

Phase Distinctions In The Compilation Of EPIGRAM

James McKinna and Edwin Brady

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

Abstract. It is commonly believed that in dependently typed programming languages, the blurring of the distinction between types and values means that no type erasure is possible at run-time. In this paper, however, we propose an alternative phase distinction. Rather than distinguishing types and values in the compilation of EPIGRAM, we distinguish compile-time and run-time evaluation, and show by a series of program transformations that values which are not required at run-time can be erased.

1 Introduction

A distinctive feature of the programming language EPIGRAM [1, 2] is that it was designed with what we dub **full spectrum** type dependency in mind: values may depend on values, types on values, and types on types (and even values on types, via universes). This is in contrast with some other recent proposals for new programming languages, or extensions to the Hindley-Milner type system, which incorporate various restricted notions of dependent type. 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 [3] or in Sheard’s Ω mega [4].

It has been commonly believed since Cardelli’s early paper [5], continuing to the present day [6], that such 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; consequently, it is believed that there should be a limitation on the allowable forms of dependency. The main technical contribution 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. In this paper, we identify such values, and show how to erase them.

Other approaches to introducing dependent types into programming languages, such as DML [7], GADTs [8] and Ω mega restrict the form of dependent types they allow; this is in part in order to preserve a clear phase distinction between types and values. Where the same piece of data must occur in both phases, it must be duplicated by hand at the type level. The justification for such programmer inserted duplications is the need to maintain an erasure semantics; in this paper, we show that such explicit duplication is both unnecessary for the programmer and removable by the compiler.

1.1 Programming in EPIGRAM

EPIGRAM is based on a strongly normalising core type theory $\mathbb{T}\mathbb{T}$ with **inductive families** [9], similar to Luo’s UTT [10], together with a sophisticated type-directed elaborator from source programs to the type theory, affording the programmer a high-level concrete syntax considerably more terse than the type theory itself [1]. Datatype families and their constructors are declared using a natural deduction style notation; functions are defined by supplying a type signature (also in nd-style) and a body, specifying steps of data decomposition (notation: $lhs \Leftarrow e$) and definition (notation: $lhs \mapsto e$).

Inductive families are simultaneously defined collections of algebraic data types which can be indexed over values as well as types. We start with the type \mathbb{N} of natural numbers, in usual Peano-style:

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

The distinguishing feature of EPIGRAM’s approach to full spectrum type dependency is that this is the “one and only” type of natural numbers, for indexing *and* for computation; no separation is made in terms of the data declaration between these various uses; addition on \mathbb{N} is the addition *function*, not a kind-level inductive relation simulating that function, *etc.*

Now we declare a “lists with length” (vector) type, the family Vect indexed over \mathbb{N} :

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

Note that the constructor ϵ only yields vectors of length zero, while $x::xs$ yields vectors of length greater than zero; again, these finer type distinctions are directly specified in terms of expressions over the indexing type \mathbb{N} , rather than via an explicit equality constraint. Such equalities do arise, however, in elaboration of decomposition steps.

One advantage of indexing vectors over their length is that any type correct function written over a vector automatically satisfies any length invariants imposed 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, again by directly specifying the type of “bounds safe indices for lookup”: this is the family of finite sets:

$$\text{data } \frac{n : \mathbb{N}}{\text{Fin } n : \star} \quad \text{where } \overline{f0 : \text{Fin } (s\ n)} \quad \frac{i : \text{Fin } n}{fs\ i : \text{Fin } (s\ n)}$$

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: $0 < s\ n$ for any n , while if $j < n$, then certainly $fs\ j < s\ n$. 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 $\text{Fin}(s\ n)$ contains one more element than $\text{Fin } n$, and thus that $\text{Fin } n$ is indeed a type containing exactly n distinct values.

