Inductive Families Need Not Store Their Indices

Edwin Brady[1], Conor McBride[1] and James McKinna[2]

[1] Department of Computer Science, University of Durham
[2] School of Computer Science, University of St. Andrews

Abstract. We consider the problem of efficient representation of dependently typed data. In particular, we consider a language TT based on Dybjer's notion of **inductive families** [11] and reanalyse their general form with a view to optimising the storage associated with their use. We introduce an execution language, ExTT, which allows the commenting out of computationally irrelevant subterms and show how to use properties of elimination rules to elide constructor arguments and tags in ExTT. We further show how some types can be collapsed entirely at run-time. Several examples are given, including a representation of the simply typed λ-calculus for which our analysis yields an 80% reduction in run-time storage requirements.

1 Introduction

Dependent type theory provides programmers with more than an integrated logic for reasoning about program correctness. It allows more precise types for programs and data in the first place, strengthening the typechecker's language of guarantees. We have richer function types $\forall x : S.\ T$ which adapt their return types to each argument; we also have richer data structures which do not just contain but *explain* data, exposing and enforcing their properties.

Moreover, we may reasonably expect more static detail about programs and data to yield better optimised dynamic behaviour. We need neither test what is guaranteed nor store what is determined by typechecking. Pollack's implicit syntax [24] already supports the omission of much redundant information from concrete syntax for similar reasons.

This paper idenitifies some space optimisations which significantly reduce the storage overheads associated with **inductive families** in the sense of [11]. These are data-indexed collections of mutually recursive datatypes, $\mathsf{D}\,\vec{x}$, available in systems such as those underlying Lego [16], Coq [9], ALF [18] and also the language we use here — Epigram [20]. A common example for illustrative purposes is Vect, the family of list types indexed by element type and length:

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

1.1 Programming with Inductive Families

Function types over inductive families can use specific indices to require and ensure properties of inputs and outputs — e.g., compatibility of length:

$$\underline{\text{let}}\ \frac{u, v\ :\ \mathsf{Vect}\,\mathbb{N}\,n}{\mathbf{vAdd}\ u\ v\ :\ \mathsf{Vect}\,\mathbb{N}\,n} \quad \begin{array}{llll} \mathbf{vAdd} & \varepsilon & \varepsilon & \mapsto \varepsilon \\ \mathbf{vAdd} & (x :: u) & (y :: v) & \mapsto (x + y) :: \mathbf{vAdd}\ u\ v \end{array}$$

The precise type prevents some bogus choices of output — we can only return ε on the first line, only a :: on the second. The input possibilities become narrower too — adding $(x :: u)$ to ε, or vice versa, is not even an issue.

By the same token, the potential for optimisation is clear. Once we know whether the first argument is ε or $(x :: u)$, we can *presuppose* the form of the second argument — we can ignore it in the ε case; in the :: case, we can safely project out y and v without checking the constructor tag. Moreover, if we inspect n, implicitly passed to **vAdd**, we need never check **Vect** constructor tags at all.

We impose invariants on inductive families to improve reliability, but this paper seeks to exploit them for performance. Such optimisations are not available in conventional functional languages — there is no way that inspecting one argument can justify presuppositions about another. If we want to write vector addition using ordinary lists, we must not only consider how to handle length mismatch in our code, we must also effectively test for it at run-time.

1.2 Underlying Type Theory

Following [20], EPIGRAM programs elaborate to well typed terms in a type theory TT, based on Luo's UTT [15] with inductive families [11] and equality as in [19]. Here is its syntax:

$$\begin{array}{lll|ll} t ::= & \star_i & \text{(type of types)} & x & \text{(variable)} \\ \mid & \forall x\!:\!t.\ t & \text{(function space)} & \mathsf{D} & \text{(inductive family)} \\ \mid & \lambda x\!:\!t.\ t & \text{(abstraction)} & \mathsf{c} & \text{(constructor)} \\ \mid & t\ t & \text{(application)} & \mathsf{D}\text{-}\mathbf{E} & \text{(elimination operator)} \end{array}$$

As usual, we may abbreviate the function space $\forall x\!:\!S.\ T$ by $S \to T$ if x is not free in T. There is an infinite hierarchy of predicative universes, $\star_i : \star_{i+1}$. We leave universe levels to the machine, as in [12].

Computation is by **β-reduction** for λ-abstractions and **ι-reduction** for elimination operators. A <u>data</u> declaration typically elaborates to declarations of a family $\mathsf{D} : \forall\vec{i}\!:\!\vec{I}.\ \star$, constructors c, and an elimination operator $\mathsf{D}\text{-}\mathbf{E}$ equipped with ι-schemes. We write $s \mapsto t$ if s β- or ι-reduces to t. In the usual way, every well typed TT term t computes to a **weak head-normal form** $\textsc{whnf}(t)$.

A typical constructor has a type like this:[1]

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

[1] To ease presentation, we keep the non-recursive arguments $\vec{a}$ to the front and permit only first-order recursive arguments — neither restriction is crucial to this work.

For example, elaborating our Vect example, we acquire $\mathsf{Vect} : \forall A{:}\star.\,\forall n{:}\mathbb{N}.\,\star$, and constructors:

$$\begin{aligned}&\varepsilon : \forall A{:}\star.\,\mathsf{Vect}\;A\;\mathsf{0}\\ &:: : \forall A{:}\star.\,\forall k{:}\mathbb{N}.\,\forall a{:}A.\,\forall v{:}\mathsf{Vect}\;A\;k.\,\mathsf{Vect}\;A\;(\mathsf{s}k)\end{aligned}$$

Note that the variables left schematic in the <u>data</u> declaration have become explicitly quantified arguments. In naïve implementations these take up space — every $\mathsf{Vect}\,A\,n$ stores the sequence $0, \ldots, n-1$, and n references to A. Even with perfect sharing, this is quite an overhead — the space implications for families with more complex invariants are quite drastic if this problem is left unchecked.

