Elimination with a Motive

Conor McBride

Department of Computer Science
University of Durham

Abstract. I present a tactic, `BasicElim`, for Type Theory based proof systems to apply *elimination rules* in a refinement setting. Applicable rules are *parametric* in their conclusion, expressing the leverage hypotheses $\vec{x}$ yield on any $\Phi\,\vec{x}$ we choose. Φ represents the *motive* for an elimination: `BasicElim`'s job is to construct a Φ suited to the goal at hand.
If these $\vec{x}$ inhabit an *instance* of Φ's domain, I adopt a technique standard in 'folklore', generalizing the $\vec{x}$ and expressing the restriction by equation. A novel notion of $=$ readily permits dependent equations, and a second tactic, `Unify`, simpifies the equational hypotheses thus appearing in subgoals.
Given such technology, it becomes effective to express properties of datatypes, relations and functions in this style. A small extension couples `BasicElim` with rewriting, allowing complex techniques to be packaged in a single rule.

1 Introduction

Computations on datatypes in the proof assistant LEGO [6], are by 'elimination rules' playing the dual rôle of 'induction principle' and 'primitive recursor'. During my PhD [8], I developed technology to help working with these rules in the cause of programming. However, this technology soon acquired wider applications: it works with any theorem resembling a datatype elimination rule. It thus pays to characterize many kinds of information in this style. My claim is that we should exploit a hypothesis not in terms of its immediate consequences, but in terms of the leverage it exerts on an arbitrary goal: we should give elimination a *motive*.

The technical purpose of this paper is to document this elimination technology for the benefit of other implementers. Its more political purpose is to persuade users to work with the properties they need in the style supported by this technology.

2 Motivation

I shall introduce the issues with the help of some examples.

2.1 Conjunction and Disjunction

Undergraduates learning natural deduction (such as myself, once upon a time) are typically taught the following elimination rules for conjunction and disjunction:

$$\wedge\textbf{-project-l}\ \frac{A \wedge B}{A} \quad \wedge\textbf{-project-r}\ \frac{A \wedge B}{B} \qquad \vee\textbf{-elim}\ \frac{A \vee B \quad \begin{matrix}[A]\\ \vdots \\ C\end{matrix} \quad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}}{C}$$

Students usually grasp the $\wedge$**-project-** rules easily, but finds $\vee$**-elim** frightening. The trouble is that C appears from nowhere: students struggle to dream up C's which will eventually lead to their goal. But when they learn proof by *refinement*, $\vee$**-elim** finally makes sense: given $A \vee B$, we can instantiate C with the our *goal*, splitting it into cases. The need to prove the goal is why we are eliminating $A \vee B$: the goal is the *motive* for the elimination. We can choose the appropriate motive precisely because 'C appears from nowhere': $\vee$**-elim** is *parametric* in its motive.

Elimination rules whose conclusion is a parameter—the *motive variable*—allow us to exploit a hypothesis whatever the goal, just as 'left rules' in sequent calculus analyse hypotheses regardless of what stands right of the turnstile. In this light, the 'simplicity' of the $\wedge$**-project-** rules is less attractive: we may only exploit $A \wedge B$ when we want to know A or we want to know B. I join the many advocates of the 'Gentzenized' alternative, exploiting $A \wedge B$, whatever our motive C:

$$\wedge\textbf{-elim}\ \frac{A \wedge B \quad \begin{array}{c}[A]\\ [B]\\ \vdots \\ C\end{array}}{C}$$

2.2 Structural Induction and Recursion

'Mathematical Induction' is another common example of elimination with a motive:

$$\mathit{\Phi}\ :\ \mathbb{N} \to \mathsf{Prop}$$

$$\mathbb{N}\textbf{-induction}\ \frac{\mathit{\Phi}\ \mathsf{0} \qquad \begin{array}{c}\mathit{\Phi}\ n\\ \cdots\cdots\\ \mathit{\Phi}\ (\mathsf{s}\ n)\end{array}}{\forall n\!:\!\mathbb{N}.\ \mathit{\Phi}\ n}$$

Here $\mathit{\Phi}$ stands for a family of propositions *indexed* by a number. Not even the most ardent 'forwardist' is bold enough to suggest that we should search our collection of established facts for a pair related by such a $\mathit{\Phi}$ in order to add $\forall n\!:\!\mathbb{N}.\ \mathit{\Phi}\ n$ to that collection. $\mathbb{N}$**-induction** needs a motive not only because, like $\vee$**-elim**, it splits the proof into cases, but also because the abstract n being eliminated is instantiated in each case. The point of induction is not just to decompose a hypothesis but to simplify the goal: where constructor symbols appear, computation can happen.

