The view from the left

Conor McBride and James McKinna
Department of Computer Science
University of Durham
South Road, Durham DH1 3LE
c.t.mcbride@durham.ac.uk
j.h.mckinna@durham.ac.uk

Abstract

Pattern matching has proved an extremely powerful and durable notion in functional programming. This paper contributes a new programming notation for type theory which elaborates the notion in various ways.

Firstly, as is by now quite well-known in the type theory community, definition by pattern matching becomes a more discriminating tool in the presence of dependent types, since it refines the explanation of types as well as values. This becomes all the more true in the presence of the rich class of datatypes known as inductive families (Dybjer, 1991).

Secondly, as proposed by Peyton Jones (Peyton Jones, 1997) for Haskell, and independently rediscovered by us, subsidiary case analyses on the results of intermediate computations, which commonly take place on the right-hand side of definitions by pattern matching, should rather be handled on the left. In simply-typed languages, this subsumes the trivial case of Boolean guards; in our setting it becomes yet more powerful.

Thirdly, elementary pattern matching decompositions have a well-defined interface given by a dependent type; they correspond to the statement of an induction principle for the datatype. More general, user-definable decompositions may be defined which also have types of the same general form. Elementary pattern matching may therefore be recast in abstract form, with a semantics given by translation. Such abstract decompositions of data generalize Wadler's notion of 'view' (Wadler, 1987). The programmer wishing to introduce a new view of a type T, and exploit it directly in pattern matching, may do so via a standard programming idiom. The type theorist, looking through the Curry-Howard lens, may see this as *proving a theorem*, one which establishes the validity of a new induction principle for T.

We develop enough syntax and semantics to account for this high-level style of programming in dependent type theory. It culminates in the development of a typechecker for the simply-typed lambda calculus, which furnishes a view of raw terms as either being well-typed, or containing an error. The implementation of this view is *ipso facto* a proof that typechecking is decidable.

1 Introduction

This paper is a contribution to declarative programming, in that it introduces a new high-level *notation* for functional programming on top of an existing low-level dependent type theory. In particular, we offer a powerful and abstract successor to *pattern matching*, as conceived by Rod Burstall (Burstall, 1969) and, to our knowledge, first implemented in Fred McBride's extension of LISP (McBride, 1970).

The key feature of pattern matching in simply typed languages is that the structure of an arbitrary *value* in a datatype is explained. Classically, pattern matching analyses *constructor* patterns on the left-hand sides of functional equations, and is defined by a subsystem of the operational semantics with hard-wired rules for computing substitutions from the pattern variables to values. For example, in Standard ML (Milner *et al.*, 1997), one might test list membership as follows:

```
fun  elem k    []      = false
  |  elem k (l :: ls) = if (k = l) then true else elem k ls
```

The clarity of the code does not hinder its efficient compilation; a key technique here is Augustsson's analysis in terms of hierarchical switching on the outermost constructor symbol, coupled with the exposure of subexpressions (Augustsson, 1985). This yields, for `elem` above, the following cascade of `case` expressions:

```
fun  elem k ls = case   ls
                   of   []      => true
                    | l :: ls' => case (k = l)
                                    of  true  => true
                                     |  false => elem k ls'
```

Pattern matching has proved such a powerful and durable notion in functional programming, that its further development has remained firmly on the research agenda. Peyton Jones' idea of *pattern guards* (Peyton Jones, 1997; Peyton Jones & Erwig, 2000) allows definitions by pattern matching to handle on the *left*-hand side of programs, subsidiary analysis of the results of intermediate computations, which are more commonly, but "clunkily" (*loc.cit.*), handled on the *right*. For `elem`, we can pull *both* tests to the left as follows:

```
elem k []                        = False
elem k (l:ls) | True  <- k == l = True
elem k (l:ls) | False <- k == l = elem k ls
```

Of course, Haskell's *Boolean* guards (Peyton Jones & Hughes, 1999) can already qualify pattern matches by tests like `k == l`, but pattern guards handle subcomputations of more complex types. Further, the guard expression can be shared via a `where` clause and the layout rule. In our notation, you can achieve the same effect by grouping the two clauses in the scope of the call to $k == l$, as follows:

$$
\begin{array}{lll}
\mathbf{elem}\, k \;\; [\,] & \mapsto & \mathsf{false} \\
\mathbf{elem}\, k\, (l :: ls) & \mid & k == l \\
 & & \quad \mathsf{true} \;\; \mapsto \;\; \mathsf{true} \\
 & & \quad \mathsf{false} \;\; \mapsto \;\; \mathbf{elem}\, k\, ls
\end{array}
$$

Dependent types add a descriptive and expressive power which makes pattern matching a more discriminating tool, refining types as well as values. Each elementary pattern matching decomposition has a well-defined interface given by a dependent type, corresponding to an induction principle for the datatype (Burstall, 1969; Nordström *et al.*, 1990). This insight flows from type theory's interplay between computation and reasoning—usually sloganised as the 'Curry-Howard correspondence', or 'propositions-as-types'. The key feature of induction is that the result *type* is *instantiated*, and hence further explained, by the patterns.

This observation bites all the more strongly in the presence of the rich class of datatypes known as **inductive families** (Dybjer, 1991). One such is So, a collection of types indexed by a Boolean value:

$$
\underline{\text{data}} \quad \frac{b \;:\; \mathsf{Bool}}{\mathsf{So}\, b \;:\; \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{oh} \;:\; \mathsf{So\,true}}
$$

The point here is that $\mathsf{So\,true}$ has one element whilst $\mathsf{So\,false}$ has none. If $p \;:\; \mathsf{So}\, b$, then 'case' on p tells us not only that p is oh, but also ('*for free*') that b must be true. Inspecting p can instantiate b and hence any type which depends on either!

We can use So to impose Boolean 'preconditions' on programs. For example, a program which requires an argument $p \;:\; \mathsf{So}\,(test_1 \;\mathbf{or}\; test_2)$ need only be defined under circumstances which make one of the test expressions evaluate to true. If such a program were to switch on the value of $test_1$, say, we should somehow 'know' that $p \;:\; \mathsf{So\,true}$ in the true case and that $p \;:\; \mathsf{So}\; test_2$ otherwise, but how might a typechecker make this connection? Our $\mid$ notation is motivated not just by convenience, but also to signal the abstraction of subcomputations from *types*.

Meanwhile, Wadler's 'views' proposal (Wadler, 1987; Burton *et al.*, 1996) allows programmers to implement new schemes for decomposing values in types (abstract datatypes, especially), extending the syntax of matching correspondingly. In our setting, user-definable decompositions—**elimination operators**—may be specified by types resembling the structural induction principles for datatypes, now the *primitives* from which higher-level analyses can be developed compositionally.

Our notation gives a pattern-based syntax to programming with *arbitrary* eliminators; the semantics is given by translation, rather than 'pattern matching' *per se*. Further, we establish a standard idiom of first-order programming for equipping a type T with a new elimination operator, by identifying a set of patterns which **cover** the values in T; such patterns may now be arbitrary expressions of type T. The type theorist, looking through the Curry-Howard lens, may see this as *proving* a new induction principle for T. A similar idea has emerged recently in Voda's untyped first-order 'Clausal Language' (Voda, 2002), which admits new forms of case analysis via theorem-proving in Peano Arithmetic.

Although the power of dependent types is widely acknowledged, sceptics rightly argue that expressibility is one thing and accessibility another. Programs should be read as well as written, often on the back of an envelope. Here, we address this issue of clarity. We claim that the existing notations of both functional languages and type theory fall short of what dependently typed programming demands, but also of what it can supply—a language of derived forms, rich, intuitive and extensible. Type theory offers the motive, the methods and the opportunity to ask anew what functional programming can aspire to be. We barely scratch the surface in this paper—nevertheless, we hope to engage your enthusiasm and your imagination.

1.1 Background

We start from a type theory with inductive families of datatypes (Dybjer, 1991), essentially Luo's UTT (Luo, 1994), as implemented in OLEG—the first author's adaptation (McBride, 1999) of Pollack's proof assistant LEGO (Luo & Pollack, 1992; Pollack, 1995). This type system is strongly normalizing (Goguen, 1994) and hence typechecking is decidable. An important and distinctive feature, which we expand upon below, is that inductive families embrace data structures richer than those available in other candidate languages for dependently-typed programming such as DML (Xi, 1998), or Cayenne (Augustsson, 1998): the former supports compile-time enforcing of finer well-formedness constraints on data which is nonetheless only Hindley-Milner typable; as to the latter, we explore an example not readily expressible in Cayenne—well-typed λ-terms over simple types—in Section 7.

Datatypes in UTT come with no intrinsic notion of pattern matching, by contrast with systems like ALF (Coquand, 1992; Magnusson, 1994). Primitive computation on datatypes is provided via 'elimination operators' (the 'introduction operators' being constructors), which behave operationally like primitive recursors, but have types which state structural induction principles.

For example, the elimination operator for the natural numbers has the following type—compare the Hindley-Milner type scheme for primitive recursion:

$$\begin{array}{lll} \mathbb{N}\text{-}\mathbf{Elim} & : & \forall P\!:\!\mathbb{N} \to \star. \\ & & P\,\mathsf{0} \to \\ & & (\forall k\!:\!\mathbb{N}.\ P\ k \to P\,(\mathsf{s}k)) \to \\ & & \forall n\!:\!\mathbb{N}.\ P\ n \end{array} \qquad \begin{array}{lll} \mathbb{N}\text{-}\mathbf{PrimRec} & : & \forall T\!:\!\star. \\ & & T \to \\ & & (\mathbb{N} \to T \to T) \to \\ & & \mathbb{N} \to T \end{array}$$

Observe that $\mathbb{N}$-**Elim** delivers an inhabitant of a **dependent function space**, in this case $\forall n : \mathbb{N}.\ P\ n$. This allows us to specify, via an arbitrary program P, the '**motive**', different outcomes intended for different values of n. Learning more about n can *change* the things we are able to do with it, hence we can express numerically indexed operations such as matrix multiplication. By contrast, $\mathbb{N}$-**PrimRec**'s type allows no connection between the number and the purpose it serves.

The arguments of $\mathbb{N}$-**Elim** which explain each case also have more informative types than in the Hindley-Milner version. We call these arguments **methods**—where the vernacular speaks only, somewhat weakly, of 'base' and 'step' cases, without naming 'the argument for such a case'—because they describe how the motive is to be pursued, depending on the value of n . Method types document explicitly the values for which we use them—a possibility only when types can depend on data.

A key point of this paper is that the types of eliminators give an *abstract* interface to pattern analysis, whatever the actual patterns are. For example, the trichotomy principle can be seen as an operator eliminating two natural numbers:

$$
\begin{array}{lllll}
\mathbb{N}\textbf{-Compare} : & & \forall P : \mathbb{N} \to \mathbb{N} \to \star. & & \\
& (\forall x, y : \mathbb{N}. & P \quad x & (x + \mathsf{s}y)) & \to \\
& (\forall x : \mathbb{N}. & P \quad x & x \quad) & \to \\
& (\forall x, y : \mathbb{N}. & P\ (y + \mathsf{s}x) & y \quad) & \to \\
& \forall m, n : \mathbb{N}. & P \quad m & n &
\end{array}
$$

We will show in Section 4 below how to use such operators in general, and in Section 6 how to construct (a variant of) $\mathbb{N}$-**Compare**, which we may then use to define functions which in ordinary programming would be computed by a combination of a boolean test and subtraction, where this operation is *rendered safe to perform* by the outcome of the test.

Elimination operators are first-class values, and their types are sufficient on their own to document their usage in programs. Hence they may be abstracted in signatures which hide their representation without further ado. Moreover, as we shall see below, for the class of datatype families which we consider, certain distinguished elimination operators may be defined automatically.

1.2 Outline of the rest of the paper

Section 2 describes the basic type theory in which we work, augmented with a concrete syntax for programming. This is then explained by elaboration into an extension of the basic type theory which uses labels in terms and types to correlate the usage of a concrete syntax program with its elaboration.

In Section 3 we focus upon the language of inductive families and their properties. We identify a taxonomy of possible type dependency in case analyses through consideration of a running example based on heterogeneous association lists.

In Section 4 we give a technical characterization of **eliminators**, together with the $\Leftarrow$ ('by') construct which supports their use, whether primitive or user-defined. We discuss in depth the method by which we exploit elimination with equational constraints to explain the notion of patterns, as well as arbitrary structured decomposition, on the left-hand sides of program definitions. In particular, we consider a useful derived form for dealing with structural recursion.

In Section 5, we discuss the general situation of decomposing the results of subcomputations. Our | ('with') construct supports this, generalizing pattern guards to the dependently-typed setting. This notation retains economy of expression, but also allows delicate type distinctions to be made during case analysis: without it, we would need explicit helper functions with much more complex type signatures.

Although elimination operators are higher-order functions, Section 6 introduces a first-order programming idiom for constructing and working with them—this is our account of **views**.

In Section 7, we conclude our technical discussion with a large example: a typechecker for simply-typed lambda calculus with explicit type labels—'Church-style' (pre-)terms in Barendregt's terminology (Barendregt, 1992). The program takes the form of a view of pre-terms as being either well-typed or containing an error. The implementation of this view is a proof that typechecking is decidable.

In an epilogue, we discuss our findings and future work.

1.3 Some history; some culture

Our background is mainly in the field of interactive theorem proving in type theory, using the LEGO/OLEG system. Consequently, the original draft of this paper had a very different emphasis: firstly, we focused on supporting an *interactive method* of programming. Indeed, while OLEG does not directly support the notations described in this paper, it does provide the tactics which inspired them—and which translate them into raw type theory. We developed all our examples *interactively* using these tactics.

Secondly, and perhaps more seriously, it was motivated from the 'logical' perspective on type theory. Regardless of the merits of this viewpoint, "dependent types" scarcely approached "practical programming" in terms of contributing to a dialogue between communities. This is not a new phenomenon: a good illustration lies in the papers by Bird and Paterson, and Altenkirch and Reus, each writing about the type of de Bruijn λ-terms, as a nested type in (Bird & Paterson, 1999), and as an inductive family in (Altenkirch & Reus, 1999). The two share but a single common reference—Wadler's "Theorems for Free!" (Wadler, 1989). Would that more researchers had Wadler's ability to speak to both communities with equal effect.

Likewise, though we were inspired by Wadler's original proposal for views, we had worked in ignorance of subsequent elaborations of that idea and related developments, not least Peyton Jones' (1997) note. Quite independently, we had arrived at essentially the same formulation, but motivated by considerations of *typing*, rather than *evaluation*. Rod Burstall used to say to us that "Proofs are harder for students to understand than programs, because once you've obtained a proof, it isn't obvious what to do with it, or what it means to run one," in spite of what Curry-Howard might lead one to believe. Our experience teaching students is that only by connecting patterns to the *types* which give rise to them, can the computational meaning and use of pattern matching be fully grasped.

Acknowledgements We gratefully acknowledge the support of the EPSRC, with grants GR/N 24988 and GR/R 72259. We also thank the organisers of Dagstuhl seminars 01141, "Semantics of Proof Search", and 01341, "Dependent Types Meets Practical Programming", where we presented preliminary versions of some of these ideas. Healf Goguen had important, and early, influence on this work supervising the first author's PhD. We have received much good advice from the anonymous referees on how to improve this paper and from our colleagues, especially Randy Pollack. At a late stage, Thorsten Altenkirch and Roland Backhouse helped us out of a tight spot with coffee and printing facilities. Our main debt, however, is to the programmers who have inspired us: Rod Burstall, Fred McBride and Phil Wadler.

2 Dependent type theory for functional programming

This section introduces the functional core of the type theory in which we work—Luo's UTT (Luo, 1994), extended with local definitions as in (Luo & Pollack, 1992; Pollack, 1995; McBride, 1999)—together with a concrete syntax for programming. The core language of UTT is summarised in Figure 1. We expect readers familiar with type theory to find its technical content largely unremarkable. The notation we employ here is not standard, being orientated more towards programming, but we hope it is nonetheless clear. For functional programmers with less prior exposure to this subject matter, we cannot expect to fill in all the blanks, but we hope that we provide enough of an introduction to give access to the ideas in this paper.

Type theory's key novelty for the functional programmer is the generalization from simple function spaces $S \to T$ to **dependent** function spaces $\forall x : S.\ T$. Here T may involve x, making the return type of the function *depend* on the value of the argument. We may still write $S \to T$ if T does not contain x. Dependency allows operations on ranges of types, selected by a prior input, such as C-style `printf` (Augustsson, 1998), or the generic 'fold' for every concrete Haskell type (Altenkirch & McBride, 2002). It also makes type theory an expressive logic.

Functions themselves are introduced by λ-terms and applications compute just by β-reduction. As we have local definition ($\underline{\text{let}}\ x \mapsto s : S.t$), we dispense with substitution in the presentation. Definitions are not recursive—the s must exist before x is bound to it. Under the $\underline{\text{let}}\ x \mapsto s : S$ binding, x has type S and reduces to s by δ-reduction, and the binding itself will vanish when x no longer occurs in scope: we call this γ-reduction—γ for 'garbage'(*cf.* (Severi & Poll, 1994)).

UTT has no special treatment of polymorphism, but we may $\forall$-quantify over types

syntax

$$
\begin{array}{rcl}
\mathit{vid} & := & x \mid \ldots \\[1ex]
\mathit{term} & := & \mathit{vid} \mid \star_0 \mid \star_1 \mid \ldots \mid \star_n \mid \ldots \\
& \mid & \forall \mathit{vid}\!:\!\mathit{term}.\ \mathit{term} \mid \lambda \mathit{vid}\!:\!\mathit{term}.\ \mathit{term} \mid \mathit{term}\ \mathit{term} \\
& \mid & \underline{\mathrm{let}}\ \mathit{vid} \mapsto \mathit{term} : \mathit{term}.\ \mathit{term} \\[1ex]
\mathit{context} & := & \cdot \mid \mathit{context}; \mathit{vid} : \mathit{term} \mid \mathit{context}; \mathit{vid} \mapsto \mathit{term} : \mathit{term}
\end{array}
$$

validity $\boxed{\mathit{context} \vdash \underline{\mathrm{valid}}}$

$$
\frac{}{\cdot \vdash \underline{\mathrm{valid}}} \qquad \frac{\Gamma \vdash S : \star_i}{\Gamma; x : S \vdash \underline{\mathrm{valid}}} \qquad \frac{\Gamma \vdash s : S}{\Gamma; x \mapsto s : S \vdash \underline{\mathrm{valid}}}
$$

typing $\boxed{\mathit{context} \vdash \mathit{term} : \mathit{term}}$

$$
\frac{\Gamma \vdash \underline{\mathrm{valid}}}{\Gamma \vdash x : S}\ \Gamma \text{ contains } x : S \text{ or } x \mapsto s : S
$$

$$
\frac{\Gamma \vdash \underline{\mathrm{valid}}}{\Gamma \vdash \star_n : \star_{n+1}}
$$

$$
\frac{\Gamma \vdash S : \star_i \quad \Gamma; x : S \vdash T : \star_i}{\Gamma \vdash \forall x\!:\!S.\ T : \star_i}
$$

$$
\frac{\Gamma; x : S \vdash t : T}{\Gamma \vdash \lambda x\!:\!S.\ t : \forall x\!:\!S.\ T}
$$

$$
\frac{\Gamma \vdash f : \forall x\!:\!S.\ T \quad \Gamma \vdash s : S}{\Gamma \vdash f\ s : \underline{\mathrm{let}}\ x \mapsto s : S.\ T}
$$

$$
\frac{\Gamma; x \mapsto s : S \vdash t : T}{\Gamma \vdash \underline{\mathrm{let}}\ x \mapsto s : S.\ t : \underline{\mathrm{let}}\ x \mapsto s : S.\ T}
$$

$$
\frac{\Gamma \vdash t : S \quad \Gamma \vdash S \preceq T}{\Gamma \vdash t : T}
$$

reduction $\boxed{\mathit{context} \vdash \mathit{term} \leadsto \mathit{term}}$ **conversion** $\boxed{\mathit{context} \vdash \mathit{term} \simeq \mathit{term}}$

$$
[\beta] \qquad \frac{}{\Gamma \vdash (\lambda x\!:\!S.\ t)\ s \leadsto \underline{\mathrm{let}}\ x \mapsto s : S.\ t}
$$

$$
[\delta] \qquad \frac{}{\Gamma; x \mapsto s : S; \Gamma' \vdash x \leadsto s}
$$

$$
[\gamma] \qquad \frac{}{\Gamma \vdash \underline{\mathrm{let}}\ x \mapsto s : S.\ t \leadsto t}\quad x \notin t
$$

plus contextual closure, and $\simeq$ as the equivalence closure of $\leadsto$

cumulativity $\boxed{\mathit{context} \vdash \mathit{term} \preceq \mathit{term}}$

$$
\frac{\Gamma \vdash S \simeq T}{\Gamma \vdash S \preceq T} \qquad \frac{\Gamma \vdash R \preceq S \quad \Gamma \vdash S \preceq T}{\Gamma \vdash R \preceq T}
$$

$$
\frac{}{\Gamma \vdash \star_n \preceq \star_{n+1}} \qquad \frac{\Gamma \vdash S_1 \simeq S_2 \quad \Gamma; x : S_1 \vdash T_1 \preceq T_2}{\Gamma \vdash \forall x\!:\!S_1.\ T_1 \preceq \forall x\!:\!S_2.\ T_2}
$$

Fig. 1. Luo's UTT plus local definition (functional core)

(and other higher-kind objects). There is no danger of paradox—types are collected in a *cumulative* hierarchy of universes $\star_n$, individually closed under $\forall$, each inhabiting and embedded in the next. These level subscripts can be managed mechanically (Harper & Pollack, 1991), so we shall freely omit them.