Basically, the elimination operator, $\mathsf{D}\text{-}\mathbf{E}$ has a type of this form:

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

The **target**, with given **indices**, explains what to eliminate; the **motive** explains what is to be achieved by the elimination; the **methods** explain how to achieve the motive for each canonical form the target can take, given appropriate inductive hypotheses. The associated ι-rules for definitional equality have this form:

$$\Gamma\vdash\mathsf{D}\text{-}\mathbf{E}\;\vec{s}\;(\mathsf{c}\;\vec{a}\;\vec{y})\;P\;\vec{m}\;=\;m_{\mathsf{c}}\;\vec{a}\;\vec{y}\;(\mathsf{D}\text{-}\mathbf{E}\;\vec{r}_1\;y_1\;P\;\vec{m})\;\ldots\;(\mathsf{D}\text{-}\mathbf{E}\;\vec{r}_j\;y_j\;P\;\vec{m})$$

When indices are used uniformly, such as the element type of Vect, we adapt the basic $\mathsf{D}\text{-}\mathbf{E}$ slightly, abstracting these parameters once for all. This yields:

$$\begin{array}{ll}\mathsf{Vect}\text{-}\mathbf{E} : & \forall A{:}\star.\,\forall n{:}\mathbb{N}.\,\forall v{:}\mathsf{Vect}\;A\;n.\\ & \forall P{:}\forall n{:}\mathbb{N}.\,\forall v{:}\mathsf{Vect}\;A\;k.\;\star.\\ & \forall m_\varepsilon{:}P\;\mathsf{0}\;(\varepsilon\;A).\\ & \forall m_{::}{:}\forall k{:}\mathbb{N}.\,\forall a{:}A.\,\forall v{:}\mathsf{Vect}\;A\;k.\,(P\;k\;v)\to P\;(\mathsf{s}k)\;(::\;A\;k\;a\;v).\\ & P\;n\;v\end{array}$$

$$\begin{array}{llllll}\mathsf{Vect}\text{-}\mathbf{E}\;A & \mathsf{0} & (\varepsilon\;A) & P\;m_\varepsilon\;m_{::} & = & m_\varepsilon\\ \mathsf{Vect}\text{-}\mathbf{E}\;A & (\mathsf{s}\;k) & (::\;A\;k\;a\;v) & P\;m_\varepsilon\;m_{::} & = & m_{::}\;k\;a\;v\;(\mathsf{Vect}\text{-}\mathbf{E}\;A\;k\;v\;P\;m_\varepsilon\;m_{::})\end{array}$$

Implementing $\mathsf{Vect}\text{-}\mathbf{E}$ appears to require non-linear matching — there are repeated arguments in both patterns, suggesting a run-time conversion check. In fact this is not needed — the repeated arguments coincide in any *well typed* application of $\mathsf{Vect}\text{-}\mathbf{E}$. We do not need to recheck the duplicate A or k in the patterns $(\varepsilon\;A)$ or $(::\;A\;k\;a\;v)$. So why store them?

In this paper we show how to streamline the implementation of ι-rules so that unnecessary testing is avoided. We introduce extensions to the TT syntax for marking parts of terms to be ignored or removed. So equipped, we consider which constructor arguments can be ignored, and then play a similar game with constructor tags. Finally, we show how to eliminate some structures entirely and make a larger example smaller — the simply typed λ-calculus.

1.3 Related Work

Correctness preserving program transformations [10, 23] provide a basis for many optimisations in simply typed functional languages. In this paper we use substitution transformations to mark unused terms for deletion in a similar manner to Berardi's pruning of simply typed λ-terms [4]. Program transformation techniques have also been applied to type theory; Magaud and Bertot [17] show an approach to changing data representation by transforming the constructors and elimination rule of a family and use this technique to change from unary natural numbers to a more efficient binary representation.

The Coq program extraction tool [22, 14] attempts to remove purely logical parts of proofs in order to produce executable programs. Our approach differs in that we do not separate the predicative and impredicative type universes and attempt to remove *all* terms which are unused.

Callaghan and Luo [7] use the well-typedness of elimination rules to avoid checking of repeated arguments, a technique which we apply and extend in this paper. Xi's DML [26] also uses dependent types for optimisation, eliminating dead code [27] and array bounds checking [28].

2 Implementing Reduction Rules for Datatypes

The elimination operator $\mathsf{D}\text{-}\mathbf{E}$ is the only means TT provides for inspecting data in the inductive family D. If we optimise $\mathsf{D}\text{-}\mathbf{E}$'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 $\mathsf{D}\text{-}\mathbf{E}$, then it is *never* needed at run-time. Let us look more closely at how ι-rules are implemented.

2.1 Pattern Syntax and its Semantics

We implement ι-rules $\mathsf{D}\text{-}\mathbf{E}\,\vec{t_i} = e_i$ by pattern matching, marking with $[\cdot]$ those parts of patterns p which *well typed* terms are *presupposed* to match. Unmarking these parts gives back a term, $|p|$.

$$\begin{array}{llll} p ::= x & \text{(pattern variable)} & \mid [t] & \text{(presupposed term)} \\ \quad\mid \mathsf{c}\,\vec{p} & \text{(constructor pattern)} & \mid [\mathsf{c}]\,\vec{p} & \text{(presupposed-constructor pattern)} \end{array}$$

For each ι-law, as above, we write a **ι-scheme**, $\mathsf{D}\text{-}\mathbf{E}\,\vec{p}_i \mapsto e_i$ with $|\vec{p}_i| = \vec{t_i}$ and e_i a term over $\vec{p}_i$'s **pattern variables**. The ι-schemes are then compiled into an efficient case-expression [2]. However, our pattern syntax will facilitate the discussion without delving into those details.

The partial function MATCH tries to compute a **matching substitution** for a pattern and term (MATCHES lifts MATCH to sequences in the obvious way):

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

The first two lines of MATCH test constructors and bind pattern variables as usual in implementations of pattern matching from [21] onwards. The remaining two lines, however, presuppose the successful outcome of testing. To justify these presuppositions, we shall require that each ι-scheme is **Γ-respectful** of well typed instances, i.e.

if $\Gamma \vdash \mathsf{D}\text{-}\mathbf{E}\,\vec{t} \,:\, T$ and $\textsc{matches}(\vec{p}_i, \vec{t}) \Longrightarrow \sigma$ then $\Gamma \vdash \mathsf{D}\text{-}\mathbf{E}\,\sigma|\vec{p}_i| = \mathsf{D}\text{-}\mathbf{E}\,\vec{t} \,:\, T$

A set of ι-schemes, $\mathsf{D}\text{-}\mathbf{E}\,\vec{p}_i \;\mapsto\; e_i$ is **Γ-well-defined** if, for any $\Gamma \vdash \mathsf{D}\text{-}\mathbf{E}\,\vec{t} : T$ of the right arity, with a constructor-headed target, we have $\textsc{matches}(\vec{p}_i, \vec{t}) \Longrightarrow \sigma$ for exactly one i. This yields ι-reduction $\mathsf{D}\text{-}\mathbf{E}\,\vec{t} \;\mapsto\; \sigma e_i$. A set of ι-schemes which is Γ-respectful and Γ-well-defined for all Γ is said to **implement** the corresponding ι-rules.