If we allow $\mathit{\Phi}$ to stand for a $\mathbb{N}$-indexed family of *types* and supply the appropriate computational behaviour, induction becomes dependently typed primitive recursion, supporting functions on n whose return type depends on n. The explicit indexing of $\mathit{\Phi}$ by numbers makes a strong connection to pattern matching and structural recursion. We can expose these patterns even in simply typed examples by making a definition:[1]

$$\begin{array}{rl}
\text{definition} & \textbf{Plus} \mapsto \lambda x, y\!:\!\mathbb{N}.\ \mathbb{N} \\[1em]
\text{goal} & ?:\ \forall x, y\!:\!\mathbb{N}.\ \textbf{Plus}\ x\ y \\[1em]
\text{induction base} & ?:\ \forall y\!:\!\mathbb{N}.\ \textbf{Plus}\ \mathsf{0}\ y \\
\text{induction step} & ?:\ \forall x\!:\!\mathbb{N}.\ (\forall y\!:\!\mathbb{N}.\ \textbf{Plus}\ x\ y) \to \forall y\!:\!\mathbb{N}.\ \textbf{Plus}\ (\mathsf{s}\ x)\ y
\end{array}$$

[1] I use $\mapsto$ for directed *computational* equalities, reserving $=$ for *propositional* equality.

The return type of the goal reads like the left-hand side of a functional program 'under construction'. Induction splits our programming problem in two: we can read off the instantiated patterns and, in the s case, the legitimate class of recursive calls.

An elimination rule with an indexed motive variable Φ justifies a kind of pattern analysis, 'matching' Φ's arguments in the conclusion (the *goal patterns*) against Φ's arguments in the premises (the *subgoal patterns*): Φ's arguments in inductive hypotheses (the *recursion patterns*) allow the corresponding recursive calls. To equip an elimination rule with a computational behaviour is to give its associated pattern matching and structural recursion an operational semantics.

2.3 Relation Induction

Inductively defined relations may also be presented with an elimination rule corresponding to induction on derivations. For example, $\leq$ may be defined as follows:[2]

$$\frac{m, n \;:\; \mathbb{N}}{m \leq n \;:\; \mathsf{Prop}} \qquad \Phi \;:\; \mathbb{N} \to \mathbb{N} \to \mathsf{Prop}$$

$$\frac{}{n \leq n} \qquad \frac{m \leq n}{m \leq \mathsf{s}\,n} \qquad \underline{\leq}\textbf{-induction}\; \frac{\Phi\,n\,n \qquad \begin{array}{c} m \leq n \quad \Phi\,m\,n \\ \vdots \\ \Phi\,m\,(\mathsf{s}\,n)\end{array}}{\forall m, n\,|\,\mathbb{N}.\; m \leq n \to \Phi\,m\,n}$$

Relation induction is easy to apply if the eliminated hypothesis, $m \leq n$, ranges over the entire domain of the relation: we can choose Φ by naïve 'undergraduate' textual matching. However, if the hypothesis is *instantiated*, we still need a Φ indexed over the whole domain, so we employ a well known goal transformation which I learned from James McKinna—use a general $m \leq n$, but constrain m and n with *equations*:

goal	Φ
$? : \forall m, n\,|\,\mathbb{N}.\; m \leq n \to P[m, n]$	$\lambda m, n{:}\mathbb{N}.\; P[m, n]$
$? : \forall m\,|\,\mathbb{N}.\; m \leq 0 \to m = 0$	not obvious...
...so generalize and add an equation	
$? : \forall m, n\,|\,\mathbb{N}.\; m \leq n \to n = 0 \to m = 0$	$\lambda m, n{:}\mathbb{N}.\; m \leq n \to n = 0 \to m = 0$

More generally, an elimination rule for an inductive relation, $R \;:\; \forall \vec{x} : \vec{X}.\, \mathsf{Prop}$, typically requires some $\Phi \;:\; \forall \vec{x}{:}\vec{X}.\, \mathsf{Prop}$ as motive[3]

typical goal	typical Φ
$? : \forall \vec{y}{:}\vec{Y}.\, R\,\vec{t}[\vec{y}] \to P[\vec{y}]$	$\lambda \vec{x}{:}\vec{X}.\, \forall \vec{y}{:}\vec{Y}.\, \vec{x} = \vec{t}[\vec{y}] \to P[\vec{y}]$

Plugging this Φ and the proof of $R\,\vec{t}[\vec{y}]$ into the rule delivers a proof of $P[\vec{y}]$ subject to trivial equations $\vec{t}[\vec{y}] = \vec{t}[\vec{y}]$. This technique gives a slightly clumsier Φ than we chose for $m \leq 0$, which only constrains one argument, so needs only one equation. It is not hard to see how to remove the unnecessary equation.

Note that our chosen $\Phi\,\vec{x}$ resembles the goal, but with some equations inserted and $R\,\vec{t}[\vec{y}]$ missing. Φ is not indexed over the proof of $R\,\vec{x}$, so elimination tells us nothing about it: we can safely omit it from the motive.

[2] I adopt Pollack's convention, using $|$ when binding parameters I wish to keep implicit; I use subscripts when I need to make them explicit.