Additionally, **implicit syntax**, a very useful mechanism also due to Pollack (Pollack, 1992), allows us to omit arguments to functions, where they may be *inferred* by unification. We mark in the concrete syntax for dependent function types whether the argument is to be supplied or omitted by default, writing $\forall_{x:S}.\ T$ to indicate the latter. We do not demand *complete* mechanical inference and indeed we may override it—if $f : \forall_{x:S}.\ T$, we may still write f_s to supply the argument s ourselves.

The core language is regulated by a system of mutually inductively defined **judgments**, of which the first (**typechecking**) and third (**conversion**) contain the most interest from a programming point of view:

$\boxed{\Gamma \vdash t : T}$ 't has type T in context Γ': terms t are typechecked with respect to a context which contains (at least) the declarations $x : S$ or definitions $x \mapsto s : S$ of every variable which may occur free within t;

$\boxed{\Gamma \vdash \underline{\text{valid}}}$ 'Γ is valid': only those contexts Γ make sense, whose declarations give variables legitimate types and whose definitions are type-correct;

$\boxed{\Gamma \vdash S \simeq T}$ 'S is convertible to T in Γ': UTT is a a *computational* theory: its types may contain and are identified up to conversion; conversion is the usual equivalence closure of a reduction relation $\boxed{\Gamma \vdash s \leadsto t}$, generated by congruence closure from a number of specified one-step contractions; $\leadsto$ embraces β-reduction, as well as other rules detailed below; we do not consider α-conversion explicitly—treatments include (McKinna & Pollack, 1999);

$\boxed{\Gamma \vdash S \preceq T}$ cumulativity polices embedding between universe levels.

This system has a number of very strong meta-theoretic properties: all programs terminate, so conversion is decidable, hence so too are cumulativity, validity and typechecking (Luo, 1990; Goguen, 1994; Pollack, 1995).

Remark on meta-notation and meta-operations

In addition to the above properties of the type theory, we also require a number of *meta*-operations. For example, $\Downarrow t$ denotes the unique normal form of t. We typically present these in 'functional' style, writing equations in the form *definiendum* $\Longrightarrow$ *definiens*, employing 'where' clauses, 'if-then-else' *etc.*

Inspired by de Bruijn's 'telescopes' (de Bruijn, 1991), we manipulate sequences of bindings and of arguments, writing sequences of terms as **vectors** $\vec{t}$ (empty vector ε), and iterated applications as $f\,\vec{t}$. Contexts, denoted by Greek capital letters, may stand for multiple bindings in $\forall$-, λ- and $\underline{\text{let}}$-expressions. That is, we write $\forall\Delta.\ T$ for the dependent function space formed by iteratively 'discharging' Δ over T:

$$\begin{aligned}\forall\cdot.\,T &\Longrightarrow T\\ \forall\Delta; x : S.\,T &\Longrightarrow \forall\Delta.\,\forall x{:}S.\,T\\ \forall\Delta; x \mapsto s : S.\,T &\Longrightarrow \forall\Delta.\,\underline{\text{let}}\; x \mapsto s : S.\,T\end{aligned}$$

Functions $\lambda\Delta.\,t$ and iterated definitions $\underline{\text{let}}\;\Delta \mapsto \vec{s}.\,t$ are accordingly abbreviated. Successive bindings with the same type, *e.g.* $m{:}\mathbb{N}; n{:}\mathbb{N}$, are abbreviated as $m, n{:}\mathbb{N}$. Finally, Δ may stand for the vector of its *declared* variables: if $\Gamma \vdash f : \forall\Delta.T$, then $\Gamma;\Delta \vdash f\Delta : T$, even if Δ contains definitions. *(End of remark).*

By the Strengthening Lemma (Luo, 1990; van Benthem Jutting *et al.*, 1994), any well-typed term $\Gamma \vdash t : T$ arises from a *minimal subcontext* of Γ, that is, there exist contexts Γ^t, Γ_t, satisfying:

- $\Gamma^t \subseteq \Gamma$ minimal such that $\Gamma^t \vdash t : T$;
- $\Gamma^t;\Gamma_t$ is a permutation of Γ;
- $\Gamma^t;\Gamma_t \vdash J$ if and only if $\Gamma \vdash J$, for any judgment J.

We shall make frequent use of this fact in the sequel. Indeed, such a context splitting (Γ^t, Γ_t) may be computed as $\textsc{strengthen}(\Gamma, t, T)$, a meta-operation defined as follows, where $\textsc{fv}(X)$ denotes the set of variables free in X:

$$\begin{array}{l}\textsc{strengthen}(\cdot, t, T) \Longrightarrow (\cdot,\cdot)\\ \textsc{strengthen}(x : S;\Gamma, t, T)\\ \quad \text{where } (\Gamma^t,\Gamma_t) \Longleftarrow \textsc{strengthen}(\Gamma, t, T)\\ \quad \Longrightarrow \text{if } x \in \textsc{fv}(\Gamma^t) \cup \textsc{fv}(t) \cup \textsc{fv}(T)\\ \qquad\quad \text{then } (x : S;\Gamma^t, \Gamma_t)\\ \qquad\quad \text{else } (\Gamma^t, x : S;\Gamma_t)\end{array}$$

2.1 Concrete Syntax for Programs

In this section, we develop our notation for programming, summarised in Figure 2.

We distinguish an extended expression language *expr* of this programming notation from the low-level *terms* of the underlying type theory. The category *expr* embraces the basic constructs of UTT, together with:

- names for datatypes *did* and their constructors *cid*;
- a category *lhs* which forms the left-hand sides of *programs*;
- a distinguished subcategory *call* of the *lhs*, which comprises the allowable invocations of functions;
- <u>let</u> notation, for local function definitions in expressions;
- <u>view</u> notation, which will be explained in detail in Section 6.

Top-level *source* code consists of a sequence of datatype declarations (of which more in Section 3 below) and definitions of new function symbols *fid*. These are

$$
\begin{array}{lll}
expr & := & vid \mid did \mid cid \mid call \\
 & \mid & expr : expr \\
 & \mid & \forall vid : expr.\ expr \mid \star \\
 & \mid & \lambda vid : expr.\ expr \mid expr\ expr \\
 & \mid & \underline{\mathrm{let}}\ sig[fid]\ program.\ expr \\
 & \mid & \underline{\mathrm{view}}\ expr \\
 & & \\
program & := & lhs \mapsto expr \\
 & \mid & lhs \Leftarrow expr\ \{seq[program]\} \\
 & \mid & lhs \mid expr\ \{program\} \\
 & & \\
decl & := & \underline{\mathrm{data}}\ sig[did]\ \underline{\mathrm{where}}\ sig[cid]^* \\
 & \mid & \underline{\mathrm{let}}\ sig[fid]\ program \\
 & & \\
source & := & seq[decl]
\end{array}
\qquad
\begin{array}{lll}
vid & := & x \mid \ldots \\
did & := & \mathsf{D} \mid \ldots \\
cid & := & \mathsf{c} \mid \ldots \\
fid & := & \mathbf{f} \mid \ldots \\
 & & \\
call & := & fid\ expr^* \\
lhs & := & call\ (\mid expr)^* \\
 & & \\
seq[thing] & := & \\
 & \mid & thing\ (;\ thing)^* \\
 & & \\
sig[id] & := & \dfrac{seq[vid\ :\ expr]}{id\ vid^*\ :\ expr}
\end{array}
$$

Fig. 2. Concrete syntax for dependently typed programs

introduced using <u>let</u>, which introduces a program with a specified type signature, given in natural deduction style:

$$\underline{\mathrm{let}} \quad \frac{\Phi}{\mathbf{f}\,\Phi\ :\ R} \quad program$$

where the syntax for *programs* departs from the traditional prioritized *list* of pattern matching equations. A *program* is a hierarchical structure, resembling those of Augustsson (Augustsson, 1985), which explains how *calls* to the function **f** should be executed—either

- 'by' ($\Leftarrow$) invoking an eliminator;
- or 'with' ($\mid$) the result of an intermediate computation added to the data under scrutiny;
- or returning ($\mapsto$) the value of a given expression once enough analysis has been done. 'Returns' $lhs \mapsto expr$ are leaves in the program structure.

To aid readability in this paper, we adopt informal spacing and layout conventions which are inevitably more sustainable in LaTeX than in `ASCII`. For example, we tend to show the hierarchical structure of programs by indentation rather than brackets and semicolons. Also, from time to time (*e.g.* in the code for **elem**), we use vertical alignment to avoid the repetition of unchanged patterns from the *lhs* of a program to those of its subprograms. We shall shortly show how programs determine the syntactic structure of their subprograms, and hence that some such convention can be implemented; we omit any further detailed discussion of such pragmatics.

2.2 From Programs to UTT

We explain the concrete syntax by **elaboration** into the underlying type theory, but to do this, we will have to augment the abstract syntax of UTT (see Figure 3).

$$
\begin{array}{rcl}
\mathit{term} & := & \ldots \\
& | & \mathit{did} \mid \mathit{cid} \\
& | & \langle \mathit{label} : \mathit{term} \rangle \\
& | & \underline{\mathrm{call}}\ \langle \mathit{label} \rangle\ \mathit{term} \\
& | & \underline{\mathrm{return}}\ \mathit{term}
\end{array}
\qquad
\mathit{label} \ := \ \mathit{fid}\ \mathit{term}^{*}\ (\mid \mathit{term})^{*}
$$

Fig. 3. Abstract syntax extensions for elaborating programs

The underlying functional core must be extended with the datatype and constructor names, and to explain the distinguished calls and returns of functions, we introduce:

- labels, $\mathit{label} \ := \ \mathit{fid}\ \mathit{term}^{*}\ (\mid \mathit{term})^{*}$, which elaborate the category *lhs*;
- labelled calls, $\underline{\mathrm{call}}\ \langle \mathit{label} \rangle\ \mathit{term}$, which associate a term with an elaborated *lhs*;
- and their correspond returns, $\underline{\mathrm{return}}\ \mathit{term}$;
- and labelled types, $\langle \mathit{label} : \mathit{term} \rangle$;

This last construct $\langle l : T \rangle$ is used to label a type T with a function invocation l which, when executed, should return a value in T. We call these labelled types **programming problems**: they are solved by elaborating *program*s.

Digression: programming problems in Lego To give an idea of our underlying motivation for labelled types, consider the following trick which you can play even in implementations of raw type theory such as Coq or Lego: suppose you want to implement the addition function $(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. You might start with this type as a top-level goal, and invoking $\mathbb{N}$**-elim**, get back the subgoals

$$
\begin{array}{l}
? \ : \ \mathbb{N} \to \mathbb{N} \\
? \ : \ \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}
\end{array}
$$

(the precise form of the interaction is not at issue here). Which instance of $\mathbb{N}$ is which? If you are unsure, it is rather easy to finish the job with a well-typed term which does not quite add up! Suppose instead that you rephrase the goal, as follows, via a defined function **Plus** which is *vacuous* in its arguments:

$$
\begin{array}{l}
\mathbf{Plus} \ \mapsto \ \lambda x, y : \mathbb{N}.\ \mathbb{N} \ : \ \mathbb{N} \to \mathbb{N} \to \star \\
? \ : \ \forall x, y : \mathbb{N}.\ \mathbf{Plus}\ x\ y
\end{array}
$$

If you normalize the goal, you can see it is just as before. With the unreduced goal, invoking $\mathbb{N}$**-elim** now yields two subgoals

$$
\begin{array}{l}
? \ : \ \forall y : \mathbb{N}.\ \mathbf{Plus}\ \mathsf{0}\ y \\
? \ : \ \forall x : \mathbb{N}.\ (\forall z : \mathbb{N}.\ \mathbf{Plus}\ x\ z) \ \to \ \forall y : \mathbb{N}.\ \mathbf{Plus}\ (\mathsf{s}x)\ y
\end{array}
$$

Again, the normal forms of these subgoals are as before, but unreduced, they tell you exactly which $\mathbb{N}$ is which. Each subgoal shows you the 'pattern' to which it corresponds: in the base case, you are asked to solve the problem "what is $\mathsf{0} + y$?", and in the step case, "what is $(\mathsf{s}x) + y$?", the inductive hypothesis shows you which are the allowable recursive calls, in this case $x + z$ for any z. *(End of digression).*

$$\boxed{\mathit{context} \;\vdash\; \mathit{label}\ \underline{\text{label}}}$$

$$\frac{\Gamma \vdash \underline{\text{valid}}}{\Gamma \vdash \mathbf{f}\ \underline{\text{label}}} \qquad \frac{\Gamma \vdash l\ \underline{\text{label}} \quad \Gamma \vdash t : T}{\Gamma \vdash l\ t\ \underline{\text{label}}} \qquad \frac{\Gamma \vdash l\ \underline{\text{label}} \quad \Gamma \vdash t : T}{\Gamma \vdash l \mid t\ \underline{\text{label}}}$$

$$\boxed{\mathit{context} \;\vdash\; \mathit{term} \;:\; \mathit{term}}$$

$$\frac{\Gamma \vdash l\ \underline{\text{label}} \quad \Gamma \vdash T : \star_n}{\Gamma \vdash \langle l : T\rangle : \star_n}$$

$$\frac{\Gamma \vdash l\ \underline{\text{label}} \quad \Gamma \vdash t : T}{\Gamma \vdash \underline{\text{return}}\ t : \langle l : T\rangle} \qquad \frac{\Gamma \vdash t : \langle l : T\rangle}{\Gamma \vdash \underline{\text{call}}\ \langle l\rangle\ t : T}$$

$$\boxed{\mathit{context} \;\vdash\; \mathit{term} \leadsto \mathit{term}}$$

$$[\rho] \qquad \frac{}{\Gamma \vdash \underline{\text{call}}\ \langle l\rangle\ (\underline{\text{return}}\ t) \leadsto t}$$

Fig. 4. Typing and conversion extensions

The vacuous arguments of **Plus** echo the use of *phantom types* in Haskell (Leijen & Meijer, 1999). These arguments enrich the descriptive power of the type, giving a more discriminating account of the purpose of its values—not just their representation. In much the same way, we distinguish $\langle l : T\rangle$ and T, and use this to manage the process of typechecking and elaborating programs by stratifying their return types, labelling them with the function calls to which they correspond.

The elaboration process relies on computation within labels, so the terms they contain must be well-typed—this is enforced by a label well-formedness judgment, $\boxed{\Gamma \vdash l\ \underline{\text{label}}}$. We give a very simple, and intuitively appealing, operational semantics to abstract call and return, by extending the reduction relation with ρ-reductions (ρ for 'return'). The new rules are shown in Figure 4.

Each program construct in our notation either refines problems into subproblems or solves them outright. For nontrivial problems, solving at a leaf is achieved by 'filling in the right-hand side' with the term whose value is to be returned. If every leaf is solved outright, then the program successfully elaborates. Such a model of successful elaboration lends itself to a fully-fledged account of type-directed *interactive* program development—with all the armoury of techniques currently employed in implementations of type theory at our disposal. We will return to this point later.

We explain which high-level programs and expressions successfully elaborate with these new judgment forms:

$\boxed{\Gamma \Vdash \ell \triangleright l}$ 'left-hand side ℓ elaborates to label l';
$\boxed{\Gamma \Vdash e \triangleright t : T}$ 'expression e elaborates to well-typed term t of type T';
$\boxed{\Gamma|\Delta \Vdash p \triangleright t : \langle l : T\rangle}$ 'in global context Γ, and local context Δ of pattern bindings, program p elaborates to well-typed term t of labelled type $\langle l : T\rangle$';
$\boxed{\Gamma \Vdash d \triangleright \Delta}$ 'in context Γ, declaration d elaborates to new context bindings Δ'.

$$\boxed{\mathit{context} \Vdash \mathit{lhs} \rhd \mathit{label}}$$

$$\frac{}{\Gamma \Vdash \mathbf{f} \rhd \mathbf{f}} \qquad \frac{\Gamma \Vdash \ell \rhd l \quad \Gamma \Vdash e \rhd t : T}{\Gamma \Vdash \ell\, e \rhd l\, t} \qquad \frac{\Gamma \Vdash \ell \rhd l \quad \Gamma \Vdash e \rhd t : T}{\Gamma \Vdash \ell \mid e \rhd l \mid t}$$

$$\boxed{\mathit{context} \Vdash \mathit{expr} \rhd \mathit{term} : \mathit{term}}$$

$$\frac{\Gamma \vdash \underline{\text{valid}}}{\Gamma \Vdash \star \rhd \star_n : \star_{n+1}} \qquad \ldots \qquad \frac{\Gamma \Vdash e \rhd t : S \quad \Gamma \vdash S \preceq T}{\Gamma \Vdash e \rhd t : T}$$

$$[\text{call}] \quad \frac{\Gamma \Vdash c \rhd l \quad \text{LOOKUP}(l, \Gamma) \Longrightarrow (t : \langle l : T \rangle)}{\Gamma \Vdash c \rhd \underline{\text{call}}\ \langle l \rangle\ t : T}$$

[view] See Section 6

Fig. 5. Elaboration of left-hand sides and expressions (edited highlights)

$$\boxed{\mathit{context} | \mathit{context} \Vdash \mathit{expr} \rhd \mathit{term} : \langle \mathit{label} : \mathit{term} \rangle}$$

$$\frac{\Gamma | \Delta \Vdash p \rhd t : \langle l : S \rangle \quad \Gamma ; \Delta \vdash S \preceq T}{\Gamma | \Delta \Vdash p \rhd t : \langle l : T \rangle}$$

$$[\text{return}] \quad \frac{\Gamma ; \Delta \Vdash \ell \rhd l \quad \Gamma ; \Delta \Vdash e \rhd t : T}{\Gamma | \Delta \Vdash \ell \mapsto e \rhd \underline{\text{return}}\, t : \langle l : T \rangle}$$

[by] See Section 4 [with] See Section 5

Fig. 6. Elaboration of programs

Interpretation We intend the judgments for elaboration of high-level programs and those of the type theory to be connected by the following soundness properties, which we conjecture follow by simple induction on the rules, together with the analysis we provide below of the elaboration rules for the various constructs:

soundness for	elaboration judgment	yields	underlying judgment
labels	$\Gamma \Vdash \ell \rhd l$	$\Rightarrow$	$\Gamma \vdash l\ \underline{\text{label}}$
expressions	$\Gamma \Vdash e \rhd t : T$	$\Rightarrow$	$\Gamma \vdash t : T$
declarations	$\Gamma \Vdash d \rhd \Delta$	$\Rightarrow$	$\Gamma ; \Delta \vdash \underline{\text{valid}}$
programs	$\Gamma | \Delta \Vdash p \rhd t : \langle l : T \rangle$	$\Rightarrow$	$\Gamma ; \Delta \vdash t : \langle l : T \rangle$

We hope to expand on such meta-theoretical treatment in future work; for now it suffices to observe that we obtain a naïve operational semantics for programs, simply by taking normal forms of elaborated terms.

The basic structural rules for left-hand sides and expressions are summarised in Figure 5; we only give selected instances of the rules for expressions, noting that we may incorporate into both forms the use of such notational conveniences as infix operators, Pollack-style implicit syntax and universe level inference, and the omission of domain types from binders where they can be inferred from usage. Of course, the real work is done by the remaining rules which explain the elaboration of the main programming constructs.

$$\boxed{\mathit{context} \Vdash \mathit{decl} \rhd \mathit{context}}$$

[data] See Subsection 3.2

$$[\text{let}] \quad \frac{\Gamma \Vdash \forall\Phi.\, R \rhd \forall\Delta.\, T \;:\; \star \qquad \Gamma|\Delta \Vdash p \rhd t \;:\; \langle \mathbf{f}\,\Delta : T\rangle}{\Gamma \Vdash \underline{\text{let}}\, \dfrac{\Phi}{\mathbf{f}\,\Phi \;:\; R}\, p \;\rhd\; f \mapsto \lambda\Delta.\, t : \forall\Delta.\, \langle \mathbf{f}\,\Delta : T\rangle}$$

Fig. 7. Elaboration of declarations

We explain how the elaboration of a datatype declaration extends the context with new bindings, in Section 3. Likewise, we defer the discussion of 'by' until Section 4, as it requires some considerable analysis—this is the heart of our account of 'structured decomposition on the left'. The elaboration rule for 'with' is explained in Section 5; in effect it constructs a 'helper function' with an extended label.