We can now write a bounds-safe **lookup** function, by recursion on the index, and then case analysis of the vector:

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

$$\text{lookup } i \ xs \Leftarrow \text{elim } i \Leftarrow \text{case } xs$$

$$\text{lookup } f0 \ (y :: ys) \mapsto y$$

$$\text{lookup } (\text{fs } j) \ (y :: ys) \mapsto \text{lookup } j \ ys$$

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. Moreover, the possible cases for i ensure that the vector in each case is non-nil, so that we always have either a head element to return, or a tail on which to recurse. These invariants are checked at compile-time by the typechecker rather than at run time by the run-time system; that the body of `lookup` possesses the type proposed is an intrinsic *proof* of these safety properties.

Having gone to the trouble of securing such strong safety guarantees, we would like to generate code from such high-level source programs which neither violates the guarantees (an obvious soundness requirement), nor requires them to be rechecked (an obviously desirable payoff for the programmer’s efforts). The fly in the ointment is the passage to code via elaborated TT terms (for `lookup`, the gory details are in AppendixA); the challenge in compiling EPIGRAM, or indeed any language with full spectrum type dependency, is to minimise any additional computational overhead arising from such elaboration.

1.2 Overview

The rest of this paper is structured as follows; Sect. 2 introduces the core type theory, TT, to which EPIGRAM elaborates. In this section we describe the problem of identifying the phase distinction, and explain how run-time erasure is more than simply removing types. There are two aspects to run-time erasure: in Sect. 3 we describe how to erase compile-time only values which are present in data structures, and in Sect. 4 we describe how to erase compile-time only values from function definitions, using the above `lookup` function as an example. Finally, Sect. 5 relates our work to other dependently typed programming languages and tools, and Sect. 6 gives some conclusions. In [6] it is suggested 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.

Acknowledgements

This paper is based on results in the second author’s PhD thesis, of which Lennart Augustsson proved a considerate but exacting examiner. It owes much to our long-standing collaboration with Conor McBride, for which all our thanks. We gratefully acknowledge the support of EPSRC through grant number GR/R72259/01.

2 EPIGRAM and its Elaboration

The high-level notation of EPIGRAM, illustrated above, is translated to a core dependently typed λ -calculus, $\mathbb{T}\mathbb{T}$, by a series of **elaboration** rules, presented in detail in [1]. For further details of $\mathbb{T}\mathbb{T}$, and its typechecking rules, see also [11]. This core type theory is similar to Luo's UTT [10] with definitions; there is an infinite hierarchy of predicative universes, $\star_i : \star_{i+1}$, although in what follows we simply work as if $\star : \star$, as universe levels can be inferred by machine, as in [12]. The syntax of $\mathbb{T}\mathbb{T}$ is as follows:

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

Elaboration inserts all of the implicit arguments and proofs of equalities necessary for typechecking made explicit. For example, elaboration of `Vect` leads to the introduction of the following constants into the context:

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

2.1 Elimination operators

When we declare an inductive family \mathbb{D} such as `Vect` above, we give the constructors which explain how to build objects in that family. Elaboration of \mathbb{D} generates an **elimination operator** (whose type we call the **elimination rule**) and corresponding reductions, which we call ι -**schemes** (given in pattern matching form; such definitions, indicated by \rightsquigarrow arrows, are distinct from EPIGRAM definitions, indicated by \mapsto). This machinery, well documented in particular by [9, 10, 13], describes and implements the allowed reduction and recursion behaviour of terms in the family. For `Vect`, it yields:

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

The arguments to the elimination operator are the **indices** (A and n here), the **target** (the object being eliminated; v here), the **motive** (a type-valued function which com-

putes the return type of the elimination; P here) and the **methods** (which describe how to achieve the motive for each constructor form).

As well as the basic elimination operator **D-Elim**, elaboration yields an operator, **D-Case**, for case analysis on D without recursion. Although definable in terms of **D-Elim**, it is more efficient to implement **D-Case** reductions directly. Other elimination operators are canonically associated with an inductive family D , but are not germane to this paper; we simply refer the interested reader for further details to [1] and [14].