2.2 Standard Implementation

Theorem. For $\mathsf{D} : \forall \vec{i} : \vec{I}.\, \star$, with typical $\mathsf{c} \;:\; \forall \vec{a} : \vec{A}.\, \mathsf{D}\,\vec{r}_1 \to \ldots \to \mathsf{D}\,\vec{r}_j \to \mathsf{D}\,\vec{s}$, this typical ι-scheme implements the ι-rules (the **standard** implementation):

$$\mathsf{D}\text{-}\mathbf{E}\,[\vec{s}]\;(\mathsf{c}\;\vec{a}\;\vec{y})\;P\;\vec{m} \;\mapsto\; m_{\mathsf{c}}\;\vec{a}\;\vec{y}\;(\mathsf{D}\text{-}\mathbf{E}\,\vec{r}_1\;y_1\;P\;\vec{m})\;\ldots\;(\mathsf{D}\text{-}\mathbf{E}\,\vec{r}_j\;y_j\;P\;\vec{m})$$

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

$$\textsc{matches}([\vec{s}]\;(\mathsf{c}\;\vec{a}\;\vec{y})\;P\;\vec{m}, \vec{s}'\;(\mathsf{c}\;\vec{a}'\;\vec{y}')\;P'\;\vec{m}') \Longrightarrow \sigma$$

but matching the other ι-schemes fails, so these schemes are Γ-well-defined.

Moreover, σ is $\vec{a}'/\vec{a} \circ \vec{y}'/\vec{y} \circ P'/P \circ \vec{m}'/\vec{m}$. Typechecking, $\mathsf{c}\vec{a}'\vec{y}' \;:\; \mathsf{D}(\vec{a}'/\vec{a} \circ \vec{y}'/\vec{y})\vec{s} = \mathsf{D}\;\sigma\vec{s}$. Hence $\sigma\vec{s} = \vec{s}'$ as $\mathsf{D}\text{-}\mathbf{E}\,\vec{s}'\;(\mathsf{c}\;\vec{a}'\;\vec{y}')$ is well-typed. Hence our typical scheme is Γ-respectful. $\Box$

The standard implementation comments out the *indices* — just as well, because there is no guarantee that they generally take the constructor form which explicit matching requires. For example, $\mathsf{Vect}\text{-}\mathbf{E}$ has standard implementation

$$\begin{array}{llllll}\mathsf{Vect}\text{-}\mathbf{E}\,[A] & [0] & (\varepsilon\;A) & P\;m_\varepsilon\;m_{::} & \mapsto & m_\varepsilon \\ \mathsf{Vect}\text{-}\mathbf{E}\,[A] & [\mathsf{s}\;k] & (::\;A\;k\;a\;v) & P\;m_\varepsilon\;m_{::} & \mapsto & m_{::}\;k\;a\;v\;(\mathsf{Vect}\text{-}\mathbf{E}\,A\;k\;v\;P\;m_\varepsilon\;m_{::})\end{array}$$

2.3 Alternative Implementations

Where the indices of a constructor's return type do happen to resemble constructor or variable patterns, we are free to consider alternative implementations of the corresponding ι-schemes. We may certainly comment out a pattern variable from the target if we can recover it by matching an index. For example, this is also an implementation of $\mathsf{Vect}\text{-}\mathbf{E}$:

$$\begin{array}{llllll}\mathsf{Vect}\text{-}\mathbf{E}\,A & 0 & (\varepsilon\;[A]) & P\;m_\varepsilon\;m_{::} & \mapsto & m_\varepsilon \\ \mathsf{Vect}\text{-}\mathbf{E}\,A & (\mathsf{s}\;k) & (::\;[A]\;[k]\;a\;v) & P\;m_\varepsilon\;m_{::} & \mapsto & m_{::}\;k\;a\;v\;(\mathsf{Vect}\text{-}\mathbf{E}\,A\;k\;v\;P\;m_\varepsilon\;m_{::})\end{array}$$

But we can do better than that. There is no need to check the constructor tags on *both* the length and the target — one check will do. We may take either

$$\begin{array}{llllll}(\dagger)\ \mathsf{Vect}\text{-}\mathbf{E}\ A & [0] & (\varepsilon\ [A]) & P\ m_\varepsilon\ m_{::} & \mapsto & m_\varepsilon \\ \quad\ \mathsf{Vect}\text{-}\mathbf{E}\ A & ([\mathsf{s}]\ k) & (::\ [A]\ [k]\ a\ v) & P\ m_\varepsilon\ m_{::} & \mapsto & m_{::}\ k\ a\ v\ (\mathsf{Vect}\text{-}\mathbf{E}\ A\ k\ v\ P\ m_\varepsilon\ m_{::})\end{array}$$

or, instead, privileging index length over vector contents

$$\begin{array}{llllll}(\ddagger)\ \mathsf{Vect}\text{-}\mathbf{E}\ A & 0 & ([\varepsilon]\ [A]) & P\ m_\varepsilon\ m_{::} & \mapsto & m_\varepsilon \\ \quad\ \mathsf{Vect}\text{-}\mathbf{E}\ A & (\mathsf{s}\ k) & ([::]\ [A]\ [k]\ a\ v) & P\ m_\varepsilon\ m_{::} & \mapsto & m_{::}\ k\ a\ v\ (\mathsf{Vect}\text{-}\mathbf{E}\ A\ k\ v\ P\ m_\varepsilon\ m_{::})\end{array}$$

In the sequel, we show how to choose good alternative implementations for elimination operators by systematically exploiting the presence of constructor symbols in indices. This leads naturally to space optimisations, where we do not merely comment out unnecessary data from patterns — we delete them entirely from the representation of datatypes.

2.4 ExTT — an execution language for TT with deleted terms

We introduce ExTT, an execution language for terms in TT. ExTT extends TT's syntax with **deleted** terms and patterns $\{t\}$, and also with deleted constructor patterns $\{\mathsf{c}\}\ \vec{p}$ corresponding to untagged tuples $\{\mathsf{c}\}\ \vec{t}$. We extend the operational semantics thus:

$$\begin{array}{l}\textsc{match}(\{t\}, \{t'\}) \Longrightarrow \textsc{id} \\ \textsc{match}(\{\mathsf{c}\}\ \vec{p}, t) \Longrightarrow \textsc{matches}(\vec{p}, \vec{t})\ \ \underline{\text{if}}\ \textsc{whnf}(t) \Longrightarrow (\{\mathsf{c}'\}\ \vec{t})\end{array}$$

We are careful to distinguish $(\{\mathsf{c}\}\ \{\vec{t}\})$, which is represented by the empty tuple, from $\{\mathsf{c}\ \vec{t}\}$, which is deleted altogether. The actual evaluation of terms in ExTT can be by any standard method, such as normalisation by evaluation [1, 5], compilation to G machine code [13] or program extraction [14].