Return from a call is straightforward to explain—rule [return], Figure 6; the elaborated right-hand side is returned, packaged with the label which elaborates the left-hand side. Given $t \;:\; T$, the problem $\langle l : T\rangle$ is solved outright.

The rule for declaring a function (see Figure 7) whose type $\forall\Phi.\, R$ and body p successfully elaborate, binds a new definition into the context: a λ-abstracted term whose type offers solutions to a class of programming problems—those whose labels represent calls to the function. For example, we may define **snoc** in terms of $+\!\!+$ ('append') as follows:

$$\underline{\text{let}} \quad \frac{\mathit{xs} \;:\; \mathsf{List}\, X \qquad x \;:\; X}{\mathbf{snoc}\, \mathit{xs}\, x \;:\; \mathsf{List}\, X} \qquad \mathbf{snoc}\, \mathit{xs}\, x \;\mapsto\; \mathit{xs} +\!\!+ (x :: [])$$

Here, the [return] rule demands that $\mathit{xs} +\!\!+ (x :: []) \;:\; \mathsf{List}\, X$, to ensure that the equation solves the top-level problem $\langle \mathbf{snoc}\, \mathit{xs}\, x : \mathsf{List}\, X\rangle$. We could write all our programs this way by applying elimination operators in gory detail 'on the right'. However, our notation exists to hide this detail, treating elimination 'on the left'.

Meanwhile, the [call] rule uses the partial (but terminating) meta-operation LOOKUP, to search the context for a variable which can be applied to deliver a solution to a programming problem with a given label—as delivered by definition. Similarly, whilst elaborating a recursive program via an induction principle, the local context will contain inductive hypotheses which 'advertise' the recursive calls they enable via labelled types, just as in our **Plus** example above.

The LOOKUP mechanism thus corresponds to a simple proof tactic—like `Immed` in LEGO. We defer its definition until Subsection 4.1, by which time the structure of inductive hypotheses will have been made precise. For now, we can say that if Γ contains an elaborated definition, $f \mapsto \cdots : \forall\Delta.\, \langle \mathbf{f}\,\Delta : T\rangle$ and $\vec{t} \;:\; \Delta$, then certainly

$$\text{LOOKUP}(\mathbf{f}\,\vec{t}, \Gamma) \implies (f\,\vec{t} : \langle \mathbf{f}\,\vec{t} : \Downarrow\underline{\text{let}}\,\Delta \mapsto \vec{t}.\, T\rangle)$$

Strictly speaking, this permits the elaboration of calls to defined functions only at exactly the arity in their signature. However, given that this arity has been specified, it is a simple matter for the elaborator to handle a call at any arity: calls which are too long becomes applications of calls; calls which are too short get η-expanded, λ-abstracting the extra arguments required.

3 Datatype families, eliminators and computation

We declare families of **datatypes** in our language by giving type signatures for the **type constructor** symbol and for its **data constructors**, in the format

$$\underline{\text{data}} \;\; \textit{type-constructor-signature} \quad \underline{\text{where}} \;\; \textit{data-constructor-signatures}$$

Simple monomorphic datatypes fit this pattern. For example, Unit and Bool:

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

$$\underline{\text{data}} \quad \frac{}{\mathsf{Bool}\ :\ \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{true}\ :\ \mathsf{Bool}} \quad \frac{}{\mathsf{false}\ :\ \mathsf{Bool}}$$

Note that we write both type and data constructors sans serif. Signatures usually take the form of natural deduction rules: for each new symbol, we give the context which types its arguments above the line, and the type of the symbol applied to those arguments below. Examples include Cartesian products and lists:

$$\underline{\text{data}} \quad \frac{A, B\ :\ \star}{A \times B\ :\ \star} \quad \underline{\text{where}} \quad \frac{a\ :\ A \quad b\ :\ B}{(a, b)\ :\ A \times B}$$

$$\underline{\text{data}} \quad \frac{X\ :\ \star}{\mathsf{List}\,X\ :\ \star} \quad \underline{\text{where}} \quad \frac{}{[]\ :\ \mathsf{List}\,X} \quad \frac{x\ :\ X \quad xs\ :\ \mathsf{List}\,X}{x :: xs\ :\ \mathsf{List}\,X}$$

$\mathsf{List}\,X$ is defined *uniformly* for any X and makes recursive references only to $\mathsf{List}\,X$. Such a **parametric** declaration introduces a collection of datatypes each actual instance of which could, more tediously, be declared by itself. **Families** of datatypes (Dybjer, 1991) generalize parametric datatypes in two ways. Firstly, they are *non-uniform*: each data constructor targets a *subset* of the type constructor's possible arguments—Dybjer calls these arguments **indices** when they are used in this non-uniform way. The So family mentioned earlier is a simple example:

$$\underline{\text{data}} \quad \frac{b\ :\ \mathsf{Bool}}{\mathsf{So}\ b\ :\ \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{oh}\ :\ \mathsf{So\,true}}$$

Secondly, datatype families are *mutually* declared: a constructor for one subset of the indices may refer recursively to other such subsets. A suitable example is the family of heterogeneous association lists ('a-lists') *with a specified domain* of Labels:

$$\underline{\text{data}} \quad \frac{ls\ :\ \mathsf{List\,Label}}{\mathsf{HAL}\ ls\ :\ \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{hnil}\ :\ \mathsf{HAL}\,[]}$$

$$\frac{l\ :\ \mathsf{Label} \quad x\ :\ X \quad h\ :\ \mathsf{HAL}\ ls}{\mathsf{hcons}_X\ l\ x\ h\ :\ \mathsf{HAL}\,(l :: ls)}$$

Here, hnil represents the empty a-list, with empty domain, and hcons adds a new association, of the *value* x, of *type* X, with label l to an existing a-list h with domain ls, yielding an a-list with domain $l :: ls$. Incidentally, we could easily require distinct labels by giving hcons an extra argument in $\mathsf{So}\,(\mathbf{not}\,(\mathbf{elem}\,l\,ls))$.

More generally, we permit datatype family declarations of this general form:

$$\underline{\text{data}} \quad \frac{\Phi}{\mathsf{D}\,\Phi\ :\ \star} \quad \underline{\text{where}} \quad \frac{\Phi_1}{\mathsf{c}_1\ \Phi_1\ :\ \mathsf{D}\,\vec{e}_1} \quad \cdots \quad \frac{\Phi_n}{\mathsf{c}_n\ \Phi_n\ :\ \mathsf{D}\,\vec{e}_n} \qquad (\dagger)$$

The $\vec{e}_i$ may differ from Φ and each other, hence a Haskell/Cayenne-style

```
data  D x y z ... = C1 ... | ... | Cn ...
```

will not serve. It is also why datatype families are so powerful. Correspondingly, case analysis on datatype families is rather more subtle than on simple datatypes. As with function type signatures, if $\forall\Phi.\ \star \rhd \forall\Theta.\star$ and $\forall\Phi_i.\,\mathsf{D}\,\vec{e}_i \rhd \forall\Delta_i.\,\mathsf{D}\,\vec{s}_i$, then we obtain $\mathsf{D} : \forall\Theta.\star$ and $\mathsf{c}_i : \forall\Delta_i.\,\mathsf{D}\,\vec{s}_i$.

Remark For readability, we adopt the typographical convention that arguments with inferrable types need not be declared explicitly in a type signature's premises—*e.g.* $X : \star$ and $ls : \mathsf{List\,Label}$ in the declaration of hcons. The missing declarations are inserted (with Pollack-style implicit quantification) among the elaborated context of arguments—we may subscript such an argument in the conclusion to determine where it goes. The signature for hcons elaborates to

$$\mathsf{hcons} \quad : \quad \forall_{X:\star}.\forall_{ls:\mathsf{List\,Label}}.\,\forall l{:}\mathsf{Label}.X \to \mathsf{HAL}\ ls \to \mathsf{HAL}\ (l :: ls)$$

This convention is implementable, by augmenting Pollack's techniques, but the details are beyond the scope of this paper. *(End of remark).*

Dependency in type families allows us to specify operations which enforce additional safety constraints *by typing alone*. For example, we can ensure that projections from an a-list apply only to labels in its domain:

$$\underline{\text{let}} \quad \frac{k\ :\ \mathsf{Label} \qquad h\ :\ \mathsf{HAL}\ ls \qquad p\ :\ \mathsf{So}\,(\mathbf{elem}\,k\ ls)}{\mathbf{typeProj}\ k\ h\ p\ :\ \star} \qquad \cdots$$

$$\underline{\text{let}} \quad \frac{k\ :\ \mathsf{Label} \qquad h\ :\ \mathsf{HAL}\ ls \qquad p\ :\ \mathsf{So}\,(\mathbf{elem}\,k\ ls)}{\mathbf{valProj}\ k\ h\ p\ :\ \mathbf{typeProj}\ k\ h\ p}$$

We develop these operations as a running example: in Subsection 3.1 below, we explore the impact of dependent case analysis on the types which arise, and in Subsection 5.1, the necessary coupling between intermediate computations and types. It is worth noting that there are other presentations of heterogeneous a-lists: we could index them by *signatures* in $\mathsf{List}\,(\mathsf{Label} \times \star)$, or we could index signatures by domain, then a-lists by signatures. Indeed, this example takes its cue from problems originally encountered by Pollack in his codings of *records* in which later field types depend on earlier field values (Pollack, 2000). In all of these variations, we find the same problems—and the same solutions.

3.1 Working with datatype families

In this section, we examine the interaction between case analysis and types—clearly nontrivial where a function's return type depends on its argument, but still more interesting once datatype families become involved. Although not yet defined, we use our high-level notation to facilitate the discussion of our examples. Our purpose here is to examine the phenomena which arise in these programs, and which must be addressed in the design of *any* notation for them.

For many simple programs, there is no interaction between case analysis and types, just as in standard functional programming. The familiar **elem** function contains two case-splits (on a $\mathsf{List\ Label}$ and on a Bool) neither of which affects types:

$$\underline{\text{let}}\quad \frac{k\ :\ \mathsf{Label}\qquad ls\ :\ \mathsf{List\ Label}}{\mathbf{elem}\,k\,ls\ :\ \mathsf{Bool}}\qquad \begin{array}{ll} \mathbf{elem}\,k\ [\,] & \mapsto\ \mathsf{false} \\ \mathbf{elem}\,k\ (l :: ls) & \left|\ \begin{array}{lll} k == l & & \\ \mathsf{true} & \mapsto & \mathsf{true} \\ \mathsf{false} & \mapsto & \mathbf{elem}\,k\,ls \end{array}\right. \end{array}$$

Examining a value from an indexed datatype family is just as straightforward if its indices may vary *freely*. In a function with type $\forall\Theta.\,\forall x:\mathsf{D}\,\Theta.\ T$, x could come from any constructor. If T does not depend on Θ or x, it will be unaffected. For example, we may compute a signature from a heterogeneous a-list:

$$\underline{\text{let}}\quad \frac{h\ :\ \mathsf{HAL}\ ls}{\mathbf{hSig}\ :\ \mathsf{List}\ (\mathsf{Label}\times\star)}$$
$$\begin{array}{lll} \mathbf{hSig} & \mathsf{hnil} & \mapsto\ [\,] \\ \mathbf{hSig} & (\mathsf{hcons}_X\ l\ x\ h') & \mapsto\ (l, X) :: (\mathbf{hSig}\ h') \end{array}$$

Once a function space depends even on a simply-typed argument, case analysis can change the return type—a phenomenon new to functional programming. For example, given a value and a list of labels, we can compute the a-list binding each label to the value:

$$\underline{\text{let}}\quad \frac{x\ :\ X\qquad ls\ :\ \mathsf{List\ Label}}{\mathbf{repeat}\,x\,ls\ :\ \mathsf{HAL}\ ls}\qquad \begin{array}{ll} \mathbf{repeat}\,x\ [\,] & \mapsto\ \mathsf{hnil} \\ \mathbf{repeat}\,x\ (l :: ls) & \mapsto\ \mathsf{hcons}\ l\ x\ (\mathbf{repeat}\,x\,ls) \end{array}$$

The return type is indexed by the list, so the more we learn about the list, the more we know about what to return. In the $[\,]$ case, the right-hand side must have type $\mathsf{HAL}\ [\,]$—hnil is the only candidate; in the step case, we need a $\mathsf{HAL}\ (l :: ls)$, which suggests applying $\mathsf{hcons}\ l$. No constructor makes a $\mathsf{HAL}\ ls$ for unknown ls, but the more of ls we can see on the left, the more we can do on the right.

When analysing values from a datatype family, *constraining* the choice of indices can rule out some cases. For example, we may shorten a *nonempty* a-list:

$$\underline{\text{let}}\quad \frac{h\ :\ \mathsf{HAL}\ (l :: ls)}{\mathbf{hTail}\,h\ :\ \mathsf{HAL}\ ls}\qquad \mathbf{hTail}\ (\mathsf{hcons}\ l\ x\ h')\ \mapsto\ h'$$

Why is there no case for hnil? Because there is no way hnil can make an inhabitant

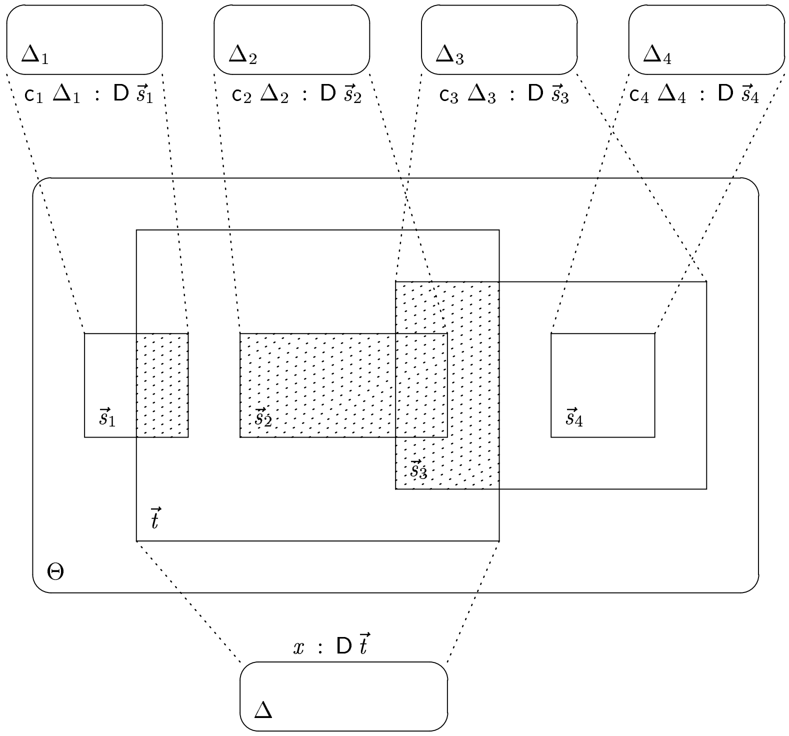

Fig. 8. Constrained case analysis on a datatype family

of HAL $(l :: ls)$! The type discipline ensures that we need only return values for constructors delivering elements whose indices lie in the subset under scrutiny. Further, a constructor may deliver suitable elements only from a portion of its domain. More generally, suppose we are writing a function $\mathbf{f}$ whose type is

$$\mathbf{f}\ :\ \forall\Delta.\,\forall x\!:\!\mathsf{D}\ \vec{t}.\ T$$

by case analysis on x, where family $\mathsf{D}\,\Theta : \star$ has constructors $\mathsf{c}_i\,\Delta_i : \mathsf{D}\,\vec{s}_i$. As Coquand observes in (Coquand, 1992), we need consider not the whole of $\mathsf{D}\ \Theta$, nor even the whole of $\mathsf{D}\ \vec{t}$, but the *intersection* between $\mathsf{D}\ \vec{t}$ and each of the $\mathsf{D}\ \vec{s}_i$ in turn, as illustrated in Figure 8.

In this hypothetical example, constructor c_4 is ruled out, just as hnil was for **hTail**, whilst every value returned by c_2 lies within $\mathsf{D}\ \vec{t}$, as was the case with hcons. However, we need only consider $\mathsf{c}_1\ \Delta_1$ for a subset of its possible arguments—those Δ_1 which make $\vec{s}_1$ coincide with $\vec{t}$—and similarly for c_3. Moreover, for each c_i, we need only consider instances of Δ—$\mathbf{f}$'s arguments—which make $\vec{t}$ coincide with $\vec{s}_i$.

This is a real departure for functional programming. Analysing one input x can not only deliver a restricted set of constructor patterns with some of their arguments already determined; it can also have a *non-local* impact, determining the values of *other* inputs on which the type of x depends. These instantiations may in turn change the types of still other inputs, and possibly even the return type of the function. Examples of these phenomena are found in our definition of **typeProj**:

$$
\underline{\mathrm{let}}\quad \frac{k \;:\; \mathsf{Label} \qquad h \;:\; \mathsf{HAL}\, ls \qquad p \;:\; \mathsf{So}\,(\mathbf{elem}\, k\, ls)}{\mathbf{typeProj}\, k\, h\, p \;:\; \star}
$$

$$
\begin{array}{ll}
\mathbf{typeProj}\, k \quad \mathsf{hnil} \quad p & \Leftarrow \mathsf{So}\text{-}\mathbf{case}\, p \\
\mathbf{typeProj}\, k\, (\mathsf{hcons}_X\; l\; x\; h')\; p & \mid\; k == l \\
 & \quad \mathsf{true} \;\mapsto\; X \\
 & \quad \mathsf{false} \;\mapsto\; \mathbf{typeProj}\, k\, h'\, p
\end{array}
$$

Analysing the $h : \mathsf{HAL}\ ls$ argument gives two cases. In the case where h is hnil, we also learn—by typing, not testing—that ls is $[]$. Hence p's type in this case is really $\mathsf{So\,false}$. The notation $\Leftarrow$ So-**case** p, introduced formally in Section 4, then invokes case analysis of p revealing no possible constructor—k cannot occur in $[]$, so there is no projection to define!

The hcons case is still more interesting: the 'information for free' here is that the domain must be $l :: ls'$, and the tail $h' : \mathsf{HAL}\ ls'$. Moreover, $p : \mathsf{So}\,(\mathbf{elem}\, k\, (l :: ls'))$. Now, $\mathbf{elem}\, k\, (l :: ls')$ is computed by testing the result of an intermediate call to $k == l$. Hence, when **typeProj** analyses $k == l$, it learns, again for free, yet more about the type of p. In the true case, this does not matter as label k has been found; in the false case, p's type becomes $\mathsf{So}\,(\mathbf{elem}\, k\, ls')$—exactly the prerequisite for the recursive call, $\mathbf{typeProj}\, k\, h'\, p$.

As you can see, some careful choreography is required to keep the testing performed by **typeProj** in step with the testing performed by its type. The '$\mid k == l$' clause not only makes the result of the test available for analysis, it abstracts that result from the type of p. We give the exact details of its elaboration in Section 5.

The **valProj** function carries out exactly the same analyses as **typeProj**:

$$
\underline{\mathrm{let}}\quad \frac{k \;:\; \mathsf{Label} \qquad h \;:\; \mathsf{HAL}\, ls \qquad p \;:\; \mathsf{So}\,(\mathbf{elem}\, k\, ls)}{\mathbf{valProj}\, k\, h\, p \;:\; \mathbf{typeProj}\, k\, h\, p}
$$

$$
\begin{array}{ll}
\mathbf{valProj}\, k \quad \mathsf{hnil} \quad p & \Leftarrow \mathsf{So}\text{-}\mathbf{case}\, p \\
\mathbf{valProj}\, k\, (\mathsf{hcons}\; l\; x\; h')\; p & \mid\; k == l \\
 & \quad \mathsf{true} \;\mapsto\; x \\
 & \quad \mathsf{false} \;\mapsto\; \mathbf{valProj}\, k\, h'\, p
\end{array}
$$

This is no idle coincidence. Each case-split in **valProj** also instantiates the return type computed by **typeProj**. This is unremarkable in the hnil case: p's type is empty anyway, just as before. For the hcons case, the subsequent analysis of $k == l$ now delivers the value not only of the same test in the type of p, but also in the **typeProj** call, by which the return type is computed. Correspondingly, where x is returned in the true case, the return type really is X. In the false case, we must return an element of $\mathbf{typeProj}\, k\, h'\, p$, which is exactly the type of $\mathbf{valProj}\, k\, h'\, p$.

