to inline **fzCase** and **fsCase**, which yields the following definition:

$$\begin{aligned} \text{lookup} &\mapsto \lambda A; n; i; v. \\ \text{case } i \text{ of} & \\ \text{f0} \langle \rangle &\rightsquigarrow v!0 \\ \text{fs} \langle i' \rangle &\rightsquigarrow \text{lookup } A \ n \ i' \ (v!1) \end{aligned}$$