2.2 Labelled Types

We note in particular the presence in $\mathbb{T}\mathbb{T}$ of **labelled types**, introduced in [1]. Labelled types are an extension to the core type theory which allow terms to be “labelled” by another term which describes its meaning. The typing and contraction rules for labelled types are as follows:

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

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

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

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

EPIGRAM supports defining programs interactively, with metavariables standing for parts of programs which have not yet been written, annotated with 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:

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

We can read $\langle \mathbf{lookup} \ A \ n \ i \ v : A \rangle$ as “**lookup** $A \ n \ i \ v$ with type A is computable”.

The elaboration of **lookup** to $\mathbb{T}\mathbb{T}$, shown in Appendix A, is given in terms of the elimination operator **Fin-Elim**. It contains proofs of equational constraints required for it to typecheck, and subexpressions which correspond to computing recursive calls on **lookup** in the $(:: A \ k \ x \ x \ s)$ case; the use of labelled types allows the types of such expressions to be correlated with the corresponding recursive call **lookup** $A \ k \ j \ x \ s$. 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.

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 not expect to have to execute them.

2.3 The supercombinator language RunTT

Compilation of TT to abstract machine code consists of two high level steps; first we translate to an intermediate representation RunTT (by lambda lifting, and by pattern matching compilation of elimination operators), then from RunTT to abstract machine code. RunTT is a language of supercombinators, i.e. higher order functions with no free variables. The syntax of RunTT is as follows:

$s ::= \lambda \vec{a} : \vec{e}. e$	(supercombinator)		
$e ::= \star_i$	(type of types)		
x	(bound variable)	f	(global name)
$D(\vec{e})$	(type constructor)	$c\langle \vec{e} \rangle$	(constructor)
$e e$	(function application)	$\forall x : e. e$	(function space)
$\text{let } a : e \mapsto e \text{ in } e$	(let binding)	$e!i$	(argument projection)
$\langle \vec{e} \rangle$	(untagged constructor)	<u>Impossible</u>	(unreachable code)
$\text{case } e \text{ of } \vec{alt}$	(case expression)		
$alt ::= c \langle \vec{x} \rangle \rightsquigarrow e$	(case alternative)		

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 (i.e. they are in supercombinator form).
- All constructor applications (including type constructors) are fully applied.
- There is a case construct — in TT case analysis is performed by elimination operators; execution of elimination operators in RunTT is by this case construct, which arise by compilation of the ι -schemes. We use a built-in argument projection operator ($e!i$) to retrieve arguments from constructors.
- For later optimisation purposes, we allow the marking of unreachable code (Impossible) and untagged constructors.

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.

Each supercombinator is compiled to a code sequence which, when executed, builds the supercombinator body. We have used the G-machine as a target language, and have found the presence of dependent types gives no additional difficulty. The compilation is presented in detail in [15].

2.4 The Phase Distinction

In traditional languages, there is a clear separation between types and values; types exist on the right hand side of the colon in the typing judgment, values on the left hand side. We compile the values, and ignore the types, since they serve no computational purpose. We can, to some extent, do the same with RunTT — at run-time we are interested in normalising the term to the left of the colon. However, this erasure is not enough. We observe, from the definitions of TT and RunTT, that types and values may be present either side of the colon. In particular, values which exist only for typechecking appear

to the left of the colon *as well as* to the right. Our aim is to erase such compile-time only values.

For example, the elaboration of **lookup** gives a large and complex term — the term includes proofs of equational constraints which justify the EPIGRAM definition. Such justification is compile-time only and need not be present at run-time. By identifying where the phase distinction lies and erasing the compile-time only values, we arrive at a compiled program which accurately reflects the programmer’s original definition.

The phase distinction, therefore, is not between values and types, but between compile-time and run-time values. In the following sections, we show how to identify and apply this phase distinction in EPIGRAM. There are two aspects to this; firstly, identifying compile-time only values which are present as arguments to data structures, and secondly, identifying compile-time only values which are present as arguments to functions.