We may summarize the interactions between case-splits and types observed in this section, by means of the following table. We categorize the examples, firstly by the type of the argument being analysed and secondly by the degree of dependency in the function space where the analysis occurs. In each meaningful category, we name an example with the stated dependency and give the argument type.

arg's type / dependency	simple D	free $\mathsf{D}\,\Phi$	constrained $\mathsf{D}\,\vec{t}$
none	[**elem**] $\mathsf{List\ Label}$	[**hSig**] $\mathsf{HAL}\ ls$	[**typeProj**] $\mathsf{So\ false}$
on indices	not applicable	[**typeProj**] $\mathsf{HAL}\ ls$	[**hTail**] $\mathsf{HAL}\ (l :: ls)$
on arg itself	[**repeat**] $\mathsf{List\ Label}$	[**valProj**] $\mathsf{HAL}\ ls$	[**valProj**] $\mathsf{So\ false}$

Programming in Hindley-Milner systems never strays beyond the top left corner of this table. Recent experiments with polymorphic recursion on *nested* types (Bird & Meertens, 1998) begin to stray into the second row, although the indices affected are always type parameters rather than actual data arguments. Further, the uniform '$\underline{\text{data}}\,\mathsf{D}\,\Theta\ =\ \ldots$' style of family means that constructors can never be ruled out by analysing a constrained $\mathsf{D}\,\vec{t}$, nor can a particular choice of constructor tell us more about the indices $\vec{t}$, as the intersection of the whole set Θ with $\vec{t}$ is just $\vec{t}$ itself.

As we work towards the more powerful techniques and programs inhabiting the bottom right corner, we must confront a number of new issues:

- How do we handle the effects of analysing one argument on other arguments and on types?
- How do we handle the potential complexity of the intersections between nontrivial argument types $\mathsf{D}\,\vec{t}$ and nontrivial constructor ranges $\mathsf{D}\,\vec{s}_i$?
- How do we handle the impact *on types* of analysing the result of an intermediate computation?

The notation we introduce in this paper is a step towards addressing these questions. However, before we present the elaboration of the programming constructs, let us be precise about the presentation of datatype families in the underlying type theory.

3.2 Elaborating data *declarations*

These '$\underline{\text{data}}$' declarations (†) of Section 3 elaborate to context extensions by the rules in Figure 9; the new bindings declare the type- and data-constructors, together with the elimination operator D**-elim**, specifying which recursive computations are permitted over instances of $\mathsf{D}\,\Theta$. The meta-operation $\textsc{hyps}(P, \Delta)$ computes the appropriate contexts of **inductive hypotheses**. Elimination operators acquire computational behaviour by extending the conversion judgment of the type theory with the 'ι-reduction' scheme.

As observed in (Callaghan & Luo, 2000), ι-reduction need not be implemented by naïve pattern matching (as it is in LEGO (Pollack, 1994)). A simple switch on the constructor c_i, in the style of Augustsson (Augustsson, 1985), suffices for the safe execution of *well-typed* programs.

$$\boxed{\mathit{context} \ \Vdash \ \mathit{decl} \ \rhd \ \mathit{context}}$$

$$[\text{data}]\quad \frac{\begin{array}{l} \Gamma \Vdash \forall\Phi.\star \ \rhd \ \forall\Theta.\star \ : \ \star \\ \Gamma;\ \mathsf{D}:\forall\Theta.\star \ \Vdash \ \forall\Phi_i.\,\mathsf{D}\,\vec{e}_i \ \rhd \ \forall\Delta_i.\,\mathsf{D}\,\vec{s}_i \ : \ \star \qquad (1 \le i \le n) \\ \text{for each } x : T \text{ in each } \Delta_i, \text{if } \mathsf{D} \in T \text{ then for some } \vec{u},\ T \text{ is } \mathsf{D}\,\vec{u} \end{array}}{\begin{array}{l} \Gamma \Vdash \underline{\text{data}}\ \dfrac{\Phi}{\mathsf{D}\,\Phi \ : \ \star} \ \underline{\text{where}}\ \dfrac{\Phi_1}{\mathsf{c}_1\ \Phi_1 \ : \ \mathsf{D}\,\vec{e}_1} \ \cdots \ \dfrac{\Phi_n}{\mathsf{c}_n\ \Phi_n \ : \ \mathsf{D}\,\vec{e}_n} \\ \quad \rhd\ \mathsf{D}:\forall\Theta.\,\star;\ \ \mathsf{c}_1:\forall\Delta_1.\,\mathsf{D}\,\vec{s}_1;\ \ \ldots\ ;\ \ \mathsf{c}_n:\forall\Delta_n.\,\mathsf{D}\,\vec{s}_n; \\ \quad \begin{array}{lll} \mathsf{D}\textbf{-elim} \ : & \forall_\Theta;\, x : \mathsf{D}\,\Theta. & \textbf{targets} \\ & \forall P{:}\forall_\Theta;\, x : \mathsf{D}\,\Theta.\ \star\,. & \textbf{motive} \\ & \left.\begin{array}{l} \forall m_1{:}\forall\Delta_1;\,\textsc{hyps}(P,\Delta_1).\ P\ (\mathsf{c}_1\ \vec{s}_1). \\ \quad\vdots \\ \forall m_n{:}\forall\Delta_n;\,\textsc{hyps}(P,\Delta_n).\ P\ (\mathsf{c}_n\ \vec{s}_n). \end{array}\right\} & \textbf{methods} \\ & P\ x & \end{array} \end{array}}$$

$$\begin{array}{lrcll} \text{where} & \textsc{hyps}(P,\cdot) & \Longrightarrow & \cdot & \\ & \textsc{hyps}(P, r:\mathsf{D}\,\vec{u};\ \Delta) & \Longrightarrow & r' : P\ r;\ \textsc{hyps}(P,\Delta) & \\ & \textsc{hyps}(P, a:A;\ \Delta) & \Longrightarrow & \textsc{hyps}(P,\Delta) & \text{otherwise} \end{array}$$

$$\boxed{\mathit{context} \ \vdash \ \mathit{term} \leadsto \mathit{term}}$$

$$[\iota]\quad \frac{}{\Gamma;\,\mathsf{D}\textbf{-elim}:\ldots;\Gamma' \ \vdash \ \mathsf{D}\textbf{-elim}\ (\mathsf{c}_i\ \Delta_i)\ P\ \vec{m} \leadsto m_i\ \Delta_i\ \textsc{recs}(P,\vec{m},\Delta_i)}$$

$$\begin{array}{lrcll} \text{where} & \textsc{recs}(P,\vec{m},\Delta_i) & : & \textsc{hyps}(P,\Delta_i) & \\ & \textsc{recs}(P,\vec{m},\cdot) & \Longrightarrow & \varepsilon & \\ & \textsc{recs}(P,\vec{m}, r:\mathsf{D}\,\vec{u};\ \Delta) & \Longrightarrow & (\mathsf{D}\textbf{-elim}\ r\ P\ \vec{m});\ \textsc{recs}(P,\vec{m},\Delta) & \\ & \textsc{recs}(P,\vec{m}, a:A;\ \Delta) & \Longrightarrow & \textsc{recs}(P,\vec{m},\Delta) & \text{otherwise} \end{array}$$

Fig. 9. Elaboration of datatype declarations

For $\mathbb{N}$, declared by $\underline{\text{data}}\ \dfrac{}{\mathbb{N} \ : \ \star}\ \ \underline{\text{where}}\ \ \dfrac{}{\mathsf{0} \ : \ \mathbb{N}}\ \ \dfrac{n \ : \ \mathbb{N}}{\mathsf{s}n \ : \ \mathbb{N}}$, we obtain

$$\begin{array}{l} \mathbb{N}:\star;\ \ \mathsf{0}:\mathbb{N};\ \ \mathsf{s}:\mathbb{N}\to\mathbb{N}; \\ \mathbb{N}\textbf{-elim} \ : \ \forall x{:}\mathbb{N}.\ \forall P{:}\mathbb{N}\to\star.\ P\ \mathsf{0} \ \to\ (\forall n{:}\mathbb{N}.\ P\ n \to P\ (\mathsf{s}n)) \ \to\ P\ x \end{array}$$

$$\begin{array}{r} \mathbb{N}\textbf{-elim}\ \mathsf{0}\ P\ m_\mathsf{0}\ m_\mathsf{s} \leadsto m_\mathsf{0} \\ \mathbb{N}\textbf{-elim}\ (\mathsf{s}n)\ P\ m_\mathsf{0}\ m_\mathsf{s} \leadsto m_\mathsf{s}\ n\ (\mathbb{N}\textbf{-elim}\ n\ P\ m_\mathsf{0}\ m_\mathsf{s}) \end{array}$$

For all the examples in this paper, it is sufficient to ignore the possibility of higher-order recursive constructors and presume that all constructor argument types mentioning D have form $\mathsf{D}\,\vec{u}$. Looser recursion regimes are now standard, as are mutual definitions, but we prefer not to complicate the presentation beyond what is needed to support the present paper. Moreover it suffices to treat datatype parameters (like the X in $\mathsf{List}\,X$) the same way we treat indices: a possible optimization is to abstract them once at the outside, rather than repeatedly in the motive and methods.

4 The 'by' construct: generalized elimination

In this section, we develop the tools we need to deploy not merely the machine-generated elimination operators for datatype families, but *any* function whose type has a suitable shape. We say that a term e is a **$\Gamma|\Delta$-eliminator** and we call its type a **$\Gamma|\Delta$-eliminator type** if, for any $\Theta, \Delta_i, \vec{s}_i, \vec{t}$,

$$\begin{array}{ll} & \Gamma;\Delta \;\vdash\; e \;:\; \forall P\!:\!(\forall\Theta.\,\star).\;\; (\forall\Delta_1.\,P\;\vec{s}_1) \;\to\; \cdots \;\to\; (\forall\Delta_n.\,P\;\vec{s}_n) \;\to\; P\;\vec{t} \\ \text{and} & \Gamma;\Theta \;\vdash\; \underline{\text{valid}} \\ \text{and} & \Gamma;\; P\!:\!(\forall\Theta.\,\star)\,;\Delta_i \;\vdash\; P\;s_i \;:\; \star \qquad (1 \le i \le n) \end{array}$$

It is this central definition, and its abstract characterization of the type-shape which drives the generalization of the primitive elimination operators in type theory. We call an eliminator's first argument its **motive**—it shows what is to be gained by the elimination; the remaining arguments, we call **methods**—they show how the motive is to be achieved in each case.

An **elimination operator** is a function $f\,:\,\forall\Delta.\,E$ in Γ, such that E is a $\Gamma|\Delta$-eliminator type. We say that the Δ are f's **targets**—they explain what is to be eliminated. Our definition thus includes, but is not restricted to the basic D**-elim** operators which come with datatype families.

Note that the traditional presentation of induction principles (as in Subsection 1.1) orders the arguments: motive, methods, targets. We put the targets first, so that an elimination operator is a function from targets to eliminators. The $\Leftarrow$-construct splits a programming problem into subproblems given an *arbitrary* eliminator. Of course, if $\Gamma;\Delta \;\vdash\; x \;:\; \mathsf{D}\;\vec{t}$, then D**-elim** x is a $\Gamma|\Delta$-eliminator.

The [by] rule explains how this splitting proceeds, directed by the eliminator's type. It is shown, with other associated definitions, in Figure 10. The main work is done by the meta-operation SPLIT, computing the combinator g with which to recombine the elaborated subprograms. The account which we give here is a simplified version of those in (McBride, 1999; McBride, 2002), adequate for all the examples in this paper. Extensions covering more complex rules or more complex combinations of recursion are routine, but require more careful bookkeeping than is justified here.

We shall explain what happens, with the help of a worked example—defining **htail**

$$\underline{\text{let}}\quad \frac{h \;:\; \mathsf{HAL}\;(l :: ls)}{\mathbf{hTail}\,h \;:\; \mathsf{HAL}\;ls} \qquad \begin{array}{lcl} \mathbf{hTail} & h & \Leftarrow\; \mathsf{HAL}\text{-}\mathbf{elim}\,h \\ \quad\mathbf{hTail} & (\mathsf{hcons}\;l\;x\;h') & \mapsto\; h' \end{array}$$

where (showing the indices, but omitting other inferrable information to save space):

$$\begin{array}{ll} \mathsf{HAL}\text{-}\mathbf{elim}_{(l::ls)}\;h \;:\; & \forall P\!:\; \forall_{ls}.\,\mathsf{HAL}\;ls \to \star. \\ & P_{[]}\;\mathsf{hnil}\;\to \\ & (\forall_{X,ls'}.\,\forall l,x,h'.\;\;P_{ls'}\;h' \;\to\; P_{(l::ls')}\;(\mathsf{hcons}\;l\;x\;h')) \;\to \\ & P_{(l::ls)}\;h \end{array}$$

For P, we need a motive such that $P_{(l::ls)}\,h$ delivers an element of $\langle\mathbf{hTail}\,h\!:\!\mathsf{HAL}\;ls\rangle$.

Heterogeneous Equality

$$\frac{a \;:\; A \qquad b \;:\; B}{a \;{}_A{=}_B\; b \;:\; \star} \qquad \frac{}{\mathsf{refl}\; a \;:\; a = a} \qquad \frac{q \;:\; a \;{}_A{=}_A\; a' \quad P \;:\; \forall_{a':A}.\, a = a' \to \star \qquad m \;:\; P_a\,(\mathsf{refl}\; a)}{\mathbf{=\text{-}elim}\; q\; P\; m \;:\; P_{a'}\; q}$$

$$\boxed{\mathit{context} \;\vdash\; \mathit{term} \leadsto \mathit{term}}$$

$$[\kappa] \qquad \frac{}{\Gamma \;\vdash \mathbf{=\text{-}elim}\,(\mathsf{refl}\; a)\; P\; m \;\leadsto\; m}$$

$$\underline{\text{let}} \quad \frac{q \;:\; a \;{}_A{=}_A\; a' \qquad P \;:\; A \to \star}{\mathbf{subst}\; q\; P \;:\; P\; a \to P\; a'} \qquad \mathbf{subst}\; q\; P \;\mapsto \mathbf{=\text{-}elim}\; q\; (\lambda_{x:A}.\, \lambda_{\_}\!: a = x.\; P\; x)$$

$$\underline{\text{let}} \quad \frac{q \;:\; a \;{}_A{=}_A\; a'}{\mathbf{sym}\; q \;:\; a' \;{}_A{=}_A\; a} \qquad \mathbf{sym}\; q \;\mapsto\; \mathbf{subst}\; q\; (\lambda x\!:\!A.\; x = a)\; (\mathsf{refl}\; a)$$

Simplification for a method

$$\begin{array}{l}
\lceil m : \forall\Delta.\; t = t \;\to\; M\rceil \\
\quad\Longrightarrow \qquad \lceil m' \;:\; \forall\Delta.\; M\rceil; \\
\qquad\qquad\quad m \mapsto \lambda\Delta.\,\lambda q.\; m'\;\Delta \\
\lceil m : \forall\Delta.\; \mathsf{chalk}\;\vec{s} = \mathsf{chalk}\;\vec{t} \to M\rceil \\
\quad\Longrightarrow \qquad \lceil m' : \forall\Delta.\; \vec{s} = \vec{t} \to M\rceil; \\
\qquad\qquad\quad m \mapsto \lambda\Delta.\,\lambda q.\; \textsc{inject}\; q\;(m'\;\Delta) \\
\lceil m : \forall\Delta.\; \mathsf{chalk}\;\vec{s} = \mathsf{cheese}\;\vec{t} \to M\rceil \;\text{ where }\; \mathsf{chalk} \neq \mathsf{cheese} \\
\quad\Longrightarrow \qquad m \mapsto \lambda\Delta.\,\lambda q.\; \textsc{conflict}\; q\; M \\
\lceil m : \forall\Delta.\; x = s \;\to\; M\rceil \;\text{ where }\; x \in \textsc{dom}\;\Delta,\; s \notin \textsc{dom}\;\Delta \\
\quad\Longrightarrow \qquad \lceil m' : \forall\Delta.\; s = x \;\to\; M\rceil; \\
\qquad\qquad\quad m \mapsto \lambda\Delta.\,\lambda q.\; m'\;\Delta\;(\mathbf{sym}\; q) \\
\lceil m : \forall\Delta.\; \mathsf{c}\;\vec{t} = x \to M\rceil \;\text{ where }\; x \prec \mathsf{c}\;\vec{t} \\
\quad\Longrightarrow \qquad m \mapsto \lambda\Delta.\,\lambda q.\; \textsc{cyclic}\; q\; M \\
\lceil m : \forall\Delta.\; t =_T x \to M\rceil \;\text{ where }\; (\Delta^t,\; \Delta^x_t; x : T; \Delta_x) \;\Longleftarrow\; \textsc{strengthen}(\Delta, t, T) \\
\quad\Longrightarrow \qquad \lceil m : \Downarrow\!\forall\Delta^t; \Delta^x_t; x \mapsto t \,:\, T; \Delta_x.\; M\rceil \\
\qquad\qquad\quad m \mapsto \lambda\Delta.\,\lambda q.\; \mathbf{subst}\; q\;(\lambda x.\,\forall\Delta_x.\; M)\;(m'\;\Delta^t\;\Delta^x_t)\;\Delta_x \\
\lceil m : M\rceil \qquad \Longrightarrow \qquad m
\end{array}$$

Simplification for a context of methods

$$\begin{array}{r}
\lceil\cdot\rceil \Longrightarrow \cdot \\
\lceil\Psi; m : M\rceil \Longrightarrow \lceil\Psi\rceil; \lceil m : M\rceil
\end{array}$$

Splitting a problem

$$\begin{array}{l}
\textsc{split}(\Delta, \langle l\!: T\rangle, E \text{ as } \forall P\!:\!(\forall\Theta.\,\star).\;\forall\Psi.\; P\;\vec{t}) \\
\quad\Longrightarrow \quad \underline{\text{let}}\; P \;\mapsto\; \lambda\Theta.\,\forall\Delta.\;\Theta = \vec{t} \;\to\; \langle l\!: T\rangle\,. \\
\qquad\qquad (\lambda\lceil\Psi\rceil.\;\lambda\Delta.\;\lambda e\!:\!E.\;\; e\;P\;\Psi\;\Delta\;(\mathsf{refl}\;\vec{t}) \\
\qquad\qquad\; : \;\forall\lceil\Psi\rceil.\;\forall\Delta.\; E \to \langle l\!: T\rangle)
\end{array}$$

$$\boxed{\mathit{context}|\mathit{context} \;\Vdash\; \mathit{expr} \;\triangleright\; \mathit{term} \;:\; \langle \mathit{label}\!: \mathit{term}\rangle}$$

$$[\text{by}] \quad \frac{\begin{array}{l}
\Gamma;\Delta \;\Vdash\; \ell \triangleright l \qquad \Gamma;\Delta \;\Vdash\; e \;\triangleright\; t \;:\; E \qquad \text{for } E \text{ a } \Gamma|\Delta\text{-eliminator type} \\
\textsc{split}(\Delta, \langle l\!: T\rangle, E) \;\Longrightarrow\; g \;:\; (\forall\Delta_1.\;\langle l_1\!: S_1\rangle) \to \cdots \to (\forall\Delta_k.\;\langle l_k\!: S_k\rangle) \\
\qquad\qquad\qquad\qquad\qquad\qquad \to\; \forall\Delta.\; E \to \langle l\!: T\rangle \\
\Gamma|\Delta_i \;\Vdash\; p_i \;\triangleright\; s_i \;:\; \langle l_i\!: S_i\rangle \qquad (1 \le i \le k)
\end{array}}{\Gamma|\Delta \;\Vdash\; \ell \Leftarrow e\;\{p_1; \ldots; p_k\} \;\triangleright\; g\;(\lambda\Delta_1.\; s_1)\;\ldots\;(\lambda\Delta_k.\; s_k)\;\Delta\;t \;:\; \langle l\!: T\rangle}$$

Fig. 10. The [by] rule and related definitions.

The problem is that although P is applied here to a *nonempty* environment, it must still abstract over *every* environment, empty or not. This is an old problem for inductive theorem proving (for example in proving 'generation lemmas' (Barendregt, 1992; McKinna & Pollack, 1993; McKinna & Pollack, 1999)) and for logic program transformation (Clark, 1978; Tamaki & Sato, 1984). How do we apply an induction principle (or an unfolding) to a *constrained* instance of a relation?