[3] $\vec{x} = \vec{t}[\vec{y}] \to$ denotes the batch of equations $x_1 = t_1[\vec{y}] \to \cdots x_n = t_n[\vec{y}] \to$

2.4 Induction for Dependent Datatypes

The datatype analogue of inductively defined relations are dependent families of datatypes, such as the vectors—lists of a given length—defined as follows:

$$\frac{A : \mathsf{Type} \quad n : \mathbb{N}}{\mathsf{Vect}\, A\, n : \mathsf{Type}} \qquad \frac{A : \mathsf{Type}}{\mathsf{vnil} : \mathsf{Vect}\, A\, 0} \qquad \frac{x : A \quad xs : \mathsf{Vect}\, A\, n}{\mathsf{vcons}\, x\, xs : \mathsf{Vect}\, A\, (\mathsf{s}\, n)}$$

$$\begin{array}{l} A \mid \mathsf{Type} \\ \Phi : \forall n|\mathbb{N}.\ \mathsf{Vect}\, A\, n \to \mathsf{Type} \end{array}$$

$$\mathsf{Vect}\text{-}\mathbf{elim}\ \frac{\Phi_0\ \mathsf{vnil} \qquad \begin{array}{c} x : A \quad \Phi_n\ xs \\ \cdots\cdots\cdots\cdots \\ \Phi_{\mathsf{s}\,n}\ (\mathsf{vcons}\ x\ xs) \end{array}}{\forall n|\mathbb{N}.\ \forall xs{:}\mathsf{Vect}\, A\, n.\ \Phi_n\ xs}$$

Proof terms for relations are interesting only for what they say about the indices—their structure is unimportant. The terms in dependent datatypes are the actual data. Correspondingly, the motive Φ of Vect-**elim** is indexed not only over the length n, but also over the vector itself: we care if a vector is vnil or vcons. On the other hand, Φ is not indexed over the element type A, which is *parametric* to the entire inductive definition.

'Constraint by equation' also works for instantiated datatypes. For example:

$$\begin{array}{rl}
\text{definition} & \mathbf{VTail} \mapsto \lambda A|\mathsf{Type}.\ \lambda m|\mathbb{N}.\ \lambda xxs{:}\mathsf{Vect}\ A\ (\mathsf{s}\ m).\ \mathsf{Vect}\ A\ m \\
\\
\text{goal} & ? : \forall A|\mathsf{Type}.\ \forall m|\mathbb{N}.\ \forall xxs{:}\mathsf{Vect}\ A\ (\mathsf{s}\ m).\ \mathbf{VTail}\ xxs \\
\\
\text{motive} & \Phi \mapsto \lambda n|\mathbb{N}.\ \lambda xs{:}\mathsf{Vect}\ A\ n.\ \forall m|\mathbb{N}.\ \forall xxs{:}\mathsf{Vect}\ A\ (\mathsf{s}\ m). \\
& \qquad\qquad\qquad\qquad n = \mathsf{s}\ m \to xs = xxs \to \\
& \qquad\qquad\qquad\qquad \mathbf{VTail}\ xxs \\
\\
\text{base case} & ? : \forall A|\mathsf{Type}.\ \forall m|\mathbb{N}.\ \forall xxs{:}\mathsf{Vect}\ A\ (\mathsf{s}\ m). \\
& \quad 0 = \mathsf{s}\ m \to \mathsf{vnil} = xxs \to \\
& \quad \mathbf{VTail}\ xxs \\
\text{step case} & ? : \forall A|\mathsf{Type}.\ \forall n|\mathbb{N}.\ \forall x{:}A.\ \forall xs{:}\mathsf{Vect}\ A\ n.\ \cdots \to \\
& \quad \forall m|\mathbb{N}.\ \forall xxs{:}\mathsf{Vect}\ A\ (\mathsf{s}\ m). \\
& \quad \mathsf{s}\ n = \mathsf{s}\ m \to \mathsf{vcons}\ x\ xs = xxs \to \\
& \quad \mathbf{VTail}\ xxs
\end{array}$$

Solving the equations refutes the base case premises and reduces the step case to

$$? : \forall A|\mathsf{Type}.\ \forall n|\mathbb{N}.\ \forall x{:}A.\ \forall xs{:}\mathsf{Vect}\ A\ n.\ \cdots \to \mathbf{VTail}\ (\mathsf{vcons}\ x\ xs)$$

The return type again shows the one pattern possible, and xs is the tail we seek.

Unlike with $\leq$, our chosen Φ did quantify over the eliminated xxs—it is matched to the xs in the goal patterns. Omitted this time is A, parametric to the definition, so kept parametric in the elimination. We must be sensitive to these distinctions in order to deliver appropriate behaviour, whatever the elimination rule.

3 Equational Constraints and Dependent Types

By now, the eagle-eyed will have noticed that I write batched equations like $\vec{x} = \vec{t}[\vec{y}]$ without worrying about type safety. Indeed, in the above example, I wrote $xs = xxs$

where $xs : \mathsf{Vect}\, A\, n$ and $xxs : \mathsf{Vect}\, A\, (\mathsf{s}\, m)$. The conventional Martin-Löf definition of $=$ forbids such *heterogeneous* equations, relating elements of different types. You can thus deduce that I am using an unconventional definition.