3 Erasure From Data Structures

The elimination operator **D-Elim** is the primitive means TT provides for inspecting data in the inductive family D; elaboration of EPIGRAM compiles all pattern matching to elimination rules, providing an abstract interface for case analysis on the family. This gives us some freedom in how to implement elimination rules; if we optimise **D-Elim**’s reduction behaviour, we optimise the programs which elaborate in terms of it. In this section, based on work presented in [16, 15], we describe optimisations which remove redundant (compile-time only) information from data structures.

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

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

We 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. An important property of elimination operators is that if the application is *well-typed*, such repeated values are guaranteed to be convertible; we can therefore learn the form of some arguments by examining other arguments.
2. The form of one argument can tell us something about other arguments. e.g. in the case of **Vect-Elim**, 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).

3.1 Presupposed Arguments in Pattern Matching

We annotate patterns to direct compilation, with parts of patterns which are *presupposed* to match marked by $[\cdot]$. Such markings are made by applying the above observations. 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. Three possible ways of making such annotations for **Vect-Elim** are:

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

The first is the **standard implementation**; execution proceeds by case analysis on the outermost constructor of the vector. This is the method used, for example, in the COQ implementation. The second and third are alternative implementations identified in [16]. In the second, we retrieve A and k from the indices, rather than the target. In the third, we additionally make the case distinction on the length index rather than the target itself. Implementations (2) and (3) suggest alternative representations of vectors: (2) suggests a representation of a list along with its length, (3) suggests a Cayenne style representation of length with iterated projection from a tuple [17].

3.2 ExTT

Annotating patterns 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. ExTT (introduced in [16, 15]) is an extension of TT which augments the syntax with **marked** terms and its operational semantics with corresponding marked patterns. For example, implementation (2) above is marked as follows:

$$\begin{aligned} &\text{Vect-Elim } A \ \{0\} \quad (\epsilon \ \{A\}) \quad P \ m_\epsilon \ m_{::} \rightsquigarrow m_\epsilon \\ &\text{Vect-Elim } A \ (\{s\} \ k) \ (:\!:\ \{A\} \ \{k\} \ a \ v) \ P \ m_\epsilon \ m_{::} \\ &\rightsquigarrow m_{::} \ k \ a \ v \ (\text{Vect-Elim } A \ k \ v \ P \ m_\epsilon \ m_{::}) \end{aligned}$$

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. In the lambda lifting translation to RunTT, marked terms are removed entirely.

3.3 Argument Erasure

The analysis of elimination operators in [16], and the translation into ExTT leads to three optimisations, each of which erases compile-time only information from data structures:

Forcing The forcing optimisation elides redundant constructor arguments from data structures. This arises from implementation (2) of **Vect-Elim** in section 3.1. These are arguments which exist for typechecking only; we call these arguments **forceable**.

Detagging The detagging optimisation elides redundant constructor tags. This arises from implementation (3) of Vect-**Elim** in section 3.1. We need not store the constructor tag if it can be determined by analysing other arguments to the elimination rule; we call such data structures **detaggable**.

Collapsing The collapsing optimisation removes data structures entirely. If a data structure is detaggable and all non-recursive arguments are forceable, then no part of the data structure will ever be examined at run-time, so we need not retain it. We call such data structures **collapsible**. In practice, we only detag structures if it leads to collapsing.

It is the phase distinction between compile-time only and run-time values which directs whether or not arguments or data structures are erased.

3.4 Examples

The following examples show some constructor applications translated into RunTT, where $\llbracket \cdot \rrbracket$ is the meta-operation which translates from TT to RunTT (we omit the intermediate marked-up ExTT terms).