Fortunately, there is also an old solution which has been exploited for many years, either by hand or mechanically, in these settings: transform 'this constrained instance' to '*any* instance which satisfies these constraints', where the constraints are expressed by *equations*:

If we could take	$P \mapsto \lambda_{ks}.\, \lambda h' {:} \mathsf{HAL}\ ks.\ ks = l :: ls \ \rightarrow\ \langle \mathbf{hTail}\, h {:} \mathsf{HAL}\ ls \rangle$
then we would have	$P_{(l::ls)}\ h \ \simeq\ l :: ls = l :: ls \ \rightarrow\ \langle \mathbf{hTail}\, h {:} \mathsf{HAL}\ ls \rangle$

This is what we need, at the cost of supplying a trivial proof. Meanwhile, the methods required would have types

$$\begin{array}{lll} m_1 & : & [\,] = l :: ls \ \rightarrow\ \langle \mathbf{hTail}\, h {:} \mathsf{HAL}\ ls \rangle \\ m_2 & : & \forall_{X,ls'}.\, \forall l', x.\, \forall h' {:} \mathsf{HAL}\ ls'. \\ & & (ls' = l :: ls \rightarrow \langle \mathbf{hTail}\, h {:} \mathsf{HAL}\ ls \rangle) \ \rightarrow \\ & & l' :: ls' = l :: ls \ \rightarrow\ \langle \mathbf{hTail}\, h {:} \mathsf{HAL}\ ls \rangle \end{array}$$

For the hnil case, m_1, we have a false equation, hence the method should be supplied vacuously. For m_2, we have an equation which implies that $ls' = ls$, and hence that, 'morally', the exposed tail h' is an acceptable return.

We can mechanize this idea in type theory, yielding the key technique for expressing high-level programs via elimination operators, hence we reprise it here. In order to do so, our type theory needs a suitable notion of equality—the **heterogeneous equality** shown in Figure 10. This presentation (McBride, 1999) is not yet standard in type theory: it allows the formation of *heterogeneous* equations between elements of any two types, and hence equations between *vectors* in a given context. We expand $\vec{a} = \vec{b}$ as a context of equational constraints $q_1 : a_1 = b_1;\ \ldots;\ q_k : a_k = b_k$, and correspondingly, $\mathsf{refl}\ \vec{t}$ as the vector $\mathsf{refl}\ t_1; \ldots; \mathsf{refl}\ t_k$.

Crucially, however, the elimination operator (with κ-reduction[1]), which gives us that equality is a congruence, only applies to *homogeneous* equations: we may only substitute elements of the same type. It is *not* the operator which a <u>data</u> declaration would generate for $=$, but it still covers all canonical proofs of equations.

Now, in the general case, we have a programming problem $\forall\Delta.\ \langle l {:} T \rangle$ and an eliminator with type $\forall P {:} (\forall\Theta.\ \star).\ \forall\Psi.\ P\ \vec{t}$. The SPLIT meta-operation chooses

$$P \ \mapsto\ \lambda\Theta.\, \forall\Delta.\, \Theta = \vec{t} \ \rightarrow\ \langle l {:} T \rangle$$

[1] κ being a nod to those authors, who have studied an additional constant K which, for the usual inductively defined equality in type theory, yields power equivalent to our notion (Streicher, 1993; Hofmann & Streicher, 1994)

Now (in scope of this definition) if we can find methods Ψ where

$$\begin{array}{lll} \Psi \text{ is} & m_1 & : \ \forall \Delta_1;\ \Delta;\ \vec{s}_1 = \vec{t}.\ \langle l : T \rangle\ ; \\ & \vdots & \\ & m_n & : \ \forall \Delta_n;\ \Delta;\ \vec{s}_n = \vec{t}.\ \langle l : T \rangle \end{array}$$

we will have

$$\lambda \Delta.\ \lambda e : E.\ e\ P\ \Psi\ \Delta\ (\mathsf{refl}\ \vec{t}) \quad : \quad \forall \Delta.\ E \to \langle l : T \rangle$$

This is the general form of the technique we used in the **hTail** example, turning a particular $\vec{t}$ into equational constraints on a freely chosen Θ described above. The instantiated constraints characterize the intersections $\vec{s}_i = \vec{t}$ in which the indices of interest lie. Further, in any inductive hypotheses given by expanding P in Δ_i, the equations give the conditions for making a recursive call. Quantifying over Δ within the motive P ensures that such inductive hypotheses are as liberal as possible. For **hTail**, the motive and the method types—now a little less tidy—are as follows:

$$\begin{array}{l} P \ \mapsto\ \lambda_{ks}.\ \lambda h' : \mathsf{HAL}\ ks. \quad \forall_{l,ls}.\ \forall h : \mathsf{HAL}\ (l :: ls). \\ \qquad\qquad ks = l :: ls\ \to\ h' = h\ \to\ \langle \mathbf{hTail}\ h : \mathsf{HAL}\ ls \rangle \\[1ex] m_1 \ :\ \forall_{l,ls}.\ \forall h : \mathsf{HAL}\ (l :: ls). \\ \qquad\qquad [] = l :: ls\ \to\ \mathsf{hnil} = h\ \to\ \langle \mathbf{hTail}\ h : \mathsf{HAL}\ ls \rangle \\ m_2 \ :\ \forall_{X,ls'}.\ \forall l', x.\ \forall h' : \mathsf{HAL}\ ls'. \\ \qquad (\forall_{l,ls}.\ \forall h : \mathsf{HAL}\ (l :: ls). ls' = l :: ls\ \to\ h' = h\ \to\ \langle \mathbf{hTail}\ h : \mathsf{HAL}\ ls \rangle)\ \to \\ \qquad \forall_{l,ls}.\ \forall h : \mathsf{HAL}\ (l :: ls). \\ \qquad\qquad l' :: ls' = l :: ls\ \to\ \mathsf{hcons}\ l'\ x\ h' = h\ \to\ \langle \mathbf{hTail}\ h : \mathsf{HAL}\ ls \rangle \end{array}$$

These methods m_i will ultimately give rise to the subproblems solved by the subprograms, but first they are simplified by first-order unification, as in (McBride, 1998; McBride, 1999; McBride, 2002), and once again here.

We present unification in Figure 10 as a meta-operation on a method binding, $\lceil m : M \rceil$, returning a context in which m still has type M, but may now be defined, either in terms of a simplified method $m' : M'$ (with the equations reduced), or without further assumption (if the equations are demonstrably absurd). Each clause of the definition explains how to simplify a *homogeneous* equational hypothesis and thus takes the form $\lceil m : \forall \Delta.\ s = t \to M \rceil \implies \cdots$. In order to resolve ambiguity, we prioritize the rules from top to bottom and shorter candidates for Δ over longer. For reasons of brevity, we omit the explicit enforcement of homogeneity and the repetition of the input method's type.

The meta-operations INJECT and CONFLICT deploy proofs that a datatype family has the 'no confusion' property. Meanwhile, CYCLIC exploits the relevant family's 'no cycles' property: the condition $x \prec \mathsf{c}\,\vec{t}$, ($x$ is **constructor-guarded** in $\mathsf{c}\,\vec{t}$), holds if either $x \simeq t_i$ or $x \prec t_i$ for some i. These properties are derived automatically when each datatype family is declared: we do not repeat the construction here, but refer the interested reader to (McBride, 1999).

In the penultimate clause, STRENGTHEN is used to ensure that t is a suitable candidate to instantiate x, whose binding must fall amongst those not needed to type-check t—this subsumes the traditional occur-check. Moreover, computing out the new definition instantiates x with t in the method's label.

What can we say about this unification algorithm? Our prioritization ensures that it is deterministic. Further, for methods $\lceil m : \forall\Delta.\ \langle l : T\rangle\rceil$, the usual induction (first on the number of non-equational hypotheses in Δ, then on the number of constructor symbols in the equations) shows that the algorithm terminates.

We can readily iterate this process across a context of methods, $\lceil\Psi\rceil$. For **hTail**, we get something of the following form, with the hnil case solved outright, and the patterns in the hcons case reduced to those the subprogram requires:

$$
\begin{array}{l}
\lceil\Psi\rceil \implies \\
\quad m_1 \mapsto \lambda_{l,ls}.\ \lambda h.\ \lambda q : [] = l :: ls. \\
\qquad\quad \text{CONFLICT}\ q\ (\mathsf{hnil} = h \to \langle \mathbf{hTail}\ h : \mathsf{HAL}\ ls\rangle); \\
\quad m_2' : \forall_{X,ls}.\ \forall l, x.\ \forall h : \mathsf{HAL}\ ls. \\
\qquad (\forall_{l,ls}.\ \forall h : \mathsf{HAL}\ (l :: ls).\ ls' = l :: ls \to h' = h \to \langle \mathbf{hTail}\ h : \mathsf{HAL}\ ls\rangle) \to \\
\qquad \langle \mathbf{hTail}\ (\mathsf{hcons}\ l\ x\ h) : \mathsf{HAL}\ ls\rangle\,; \\
\quad m_2^3 \mapsto ..\ \mathbf{subst}\ ..\ m_2';\quad m_2^2 \mapsto ..\ \mathbf{subst}\ ..\ m_2^3;\quad m_2^1 \mapsto ..\ \mathbf{subst}\ ..\ m_2^2; \\
\quad m_2 \mapsto ..\ \text{INJECT}\ ..\ m_2^1
\end{array}
$$

Crucially, $\lceil\Psi\rceil$ still binds every method in Ψ, so the SPLIT operation used in the [by]-rule is well-defined: the combinator it computes just abstracts over the simplified problems, but passes the terms derived for the $k \le n$ unsimplified methods to the eliminator, solving the original problem. The [by] rule checks that these simplified problems are solved by the subprograms.

4.1 Derived eliminators

As has often been observed, many 'obviously' terminating functions do not directly fit the pattern of computation supported by **D-elim** operators—one step of case analysis, with recursion on the immediately exposed subterms. Some, such as the Fibonacci function, require access more than one step back down the course of values. Others, such as McBride's dependently typed implementation of first-order unification (McBride, 2001), perform case analysis on a datatype family (the terms), but recursion on an *index* of that family (the number of unsolved variables).

One remedy, certainly adequate for these two examples, is to follow Coquand's suggestion and separate case analysis from recursion. Giménez achieves this in Coq (Giménez, 1994; Giménez, 1998) by equipping the type theory with primitive `Case` and `Fix` constructs. The latter permits recursion on any constructor-guarded subterm (*c.f.* the previous Section) of the argument it addresses.

One does not need the full machinery of an extension by fixpoint constructs, however; the first author's version of the same idea is to *derive* separate case analysis and recursion operators automatically, given the primitive elimination operator. The type of the case analysis operator is computed simply by discarding the inductive hypotheses from the primitive elimination operator:

$$\begin{aligned}\mathsf{D}\textbf{-case} \; : \; & \forall_\Theta; x : \mathsf{D}\,\Theta.\, \forall P{:}(\forall_\Theta; x : \mathsf{D}\,\Theta.\, \star).\\ & \forall m_1{:}\forall\Delta_1.\, P\,(\mathsf{c}_1\,\vec{s}_1).\;\ldots\;\forall m_n{:}\forall\Delta_n.\, P\,(\mathsf{c}_n\,\vec{s}_n).\, P\,x\end{aligned}$$

The intrinsic action of ι-reduction on constructor-headed arguments is harnessed to account for constructor-guarded recursion, via a memoization technique:

$$\begin{aligned}\mathsf{D}\textbf{-rec} \; : \; & \forall_\Theta; x : \mathsf{D}\,\Theta.\, \forall P{:}(\forall_\Theta; x : \mathsf{D}\,\Theta.\, \star).\\ & (\forall_\Theta; x : \mathsf{D}\,\Theta.\, \mathsf{D}\textbf{-memo}\,P\,x \;\to\; P\,x) \;\to\\ & P\,x\end{aligned}$$

The predicate transformer $\mathsf{D}\textbf{-memo}$ computes a 'course-of-values' data structure storing a value in $P\ y$ for every y structurally smaller than the given x. This structure is just a big tuple, computed by primitive recursion over x. We write $\mathsf{D}\textbf{-memo}$ informally in pattern matching style—these laws hold as conversions—but the eliminator translation is straightforward.

$$\mathsf{D}\textbf{-memo}\,P\;(\mathsf{c}_i\;\Delta_i) \simeq \times(\textsc{hyps}(\mathsf{D}\textbf{-memo}\,P, \Delta_i);\; \textsc{hyps}(P, \Delta_i))$$

where $\times(x_1 : T_1; \ldots; x_n : T_n)$ denotes the Cartesian product $T_1 \times \ldots \times T_n$. We take $\times\cdot$ to be Unit. For $\mathbb{N}$, this gives

$$\begin{aligned}&\mathbb{N}\textbf{-memo}\,P\quad \mathsf{0} \;\leadsto^*\; \mathsf{Unit}\\ &\mathbb{N}\textbf{-memo}\,P\,(\mathsf{s}n) \leadsto^* (\Downarrow\mathbb{N}\textbf{-memo}\,P\,n) \times P\,n\end{aligned}$$

The term justifying $\mathsf{D}\textbf{-case}$ is trivial to construct; that for $\mathsf{D}\textbf{-rec}$ is a little more complex—we refer the interested reader to (McBride, 1999). We may use $\mathsf{D}\textbf{-case}\,x$ repeatedly, or other means, to instantiate $\mathsf{D}\textbf{-memo}\,P\,x$ with constructor-prefixed terms, allowing it to unfold and reveal hypotheses for the guarded subterms. The meta-operation LOOKUP must therefore be able to search these tuples in order to project out the solutions to the programming problems corresponding to recursive calls. Consider, for example, the Fibonacci function:

$$\underline{\text{let}}\quad \frac{x \;:\; \mathbb{N}}{\textbf{fib}\,x \;:\; \mathbb{N}} \qquad \begin{array}{llll} \textbf{fib} & x & \Leftarrow & \mathbb{N}\textbf{-rec}\;x\\ \;\textbf{fib} & x & \Leftarrow & \mathbb{N}\textbf{-case}\;x\\ \;\;\textbf{fib} & \mathsf{0} & \mapsto & \mathsf{0}\\ \;\;\textbf{fib} & (\mathsf{s}x') & \Leftarrow & \mathbb{N}\textbf{-case}\;x'\\ \;\;\;\textbf{fib} & (\mathsf{s}\mathsf{0}) & \mapsto & \mathsf{s}\mathsf{0}\\ \;\;\;\textbf{fib} & (\mathsf{s}(\mathsf{s}x'')) & \mapsto & \textbf{fib}\;x'' + \textbf{fib}\;(\mathsf{s}x'') \end{array}$$

Here, the initial $\Leftarrow \mathbb{N}\textbf{-rec}\,x$ will select the following motive and add the corresponding memo-structure to the context:

$$\begin{aligned}&P \;\mapsto\; \lambda n.\, \forall x.\, n = x \;\to\; \langle \textbf{fib}\,x : \mathbb{N}\rangle\\ &\mathit{memo}_x \;:\; \mathbb{N}\textbf{-memo}\,P\,x\end{aligned}$$

$$
\begin{array}{lll}
\textsc{lookup}(l,\ \Gamma; x \mapsto s : S) \Longrightarrow & \text{try} & \textsc{unpack}(\cdot, (\varepsilon,\varepsilon), x,\ \Downarrow S) \\
& \text{before} & \textsc{lookup}(l, \Gamma) \\
\textsc{lookup}(l,\ \Gamma; x : S) \qquad \Longrightarrow & \text{try} & \textsc{unpack}(\cdot, (\varepsilon,\varepsilon), x,\ \Downarrow S) \\
& \text{before} & \textsc{lookup}(l, \Gamma)
\end{array}
$$

where

$$
\begin{array}{ll}
\textsc{unpack}(\Delta, (\vec{s}, \vec{t}), x, \langle l' : T\rangle) & \text{where } (\Delta \mapsto \vec{u}) \text{ unifies } l' \text{ with } l \text{ and } \vec{s} \text{ with } \vec{t} \\
& \qquad \Gamma; \Delta \mapsto \vec{u} \ \vdash \ x \ : \ \langle l : T\rangle \\
& \Longrightarrow \ (\ \Downarrow \underline{\mathsf{let}}\ \Delta \mapsto \vec{u}.\ x \ : \ \Downarrow \underline{\mathsf{let}}\ \Delta \mapsto \vec{u}.\ \langle l : T\rangle) \\
\textsc{unpack}(\Delta, (\vec{s}, \vec{t}), f, \forall x : S.\ T) & \text{where } x \in T \\
& \Longrightarrow \ \textsc{unpack}(\Delta; x : S, (\vec{s}, \vec{t}), f\ x, T) \\
\textsc{unpack}(\Delta, (\vec{s}, \vec{t}), qf, s = t \to T) & \Longrightarrow \ \textsc{unpack}(\Delta, (\vec{s}; s, \vec{t}; t), qf\ (\mathsf{refl}\ s), T) \\
\textsc{unpack}(\Delta, (\vec{s}, \vec{t}), xy, X \times Y) & \Longrightarrow \ \ \text{try} \quad \textsc{unpack}(\Delta, (\vec{s}, \vec{t}), \mathbf{snd}\ xy, Y) \\
& \qquad\ \text{before} \quad \textsc{unpack}(\Delta, (\vec{s}, \vec{t}), \mathbf{fst}\ xy, X)
\end{array}
$$

Fig. 11. The LOOKUP meta-operation

In the recursive case, x has been instantiated, and the memo-structure becomes

$$
\begin{array}{lll}
\mathit{memo}_x \ : \ \mathbb{N}\text{-}\mathbf{memo}\ P\ (\mathsf{s}(\mathsf{s}x'')) & \leadsto^* & ((\ \Downarrow \mathbb{N}\text{-}\mathbf{memo}\ P\ x'') \ \times \\
& & \quad (\forall x.\ x'' = x \ \to \ \langle \mathbf{fib}\ x : \mathbb{N}\rangle)) \ \times \\
& & (\forall x.\ \mathsf{s}x'' = x \ \to \ \langle \mathbf{fib}\ x : \mathbb{N}\rangle)
\end{array}
$$

So, LOOKUP must handle more than just the bindings, $f \mapsto \cdots : \forall\Delta.\ \langle \mathbf{f}\ \Delta : T\rangle$, yielded by the [let] rule; it must extract solutions from hypotheses tupled or constrained by equations. We define it in Figure 11, giving only the patterns which lead to progress—if the match fails, so does the operation.

For each binding in Γ, LOOKUP inspects the normal form of its type to check if it can match the required label l. The real work is done by the auxiliary meta-operation $\textsc{unpack}(\Delta, (\vec{s}, \vec{t}), x, X)$, which builds a candidate solution x, whilst accumulating a context Δ which must be instantiated, and a pair of vectors $(\vec{s}, \vec{t})$ which must be equal, for the candidate to succeed with type X. This X determines the search strategy: if it is $\forall$-quantified, try application; if it demands an equation, try a reflexive proof; if it is a pair, try each projection in turn. Eventually, if UNPACK reaches a candidate for a programming problem $\langle l' : T\rangle$, it checks that l' subsumes l by unifying the labels and the accumulated constraints, then typechecking the instantiated candidate: we use ordinary first-order unification on normalized terms.

For the **fib** example, LOOKUP does indeed find that

$$
\begin{array}{l}
\mathbf{snd}\ (\mathbf{fst}\ \mathit{memo}_x)\ x''\ (\mathsf{refl}\ x'') \ : \ \langle \mathbf{fib}\ x'' : \mathbb{N}\rangle \\
\mathbf{snd}\ \mathit{memo}_x\ (\mathsf{s}x'')\ (\mathsf{refl}\ (\mathsf{s}x'')) \ : \ \langle \mathbf{fib}\ (\mathsf{s}x'') : \mathbb{N}\rangle
\end{array}
$$

This definition of LOOKUP is certainly adequate to unpack the solutions to programming problems exposed by **D-case** in the memo-structures installed by **D-rec**. However, the latter are just particular instances of the general notion of elimination operator, defined in Section 4, and could have been defined by a programmer using **D-elim**; but since they may be generated automatically, we may take them as given. They capture an important class of allowable recursions; user-defined elimination operators which capture other interesting recursive call patterns have been considered elsewhere (McKinna, 2002) and remain the subject of ongoing study.

Of course, **htail** and **fib**, as presented in full above, have rather more bulky code than functional programmers normally expect to write. Especially annoying is the fact that the calls we eventually write on either side already carry the evidence of the case analysis and structural recursion which explain them—constructor symbols.

We can alleviate this problem somewhat by taking a combination of outer D-**rec** and inner D-**case** applications to be the default explanation of a *non-empty* block of programs wherever a single program is expected. The constructor patterns in these programs bound the depth of the splitting which can possibly produce them, and there are only finitely many ways to combine recursions lexicographically, hence there is at least a clumsy elaboration method. More sophisticated approaches may be found in (Cornes, 1997; Abel & Altenkirch, 2000).