I define $=$ as follows:

$$\frac{a : A \quad b : B}{a = b : \mathsf{Prop}} \qquad \frac{a : A}{\mathsf{refl}\, a : a = a} \qquad =\text{-}\mathbf{elim}\ \frac{\begin{array}{l} a : A \\ \Phi : \forall a' {:} A.\, a = a' \to \mathsf{Type} \\ \\ \Phi\, a\, (\mathsf{refl}\, a) \end{array}}{\forall a' {:} A.\, \forall q {:} a = a'.\, \Phi\, a'\, q}$$

This $=$ can compare anything to a, even if it is not in A. Correspondingly, we may form heterogeneous sequences $\vec{s} = \vec{t}$. However, the introduction and elimination rules follow the conventional *homogeneous* definition: we shall only treat something as an equal of a if its type really is A. I call this 'John Major' equality, because it widens aspirations to equality without affecting the practical outcome.

If $\vec{s}$ and $\vec{t}$ are vectors in the same *telescope* [3], then the leftmost equation $s_1 = t_1$ is homogeneous and thus vulnerable to elimination. Homogeneity is a maintainable invariant: solving $s_1 = t_1$ ipso facto unifies the types of s_2 and t_2, and so on.

'John Major' equality is equivalent to extending Martin-Löf equality with Altenkirch and Streicher's 'uniqueness of identity proofs' axiom, often referred to as 'axiom K'. It is clear that the new equality subsumes the old. On the other hand, we can write a heterogeneous equation $a = b$ as a homogeneous equation between pairs $(A, a) = (B, b)$ in the type $\Sigma T {:} \mathsf{Type}.\, T$. Clearly (A, a) equals itself. The elimination rule follows if $a = a'$ is a consequence of $(A, a) = (A, a')$, and this is a well-known variant of axiom K. The details of this construction can be found in my thesis.

4 What does an Elimination Rule Eliminate?

In order to mechanize elimination, we shall need some means to determine what kind of thing a given rule eliminates: what does the rule *target*? The **-elim** rules which come with inductive definitions clearly target inhabitants of the datatype, relation or family being defined. However, if we wish our tactic to apply more widely, we should perhaps think a little more carefully about this issue.

Firstly, let us establish a minimum requirement. Suppose our rule needs a motive $\Phi : \forall \vec{x} {:} \vec{X}.\, \mathsf{Type}$. A basic goal and motive are given by:

$$? : \forall \vec{y} {:} \vec{Y}.\, P[\vec{y}]$$

$$\Phi\, \vec{x} \mapsto \forall \vec{y} {:} \vec{Y}.\, \vec{x} = \vec{t}[\vec{y}] \to P[\vec{y}]$$

When we apply the rule, the equations should become reflexive. This is ensured by instantiating enough of the rule's arguments to fill in the *goal patterns*: once the instantiated rule delivers $\Phi\, \vec{t}[\vec{y}]$, our choice of equations becomes clear. Hence, we must demand enough information from the user to determine the arguments on which the goal patterns depend.

Looking back at our examples, we can see that our requirement is satisfied for $\vee$-elimination even before we select a disjunction, although it would seem foolish to apply the rule without one in mind. For $\mathbb{N}$-**induction**, we need to choose a number.

Induction on Vect needs both a vector and its length, but the length can be inferred from the type of the vector, so the user need only indicate the latter. To form the motive in a $\leq$ induction, we must identify the numbers being compared, but it makes more sense to infer these by matching with a hypothesis of form $m \leq n$.

We can permit rules with several targets: the 'double induction' principle for a datatype implements lexicographic recursion on two arguments with that type. We can also imagine rules whose application is restricted by a side-condition whose proof we would prefer to defer. It seems unlikely that a naïve machine strategy could divine from an arbitrary rule what we must point to when we say 'eliminate that'. Ingenious machine strategies disturb me, so I propose to make the 'manufacturers' responsible: we should expect elimination rules to come with 'operating instructions'.

We might describe how to use a rule with type $\forall \vec{u}\!:\!\vec{U}.T[\vec{u}]$ by giving a list of *targetting expressions* over the $\vec{u}$ for which the user is to supply the actual *targets*, perhaps with the aid of a mouse, such that the targets unify with the expressions, solving for some of the $\vec{u}$. With this selection complete, we may proceed with the elimination, provided the instantiated rule type reduces to something in *fully targetted* form:

$$\cdots \forall \Phi\!:\!\forall \vec{x}\!:\!\vec{X}.\,\mathsf{Type}.\ \cdots \Phi\,\vec{t}[\vec{y}]$$

In effect, an explicit targetting procedure allows us to delay the appearance of the motive variable Φ. In the computational world of Type Theory, we may thus choose our targets first and compute an appropriate rule afterwards. Later we shall see the 'constructors injective and disjoint' property of datatypes expressed by one rule which selects its effect by case analysis on the constructors involved.