The erasure of Vect depends on whether we choose to detag the constructors or not. If we do choose to detag, we get the following translation into RunTT, where a vector is represented either as the empty tuple, or a pair of the head and tail:

$$\begin{aligned} \llbracket \epsilon A \rrbracket &\Longrightarrow \langle \rangle \\ \llbracket :: A k x xs \rrbracket &\Longrightarrow \langle x, xs \rangle \end{aligned}$$

Erasure of arguments from Fin’s constructors leads to a representation with the same “shape” as \mathbb{N} (i.e. a zero constructor with no arguments and a successor constructor with one recursive argument). This should not be surprising, as we have used Fin to represent bounded numbers.

$$\begin{aligned} \llbracket f0 n \rrbracket &\Longrightarrow f0 \langle \rangle \\ \llbracket fs n j \rrbracket &\Longrightarrow fs \langle j \rangle \end{aligned}$$

We can take advantage of this similarity of shape; if we optimise the representation of \mathbb{N} , for example via GMP, we can clearly do the same with any structure of the same shape, such as Fin.

An important example of a collapsible family is the heterogenous equality relation, due to McBride [13] and built in to TT:

$$\frac{a : A \quad b : B}{a = b : \star} \quad \frac{A : \star \quad a : A}{\text{refl } a : a = a}$$

The ι -schemes which implement the elimination operator for this family are marked as follows; note that all of the necessary information for reduction can be obtained from the indices, so there is no need to keep the data structure at run-time:

$$= \mathbf{-elim} \ A \ a \ \{a\} \ \{\text{refl } A \ a\} \ P \ m_{\text{refl}} \ \rightsquigarrow \ m_{\text{refl}} \ A \ a$$

This is an important example because it is by reasoning with this relation that the elaboration can remove impossible case analyses, as in **lookup**. Space prevents us giving further detailed examples; a larger variety of inductive families and their optimisation are presented in detail in [15].

4 Erasure From Function Application

Having removed the redundant and duplicated (i.e. compile-time only) information from data structures, we are in a position to apply transformations to the resulting RunTT terms. We have chosen these transformations with one specific goal in mind: to remove compile-time only values from run-time terms. The correctness property that these transformations satisfy is that if a term t reduces to $c\langle\vec{x}\rangle$ in a naïve (i.e., no erasure) compilation setting, then t' reduces to $c\langle\vec{y}\rangle$ in an optimised setting, where t' is the optimised form of t . i.e., with or without erasure, evaluation produces the same constructor at the head.

4.1 Unfolding Elimination Operators

The purpose of the elimination operator **D-Elim** 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. In Sect. 2.2 we introduced labelled types; the labelling gives us the *meaning* of each inductive hypothesis. e.g. the term $\text{call } \langle \text{lookup } A \ k \ i \ v \rangle \ ih$ expresses that ih represents a call of $\text{lookup } A \ k \ i \ v$. At run-time, there is no need for this 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:

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

The effect of this in ExTT is that **D-Elim**'s inductive hypotheses are no longer used. Since **D-Case** is just **D-Elim** without the inductive hypotheses, we can replace **D-Elim** with **D-Case**. Furthermore, the typing rules for labelled types suggest that we can remove labels entirely:

$$\begin{aligned} \llbracket \langle l : T \rangle \rrbracket &\implies T \\ \llbracket \text{call } \langle l \rangle t \rrbracket &\implies t \\ \llbracket \text{return } t \rrbracket &\implies t \end{aligned}$$

The TT terms remain type correct (by the typing rules), and constructor headed terms are unchanged. In itself, this does not have a great effect, but when combined with inlining the effect is amplified, as we shall see.

4.2 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; [18] 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 — an argument to the elimination operator which is necessary at compile-time to compute the type of an elimination, but removable at run-time.

4.3 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 arguments are programmer inserted. With EPIGRAM's implicit arguments, arguments may be inserted into elaborated terms for type safety. 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.

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 other cases, we retain a wrapper function with no arguments removed.

4.4 Unreachable Code

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, `False`, which is declared as follows:

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

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.