As a consequence of this defaulting strategy, we may suppress the $\Leftarrow$-clause in **htail**, recovering our earlier statement of the program

$$\underline{\text{let}} \quad \frac{h \ : \ \mathsf{HAL}\,(l :: ls)}{\mathbf{hTail}\,h \ : \ \mathsf{HAL}\,ls} \qquad \mathbf{hTail}\,(\mathsf{hcons}\ l\ x\ h') \ \mapsto\ h'$$

We may also remove all but the three equations from the program for **fib**, yielding the more familiar

$$\underline{\text{let}} \quad \frac{n \ : \ \mathbb{N}}{\mathbf{fib}\,n \ : \ \mathbb{N}} \qquad \begin{array}{lcl} \mathbf{fib}\ \ \ \mathsf{0} & \mapsto & \mathsf{0} \\ \mathbf{fib}\ \ (\mathsf{s0}) & \mapsto & \mathsf{s0} \\ \mathbf{fib}\,(\mathsf{s}(\mathsf{s}n'')) & \mapsto & \mathbf{fib}\,n'' + \mathbf{fib}\,(\mathsf{s}n'') \end{array}$$

Indeed, in the general case, the only **-case**-splits which we must retain are those which yield no cases! The undecidability of type inhabitation obliges us to be explicit in such situations. In the absence of evidence in the form of a constructor pattern, which points to a particular argument type being empty, there is no basis on which to reconstruct the correct **-case**-term. Examples of this arise with the occurrence of So false in the hnil branches of **typeProj** and **valProj**.

With the derived case analysis and recursion operators, and using this convention, our type theory can support—by elaboration into large and unreadable terms—every program admitted by Coquand's proposed pattern matching language (Coquand, 1992), as partially implemented in ALF (Magnusson, 1994). Such is the principal result of the first author's PhD thesis (McBride, 1999), in which the original objective had been to dispense with eliminators in favour of pattern matching. With hindsight, we would recommend *exactly the opposite*. In our terms, Coquand's system hard-wires splitting as if by D-**case** (with intersections computed by a unification oracle) and presents recursion only as if by D-**rec**.

We conclude this section with a simple example using a non-standard eliminator—the 'target-first' variant of $\mathbb{N}$-**Compare** from the Introduction, of type

$$\begin{array}{lll} \mathbb{N}\textbf{-compare} & : & \forall m, n{:}\mathbb{N}.\ \forall P : \mathbb{N} \to \mathbb{N} \to \star. \\ & & (\forall x, y{:}\mathbb{N}.\ P\ x\ (x + \mathsf{s}y)) \to \\ & & (\forall x{:}\mathbb{N}.\ P\ x\ x) \to \\ & & (\forall x, y{:}\mathbb{N}.\ P\ (y + \mathsf{s}x)\ y) \to \\ & & P\ m\ n \end{array}$$

With it, we may define the 'absolute difference' function for $\mathbb{N}$:

$$\underline{\text{let}}\quad \frac{m, n\ :\ \mathbb{N}}{\textbf{absDiff}\ m\ n\ :\ \mathbb{N}} \qquad \begin{array}{llll} \textbf{absDiff} & m & n & \Leftarrow \mathbb{N}\textbf{-compare}\ m\ n \\ \quad\textbf{absDiff} & x & (x + \mathsf{s}y) & \mapsto\ \mathsf{s}y \\ \quad\textbf{absDiff} & x & x & \mapsto\ \mathsf{0} \\ \quad\textbf{absDiff} & (y + \mathsf{s}x) & y & \mapsto\ \mathsf{s}x \end{array}$$

In the original spirit of pattern matching, a testing operation, comparison, has been safely and clearly combined with a selection operation, subtraction. We shall present more sophisticated examples in Section 6, where we develop an idiom for constructing non-standard eliminators by first-order programming.

5 Abstracting Intermediate Computations

In this section, we introduce our analogue to the proposed **pattern guard** notation in Haskell (Peyton Jones, 1997; Peyton Jones & Erwig, 2000)—the **with** construct, *lhs* | *expr* {*program*}. Pattern guards allow an intermediate computation to be matched against a single acceptable pattern—if the subsidiary match fails, control passes to the next line of the program. For example, pattern guards provide a convenient way to unpack a recursively computed tuple:

```
unzip []                                 =   ([],  [])
unzip ((x,y):xys) | (xs,ys) <- unzip xys = (x:xs,y:ys)
```

The basic function of '| e' is to add the result of e to the collection of values under scrutiny on the left. Subsequent 'matching' comes from the $\Leftarrow$ construct (implicitly, for standard **-case** operators) as usual. The effect is similar to defining a helper function over all the original 'pattern variables' together with the extra value, but the | is much more compact. With our layout convention, the above becomes:

$$\underline{\text{let}}\quad \frac{xys\ :\ \mathsf{List}\,(A \times B)}{\textbf{unzip}\ xys\ :\ \mathsf{List}\,A\ \times\ \mathsf{List}\,B}$$

$$\begin{array}{ll} \textbf{unzip}\ [] & \mapsto\ ([], []) \\ \textbf{unzip}\ ((x, y) :: xys) & \left|\ \begin{array}{l} \textbf{unzip}\ xys \\ \quad (xs, ys)\ \ \mapsto\ \ (x :: xs, y :: ys) \end{array}\right. \end{array}$$

Once we have an intermediate value, we can consider more than one case of it, as in our version of **elem**. Haskell's guards also reduce the tendency of programs which mix analysis of their arguments and intermediate values to degenerate into gangling right-hand sides built by `if` and `case`. This function, counting the number of times a given tree occurs within another, shows but the tip of the iceberg:

```
count s t = if s == t then 1
            else case t of
                   Leaf      -> 0
                   t1 :^: t2 -> count s t1 + count s t2
```

To connect `count`'s arguments with the analysis on the right, we must observe the recurrence of `t`. Longer trails of repeated identifiers can easily become confusing, and certainly make it harder to tell at a glance what a program does. Here, even a Boolean guard is enough to reconnect the program, expressing its analysis clearly and concisely on the left:

```
count s t  | s == t = 1
count s Leaf        = 0
count s (t1 :^: t2) = count s t1 + count s t2
```

Even without special sugar for booleans or 'fall-through', our notation tabulates exactly the analysis performed: its 'laws' are as clear as its mechanism.

$$\underline{\mathrm{let}}\quad \frac{s, t\ :\ \mathsf{tree}}{\mathbf{count}\ s\ t\ :\ \mathbb{N}} \qquad \begin{array}{lr|lll} \mathbf{count}\ s & t & s == t & & \\ & & \mathsf{true} & \mapsto & \mathsf{s0} \\ & \mathsf{leaf} & \mathsf{false} & \mapsto & 0 \\ & (t_1\ \mathsf{node}\ t_2) & \mathsf{false} & \mapsto & \mathbf{count}\ s\ t_1 + \mathbf{count}\ s\ t_2 \end{array}$$

5.1 Abstracting from types

Clarity notwithstanding, type dependency provides a second motivation for treating subcomputations on the left—their impact on *types*. We have already observed this informally with the **elem**, **typeProj**, **valProj** example. In order to connect the intermediate label tests in **typeProj** and **valProj** with the **elem** computations at the type level, we must abstract the tests from types as well as in the patterns.

Our 'with' notation corresponds directly to an established technique in theorem proving—generalizing a goal by abstracting a subexpression, perhaps to strengthen an induction—as implemented by the `Pattern` tactic in Coq (Coq, 2001). Its elaboration rule is shown in Figure 12.

Using the meta-operation ABST (whose obvious definition as an inverse to substitution is omitted), the elaborator computes abstractions (l_x, on labels, and Δ_x on contexts): these abstractions must be typechecked again, to ensure that replacing the elaborated term s by a variable has not compromised validity. The elaborator then constructs a helper function t from subprogram p, with an extended label—the main program calls the helper. The normalization of $\mathbf{elem}\ k\ (l :: ls)$, goes thus:

$$\boxed{\mathit{context}|\mathit{context} \;\Vdash\; \mathit{expr} \;\triangleright\; \mathit{term} \;:\; \langle \mathit{label}\!:\!\mathit{term}\rangle}$$

$$[\text{with}]\quad \frac{\begin{array}{l}\Gamma;\Delta \;\Vdash\; \ell \triangleright l_s \qquad \Gamma;\Delta \;\Vdash\; e \;\triangleright\; s \;:\; S \\ (\Delta^s, \Delta_s) \;\Longleftarrow\; \textsc{strengthen}(\Delta, s, S) \\ l_x \;\Longleftarrow\; \textsc{abst}(s, x, l_s) \qquad \Delta_x \;\Longleftarrow\; \textsc{abst}(s, x, \Delta_s) \\ \Gamma; \Delta^s; x : S; \Delta_x \;\vdash\; \langle l_x \mid x : T\rangle \;:\; \star \\ \Gamma|\Delta^s; x : S; \Delta_x \;\Vdash\; p \;\triangleright\; t \;:\; \langle l_x \mid x : T\rangle\end{array}}{\Gamma|\Delta \;\Vdash\; \ell \mid e\ \{p\} \;\triangleright\; \underline{\mathrm{let}}\ x \mapsto s : S.\, \underline{\mathrm{return}}\ (\underline{\mathrm{call}}\ \langle l_x \mid x\rangle\ t) \;:\; \langle l_s : \underline{\mathrm{let}}\ x \mapsto s : S.T\rangle}$$

Fig. 12. Elaboration of 'with' notation

$$\begin{array}{ll} & \underline{\mathrm{call}}\ \langle \mathbf{elem}\, k\ (l :: ls)\rangle\ \mathsf{List\text{-}\mathbf{rec}}\ \ldots \\ \rightsquigarrow^* & \underline{\mathrm{call}}\ \langle \mathbf{elem}\, k\ (l :: ls)\rangle\ \underline{\mathrm{return}}\ (\underline{\mathrm{call}}\ \langle \mathbf{elem}\, k\ (l :: ls) \mid (\underline{\mathrm{call}}\ \langle k == l\rangle\ \ldots)\rangle\ \ldots) \\ \rightsquigarrow & \underline{\mathrm{call}}\ \langle \mathbf{elem}\, k\ (l :: ls) \mid (\underline{\mathrm{call}}\ \langle k == l\rangle\ \ldots)\rangle\ \ldots \end{array}$$

Correspondingly, when checking $\mathbf{typeProj}\, k\ (\mathsf{hcons}_X\ l\, x\, h)\ p \mid k == l\ \{\ldots\}$, we start in the context

$$k, l : \mathsf{Label};\ \ldots\ ;\ p : \mathsf{So}\ (\underline{\mathrm{call}}\ \langle \mathbf{elem}\, k\ (l :: ls) \mid (\underline{\mathrm{call}}\ \langle k == l\rangle\ \ldots)\rangle\ \ldots)$$

The term being abstracted, $k == l$, elaborates to the same $(\underline{\mathrm{call}}\ \langle k == l\rangle\ \ldots)$ as is found in the type of p, so the subprogram is checked in the context

$$k, l : \mathsf{Label};\ b : \mathsf{Bool};\ \ldots\ ;\ p : \mathsf{So}\ (\underline{\mathrm{call}}\ \langle \mathbf{elem}\, k\ (l :: ls) \mid b\rangle\ \ldots)$$

Of course, the $\langle k == l\rangle$ call is abstracted from the term implementing the $\langle \mathbf{elem}\ \ldots\rangle$ call, not just from the label. The subsequent analysis of b then allows the type of p to reduce further. The [with] rule gives the correct behaviour for **valProj** too, with abstraction from types working even harder to our benefit.

6 Views: a programming idiom

We have shown how abstracting an intermediate computation can have useful effects on types which depend on it. Case analysis on an intermediate value can also instantiate other *patterns*, if that value comes from a dependent family. In this section, we will illustrate this possibility, and show how it leads to an account of *views*, as proposed by Wadler (Wadler, 1987).

It is a commonplace to equip a datatype with an ordering by implementing a binary operator returning an element of the enumeration Ordering, given by $\{\mathsf{lt}, \mathsf{eq}, \mathsf{gt}\}$. For $\mathbb{N}$, we might write

$$\underline{\mathrm{let}}\quad \frac{m, n \;:\; \mathbb{N}}{\mathbf{cmp}\ m\ n \;:\; \mathsf{Ordering}} \qquad \begin{array}{llll} \mathbf{cmp} & 0 & 0 & \mapsto\ \mathsf{eq} \\ \mathbf{cmp} & 0 & (\mathsf{s}n) & \mapsto\ \mathsf{lt} \\ \mathbf{cmp} & (\mathsf{s}m) & 0 & \mapsto\ \mathsf{gt} \\ \mathbf{cmp} & (\mathsf{s}m) & (\mathsf{s}n) & \mapsto\ \mathbf{cmp}\ m\ n \end{array}$$

We might then write the **absDiff** function, by inspecting the result of an intermediate comparison:

$$\underline{\text{let}} \quad \ldots \quad \mathbf{absDiff}\ m\ n \;\Big|\; \begin{array}{lll} \mathbf{cmp}\ m\ n & & \\ \quad \mathsf{lt} & \mapsto & n - m \\ \quad \mathsf{eq} & \mapsto & 0 \\ \quad \mathsf{gt} & \mapsto & m - n \end{array}$$

A minor problem with this approach is that subtraction for $\mathbb{N}$ must return bogus answers when its second argument is the larger, in order to be a total function. More annoying is the fact that **cmp** has basically done the subtraction, but thrown the answer away. We could get around this by extending Ordering with difference information, but datatype families offer a more subtle approach.

We can define a binary *relation* on $\mathbb{N}$, with three canonical ways to show that two given numbers are comparable:

$$\underline{\text{data}} \quad \frac{x, y \ :\ \mathbb{N}}{\mathsf{Compare}\ x\ y} \quad \underline{\text{where}} \quad \begin{array}{c} \overline{\mathsf{lt}\ x\ y \ :\ \mathsf{Compare}\ x\ (x + \mathsf{s}y)} \\[1ex] \overline{\mathsf{eq}\ x \ :\ \mathsf{Compare}\ x\ x} \\[1ex] \overline{\mathsf{gt}\ x\ y \ :\ \mathsf{Compare}\ (y + \mathsf{s}x)\ y} \end{array}$$

Of course, every two numbers are comparable in one of these three ways. We can prove this by writing a program not much more complex than **cmp** above:

$$\begin{array}{l} \underline{\text{let}} \quad \overline{\mathbf{compare}\ x\ y \ :\ \mathsf{Compare}\ x\ y} \\ \quad \begin{array}{llll} \mathbf{compare} & 0 & 0 & \mapsto \ \mathsf{eq}\ 0 \\ \mathbf{compare} & 0 & (\mathsf{s}n) & \mapsto \ \mathsf{lt}\ 0\ n \\ \mathbf{compare} & (\mathsf{s}m) & 0 & \mapsto \ \mathsf{gt}\ m\ 0 \\ \mathbf{compare} & (\mathsf{s}m) & (\mathsf{s}n) & \Big|\ \mathbf{compare}\ m\ n \\ \quad \mathbf{compare} & (\mathsf{s}x) & (\mathsf{s}(x + \mathsf{s}y)) & \Big|\ \quad \mathsf{lt}\ x\ y \ \mapsto \ \mathsf{lt}\ (\mathsf{s}x)\ y \\ \quad \mathbf{compare} & (\mathsf{s}x) & (\mathsf{s}x) & \Big|\ \quad \mathsf{eq}\ x \ \mapsto \ \mathsf{eq}\ (\mathsf{s}x) \\ \quad \mathbf{compare} & (\mathsf{s}(y + \mathsf{s}x)) & (\mathsf{s}y) & \Big|\ \quad \mathsf{gt}\ x\ y \ \mapsto \ \mathsf{gt}\ x\ (\mathsf{s}y) \end{array} \end{array}$$

What has happened here? For the base cases, it is easy to choose the appropriate constructor and its arguments. To compare $\mathsf{s}m$ with $\mathsf{s}n$, however, we must 'update' the result of comparing m with n, hence we abstract it. But when we analyse a value in the datatype Compare m n, the arguments m and n become instantiated via the more informative constructor types. Inspecting an intermediate value has simultaneously told us more about the arguments from which it was computed.

Analysing the value of **compare** m n now does the job of comparison, subtraction, **max** and **min**. We can now write

$$\underline{\text{let}} \quad \ldots \quad \begin{array}{lll} \mathbf{absDiff} & m & n \\ \quad \mathbf{absDiff} & x & (x + \mathsf{s}y) \\ \quad \mathbf{absDiff} & x & x \\ \quad \mathbf{absDiff} & (y + \mathsf{s}x) & y \end{array} \;\Bigg|\; \begin{array}{lll} \mathbf{compare}\ m\ n & & \\ \quad \mathsf{lt}\ x\ y & \mapsto & \mathsf{s}y \\ \quad \mathsf{eq}\ x & \mapsto & 0 \\ \quad \mathsf{gt}\ x\ y & \mapsto & \mathsf{s}x \end{array}$$

The instantiated patterns now make quite clear the relationship between the inputs and the outputs in each case. We emphasize again that the nonlinear and '+' patterns do not require any ingenious operational behaviour: this is just a clearer way to write programs with basically the same operation as **cmp**.

One can perhaps imagine other suites of related testing and selection functions being combined into more general analysis methods which deliver informative patterns: Haskell's `takeWhile`, `dropWhile`, `exists`, `all`, ... each extract different functionality from the common process of applying a test successively to the elements of a list until it succeeds (or fails). By giving that process a type which shows whether and how the list is split at a particular point, *all* of these functions, together with particular instances like **elem**, can be combined. We leave this as an exercise.

The curious thing about **compare** $m\ n$ is that once we have seen the patterns it yields for m and n, we no longer care about its actual value! The column of patterns with lt, and so on, in **absDiff** is unnecessary noise. We can tidy up this idiom of testing and selection by examining case analysis over an inductively defined *relation*.

6.1 From relations to views

Wadler's original views proposal (Wadler, 1987) fits well with the notion of user-defined elimination operators. He suggests that any (possibly abstract) datatype T may be equipped with a notion of pattern matching by defining an isomorphism between T and a datatype D: elements of T may be matched against or built by D's constructors $\mathsf{d}_1, \ldots, \mathsf{d}_n$, with the compiler inserting either component of the isomorphism, **out** $: T \to \mathsf{D}$ or **in** $: \mathsf{D} \to T$, as required. Of course, there is no guarantee that **in** and **out** are either total or mutually inverse. In our setting, such a view may be expressed by replacing **out** with an elimination operator,

$$\begin{array}{lll}
T\textbf{-view} & : & \forall t\!:\!T.\ \ \forall P : T \to \star. \\
& & (\forall \vec{x}_1 \!:\! \vec{X}_1.\ P\ (\mathbf{d}_1\ \vec{x}_1)) \to \\
& & \qquad\qquad \vdots \\
& & (\forall \vec{x}_n \!:\! \vec{X}_n.\ P\ (\mathbf{d}_n\ \vec{x}_n)) \to \\
& & \qquad\quad P \quad\ t
\end{array}$$

where $\mathbf{d}_i$ is the *defined* operation by which **in** interprets d_i. Moreover, this type makes it clear that the t we put in is exactly the $(\mathbf{d}_i\ \vec{x}_i)$ we get out.