The unmarking operation $|\cdot|$ takes both patterns and terms in ExTT back to terms in TT by stripping out both $[\cdot]$ and $\{\cdot\}$ marks. Terms in ExTT arise only by optimisations from well typed TT terms hence ExTT needs no typing rules provided that these optimisations are safe.

We specify an **optimisation** by giving a substitution $[\![\cdot]\!]$ from TT identifiers to ExTT terms, $\textsc{id}$ by default, together with the optimised ExTT ι-schemes. For ι-rules $\Gamma \vdash \mathsf{D}\text{-}\mathbf{E}\ \vec{t_i} = e_i$, these have form $\mathsf{D}\text{-}\mathbf{E}\ \vec{p}_i \ \mapsto\ d_i$, where $|\vec{p}_i| = \vec{t}_i$, $|d_i| = e_i$ and every undeleted free variable in d_i is a pattern variable in $\vec{p}_i$. For all Γ, these schemes must be Γ-well-defined in the obvious way, and Γ-respectful in that

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

The rôle of τ is to instantiate the variables free in e_i, but deleted in d_i—these are not needed when executing ExTT terms, hence they need not be matched.

In the following sections, we establish several such optimisations.

3 Eliding Redundant Constructor Arguments

Recall the alternative implementation of $\mathsf{Vect}\text{-}\mathbf{E}$ ($\dagger$ above) which matches A and k in the indices rather than the target. When can we do this, in general?

Whenever $\mathsf{c}\ \vec{a},\ \mathsf{c}\ \vec{b}\ :\ \mathsf{D}\ \vec{s}$ implies $a_i = b_i$, we say that the ith argument of c is **forceable**. eg., the A argument to ε is forceable since if $\varepsilon\ a,\ \varepsilon\ b\ :\ \mathsf{Vect}\ A\ 0$ then clearly $a = b = A$. For ::, A and k are forceable in the same way.

Constructor arguments which have been commented out owing to their repetition in a ι-scheme are forceable. This is to be expected; such repeated arguments arise from the patterns describing constructor indices.

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

To express $\vec{s}$ as $|\vec{p}|$, we write a program to extract from a term a linear pattern with its variable set:

```
PAT ( $V$, $x$ ) $\Longrightarrow$ $(x \cup V, x)$  if $x \notin V$
PAT ( $V, \mathsf{c}\,\vec{t}$ ) $\Longrightarrow$ $(V', \text{LAZY}(\mathsf{c}, \vec{p}))$  if PATS $(V, \vec{t}) \Longrightarrow (V', \vec{p})$
PAT ( $V$, $t$ ) $\Longrightarrow$ $(V, [t])$
PATS( $V$, $\cdot$ ) $\Longrightarrow$ $(V, \cdot)$
PATS( $V, t\,\vec{t}$ ) $\Longrightarrow$ $(V'', p\,\vec{p})$
                    if PAT $(V, t) \Longrightarrow (V', p)$ and PATS $(V', \vec{t}) \Longrightarrow (V'', \vec{p})$
LAZY( $\mathsf{c}$, $[\vec{p}]$ ) $\Longrightarrow$ $[\mathsf{c}\,\vec{p}]$
LAZY( $\mathsf{c}$, $\vec{p}$ ) $\Longrightarrow$ $[\mathsf{c}]\,\vec{p}$  otherwise
```

For our typical constructor c, we can extract the patterns which D-**E** will match by PATS $(\emptyset, \vec{s}) \Longrightarrow (V, \vec{p})$. If an argument $a_i \in V$ then a_i is concretely forceable. It is instantiated by matching $\vec{p}$, hence we may presuppose it when we match the target, yielding the same result. Hence, we may then choose the alternative implementation:

$$\mathsf{D}\text{-}\mathbf{E}\ \vec{p}\ (\mathsf{c}\ \vec{a}^{[V]}\ \vec{y})\ P\ \vec{m}\ \mapsto\ m_{\mathsf{c}}\ \cdots \qquad \underline{\text{where}}\ a^{[V]} \Longrightarrow [a]\ \underline{\text{if}}\ a \in V; \quad a^{[V]} \Longrightarrow a\ \underline{\text{otherwise}}$$

Theorem. The following is an optimisation (**forcing**):

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

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

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

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

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

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

For our Vect example, forcing is given by:

$$\begin{array}{l} [\![\varepsilon]\!] \Longrightarrow \lambda A.\ \varepsilon\ \{A\} \\ [\![::]\!] \Longrightarrow \lambda A; k; a; v.\ ::\ \{A\}\ \{k\}\ a\ v \end{array}$$

$$\begin{array}{lllll} \mathsf{Vect}\text{-}\mathbf{E}\ A & [\mathsf{0}] & (\varepsilon\ \{A\}) & P\ m_\varepsilon\ m_{::} \mapsto & m_\varepsilon \\ \mathsf{Vect}\text{-}\mathbf{E}\ A & ([\mathsf{s}]\ k) & (::\ \{A\}\ \{k\}\ a\ v) & P\ m_\varepsilon\ m_{::} \mapsto & m_{::}\ k\ a\ v\ (\mathsf{Vect}\text{-}\mathbf{E}\ A\ k\ v\ P\ m_\varepsilon\ m_{::}) \end{array}$$

In the implementation the deleted arguments really are removed from the now fully applied constructors. This is safe because these terms are only decomposed by $\mathsf{Vect}\text{-}\mathbf{E}$ which does not expect the deleted arguments.

4 Eliding Redundant Constructor Tags

Recall the second alternative implementation of $\mathsf{Vect}\text{-}\mathbf{E}$ (‡) where case selection is by analysis of the length index rather than the target itself. For which types can we do case selection on an argument other than the target?

If $\mathsf{c}\,\vec{a}, \mathsf{c}'\,\vec{b}\ :\ \mathsf{D}\,\vec{s}$ implies $\mathsf{c} = \mathsf{c}'$, we say that the family D is **detaggable**. Vect is detaggable because the length index determines whether the constructor is ε (if the length index is $\mathsf{0}$) or $::$ (if the length index is $\mathsf{s}k$).