Furthermore, we can be sure that *any* function taking an argument of type `False`, or which returns a value of type `False`, will never be executed. Any function which takes an argument of type `False` or returns `False` has its body replaced with the constant `Impossible`. Such functions can obviously be inlined.

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

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

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

$$\mathbf{vTail} \ (y :: ys) \mapsto ys$$

After application of forcing (removing the indices of `Vect`), compilation to `RunTT` and the marking of impossible to execute terms, we are left with:

$$\mathbf{vTail} \mapsto \lambda A; n; xs. \underline{\text{case } xs \text{ of}}$$

$$\epsilon \langle \rangle \rightsquigarrow \text{Impossible}$$

$$:: \langle y, ys \rangle \rightsquigarrow (xs!1)$$

The constant Impossible arises because the elaborator has constructed a proof that the case branch can *never* be executed; there is *no need* for a run-time check, since we know this statically. Since there is only one possible case remaining, we can collapse the case expression entirely, and compile **vTail** to a function which simply returns a pointer to the tail, as we would expect from the original definition.

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

Note the use of argument projection to retrieve the tail of the vector, rather than using the name bound by the case expression. Using argument projection is important to allow the case collapsing optimisation to work.

The type of **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.

4.5 lookup revisited

After erasure from the Vect and Fin data structures, the elaborated **lookup** (from section 2) translates to RunTT as follows:

- **dMotive** computes a type, which cannot be analysed at run-time 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** and **fsCase**, after inlining and case collapsing, produce the following RunTT code:

$$\begin{aligned} \mathbf{fzCase} &\mapsto \lambda A; n; k; xs; p. xs!0 \\ \mathbf{fsCase} &\mapsto \lambda A; n; k; i; i'; xs; ih; p. \mathbf{lookup} A n i xs \end{aligned}$$

In each case, the empty vector case has been collapsed as it includes an application of **False-Elim** and so can never be executed.

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

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

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

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

Finally, we observe that A and n are passive arguments, hence can be delegated to a wrapper. `lookup` is defined in full as follows:

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

5 Related Work

For EPIGRAM, we have chosen a point in the design space where types and values are indistinguishable. Other recent experiments with forms of dependent types, such as Generalised Algebraic Data Types [8], Ω mega [6] and Applied Type System [19], prefer to maintain the separation between types and values. This is partly to retain an erasure semantics, and partly to allow programs to retain a traditional Haskell-style look and feel. In contrast, EPIGRAM offers the power of full spectrum type dependency and we are experimenting with the expressive style of programming this affords us [20, 21].

There have been several other experiments in introducing dependent types into practical programming languages. DML [7] 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 [22] and array bounds check elimination [23]. Again, the phase distinction is maintained in DML by choosing a restrictive expression language at the type level. Cayenne [17] 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.

There are other related approaches to run-time erasure in settings with full spectrum dependent types. The extraction mechanism of the theorem prover COQ [24, 25] relies on a distinction between a universe of sets and propositions, removing propositions from extracted code. We can view this too as a phase distinction, not between types and values but between computational values and computationally irrelevant values. 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 [26] introduce an approach to erasure of such implicit arguments using monads.

The focus of the paper has been twofold: uncovering the phase distinction and optimising run-time execution. Of course, since we must also do computation at compile-time in order to typecheck EPIGRAM terms, we should consider how to improve evaluation at compile-time. The current implementation uses normalisation by evaluation [27], but we can also consider Grégoire and Leroy's technique [28], as applied in COQ. The forcing and detagging optimisations described in Sect. 3 are also applicable at compile time and can be applied *before* typechecking — the marked values are marked because they are duplicated, and hence have already been typechecked. [15] describes this optimisation further, including a proof that typechecking in this setting is sound and complete.