The 'operating instructions' approach might also benefit user interfaces. Given a catalogue of known elimination rules and what they target, we can point at a hypothesis and ask 'which rules would eliminate that?'. The machine could even sort the responses to give the best matching rule first.

5 An Elimination Tactic: `BasicElim`

`BasicElim` implements the ideas above. Its first argument is a rule, typically:

$$\mathbf{elim}\ :\ \forall \vec{u}\!:\!\vec{U}.\,T[\vec{u}]$$

$$?\ :\ \forall \vec{y}\!:\!\vec{Y}.\,P[\vec{y}]$$

$$>\ \texttt{BasicElim}\ \mathbf{elim}\ \ldots;$$

The remaining arguments are some $\vec{g}[\vec{y}]$—the user's chosen *targets*. `BasicElim` has three phases:

- plug in the $\vec{g}[\vec{y}]$ to make the instantiated rule *fully targetted*
- construct the *motive*, by adding equations to the goal, then simplifying
- perform the refinement step, leaving the rest of the rule's premises as subgoals

5.1 Targetting

`BasicElim` builds a refinement by applying **elim**. The first phase constructs and maintains a full application of **elim** to terms $\vec{s}[\vec{u}, \vec{y}]$ over the goal's premises $\vec{y}$, but containing holes[4] $\vec{u}$. At the same time, it keeps the list of the user's nominated targets $\vec{g}[\vec{y}]$ so far unmatched. I write such a *state* as follows:

$$\mathbf{elim}\,\vec{s}[\vec{u},\vec{y}]\ :\ E[\vec{u},\vec{y}] \quad \text{unknowns}\ ?\vec{u}\ :\ \vec{U} \quad \text{targets}\ \vec{g}[\vec{y}]$$

By 'full application', I mean that $E[\vec{u},\vec{y}]$ is in weak head normal form and is not a $\forall$-type. I presume some way to annotate $E[\vec{u},\vec{y}]$ with a *targetting expression* $e[\vec{u}]$: I shall denote this $\langle e[\vec{u}]\rangle E[\vec{u},\vec{y}]$. This phase successively fills in the $\vec{u}$ by unifying these $e[\vec{u}]$'s with the user's $\vec{g}[\vec{y}]$.

- **initialization**: start in state
$$\mathbf{elim}\,\vec{u}\ :\ T[\vec{u}] \quad \text{unknowns}\ ?\vec{u}\ :\ \vec{U} \quad \text{targets}\ \vec{g}[\vec{y}]$$
- **loop**: while the state has form...
$$\mathbf{elim}\,\vec{s}[\vec{u},\vec{y}]\ :\ \langle e[\vec{u}]\rangle E[\vec{u},\vec{y}] \quad \text{unknowns}\ ?\vec{u}\ :\ \vec{U} \quad \text{targets}\ g[\vec{y}];\vec{g}[\vec{y}]$$
  The type is marked with a targetting expression $e[\vec{u}]$, so collect the next unused target $g[\vec{y}]$ and try to unify them. If this fails, then `BasicElim` fails. Otherwise, we have a unifier σ solving some of $\vec{u}$, and leaving a residue $\vec{u}'\ :\ \vec{U}'$ unsolved. The instantiated type may reduce further to weak head normal form, perhaps revealing more unknown arguments $\vec{v}$ for the rule or more targetting expressions:
$$E[\sigma\vec{u},\vec{y}] \leadsto_{wh} \forall\vec{v}\!:\!\vec{V}.\, E'[\vec{u}';\vec{v},\vec{y}]$$
  Hence move to state
$$\mathbf{elim}\,\vec{s}[\sigma\vec{u},\vec{y}]\,\vec{v}\ :\ E'[\vec{u}';\vec{v},\vec{y}] \quad \text{unknowns}\ ?\vec{u}';\vec{v}\ :\ \vec{U}';\vec{V} \quad \text{targets}\ \vec{g}[\vec{y}]$$
- **postcondition**: the state must have this form
$$\mathbf{elim}\,\vec{s}[\vec{u},\vec{y}]\ :\ u_i\,\vec{t}[\vec{y}] \quad \text{unknowns}\ ?\vec{u}\ :\ \vec{U} \quad \text{targets}\ \textit{none}$$
  where $u_i\ :\ \forall\vec{x}\!:\!\vec{X}[\vec{y}].\ \mathsf{Type}$

This phase, if successful, consumes all targets, matches the targetting expressions and constructs an application of **elim** with some holes, one of which, u_i, heads the return type. Rename it Φ: it is the *motive variable*. The postcondition ensures that Φ's argument types $\vec{X}$ and arguments $\vec{t}[\vec{y}]$ contain no holes—the instantiated rule is now *fully targetted*.[5]

5.2 Constructing the Motive

Our basic choice of motive copies the goal, inserting some equations. This phase refines this choice to avoid useless constraints, maximize the amount of rewriting done by instantiation in subgoals and remove premises which are being eliminated but are not mentioned in the goal patterns, as found with rules like $\vee$**-elim** and $\leq$**-induction**. We start by guessing

$$\Phi \mapsto \lambda\vec{x}\!:\!\vec{X}.\,\forall\vec{y}'\!:\!\vec{Y}.\,\vec{x} = \vec{t}[\vec{y}'] \to P[\vec{y}']$$

The tactic then performs the following refinements:

[4] OLEG, a type theory with holes (also known as metavariables and a host of other names) adequate to support `BasicElim`, is a useful byproduct of my thesis [8].