For any set of ι-schemes, if the index patterns are already mutually exclusive, we can decide which scheme applies without checking the target's constructor tag. The following program checks if two patterns are guaranteed to match disjoint sets of terms:

$$\begin{array}{l} \textsc{disjoint}(\ \mathsf{c}\,\vec{p}\ ,\ \mathsf{c}'\,\vec{q}\,) \Longrightarrow \mathsf{true}\ \ \underline{\text{if}}\ \mathsf{c} \neq \mathsf{c}' \\ \textsc{disjoint}(\ \mathsf{c}\,\vec{p}\ ,\ \mathsf{c}\,\vec{q}\ ) \Longrightarrow \exists i. \textsc{disjoint}(p_i, q_i) \\ \textsc{disjoint}(\ [\mathsf{c}]\,\vec{p}, [\mathsf{c}]\,\vec{q}) \Longrightarrow \exists i. \textsc{disjoint}(p_i, q_i) \\ \textsc{disjoint}(\ \ p\ \ ,\ \ q\ \ ) \Longrightarrow \mathsf{false}\ \ \underline{\text{otherwise}} \end{array}$$

Of course if we are to match on the indices then we must actually examine their constructors, so the previous lazy definition of PATS is not sufficient. We compute the patterns we need for this optimisation with EPATS — the same as PATS but with LAZY replaced by EAGER:

$$\textsc{eager}(\ \mathsf{c}, \vec{p}) \Longrightarrow \mathsf{c}\,\vec{p}$$

Given a family D with constructors $\mathsf{c}_i\ :\ \forall \vec{x}\!:\!\vec{X}_i.\,\mathsf{D}\,\vec{s}_i$ where $\textsc{epats}\,(\emptyset, \vec{s}_i) \Longrightarrow (V_i, \vec{p}_i)$, we say D is **concretely detaggable** if

$$\forall\, i \neq j.\, \exists k.\ \textsc{disjoint}(p_{ik}, p_{jk}) \Longrightarrow \mathsf{true}$$

Theorem. We may optimise (**detag**) such a concretely detaggable D thus:

$$\begin{array}{l} [\![\mathsf{c}_i]\!] \ \Longrightarrow\ \lambda\vec{x}.\ \{\mathsf{c}_i\}\ \vec{x}^{\{V\}} \\ \mathsf{D}\text{-}\mathbf{E}\,\vec{p}_i\ (\{\mathsf{c}_i\}\ \vec{x}^{\{V\}})\ P\ \vec{m}\ \mapsto\ e_i \end{array}$$

Proof. These schemes are Γ-respectful for all Γ by the same argument as for forcing—the switch to eager patterns does not affect the set of variables matched from the indices, nor the success of matching well-typed values. Deleting the constructor in the target can only improve the possibility of a match, but the disjointness condition directly ensures that the schemes remain Γ-well-defined. $\square$

For our Vect example, detagging is given by:

$$\llbracket \varepsilon \rrbracket \Longrightarrow \lambda A.\ \{\varepsilon\}\ \{A\}$$
$$\llbracket :: \rrbracket \Longrightarrow \lambda A; k; a; v.\ \{::\}\ \{A\}\ \{k\}\ a\ v$$

$$\begin{array}{lllll} \mathsf{Vect}\text{-}\mathbf{E}\ A & \mathsf{0} & (\{\varepsilon\}\ \{A\}) & P\ m_\varepsilon\ m_{::} \mapsto & m_\varepsilon \\ \mathsf{Vect}\text{-}\mathbf{E}\ A & (\mathsf{s}\ k) & (\{::\}\ \{A\}\ \{k\}\ a\ v) & P\ m_\varepsilon\ m_{::} \mapsto & m_{::}\ k\ a\ v\ (\mathsf{Vect}\text{-}\mathbf{E}\ A\ k\ v\ P\ m_\varepsilon\ m_{::}) \end{array}$$

We achieve this space optimisation at the cost of using eager rather than lazy patterns. The number of constructor tests required increases by a constant (possibly zero!) factor and indices may sometimes be computed where they would previously be ignored. Clearly a real implementation would minimise the number of eager patterns required to make the distinction. An analysis of this space/time trade-off is beyond the scope of this paper, but for Vect it seems likely to be worthwhile since we have swapped one constructor test for another.

5 Run-Time Optimisation

In our Vect-**E** example, we have already deleted both ε and its argument. We might be tempted to go a step further, and comment out that entire target.

$$\mathsf{Vect}\text{-}\mathbf{E}\ A\ \mathsf{0}\ [\{\varepsilon\}\ \{A\}]\ P\ m_\varepsilon\ m_{::} \mapsto\ m_\varepsilon$$

However, this ι-scheme is not respectful and breaks subject reduction thus:

$$\begin{array}{rl} \ldots ; x : \mathsf{Vect}\ A\ \mathsf{0} \vdash \mathsf{Vect}\text{-}\mathbf{E}\ A\ \mathsf{0}\ x\ P\ m_\varepsilon\ m_{::} & : P\ \mathsf{0}\ x \\ \mapsto\ m_\varepsilon & : P\ \mathsf{0}\ \varepsilon \end{array}$$

The pattern $(\{\varepsilon\}\ \{A\})$ may not test tags or extract arguments, but it still only matches targets whose weak head-normal forms are constructor applications. The optimisations we have seen thus far are safe to use in any context, and we need to reduce under binders when performing the equality checks which ensure that EPIGRAM programs elaborate to well typed terms.

However, at run-time, we can employ a much more restricted notion of computation, reducing only in the *empty* context, $\mathcal{E}$. In this scenario, we can exploit the **adequacy** property of TT — if $\mathcal{E} \vdash t : \mathsf{D}\ \vec{s}$ then $\textsc{whnf}(t)$ is $\mathsf{c}\ \vec{t}$ for some $\vec{t}$ — to gain further optimisations, not available in a general context.

In effect, we may employ weaker criteria for alternative implementations of elimination operators in run-time execution. We say that a **run-time optimisation** is given by a substitution and ι-schemes in ExTT as before, except that these schemes need only be $\mathcal{E}$-respectful and $\mathcal{E}$-well-defined.

The adequacy property tells us that the target will always match a constructor pattern at run-time, hence we may safely presuppose a pattern from which no information is gained, as suggested above. Moreover, by applying this observation inductively, we can sometimes extract another, more drastic optimisation from the guarantee of adequacy at run-time.