6 Conclusions

In this paper we have seen how to establish a phase distinction between compile-time only and run-time values in a full spectrum dependently typed programming language, EPIGRAM. The style of programming encouraged by EPIGRAM involves extensive use of indices on inductive families to maintain invariant properties of programs which makes the identification of the phase distinction all the more important, as a significant number of compile-time only values can be introduced into programs. In the course of developing an implementation of the core language, TT, we have found:

- EPIGRAM is executable on a stock architecture for execution of lazy functional languages with only minor modifications.
- 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.
- Datatypes are accessed only through an abstract interface given by the type of the eliminator and its associated equational theory given by ι -schemes. We can therefore choose any representation of the datatype for which this interface can be implemented. In future work this may 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 with constant time **lookup**.

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.

References

1. McBride, C., McKinna, J.: The view from the left. *Journal of Functional Programming* **14** (2004) 69–111
2. McBride, C.: Epigram: Practical programming with dependent types. *Lecture Notes, International Summer School on Advanced Functional Programming* (2004)
3. McBride, C.: Faking it – simulating dependent types in Haskell. *Journal of Functional Programming* **12** (2002) 375–392

4. Sheard, T.: Languages of the future. In: OOPSLA '04. (2004)
5. Cardelli, L.: Phase distinctions in type theory. Manuscript (1988)
6. Sheard, T., Hook, J., Linger, N.: GADTs + extensible kinds = dependent programming (2005)
7. Xi, H.: Dependent Types in Practical Programming. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University (1998)
8. Peyton Jones, S., Washburn, G., Weirich, S.: Wobbly types: type inference for generalised algebraic data types (2004) Submitted to POPL 2005.
9. Dybjer, P.: Inductive families. *Formal Aspects Of Computing* **6** (1994) 440–465
10. Luo, Z.: *Computation and Reasoning – A Type Theory for Computer Science*. International Series of Monographs on Computer Science. OUP (1994)
11. Chapman, J., Altenkirch, T., McBride, C.: EPIGRAM reloaded: A standalone typechecker for ETT. Paper presented at TFP'05, Tallinn, Estonia. (2005)
12. Harper, R., Pollack, R.: Type checking with universes. *Theoretical Computer Science* **89** (1991) 107–136
13. McBride, C.: *Dependently Typed Functional Programs and their proofs*. PhD thesis, University of Edinburgh (2000)
14. Goguen, H., McBride, C., McKinna, J.: Eliminating dependent pattern matching (2005) Draft.
15. Brady, E.: *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham (2005)
16. Brady, E., McBride, C., McKinna, J.: Inductive families need not store their indices. In Berardi, S., Coppo, M., Damiani, F., eds.: *Types for Proofs and Programs 2003*. Volume 3085., Springer (2004) 115–129
17. Augustsson, L.: Cayenne - a language with dependent types. In: *International Conference on Functional Programming*. (1998) 239–250
18. Peyton Jones, S., Marlow, S.: Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming* **12** (2002) 393–434
19. Xi, H.: Applied type system. In Berardi, S., Coppo, M., Damiani, F., eds.: *Types for Proofs and Programs 2003*. Volume 3085., Springer (2004) 394–408
20. Altenkirch, T., McBride, C., McKinna, J.: Why dependent types matter (2005) Draft.
21. Morris, P., McBride, C., Altenkirch, T.: Exploring the regular tree types. In: *Types for Proofs and Programs 2004*. (2005)
22. Xi, H.: Dead code elimination through dependent types. In: *The First International Workshop on Practical Aspects of Declarative Languages*, San Antonio (1999) 228–242
23. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal (1998) 249–257
24. Paulin-Mohring, C.: *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Paris 7 (1989)
25. Letouzey, P.: A new extraction for Coq. In Geuvers, H., Wiedijk, F., eds.: *Types for proofs and programs*. LNCS, Springer (2002)
26. Letouzey, P., Spitters, B.: Implicit and non-computational arguments using monads (2005) Draft.
27. Berger, U., Schwichtenberg, H.: An inverse of the evaluation functional for typed λ -calculus. In Vemuri, R., ed.: *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press (1991) 203–211
28. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: *International Conference on Functional Programming*. (2002) 235–246

A Elaboration of lookup

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