[5] I have written Type for the universe which $\Phi\,\vec{x}$ inhabits, but any universe is acceptable.

– **fix 'unhelpful' premises**

Our basic motive contains local copies $\vec{y}'$ of all the premises $\vec{y}$ in the goal. However, it is sometimes better to forgo this generality and use the originals, effectively *fixing* them for the elimination: because they are not local to Φ, they will not be local to any inductive hypotheses and thus cannot change with recursion. We fix a premise y'_i as follows:

substitute y_i for bound occurrences of y'_i; remove the binder $\forall y'_i$

There are three classes of 'unhelpful' premise we should fix:

parametric premises, such as the element type for Vect

A *parametric* y'_i is the local copy of a y_i found in the type of Φ: it is thus parametric to the subgoal structure and should remain constant.

large premises

A *large* premise has a type too big for the universe which $\Phi\,\vec{x}$ inhabits: we must fix such premises to keep Φ well-typed.

irrelevant premises, such as the proof of $m \le n$ in $\le -$**elim**

An irrelevant premise is the local copy of a y_i occurring in the arguments of **elim** computed by targetting, but not in the instantiated *goal patterns*. It is being eliminated, but the subgoals will tell us nothing new about it, so we may as well fix it.

We could, of course, fix more premises, but the remaining $\vec{y}'$ are 'helpful' in that they yield stronger inductive hypotheses.

– **delete duplicating constraints**

for increasing j, if $y'_i : X_j$ and the x_j constraint is $x_j = y'_i$, then remove it, substitute x_j for y'_i and delete $\forall y'_i$

There is no point having a $\forall y'_i$ in Φ if y'_i must then equal one of Φ's arguments x_j. Provided the two have the same type, we can just use x_j and remove the constraint. This ensures that we only get equations which are really necessary (as in our example with $m \le 0$). We should search from left to right, as earlier deletions may unify later types.

`BasicElim` now plugs in the motive, with its 'helpful' $\vec{y}'$ and necessary equations:

$$\Phi \;\mapsto\; \lambda\vec{x}\!:\!\vec{X}.\,\forall\vec{y}'\!:\!\vec{Y}'.\,\widehat{\vec{x} = \vec{t}[\vec{y}']} \;\to\; P[\vec{y}',\vec{y}]$$

5.3 Performing the Refinement

Having computed the motive, our application is now typed as follows:

$$\textbf{elim}\;\ldots\;:\;\forall\vec{y}'\!:\!\vec{Y}'.\,\widehat{\vec{t}[\vec{y}] = \vec{t}[\vec{y}']} \;\to\; P[\vec{y}',\vec{y}] \qquad \text{unknowns}\;\;?\,\vec{w}\;:\;\vec{W}[\vec{y}]$$

`BasicElim` now adds arguments:

- for each local y'_i, the goal premise it copies y_i
- for each constraint, now $t_j[\vec{y}] = t_j[\vec{y}]$, its proof by refl

$$\textbf{elim}\;\ldots\,y_i\ldots\,(\mathsf{refl}\;t_j[\vec{y}])\ldots\;:\;P[\vec{y}] \qquad \text{unknowns}\;\;?\,\vec{w}\;:\;\vec{W}[\vec{y}]$$

The type W_j of each hole may depend on some subset $\vec{y}_j$ of the goal premises $\vec{y}$, usually *parametric*. To build a refinement, we must abstract each w_j over its $\vec{y}_j$:

$$w_j \;\mapsto\; w'_j\,\vec{y}_j \qquad \text{unknowns}\;\;?\,w'_j\;:\;\forall\vec{y}_j\!:\!\vec{Y}_j.\,\vec{W}[\vec{y}_j]$$

These generalized holes $\vec{w}' : \vec{W}'$ now have local copies of all the premises they need, so they have the same context as the original goal. We can thus λ-abstract them:

$$\lambda \vec{w}' : \vec{W}'.\, \lambda \vec{y} : \vec{Y}.\, \mathbf{elim}\ \ldots\ :\ \forall \vec{w}' : \vec{W}'.\, \forall \vec{y} : \vec{Y}.\, P[\vec{y}]$$

`BasicElim` refines by this term, solving the goal $\forall \vec{y} : \vec{Y}.\, P[\vec{y}]$, with $\vec{W}'$ as subgoals.