It is easy to extract these eliminators from programs like **compare** above. To see how, examine the following two typed terms:

$$\begin{array}{l|l}
\mathbb{N}\textbf{-compare}\ m\ n\ : & \mathsf{Compare}\textbf{-case}\ (\textbf{compare}\ m\ n)\ : \\
\qquad\quad \forall P : \mathbb{N} \to \mathbb{N} \to \star. & \qquad\quad \forall P' : \forall_m.\ \forall_n.\mathsf{Compare}\ m\ n \to \star. \\
(\forall x, y.\ \ P \quad x \quad (x + \mathsf{s}y)) \to & (\forall x, y.\ \ P'\ {}_{x\ (x+\mathsf{s}y)}\ (\mathsf{lt}\ x\ y)\ ) \to \\
(\forall x.\ \ \ \ P \quad x \qquad x \quad) \to & (\forall x.\ \ \ \ P'\ {}_{x\ \ x}\ \ (\mathsf{eq}\ x)\ ) \to \\
(\forall x, y.\ \ P\ (y + \mathsf{s}x) \quad y \quad) \to & (\forall x, y.\ \ P'\ {}_{(y+\mathsf{s}x)\ y}\ (\mathsf{gt}\ x\ y)\ ) \to \\
\qquad\quad P \quad m \qquad n & \qquad\quad P'\ {}_{m\ \ n}\ (\textbf{compare}\ m\ n)
\end{array}$$

$$\boxed{\textit{context} \Vdash \textit{expr} \triangleright \textit{term} : \textit{term}}$$

$$[\text{view}] \quad \frac{\begin{array}{l}\Gamma \Vdash e \triangleright t : \mathsf{D}\,\vec{t} \\ \Gamma \vdash \textbf{D-case}\; t : \forall P' {:} (\forall_\Phi .\, \mathsf{D}\,\Phi \to \star).\; \ldots\; (\forall \Phi_i .\, P'_{\vec{s}_i}\,(\mathsf{c}_i\,\Delta_i)) \to \ldots \to P'\,t\end{array}}{\begin{array}{l}\Gamma \Vdash \underline{\text{view}}\; e \triangleright \lambda P {:} \forall \Phi.\, \star .\; \textbf{D-case}\; t\,(\lambda_\Phi .\, \lambda_{\_} {:} \mathsf{D}\,\Phi.\, P\,\Phi) \\ \qquad : \forall P {:} \forall \Phi.\, \star .\; \ldots\; (\forall \Delta_i .\, P\,\vec{s}_i) \to \ldots \to P\,\vec{t}\end{array}}$$

Fig. 13. Elaboration of view

These are almost the same, except that P' (on the right) takes an extra argument—the actual value from the $\mathsf{Compare}$ family. However, given a candidate motive P for $\mathbb{N}$-**compare**, we can choose to instantiate P' with

$$P' \mapsto \lambda_{m,n} .\, \lambda_{\_} {:} \mathsf{Compare}\; m\; n.\, P\; m\; n$$

This motive ignores its $\mathsf{Compare}$ argument and applies P to just the indices—the patterns we wish to keep. Observe then that the following judgment holds:

$$\begin{array}{lcl}\lambda P {:}\; \forall m, n {:} \mathbb{N}.\, \star . & : & \quad \forall P : \mathbb{N} \to \mathbb{N} \to \star . \\ \quad \mathsf{Compare}\textbf{-case}\; (\textbf{compare}\; m\; n) & & (\forall x, y.\;\; P \quad x \quad (x + \mathsf{s}y)) \to \\ \qquad (\lambda_{m,n} .\, \lambda c {:} \mathsf{Compare}\; m\; n.\, P\; m\; n) & & (\forall x. \quad\; P \quad x \qquad x \quad\;) \to \\ & & (\forall x, y.\;\; P\,(y + \mathsf{s}x) \quad y \quad\;) \to \\ & & \qquad\quad P \quad\; m \qquad n\end{array}$$

We have just built $\mathbb{N}$-**compare**! This construction is just what we mean by the concrete syntax view **compare** m n. Figure 13 shows the elaboration rule.

There is a general recipe for establishing that a type T can be viewed via patterns p_1 (over Δ_1) to p_n (over Δ_n)—it readily extends to views of vectors of values. First, declare the relation

$$\underline{\text{data}}\; \frac{t : T}{\mathsf{View}{-}T\; t : \star} \quad \underline{\text{where}}\; \frac{\Delta_1}{\mathsf{c}_1\,\Delta_1 : \mathsf{View}{-}T\; p_1} \;\cdots\; \frac{\Delta_n}{\mathsf{c}_n\,\Delta_n : \mathsf{View}{-}T\; p_n}$$

Second, write the **covering** function which shows that the view applies to all of T:

$$\underline{\text{let}} \quad \frac{}{\textbf{view-}T\; t : \mathsf{View}{-}T\; t} \quad \cdots$$

The view may be invoked in a function using the 'by' construct,

$$\textit{lhs} \Leftarrow \underline{\text{view}}\; \textbf{view-}T\; t\; \{\textit{programs}\}$$

Indeed, as view t is meaningful for any t which belongs to a datatype, we can, in particular, use view to show the effect on patterns of the covering function's own recursive calls. The actual code for **compare** in Figure 14 demonstrates this.

What we have done is to explain non-standard 'pattern matching' via the refinement of index information which naturally accompanies the standard notion of case analysis for datatype families, whilst hiding their actual constructors. We hope that the intermediate data structures we conceal when a view is invoked can also be elim-

$$
\underline{\text{let}}\quad \frac{}{\mathbf{compare}\, m\, n \;:\; \mathsf{Compare}\, m\, n}
\qquad
\begin{array}{llll}
\mathbf{compare} & 0 & 0 & \mapsto \mathsf{eq}\,0 \\
\mathbf{compare} & 0 & (\mathsf{s}n) & \mapsto \mathsf{lt}\,0\,n \\
\mathbf{compare} & (\mathsf{s}m) & 0 & \mapsto \mathsf{gt}\,m\,0 \\
\mathbf{compare} & (\mathsf{s}m) & (\mathsf{s}n) & \Leftarrow \underline{\text{view}}\ \mathbf{compare}\, m\, n \\
\quad\mathbf{compare} & (\mathsf{s}x) & (\mathsf{s}(x+\mathsf{s}y)) & \mapsto \mathsf{lt}\,(\mathsf{s}x)\,y \\
\quad\mathbf{compare} & (\mathsf{s}x) & (\mathsf{s}x) & \mapsto \mathsf{eq}\,(\mathsf{s}x) \\
\quad\mathbf{compare} & (\mathsf{s}(y+\mathsf{s}x)) & (\mathsf{s}y) & \mapsto \mathsf{gt}\,x\,(\mathsf{s}y)
\end{array}
$$

Fig. 14. Comparison of natural numbers

inated from compiled code by *deforestation*, a technique for which we also have Wadler to thank (Wadler, 1990).

Wadler conceived his view notation as syntactic sugar for the insertion of mutually inverse coercions between datatypes, one of which admits pattern-matching, the other potentially abstract. The idea that a signature for an abstract data structure might hide its actual representation, but nonetheless export a notion of 'pattern decomposition', overcomes a genuine problem in the engineering of modular code. Programming with such programmer-definable patterns is exactly what the $\Leftarrow$ construct permits, with the bonus that the interface is given by a *type* which can be required of an exported method in the usual way. Moreover, this type precisely witnesses the 'no junk' direction of the bijection: Wadler is forced by an inexpressive type system to trust the programmer.

The presentation of views through datatype families also makes it easy to state a 'no confusion' property, by stipulating that the covering function **view-**T delivers the only possible value in each case. We describe a view for which this property holds as **unambiguous**. To prove that such a property holds, we write a program with the following signature:

$$
\underline{\text{let}}\quad \frac{x \;:\; \mathsf{View}{-}T\ t}{\textbf{view-}T\textbf{-unique}\ x \;:\; \textbf{view-}T\ t = x} \qquad \cdots
$$

7 An extended example: typechecking

This section shows views in action. We develop a typechecker for Church-style preterms in simply-typed λ-calculus. Our language of *simple type expressions* has a base type and function spaces:

$$
\underline{\text{data}}\quad \frac{}{\mathsf{TExp} \;:\; \star} \qquad \underline{\text{where}}\quad \frac{}{\mathsf{o} \;:\; \mathsf{TExp}} \qquad \frac{S, T \;:\; \mathsf{TExp}}{S \Rightarrow T \;:\; \mathsf{TExp}}
$$

Contexts are represented (back-to-front) by lists $\Gamma \;:\; \mathsf{List\ TExp}$ of such. We use a de Bruijn index (de Bruijn, 1972) representation of variables, rendered in type theory as usual by the datatype family $\mathsf{Fin} : \mathbb{N} \to \star$, where $\mathsf{Fin}\ n$ has n elements.

$$\underline{\text{data}}\ \ \frac{n\ :\ \mathbb{N}}{\mathsf{Fin}\,n\ :\ \star}\ \ \underline{\text{where}}\ \ \frac{}{\bullet\ :\ \mathsf{Fin}\,\mathsf{s}n}\ \ \frac{i\ :\ \mathsf{Fin}\,n}{\uparrow i\ :\ \mathsf{Fin}\,\mathsf{s}n}$$

Our source language, $\mathsf{Expr}\,n$, is the datatype of well-scoped but untyped expressions with n free variables, the *pre-terms.* This is quite close to the representation of untyped terms in (Bird & Paterson, 1999).

$$\underline{\text{data}}\ \ \frac{n\ :\ \mathbb{N}}{\mathsf{Expr}\ n\ :\ \star}\ \ \underline{\text{where}}\ \ \frac{i\ :\ \mathsf{Fin}\ n}{\mathsf{eVar}\ i\ :\ \mathsf{Expr}\ n}\ \ \frac{f, s\ :\ \mathsf{Expr}\ n}{\mathsf{eApp}\,f\ s\ :\ \mathsf{Expr}\ n}$$
$$\frac{S\ :\ \mathsf{TExp}\quad t\ :\ \mathsf{Expr}\,(\mathsf{s}n)}{\mathsf{eLam}\ S\ t\ :\ \mathsf{Expr}\ n}$$

Our aim is to write a typechecker for pre-terms, relative to a given context Γ, of length $|\Gamma|$; we implement the typechecker for expressions in $\mathsf{Expr}\ |\Gamma|$, by defining three *views* respectively:

- for looking up variables in the context;
- for testing equality of simple types;
- for typechecking pre-terms.

Each of these views has a similar flavour: they capture the extraction of structured data (like well-typed terms or error diagnostics) from less structured data (like pre-terms) by showing that the latter can be viewed as the *forgetful image* of the former. Let us warm up by considering variables.

7.1 The **find** *view*

We may define the *membership* relation of a list inductively as follows:

$$\underline{\text{data}}\ \frac{xs\ :\ \mathsf{List}\,X\quad x\ :\ X}{\mathsf{In}\,xs\ x\ :\ \star}\ \ \underline{\text{where}}\ \ \frac{}{\bullet\ :\ \mathsf{In}\,(x :: xs)\ x}\ \ \frac{i\ :\ \mathsf{In}\,xs\ y}{\uparrow i\ :\ \mathsf{In}\,(x :: xs)\ y}$$

An element of $\mathsf{In}\,xs\ x$ encodes a reference to a particular x in a list xs. We think of such a reference as a de Bruijn index into a list, *labelled* by the x to which it points, which is why we have overloaded the constructors. We shall use $\mathsf{In}\,\Gamma\ S$ to represent variables of type S over contexts Γ in our definition of well-typed terms.

There is an obvious forgetful map $|i|^x$ from In to Fin, which strips the label. We usually overload such forgetful maps as $|-|$, superscripting what the map forgets, if we ourselves wish to remember it.

$$\underline{\text{let}}\ \ \frac{i\ :\ \mathsf{In}\,xs\ x}{|i|^x\ :\ \mathsf{Fin}\ |xs|}\qquad \begin{aligned} |\bullet|^x &\mapsto \bullet \\ |\uparrow i|^x &\mapsto \uparrow |i|^x \end{aligned}$$

If we have an unlabelled index in $\mathsf{Fin}\ |xs|$, we can look it up in xs by 'unforgetting' the label. That is, we explain how every unlabelled index arises as the forgetful image of a labelled index, by means of the following *view*:

$$\underline{\text{data}}\quad \frac{xs\ :\ \mathsf{List}\ X \qquad i\ :\ \mathsf{Fin}\ |xs|}{\mathsf{Find}\ xs\ i\ :\ \star} \qquad \underline{\text{where}}\quad \frac{i\ :\ \mathsf{In}\ xs\ x}{\mathsf{found}\ x\ i\ :\ \mathsf{Find}\ xs\ |i|^{x}}$$

$$\underline{\text{let}}\quad \frac{}{\mathbf{find}\ xs\ i\ :\ \mathsf{Find}\ xs\ i} \qquad \begin{array}{lll} \mathbf{find}\ (x :: xs)\quad \bullet & \mapsto & \mathsf{found}\ x\ \bullet \\ \mathbf{find}\ (x :: xs)\quad (\uparrow i) & \Leftarrow & \underline{\text{view}}\ \mathbf{find}\ xs\ i \\ \qquad\qquad (\uparrow |i|^{x}) & \mapsto & \mathsf{found}\ x\ (\uparrow i) \end{array}$$

This program fragment shows how we use this view:

$$\begin{array}{lll} \mathbf{check}\ \Gamma\quad (\mathsf{eVar}\ i) & \Leftarrow & \underline{\text{view}}\ \mathbf{find}\ \Gamma\ i \\ \qquad (\mathsf{eVar}\ |i|^{S}) & \mapsto & \cdots \end{array}$$

7.2 The type of well-typed terms

Now that we can represent typed variables, let us define the well-typed terms, in a similar fashion to (Altenkirch & Reus, 1999):

$$\underline{\text{data}}\quad \frac{\Gamma\ :\ \mathsf{List}\ \mathsf{TExp} \qquad T\ :\ \mathsf{TExp}}{\mathsf{Term}\ \Gamma\ T\ :\ \star}$$

$$\underline{\text{where}}\quad \frac{i\ :\ \mathsf{In}\ \Gamma\ S}{\mathsf{var}\ i\ :\ \mathsf{Term}\ \Gamma\ S} \qquad \frac{t\ :\ \mathsf{Term}\ (S :: \Gamma)\ T}{\mathsf{lam}\ S\ t\ :\ \mathsf{Term}\ \Gamma\ (S \Rightarrow T)}$$

$$\frac{f\ :\ \mathsf{Term}\ \Gamma\ (S \Rightarrow T) \qquad s\ :\ \mathsf{Term}\ \Gamma\ S}{\mathsf{app}\ f\ s\ :\ \mathsf{Term}\ \Gamma\ T}$$

These constructors just give the typing rules in syntax-directed form. There is an obvious forgetful map from Term to Expr:

$$\underline{\text{let}}\quad \frac{t\ :\ \mathsf{Term}\ \Gamma\ T}{|t|^{T}\ :\ \mathsf{Expr}\ |\Gamma|} \qquad \begin{array}{lll} |\mathsf{var}\ i|^{S} & \mapsto & \mathsf{eVar}\ |i|^{S} \\ |\mathsf{lam}\ S\ t|^{S \Rightarrow T} & \mapsto & \mathsf{eLam}\ S\ |t|^{T} \\ |\mathsf{app}\ f\ s|^{T} & \mapsto & \mathsf{eApp}\ |f|^{S \Rightarrow T}\ |s|^{S} \end{array}$$

7.3 The eq? *view*

Imagine we are in the process of typechecking an application. On one hand, we have a function, which we have checked has an $\Rightarrow$-type: that is, we have some $|f|^{S \Rightarrow T}$. On the other, we have an argument, which is some well-typed term $|s|^{A}$. What we do not yet know is whether S and A are the *same*. How will we find out?

We could compute the value of $S == A$, the usual Boolean equality test. If false, the application is ill-typed, so we can reject it. But if true, whilst *we* may know that $==$ tests equality the *typechecker* just knows that $S, A : \mathsf{TExp}$; $\mathsf{true}\ :\ \mathsf{Bool}$. A successful $==$ test does not tell the typechecker that S and A are the same, hence we cannot yet build $\mathsf{app}\ f\ s$. The trouble is that a Bool is a bit uninformative. We can remedy this by presenting equality via a *view*.

As usual, we declare a relation

The positive cases of eq?

$$\underline{\text{let}}\quad \frac{}{\mathbf{eq?}\ S\ T\ :\ \mathsf{Eq?}\ S\ T}$$

$$\begin{array}{llll}
\mathbf{eq?} & \mathsf{o} & \mathsf{o} & \mapsto\ \mathsf{same} \\
\mathbf{eq?} & \mathsf{o} & (S_2 \Rightarrow T_2) & \mapsto\ \mathsf{diff}\ ?_1 \\
\mathbf{eq?} & (S_1 \Rightarrow T_1) & \mathsf{o} & \mapsto\ \mathsf{diff}\ ?_2 \\
\mathbf{eq?} & (S_1 \Rightarrow T_1) & (S_2 \Rightarrow T_2) & \Leftarrow\ \underline{\text{view}}\ \mathbf{eq?}\ S_1\ S_2 \\
\mathbf{eq?} & (S \Rightarrow T_1) & (S \Rightarrow T_2) & \Leftarrow\ \underline{\text{view}}\ \mathbf{eq?}\ T_1\ T_2 \\
\mathbf{eq?} & (S \Rightarrow T) & (S \Rightarrow T) & \mapsto\ \mathsf{same} \\
\mathbf{eq?} & (S \Rightarrow T) & (S \Rightarrow T'\backslash T) & \mapsto\ \mathsf{diff}\ ?_3 \\
\mathbf{eq?} & (S \Rightarrow T_1) & (S'\backslash S \Rightarrow T_2) & \mapsto\ \mathsf{diff}\ ?_4
\end{array}$$

Filling in the negative cases

$$\begin{array}{lc|l}
 & \underline{\text{data}}\ \dfrac{S\ :\ \mathsf{TExp}}{\mathsf{Isnt}\ S\ :\ \star}\ \underline{\text{where}} & \underline{\text{let}}\ \dfrac{T\ :\ \mathsf{Isnt}\ S}{T\backslash S\ :\ \mathsf{TExp}} \\
[?_1] & \dfrac{}{\mathsf{isnto}\ S_2\ T_2\ :\ \mathsf{Isnt}\ \mathsf{o}} & \mathsf{isnto}\ S_2\ T_2\ \backslash\ \mathsf{o} \mapsto S_2 \Rightarrow T_2 \\
[?_2] & \dfrac{}{\mathsf{isnt}{\Rightarrow}\ S_1\ T_1\ :\ \mathsf{Isnt}\ (S_1 \Rightarrow T_1)} & \mathsf{isnt}{\Rightarrow}\ S_1\ T_1\ \backslash\ (S_1 \Rightarrow T_1) \mapsto \mathsf{o} \\
[?_3] & \dfrac{T'\ :\ \mathsf{Isnt}\ T}{\mathsf{isntR}\ T'\ :\ \mathsf{Isnt}\ (S \Rightarrow T)} & \mathsf{isntR}\ T'\ \backslash\ (S \Rightarrow T) \mapsto S \Rightarrow T'\backslash T \\
[?_4] & \dfrac{S'\ :\ \mathsf{Isnt}\ S \quad T_2\ :\ \mathsf{TExp}}{\mathsf{isntL}\ S'\ T_2\ :\ \mathsf{Isnt}\ (S \Rightarrow T_1)} & \mathsf{isntL}\ S'\ T_2\ \backslash\ (S \Rightarrow T_1) \mapsto S'\backslash S \Rightarrow T_2
\end{array}$$

Fig. 15. The equality view

$$\underline{\text{data}}\quad \frac{S, T\ :\ \mathsf{TExp}}{\mathsf{Eq?}\ S\ T\ :\ \star}\quad \underline{\text{where}}\quad \frac{}{\mathsf{same}\ :\ \mathsf{Eq?}\ S\ S}\quad \frac{T\ :\ \mathsf{Isnt}\ S}{\mathsf{diff}\ T\ :\ \mathsf{Eq?}\ S\ (T\backslash S)}$$

The first constructor is clear enough, but what is this $\mathsf{Isnt}\ S$, and what is $(S\backslash T)$? The former is a type representing evidence of difference from S, and the latter is its forgetful map back to TExp (which binds more tightly than $\Rightarrow$). We do not write $\lfloor T \rfloor^S$, to avoid clashing with the forgetful map for Term. There are many ways to define Isnt. One obvious candidate is to use existential quantification (or **dependent pairs**).

$$\mathsf{Isnt}\ S \ \mapsto\ \exists T : \mathsf{TExp}.\ S = T \to \bot \qquad (T, p)\backslash S\ \mapsto\ T$$

Another possibility is to define Isnt by recursion on S. We shall declare it as a datatype family, but we defer the definition until after our first attempt to write the covering function, **eq?**. At the top of Figure 15, we write what we can without fully declaring Isnt.