6 Collapsing Content-Free Families at Run-Time

Consider the less than or equal relation, declared and elaborated as follows:

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

$$\begin{array}{l} \leq \ : \ \mathbb{N} \to \mathbb{N} \to \star \\ \mathsf{leO} \ : \ \forall y{:}\mathbb{N}.\ \leq 0\ y \\ \mathsf{leS} \ : \ \forall x, y{:}\mathbb{N}.\ \leq x\ y \to \ \leq (\mathsf{s}x)\ (\mathsf{s}y) \end{array}$$

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. Forcing and detagging yield:

$$\begin{array}{l} [\![\mathsf{leO}]\!] \implies \lambda y.\ (\{\mathsf{leO}\}\ \{y\}) \\ [\![\mathsf{leS}]\!] \implies \lambda x; y; p.\ (\{\mathsf{leS}\}\ \{x\}\ \{y\}\ p) \\ \leq\text{-}\mathbf{E}\ 0 \quad\ y \quad\ (\{\mathsf{leO}\}\ \{y\}) \quad P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}} \ \mapsto\ m_{\mathsf{leO}}\ y \\ \leq\text{-}\mathbf{E}\ (\mathsf{s}x)\ (\mathsf{s}y)\ (\{\mathsf{leS}\}\ \{x\}\ \{y\}\ p)\ P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}} \\ \qquad\qquad \mapsto\ m_{\mathsf{leS}}\ x\ y\ p\ (\leq\text{-}\mathbf{E}\ x\ y\ p\ P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}}) \end{array}$$

Now we are left with only one undeleted argument, the recursive p in leS. This argument serves two purposes — firstly it is the target of the recursive call and secondly it is passed to the method m_{leS}. We might think that p can also be elided — ultimately it can only by examined by $\leq$-**E** which, by induction, can be shown never to examine it. In a partial evaluation setting, however, where we may reduce under binders, we must at least check that the target is canonical for reduction to be possible. If not, we run the risk of reducing a proof of something which cannot be constructed, such as $5{\leq}4$!

At run-time, on the other hand, we never need to check that 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:

$$\begin{array}{l} [\![\mathsf{leO}]\!] \implies \lambda y.\ (\{\mathsf{leO}\ y\}) \\ [\![\mathsf{leS}]\!] \implies \lambda x; y; p.\ (\{\mathsf{leS}\ x\ y\ p\}) \\ [\![\leq\text{-}\mathbf{E}]\!] \implies \lambda x; y; p; P; m_{\mathsf{leO}}; m_{\mathsf{leS}}.\ \leq\text{-}\mathbf{E}\ x\ y\ \{p\}\ P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}} \end{array}$$

For which families can we do this run-time optimisation? If $x, y \ : \ \mathsf{D}\ \vec{s}$ implies $x = y$ we say that the family D is **collapsible**. $\leq$ is collapsible because any value in $x{\leq}y$ is determined entirely by the indices x and y.

$$\begin{array}{l} \leq\text{-}\mathbf{E}\ 0 \quad\ y \quad\ \{\mathsf{leO}\ y\} \quad P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}} \ \mapsto\ m_{\mathsf{leO}}\ y \\ \leq\text{-}\mathbf{E}\ (\mathsf{s}x)\ (\mathsf{s}y)\ \{\mathsf{leS}\ x\ y\ p\}\ P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}} \\ \qquad\qquad \mapsto\ m_{\mathsf{leS}}\ x\ y\ (\{p\})\ (\leq\text{-}\mathbf{E}\ x\ y\ \{p\}\ P\ m_{\mathsf{leO}}\ m_{\mathsf{leS}}) \end{array}$$

We say a family is **concretely collapsible** if it is detaggable and for each constructor $\mathsf{c} : \forall \vec{a} : \vec{A}.\, \mathsf{D}\, \vec{r}_1 \to \ldots \to \mathsf{D}\, \vec{r}_j \to \mathsf{D}\, \vec{s}$, $\textsc{epats}\,(\emptyset, \vec{s}) \Longrightarrow (\vec{a}, \vec{p})$. That is, the constructor tag and all the non-recursive arguments are *cheaply* recoverable from the indices.

Theorem. We may optimise a concretely collapsible family *at run-time*:

$$\begin{array}{l} \mathsf{D}\text{-}\mathbf{E}\, \vec{p}\ \{\mathsf{c}\ \vec{a}\ \vec{y}\}\ P\ \vec{m} \\ \qquad \mapsto\ m_{\mathsf{c}}\ \vec{a}\ (\{y_1\})\ \ldots\ (\{y_n\})\ (\mathsf{D}\text{-}\mathbf{E}\, \vec{r}_1\ \{y_1\}\ P\ \vec{m})\ \ldots\ (\mathsf{D}\text{-}\mathbf{E}\, \vec{r}_n\ \{y_n\}\ P\ \vec{m}) \\ [\![\mathsf{c}]\!] \Longrightarrow \lambda \vec{a}; \vec{y}.\ (\{\mathsf{c}\ \vec{a}\ \vec{y}\}) \\ [\![\mathsf{D}\text{-}\mathbf{E}]\!] \Longrightarrow \lambda \vec{i}; x; p; \vec{m}.\ \mathsf{D}\text{-}\mathbf{E}\, \vec{i}\ \{x\}\ P\ \vec{m} \end{array}$$

Proof. These schemes are $\mathcal{E}$-well-defined by the same argument as for detagging. They are $\mathcal{E}$-respectful because the only possible left-hand sides have the form $\mathcal{E} \vdash \mathsf{D}\text{-}\mathbf{E}\, \vec{s}'\ (\mathsf{c}\ \vec{a}'\ \vec{y}')\ P'\ \vec{m}'$, hence, by disjointness, the only possible match, even with the target deleted, is with the scheme for c, with matching substitution $\sigma = \vec{a}'/\vec{a} \circ P'/P \circ \vec{m}'/\vec{m}$, binding all the undeleted free variables on the right-hand side because $\textsc{epats}\,(\emptyset, \vec{s}) \Longrightarrow (\vec{a}, \vec{p})$. Taking $\tau = \vec{y}'/\vec{y}$, we see that

$$\mathcal{E} \vdash \tau\, |\sigma(\mathsf{D}\text{-}\mathbf{E}\, \vec{p}\ \{\mathsf{c}\ \vec{a}\ \vec{y}\}\ P\ \vec{m})| = \mathsf{D}\text{-}\mathbf{E}\, \vec{s}'\ (\mathsf{c}\ \vec{a}'\ \vec{y}')\ P'\ \vec{m}'$$

hence these schemes are $\mathcal{E}$-respectful. □

7 Examples

7.1 The Finite Sets

The finite sets, indexed over a natural number n, are a family of types with n elements. Effectively, they are a representation of bounded numbers and are declared as follows:

$$\underline{\text{data}}\quad \frac{n : \mathbb{N}}{\mathsf{Fin}\, n : \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{f0} : \mathsf{Fin}\, \mathsf{s}n} \quad \frac{i : \mathsf{Fin}\, n}{\mathsf{fs}\, i : \mathsf{Fin}\, \mathsf{s}n}$$