6 Eliminating Equational Constraints with `Unify`

`BasicElim` applies a rule to an instantiated hypothesis by converting those instantiations into equations. Hence we expect equational premises in the subgoals, instantiated with the *subgoal patterns*. The approach of [7] was to treat these equations as a *unification* problem [11], leading to a tactic which solves such problems. The absence of a unifier indicates that a subgoal holds vacuously; a unique most general unifier simplifies a subgoal, turning equations back into instantiations. That tactic from [7] is the ancestor of the tactic `Unify` presented here. Now, as then, we may observe that, for any given datatype, these rule schemes are derivable:

deletion $\dfrac{\Phi}{x = x \ \to\ \Phi}$

coalescence $\dfrac{\Phi\, x}{\forall y.\, x = y \ \to\ \Phi\, y}$ x, y distinct variables

conflict $\dfrac{}{c\, \vec{s} = c'\, \vec{t} \ \to\ \Phi}$ c, c' distinct constructors

injectivity $\dfrac{\vec{s} = \vec{t} \ \to\ \Phi}{c\, \vec{s} = c\, \vec{t} \ \to\ \Phi}$ c a constructor

substitution $\dfrac{\Phi\, t}{\forall x.\, x = t \ \to\ \Phi\, x}$ $x \notin FV(t)$

cycle $\dfrac{}{x = t \ \to\ \Phi}$ x constructor-guarded in t

Just as in [7], these rules are seen as the transition rules for a unification algorithm operating by refinement on problems expressed as equational premises in an arbitrary goal. However, there are some key differences with the earlier work:

- 'John Major' equality now allows us to consider equations over an arbitrary *telescope*, overcoming the previous restriction to simple types.
- Consequently, the **conflict**, **injectivity** and **cycle** rules require more subtle proofs than in the simply typed fragment.
- The transition rules can be seen as *elimination rules* targetting an equation. Apart from **cycle**, the `BasicElim` tactic can apply them.

`Unify` demands that the goal's equational premises $\vec{s} = \vec{t}$ relate vectors from the same telescope of first-order terms in *constructor form*: i.e., composed solely of variables and constructor symbols. Given such a goal, `Unify` behaves as follows:

While the goal remains with equational premises, eliminate the leftmost by the appropriate rule (applying symmetry where necessary).

The precondition ensures that the leftmost equation is homogeneoue, and is preserved by the transitions with subgoals. The process is sound, complete and terminating by the same arguments as before: `Unify` either proves the goal if there is no unfier, or leaves a subgoal simplified by a most general unifier.

Of course, in order to use these rules, we must first prove them: **deletion** is trivial; **coalescence** and **substitution** follow easily from **=-elim**. The other three must be proven specifically for each datatype.

7 Derived Elimination Rules for Datatypes

This section sketches the construction of some useful classes of theorem which can be proven for each inductive family of datatypes. Included are

- the separation of induction into **case** analysis and structural **recursion**
- the proof that constructors are injective and disjoint, a property often dubbed '**no confusion**'
- the proof that datatypes contain **no cycles**

These theorems are all given as elimination rules. For the sake of readability, I shall give the proofs for concrete but typical examples—vectors for **case**, **recursion** and **no confusion**, binary trees for **no cycles**. The latter are defined:

$$\frac{}{\mathsf{Tree}\ :\ \mathsf{Type}} \qquad \frac{}{\mathsf{leaf}\ :\ \mathsf{Tree}} \qquad \frac{s, t\ :\ \mathsf{Tree}}{\mathsf{node}\, s\, t\ :\ \mathsf{Tree}}$$

The general constructions can be found in my thesis [8]. These results extend easily to mutual definitions: in any case, a mutual definition can always be recast as an inductive family indexed by a finite datatype representing 'choice of branch'.

I begin by decoupling the **elim** rule for a datatype into its **case** and **recursion** principles, recovering the flexibility of Coq's `Case` and `Fix` primitives [1] in a way which is readily extensible to instances of dependent families. This presentation makes only the necessary connection between case analysis and structural recursion: the former exposes the 'predecessors' for which the latter is valid. The **recursion** rule makes no choice of case analysis strategy, whereas **elim** on an given x analyses x straight away and forces one-step recursion.

We gain more than just 'Fibonacci and friends': **recursion** and **case** on the *indices* of a dependent type often work differently from their counterparts on the type itself, and now we can combine them as we wish. For example, the unification algorithm presented in [9] indexes terms with the number of variables they may use—the outer recursion is on this index, but the initial case analysis is on terms.

7.1 case

The case analysis principle for a datatype is formed by deleting the inductive hypotheses from the step cases of the induction principle, **elim**.

$$\begin{array}{l} A \;:\; \mathsf{Type} \\ \Phi \;:\; \forall n \,|\, \mathbb{N}.\ \mathsf{Vect}\, A\, n \;\to\; \mathsf{Type} \end{array}$$

$$\mathsf{Vect}\text{-}\mathbf{case}\ \frac{\Phi_0\ \mathsf{vnil} \qquad \dfrac{x : A \quad xs : \mathsf{Vect}\, A\, n \quad \cancel{\Phi_n\ xs}}{\Phi_{\mathsf{s}\,n}\ (\mathsf{vcons}\ x\ xs)}}{\forall n \,|\, \mathbb{N}.\ \forall xs : \mathsf{Vect}\, A\, n.\ \Phi_n\ xs}$$

Effectively, **case** splits a 'pattern variable' from into constructor cases, exposing the 'predecessors'. Of course, `Unify` can then simplify, removing some impossible cases.