Now, we need elements of Isnt types in four places—two for 'different constructors', and two for differences left or right of $\Rightarrow$. The easiest way to define Isnt is just to give it constructors for these cases, packing up exactly the information available where they are used. The constructor forms declared at the bottom of Figure 15 go in the 'holes in the program' as indicated. Or rather, the constructor forms *come from*

The positive cases of check

$$\underline{\text{data}}\quad \frac{\Gamma\ :\ \mathsf{List\ TExp}\qquad e\ :\ \mathsf{Expr}\ |\Gamma|}{\mathsf{Check}\ \Gamma\ e\ :\ \star}$$

$$\underline{\text{where}}\quad \frac{t\ :\ \mathsf{Term}\ \Gamma\ T}{\mathsf{term}\ T\ t\ :\ \mathsf{Check}\ \Gamma\ |t|^{T}}\qquad \frac{err\ :\ \mathsf{Error}\ \Gamma}{\mathsf{error}\ err\ :\ \mathsf{Check}\ \Gamma\ |err|}$$

$$\underline{\text{let}}\quad \frac{}{\mathbf{check}\ \Gamma\ e\ :\ \mathsf{Check}\ \Gamma\ e}$$

$$\begin{array}{l}
\mathbf{check}\ \Gamma\ (\mathsf{eVar}\ i\ ) \Leftarrow \underline{\text{view}}\ \mathbf{find}\ \Gamma\ i\\
\quad \mathbf{check}\ \Gamma\ (\mathsf{eVar}\ |i|^{S}) \mapsto \mathsf{term}\ S\ (\mathsf{var}\ i)\\
\mathbf{check}\ \Gamma\ (\mathsf{eLam}\ S\ t\ ) \Leftarrow \underline{\text{view}}\ \mathbf{check}\ (S :: \Gamma)\ t\\
\quad \mathbf{check}\ \Gamma\ (\mathsf{eLam}\ S\ |t|^{T}\ ) \mapsto \mathsf{term}\ (S \Rightarrow T)\ (\mathsf{lam}\ S\ t)\\
\quad \mathbf{check}\ \Gamma\ (\mathsf{eLam}\ S\ |err|) \mapsto \mathsf{error}\ ?_1\\
\mathbf{check}\ \Gamma\ (\mathsf{eApp}\ f\ s\ ) \Leftarrow \underline{\text{view}}\ \mathbf{check}\ \Gamma\ f\\
\quad \mathbf{check}\ \Gamma\ (\mathsf{eApp}\ |f|^{\mathsf{o}}\ s\ ) \mapsto \mathsf{error}\ ?_2\\
\quad \mathbf{check}\ \Gamma\ (\mathsf{eApp}\ |f|^{S\Rightarrow T}\ s\ ) \Leftarrow \underline{\text{view}}\ \mathbf{check}\ \Gamma\ s\\
\quad\quad \mathbf{check}\ \Gamma\ (\mathsf{eApp}\ |f|^{S\Rightarrow T}\ |s|^{A}\ ) \Leftarrow \underline{\text{view}}\ \mathbf{eq?}\ S\ A\\
\quad\quad\quad \mathbf{check}\ \Gamma\ (\mathsf{eApp}\ |f|^{S\Rightarrow T}\ |s|^{S}\ ) \mapsto \mathsf{term}\ T\ (\mathsf{app}\ f\ s)\\
\quad\quad\quad \mathbf{check}\ \Gamma\ (\mathsf{eApp}\ |f|^{S\Rightarrow T}\ |s|^{A\backslash S}) \mapsto \mathsf{error}\ ?_3\\
\quad\quad \mathbf{check}\ \Gamma\ (\mathsf{eApp}\ |f|^{S\Rightarrow T}\ |err|\ ) \mapsto \mathsf{error}\ ?_4\\
\quad \mathbf{check}\ \Gamma\ (\mathsf{eApp}\ |err|\ s\ ) \mapsto \mathsf{error}\ ?_5
\end{array}$$

Filling in the negative cases

$$\begin{array}{l|l}
\underline{\text{data}}\quad \dfrac{\Gamma\ :\ \mathsf{List\ TExp}}{\mathsf{Error}\ \Gamma\ :\ \star}\quad \underline{\text{where}} & \underline{\text{let}}\quad \dfrac{e\ :\ \mathsf{Error}\ \Gamma}{|e|\ :\ \mathsf{Expr}\ |\Gamma|}\\
[?_1]\quad \dfrac{err\ :\ \mathsf{Error}\ (S :: \Gamma)}{\mathsf{bodyE}\ S\ err\ :\ \mathsf{Error}\ \Gamma} & |\mathsf{bodyE}\ S\ err| \mapsto \\ & \quad \mathsf{eLam}\ S\ |err|\\
[?_2]\quad \dfrac{f\ :\ \mathsf{Term}\ \Gamma\ \mathsf{o}\qquad s\ :\ \mathsf{Expr}\ |\Gamma|}{\mathsf{notFunE}\ f\ s\ :\ \mathsf{Error}\ \Gamma} & |\mathsf{notFunE}\ f\ s| \mapsto \\ & \quad \mathsf{eApp}\ |f|^{\mathsf{o}}\ s\\
[?_3]\quad \dfrac{f\ :\ \mathsf{Term}\ \Gamma\ (S \Rightarrow T)\qquad s\ :\ \mathsf{Term}\ \Gamma\ (A\backslash S)}{\mathsf{mismatchE}\ f\ s\ :\ \mathsf{Error}\ \Gamma} & |\mathsf{mismatchE}\ f\ s| \mapsto \\ & \quad \mathsf{eApp}\ |f|^{S\Rightarrow T}\ |s|^{A\backslash S}\\
[?_4]\quad \dfrac{f\ :\ \mathsf{Term}\ \Gamma\ (S \Rightarrow T)\qquad err\ :\ \mathsf{Error}\ \Gamma}{\mathsf{argE}\ f\ err\ :\ \mathsf{Error}\ \Gamma} & |\mathsf{argE}\ f\ err| \mapsto \\ & \quad \mathsf{eApp}\ |f|^{S\Rightarrow T}\ |err|\\
[?_5]\quad \dfrac{err\ :\ \mathsf{Error}\ \Gamma\qquad s\ :\ \mathsf{Expr}\ |\Gamma|}{\mathsf{funE}\ err\ s\ :\ \mathsf{Error}\ \Gamma} & |\mathsf{funE}\ err\ s| \mapsto \\ & \quad \mathsf{eApp}\ |err|\ s
\end{array}$$

Fig. 16. The typechecking view

the holes in the program as indicated. The forgetful map is generated accordingly. We see no reason why, in an interactive setting, we cannot extract the 'remainder' family from the unsolved programming problems.

We are now ready to write the typechecker.

7.4 The **check** *view*

We define typechecking as a view $\mathsf{Check}\,\Gamma\;e$ on contexts and pre-terms, expressing any e : $\mathsf{Expr}\;|\Gamma|$ as the forgetful image either of a Term, or of an Error. Again, we shall defer giving the constructors of Error until we have identified the holes in the program **check** Γ e which establishes the view. At the top of Figure 16, we develop the algorithm as usual, by case analysis on e, followed by recursive calls to **check**:

- in the eVar case, there is nothing further to do, as variables are well-scoped; it suffices to look up the type from the context, using the **find** view;
- in the eLam case, we typecheck the body in an extended context;
- in the eApp case, we successively check first the function, then the argument, and finally match the computed types using the **eq**? view.

The view of each recursive call on **check**, yields two cases, according as typechecking succeeds or fails; in the case of success, the pattern lays bare precisely the data required for the next call. As with the equality view, we now choose constructors and define a forgetful map for Error with which we can fill in the five remaining holes, packing up the information exposed by each of the possible sources of typechecking failure—see the bottom of Figure 16.

The function **check** is not just a program: it is a *proof* that typechecking is decidable for the pre-terms. It does not merely say 'yes' or 'no', but rather explains each pre-term as deriving, by a forgetful map, either from a well-typed term or an error term. Its type guarantees that the term being checked really is the term it is given. Its analysis is concisely stated and imposes the conditions for well-typedness (and its complement) just as they are expressed by the typing rules.

Moreover, as its recursive calls show, it represents these two possibilities in a 'pattern matching' style, visibly delivering either a well-typed term which may be passed to an exception-free interpreter in the style of Augustsson and Carlsson (Augustsson & Carlsson, 1999), or a useful error diagnostic. The latter locates the *leftmost* type error in a pre-term. It could easily be adapted to find *every* application of a well-typed non-function or mismatched application between two well-typed terms—useful information not only for error reporting, but also for type debugging and repair, as investigated by McAdam (1999).

Epilogue

The main discovery we have made in the light of this research is how little is known, not least by ourselves, about functional programming with dependent types. It is no longer credible to conceive of dependently typed programming merely as a means to relegitimize programs which were lost to us when we moved from untyped languages to the Hindley-Milner system. We take its inherent complexity as an *opportunity*, rather than a *problem*, and in so doing, we see emerging a very different possibility for declarative programming, which we have barely begun to explore.

This paper has introduced a specific programming notation on top of an existing type theory, and shown in detail, through examples and a skeletal formal definition which explains how the main constructs are translated, some of the power, as well as weight, that is available in this new world. We have extended the notion of 'pattern matching' to embrace any user-definable structured decomposition of data on the left, including the use of, and interplay with, intermediate computations and result types. We have further related our work specifically to two proposals in the functional programming community for extensions to the classical notion of pattern matching, Peyton Jones' pattern guards (1997), and Wadler's views (1987).

The former remarks that the potential uses of pattern guards are, can, and should be ubiquitous, as they allow "a useful class of programs to be written much more elegantly". We would certainly argue that this is all the more surely the case in our setting—with the greater expressivity available with dependent types, that class of programs becomes much more interesting. And in our notation, we would argue, without any loss of that elegance. Neither we, nor anyone else for that matter, have even begun to exhaust the possibilities of programming in such a style.

As to the latter, we have given a thorough analysis of how views may be presented using dependent types, as well as variety of examples of views, and uses of views not previously considered in the literature. Our general picture allows us to consider partial and ambiguous views, to explore trade-offs between recursive and non-recursive views, as well as looking at termination proofs and varieties of recursion induction (Bove & Capretta, 2001).

More generally, we take the explosion of power which dependent types bring to programming, as delineated in Section 3 as a cue to re-evaluate design choices about the language within which we express programs, the tools with which we construct programs, and the programs we choose to write in the first place. This includes reassessing the interfaces and implementations of standard data structures and algorithms, no less than any other programs.

We believe that such new languages, tools and libraries as emerge in the future will also profit considerably from the experience gained in the wider domain of interactive problem-solving with dependent types. While we have downplayed that aspect of our research in this paper, our new analysis of the left-hand sides of functional programs is strongly rooted in logical considerations and the techniques which are supported by existing interactive proof assistants based on type theory. We intend in future work to elaborate on these aspects, and the contribution our notation may make to declarative *proof*.

There is much work to do here in building such a future—in Durham, we have dubbed our programme of research EPIGRAM, embracing language, meta-theory, implementation and applications. The first author's experimental extensions to

LEGO (1999; 2002) provided tactics for inductive proof supporting the constructions which underpin the [by] and [with] elaboration rules. These tactics are sufficient to develop the examples in this paper, but do not support a concrete syntax for programs as such.

This paper lays the groundwork for a formal language definition for EPIGRAM; we are now working on a new prototype implementation based on this definition. Clearly many interesting issues remain to be explored, not least at the run-time level, studying the operational behaviour of elaborated programs.

In closing, we return to Wadler, crediting him with the insight that, by constructing views, we can and should choose to adapt our perceptions of data to match our conceptions of data. We are able to reify his views directly, by using dependent types, and by our treatment of the left. So hurrah for Wadler! Welcome to the new programming.

References

Abel, Andreas, & Altenkirch, Thorsten. (2000). A predicative analysis of structural recursion. *Journal of Functional Programming*.

Altenkirch, Thorsten, & McBride, Conor. (2002). Generic Programming within Dependently Typed Programming. Gibbons, Jeremy, & Jeuring, Johan (eds), *Proceedings of the IFIP 2.1 Working Conference on Generic Programming, 2002*. Kluwer.

Altenkirch, Thorsten, & Reus, Bernhard. (1999). Monadic presentations of lambda-terms using generalized inductive types. *Computer Science Logic 1999*.

Augustsson, Lennart. (1985). Compiling Pattern Matching. *Pages 368–381 of:* Jouannaud, Jean-Pierre (ed), *Functional Programming Languages and Computer Architecture*. LNCS, vol. 201. Springer-Verlag.

Augustsson, Lennart. (1998). Cayenne—a language with dependent types. *ACM International Conference on Functional Programming '98*. ACM.

Augustsson, Lennart, & Carlsson, Magnus. (1999). *An exercise in dependent types: A well-typed interpreter*. Available at `http://www.cs.chalmers.se/~augustss/cayenne/interp.ps`.

Barendregt, Henk. (1992). Lambda Calculi with Types. Abramsky, Samson, Gabbay, Dov, & Maibaum, Tom (eds), *Handbook of Logic in Computer Science*, vol. II. Oxford University Press.

Bird, Richard, & Meertens, Lambert. (1998). Nested Datatypes. *Pages 52–67 of: Mathematics of Program Construction*. LNCS, vol. 1422. Springer-Verlag.

Bird, Richard, & Paterson, Ross. (1999). de Bruijn notation as a nested datatype. *Journal of Functional Programming*, **9**(1), 77–92.

Bove, Ana, & Capretta, Venanzio. (2001). Nested General Recursion and Partiality in Type Theory. Richard Boulton and Paul Jackson (ed), *Theorem Proving in Higher Order Logics, TPHOLs 2001*. LNCS, vol. 2152. Springer-Verlag.

Burstall, Rod. (1969). Proving properties of programs by structural induction. *Computer Journal*, **12**(1), 41–48.

Burton, Warren, Meijer, Erik, Samson, Patrick, Thompson, Simon, & Wadler, Philip. (1996). *Views: An Extension to Haskell Pattern Matching*. Available from `http://www.haskell.org/development/views.html`.

Callaghan, Paul, & Luo, Zhaohui. (2000). Implementation Techniques for Inductive Types in Plastic. *Pages 94–113 of: Proceedings TYPES'99*. LNCS, vol. 1956. Springer-Verlag.

Clark, Keith. (1978). Negation as failure. *Pages 292–322 of:* Hervé Gallaire and Jack Minker (ed), *Logic and data bases*. Plenum Press.

Coq, L'Équipe. (2001). *The Coq Proof Assistant Reference Manual*. `http://pauillac.inria.fr/coq/doc/main.html`.

Coquand, Thierry. (1992). Pattern Matching with Dependent Types. Nordström, Bengt, Petersson, Kent, & Plotkin, Gordon (eds), *Electronic Proceedings of the Third Annual BRA Workshop on Logical Frameworks (Båstad, Sweden)*. Available in `http://www.lfcs.informatics.ed.ac.uk/research/types-bra/proc/proc92.ps.gz`.

Cornes, Cristina. (1997). *Conception d'un langage de haut niveau de répresentation de preuves*. Ph.D. thesis, Université Paris VII.

de Bruijn, Nicolas G. (1972). Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes mathematicæ*, **34**, 381–392.

de Bruijn, Nicolas G. (1991). Telescopic Mappings in Typed Lambda-Calculus. *Information and computation*, **91**, 189–204.

Dybjer, Peter. (1991). Inductive Sets and Families in Martin-Löf's Type Theory. Huet, Gérard, & Plotkin, Gordon (eds), *Logical Frameworks*. CUP.

Giménez, Eduardo. (1994). Codifying guarded definitions with recursive schemes. *Pages 39–59 of:* Dybjer, Peter, Nordström, Bengt, & Smith, Jan (eds), *Types for Proofs and Programs, '94*. LNCS, vol. 996. Springer-Verlag.

Giménez, Eduardo. (1998). Structural Recursive Definitions in Type Theory. *Proceedings of ICALP '98*. LNCS, vol. 1443. Springer-Verlag.

Goguen, Healfdene. (1994). *A Typed Operational Semantics for Type Theory*. Ph.D. thesis, Laboratory for Foundations of Computer Science, University of Edinburgh. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/94/ECS-LFCS-94-304/`.

Harper, Robert, & Pollack, Randy. (1991). Type checking with universes. *Theoretical Computer Science*, **89**, 107–136.

Hofmann, Martin, & Streicher, Thomas. (1994). A groupoid model refutes uniqueness of identity proofs. *Pages 208–212 of: Proc. Ninth Annual Symposium on Logic in Computer Science (LICS) (Paris, France)*. IEEE Computer Society Press.

Huet, Gérard, & Plotkin, Gordon (eds). (1990). *Electronic Proceedings of the First Annual BRA Workshop on Logical Frameworks (Antibes, France)*. Available in `http://www.lfcs.informatics.ed.ac.uk/research/types-bra/proc/proc90.ps.gz`.

Leijen, Daan, & Meijer, Erik. (1999). Domain specific embedded compilers. *2nd Conference on Domain-Specific Languages (DSL)*. USENIX. Available from `http://www.cs.uu.nl/people/daan/papers/dsec.ps`.

Luo, Zhaohui. (1990). *ECC: An Extended Calculus of Constructions*. Ph.D. thesis, University of Edinburgh. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/90/ECS-LFCS-90-118/`.

Luo, Zhaohui. (1994). *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press.

Luo, Zhaohui, & Pollack, Robert. (1992). *LEGO Proof Development System: User's Manual*. Tech. rept. ECS-LFCS-92-211. Laboratory for Foundations of Computer Science, University of Edinburgh.

Magnusson, Lena. (1994). *The implementation of ALF—A Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. Ph.D. thesis, Chalmers University of Technology, Göteborg.

McAdam, Bruce J. (1999). Generalising techniques for type explanation. *Pages 243–252* *of: Scottish functional programming workshop.* Heriot-Watt Department of Computing and Electrical Engineering Technical Report RM/99/9.

McBride, Conor. (1998). Inverting inductively defined relations in LEGO. *Pages 236–253 of:* Giménez, E., & Paulin-Mohring, C. (eds), *Types for proofs and programs, '96.* LNCS, vol. 1512. Springer-Verlag.

McBride, Conor. (1999). *Dependently Typed Functional Programs and their Proofs.* Ph.D. thesis, University of Edinburgh. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.

McBride, Conor. (2001). *First-Order Unification by Structural Recursion.* To appear in the Journal of Functional Programming.

McBride, Conor. (2002). Elimination with a Motive. Callaghan, Paul, Luo, Zhaohui, McKinna, James, & Pollack, Robert (eds), *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00).* LNCS, vol. 2277. Springer-Verlag.

McBride, Fred. (1970). *Computer aided manipulation of symbols.* Ph.D. thesis, Queen's University of Belfast.

McKinna, James. (2002). *Views for recursion.* Talk given at the workshop on Termination and Type Theory, Hindås, Sweden.

McKinna, James, & Pollack, Robert. (1993). Pure type systems formalized. Bezem, Marc, & Groote, Jan-Friso (eds), *Int. Conf. Typed Lambda Calculi and Applications TLCA'93.* LNCS, vol. 664. Springer-Verlag.

McKinna, James, & Pollack, Robert. (1999). Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, **23**, 373–409. (Special Issue on Formal Proof, editors Gail Pieper and Frank Pfenning).

Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, David. (1997). *The Definition of Standard ML, revised edition.* MIT Press.

Peyton Jones, Simon. (1997). *A new view of guards.* Available from `http://research.microsoft.com/Users/simonpj/Haskell/guards.html`.

Peyton Jones, Simon, & Erwig, Martin. (2000). *Pattern guards and transformational patterns.* In Proceedings of the 2000 Haskell Workshop. Available from `http://research.microsoft.com/Users/simonpj/Haskell/pat.ps.gz`.

Peyton Jones, Simon, & Hughes, John (eds). (1999). *Haskell'98: A Non-Strict Functional Language.* Available from `http://www.haskell.org/definition`.

Pollack, Robert. (1992). *Implicit syntax.* Available from `ftp://ftp.dcs.ed.ac.uk/pub/lego/ImplicitSyntax.ps.Z`. An earlier version of this paper appeared in (Huet & Plotkin, 1990).

Pollack, Robert. (1994). *Incremental Changes in LEGO:1994.* Available from `ftp://ftp.dcs.ed.ac.uk/pub/lego/changes94.ps.gz`.

Pollack, Robert. (1995). *The Theory of LEGO.* Ph.D. thesis, University of Edinburgh. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/95/ECS-LFCS-95-323/`.

Pollack, Robert. (2000). Dependently Typed Records for Representing Mathematical Structure. Aagard, Mark, & Harrison, John (eds), *Theorem Proving in Higher Order Logics, TPHOLs 2000.* LNCS, vol. 1869. Springer-Verlag.

Nordström, Bengt, Petersson, Kent, & Smith, Jan. (1990). *Programming in Martin-Löf's type theory: an introduction.* Oxford University Press.

Severi, Paula, & Poll, Erik. (1994). Pure Type Systems with definitions. *Pages 316–328 of:* Anil Nerode and Yuri Matijasevič (ed), *Proceedings of LFCS'94.* LNCS, vol. 813. Springer-Verlag.

Streicher, Thomas. (1993). *Investigations into intensional type theory.* Habilitation Thesis, Ludwig Maximilian Universität.

Tamaki, Hisao, & Sato, Taisuke. (1984). Unfold/fold transformation of logic programs. *Pages 127–138 of:* Sten-Åke Tårnlund (ed), *Proceedings 2nd International Logic Programming Conference.*

van Benthem Jutting, Bert, McKinna, James, & Pollack, Robert. (1994). Checking Algorithms for Pure Type Systems. Barendregt, Henk, & Nipkow, Tobias (eds), *Types for proofs and programs.* LNCS 806. Springer-Verlag. Selected papers from the Int. Workshop TYPES '93, Nijmegen, May 1993.

Voda, Paul. (2002). *What do we gain by integrating a programming language with a theorem prover?* Talk given at the workshop on Termination and Type Theory, Hindås, Sweden.

Wadler, Philip. (1987). Views: A way for pattern matching to cohabit with data abstraction. *Proceedings of POPL '87.* ACM.

Wadler, Philip. (1989). Theorems for Free! *Proceedings of FPCA '89.* ACM.

Wadler, Philip. (1990). Deforestation: transforming programs to eliminate trees. *Theoretical computer science*, **73**, 231–248. (Special issue of selected papers from 2'nd ESOP.).

Xi, Hongwei. (1998). *Dependent types in practical programming.* Ph.D. thesis, Department of Mathematical Sciences, Carnegie Mellon University.