The forcing optimisation elides the indices from the elaborated constructors:

$$\begin{array}{l} [\![\mathsf{f0}]\!] \Longrightarrow \lambda n.\ \mathsf{f0}\ \{n\} \\ [\![\mathsf{fs}]\!] \Longrightarrow \lambda n; i.\ \mathsf{fs}\ \{n\}\ i \end{array}$$

After stripping the forceable arguments, the shape of the resulting type matches that of $\mathbb{N}$ — that is, the base constructor takes no arguments and the step constructor takes a single recursive argument. In principle, any optimisations which apply to $\mathbb{N}$ such as Magaud and Bertot's binary representation [17] should also apply to Fin. We hope to recover Xi's efficient treatment of bounded numbers in this way [28] and perhaps extend it to other forms of validation.

7.2 Comparison of Natural Numbers

The $\mathsf{Compare}$ family from [20] represents the result of comparing two numbers, storing which is the greater and by how much:

$$\underline{\text{data}} \quad \frac{m, n \;:\; \mathbb{N}}{\mathsf{Compare}\; m\; n \;:\; \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{lt}\; y \;:\; \mathsf{Compare}\; x\; (x + (\mathsf{s}\; y))} \quad \frac{}{\mathsf{eq} \;:\; \mathsf{Compare}\; x\; x} \quad \frac{}{\mathsf{gt}\; x \;:\; \mathsf{Compare}\; (y + (\mathsf{s}\; x))\; y}$$

$\mathsf{Compare}$ is an example of a family which is collapsible, but not concretely collapsible. Clearly there is only one possible element of $\mathsf{Compare}\; m\; n$ for each m and n, and given this element we can extract their difference in constant time. If we were to collapse $\mathsf{Compare}$ we would replace this simple inspection by the recomputation of the difference each time the same value was used. We restrict concretely collapsible families to those where the recomputation of values is cheap. Nonetheless, by forcing, $\mathsf{Compare}$ need only store which index is larger and by how much:

$$\begin{aligned} [\![\mathsf{lt}]\!] &\Longrightarrow \lambda x; y.\; \mathsf{lt}\; \{x\}\; y \\ [\![\mathsf{eq}]\!] &\Longrightarrow \lambda x.\; \mathsf{eq}\; \{x\} \\ [\![\mathsf{gt}]\!] &\Longrightarrow \lambda x; y.\; \mathsf{gt}\; x\; \{y\} \end{aligned}$$

7.3 Accessibility Predicates

In [6], Bove and Capretta use special-purpose accessibility predicates to prove termination of general recursive functions. For example, **quicksort** terminates on the nil, and it terminates on $\mathsf{cons}\; x\; xs$ if it terminates on **filter** $(<\; x)\; xs$ and **filter** $(\geq\; x)\; xs$. This is expressed by the qsAcc predicate below:

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

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

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

quicksort itself is defined by induction over qsAcc, so a naïve implementation would need to store the proofs. However, qsAcc is concretely collapsible:

$$\begin{aligned} [\![\mathsf{qsNil}]\!] &\Longrightarrow \{\mathsf{qsNil}\} \\ [\![\mathsf{qsCons}]\!] &\Longrightarrow \lambda x; xs; qsl; qsr.\; \{\mathsf{qsCons}\; x\; xs\; qsl\; qsr\} \end{aligned}$$

Collapsing replaces computation over qsAcc by computation over its indices, restoring the intended operational semantics of the original program! These accessibility predicates are concretely collapsible because their indices are constructed from the constructor patterns of programs.

7.4 The Simply Typed λ-calculus

We define the simply typed λ-calculus in a similar fashion to [20], making extensive use of inductive families to specify invariants on the data structures. We begin with STy, representing simple monomorphic types:

$$\underline{\text{data}}\quad \frac{}{\mathsf{STy}\ :\ \star}\quad \underline{\text{where}}\quad \frac{}{\iota\ :\ \mathsf{STy}}\quad \frac{s,t\ :\ \mathsf{STy}}{s \Rightarrow t\ :\ \mathsf{STy}}$$

We represent contexts by Vects of types, $\mathsf{Ctx} = \mathsf{Vect}\,\mathsf{STy}$. The explicit length allows a safe de Bruijn representation of variables, via the Fin family, hence our untyped terms, Expr, are at least well scoped—the length is forceable for each constructor:

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

$$\underline{\text{where}}\quad \frac{i\ :\ \mathsf{Fin}\ n}{\mathsf{eVar}\ i\ :\ \mathsf{Expr}\ n}\quad \frac{S\ :\ \mathsf{STy}\quad t\ :\ \mathsf{Expr}\ \mathsf{s}n}{\mathsf{eLam}\ S\ t\ :\ \mathsf{Expr}\ n}\quad \frac{f,s\ :\ \mathsf{Expr}\ n}{\mathsf{eApp}\ f\ s\ :\ \mathsf{Expr}\ n}$$

The Var relation gives types to variables. $\mathsf{Var}\,G\,i\,T$ states that the ith member of the context G has type T. Clearly Var is concretely collapsible.

$$\underline{\text{data}}\quad \frac{G\ :\ \mathsf{Ctx}\ n\quad i\ :\ \mathsf{Fin}\ n\quad T\ :\ \mathsf{STy}}{\mathsf{Var}\ G\ i\ T\ :\ \star}$$

$$\underline{\text{where}}\quad \frac{}{\mathsf{stop}\ :\ \mathsf{Var}\ (S{::}G)\ \mathsf{f0}\ S}\quad \frac{v\ :\ \mathsf{Var}\ G\ i\ T}{\mathsf{pop}\ v\ :\ \mathsf{Var}\ (S{::}G)\ (\mathsf{fs}\ i)\ T}$$

Finally, we have the well typed terms, indexed over contexts, the original raw terms and types. This gives us a particularly safe representation — no typechecker can return the wrong well typed term. This indexing also enables us to synchronise terms safely with value environments during evaluation in the style of Augustsson and Carlsson [3].

$$\underline{\text{data}}\quad \frac{G\ :\ \mathsf{Ctx}\ n\quad e\ :\ \mathsf{Expr}\ n\quad T\ :\ \mathsf{STy}}{\mathsf{Term}\ G\ e\ T\ :\ \star}$$

$$\underline{\text{where}}\quad \frac{v\ :\ \mathsf{Var}\ G\ i\ T}{\mathsf{var}\ v\ :\ \mathsf{Term}\ G\ (\mathsf{eVar}\ i)\ T}\quad \frac{b\ :\ \mathsf{Term}\ (S{::}G)\ e\ T}{\mathsf{lam}\ b\ :\ \mathsf{Term}\ G\ (\mathsf{eLam}\ S\ e)\ (S \Rightarrow T)}$$

$$\frac{f\ :\ \mathsf{Term}\ G\ \mathit{fe}\ (S \Rightarrow T)\quad a\ :\ \mathsf{Term}\ G\ \mathit{ae}\ S}{\mathsf{app}\ f\ a\ :\ \mathsf{Term}\ G\ (\mathsf{eApp}\ \mathit{fe}\ \mathit{ae})\ T}$$

Term seems to involve a horrifying amount of duplication. Fortunately, many of the arguments are forceable and thanks to the indexing over raw terms, Term is detaggable. After optimisation, this is all that remains:

$$\begin{aligned}
[\![\mathsf{var}]\!] &\Longrightarrow \lambda n; G; i; T; v.\ \{\mathsf{var}\}\ \{n\}\ \{G\}\ \{i\}\ \{T\}\ \{v\}\\
[\![\mathsf{lam}]\!] &\Longrightarrow \lambda n; G; S; e; T; b.\ \{\mathsf{lam}\}\ \{n\}\ \{G\}\ \{S\}\ \{e\}\ \{T\}\ b\\
[\![\mathsf{app}]\!] &\Longrightarrow \lambda n; G; \mathit{fe}; S; T; f; \mathit{ae}; a.\ \{\mathsf{app}\}\ \{n\}\ \{G\}\ \{\mathit{fe}\}\ S\ \{T\}\ f\ \{\mathit{ae}\}\ a
\end{aligned}$$

The only non-recursive arguments which survive are the domain types of applications. Typechecking thus consists of ensuring that these can be determined.

8 Conclusions and Further Work

The ideas presented here have been tested in a prototype implementation. Execution in this system is by extraction to a Haskell coding of ExTT values without the deleted subterms. We have used the GHC profiling tools [25] to assess the space usage of programs.

Our experiments show a significant reduction in space requirements over a naïve implementation particularly where there is extensive indexing. For vector operations, a 10-20% saving in memory usage is typical (depending on the length of the vector), but for the typechecker, a saving of over 80% has been observed.

Although remarkably straightforward, these optimisations only present themselves when one takes dependently typed programming seriously. The forcing optimisation largely overcomes the space penalty of adopting dependent types, but detagging derives new dynamic benefit from previously unavailable static information. Collapsing, too, has significant consequences, deleting accessibility arguments and all the equational reasoning from run-time code, not because we deem them to be proof-irrelevant, but because they actually are.

We suspect that these optimisations are the first of many. For example, as we erase forceable indices, it is worth identifying operations which affect nothing else, such as **weaken** $:$ $\mathsf{Fin}\ n \to \mathsf{Fin}\ (\mathsf{s}n)$, which embeds a value in a higher indexed set — this is effectively the identity function. This optimisation applies wherever functions exist only to manage invariants.

We might also consider the low level implementation of high level types, such as the natural numbers. By replacing $\mathbb{N}$-**E** with an appropriate elimination rule for unbounded binary numbers [17] we can achieve a significant speed-up. Any other data structure with the same shape *after optimisation*, eg. Fin, can be treated similarly. In a practical implementation, such optimisations are essential for comparable performance to its conventional counterparts.

Optimisation of a new language with a new type system naturally presents new problems and new opportunities. While we can never hope to produce a completely optimal program in all cases, this research leads us to believe that the presence of much more static information can only give us greater scope for optimisation in both time and space.

References

1. Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof, 1996.
2. Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 368–381. Springer-Verlag, September 1985.
3. Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. http://www.cs.chalmers.se/ augustss/cayenne/, 1999.
4. Stefano Berardi. Pruning simply typed lambda terms. *Journal of Logic and Computation*, 6(5):663–681, 1996.

5. U. Berger and H. 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.
6. Ana Bove and Venanzio Capretta. Modelling general recursion in type theory, September 2002.
7. Paul Callaghan and Zhaohui Luo. Implementation techniques for inductive types in plastic. In Bengt Nordström Thierry Coquand, Peter Dybjer and Jan Smith, editors, *Types for Proofs and Programs*, volume 1956 of *LNCS*, pages 94–113. Springer-Verlag, 1999.
8. Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors. *Types for Proofs and Programs*, volume 2277 of *LNCS*. Springer-Verlag, 2000.
9. Coq Development Team. The Coq proof assistant — reference manual, 2001.
10. André Luís de Medeiros Santos. *Compilation By Transformation In Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.
11. Peter Dybjer. Inductive families. *Formal Aspects Of Computing*, 6:440–465, 1994.
12. Robert Harper and Randy Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136, 1991.
13. Thomas Johnsson. Efficient compilation of lazy evaluation, 1984.
14. Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for proofs and programs*, volume 2646 of *LNCS*. Springer-Verlag, 2002.
15. Zhaohui Luo. *Computation and Reasoning – A Type Theory for Computer Science*. International Series of Monographs on Computer Science. OUP, 1994.
16. Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical report, LFCS, University of Edinburgh, 1992.
17. Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In Callaghan et al. [8], pages 181–196.
18. Lena Magnusson. *The implementation of ALF – A Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitutions*. PhD thesis, Chalmers University of Technology, Göteborg, 1994.
19. Conor McBride. *Dependently Typed Functional Programs and their proofs*. PhD thesis, University of Edinburgh, May 2000.
20. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
21. Fred McBride. *Computer Aided Manipulation of Symbols*. PhD thesis, Queen's University of Belfast, 1970.
22. Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Paris 7, 1989.
23. Simon L Peyton Jones and André L M Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32:3–47, 1998.
24. Randy Pollack. Implicit syntax. Technical report, LFCS, University of Edinburgh, 1992.
25. Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher order functional languages, 1995.
26. Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, December 1998.
27. Hongwei Xi. Dead code elimination through dependent types, 1999.
28. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types, 1998.