Having stronger premises, **case** follows directly from **elim**. We may also prove **case** for a relation, where it is often called the *inversion* principle. The treatment of inversion in [7] relies clumsily on equations to constrain the indices of relations: **case** gives a neater rule with an indexed motive, and any equations required for a particular inversion are supplied by `BasicElim`.

7.2 recursion

Let us now facilitate recursion on *guarded* subterms, after the fashion of Giménez [4, 5]. The technique is to introduce an auxiliary structure which collects inductive hypotheses. For any motive Φ and datatype inhabitant t, $\Phi \prec t$ contains a proof of Φs for each s strictly smaller than t. Giménez defines $\prec$ inductively, but computation is enough. For Vect:

$$\frac{\Phi \;:\; \forall n \,|\, \mathbb{N}.\ \mathsf{Vect}\, A\, n \;\to\; \mathsf{Type} \qquad xs \;:\; \mathsf{Vect}\, A\, n}{\Phi \prec xs \;:\; \mathsf{Type}}$$

$$\begin{array}{rcl} \Phi \prec \mathsf{vnil} & \mapsto & 1 \\ \Phi \prec \mathsf{vcons}\, x\, xs & \mapsto & \Phi \preceq xs \\ \text{where } \Phi \preceq xs & \mapsto & \Phi \prec xs \;\times\; \Phi\ xs \end{array}$$

$\Phi \prec xs$ is primitive recursive, thus easily defined via Vect-**elim**. Generally, we have $\Phi \prec c\,\vec{s} \;\mapsto\; \times_i\, \Phi \preceq s_i$, e.g. for Tree:

$$\begin{array}{rcl} \Phi \prec \mathsf{leaf} & \mapsto & 1 \\ \Phi \prec \mathsf{node}\, s\, t & \mapsto & \Phi \preceq s \;\times\; \Phi \preceq t \\ \text{where } \Phi \preceq t & \mapsto & \Phi \prec t \;\times\; \Phi\, t \end{array}$$

We can now state Vect-**recursion**:

$$\begin{array}{l} A \;:\; \mathsf{Type} \\ \Phi \;:\; \forall n \,|\, \mathbb{N}.\ \mathsf{Vect}\, A\, n \;\to\; \mathsf{Type} \end{array}$$

$$\mathsf{Vect}\text{-}\mathbf{recursion}\ \frac{\forall n \,|\, \mathbb{N}.\ \forall xs : \mathsf{Vect}\, A\, n.\ \Phi \prec xs \;\to\; \Phi\ xs}{\forall n \,|\, \mathbb{N}.\ \forall xs : \mathsf{Vect}\, A\, n.\ \Phi\ xs}$$

Vect-**recursion** weakens a goal with a 'hypothesis collector', $\Phi \prec xs$. When we then apply **case** to xs or its subterms, $\Phi \prec$ unfolds, revealing the inductive hypothesis for the newly exposed subterm. The proof uses Giménez's argument, fixing A, Φ and the premise (**step**, say). The conclusion holds by projection from the lemma

$$\forall n \,|\, \mathbb{N}.\ \forall xs\!:\!\mathsf{Vect}\ A\ n.\ \Phi \preceq xs$$

proven with Vect-**elim**. Each subgoal conclusion computes to some $\Phi \prec c\,\vec{s} \times \Phi(c\,\vec{s})$: **step** gives $\Phi\,(c\,\vec{s})$ from $\Phi \prec c\,\vec{s}$, which we may then unfold further: in the vnil case, to 1; for the step, $\Phi \prec \mathsf{vcons}\ x\ xs \leadsto \Phi \preceq xs$, exactly the inductive hypothesis. (Generally, we have hypotheses $\Phi \preceq s_i$ and a goal which is their product.)

To see decoupling at work, let us compute the last element of a nonempty vector:

$$?:\ \forall A \,|\, \mathsf{Type}.\ \forall m \,|\, \mathbb{N}. \forall xs : \mathsf{Vect}\ A\ (\mathsf{s}\ m).\ A$$

Eliminating with Vect-**recursion** introduces a collector $\Phi \prec xs$ for

$$\Phi_n\ xs\ \mapsto\ \forall m \,|\, \mathbb{N}.\ \forall ys\!:\!\mathsf{Vect}\ A\ (\mathsf{s}m).\ n = \mathsf{s}\ m\ \rightarrow\ xs = ys\ \rightarrow\ A$$

The equational constraints thus appearing in $\Phi \prec xs$ as it unfolds allow recursion only on nonempty vectors. Length is clearly crucial—we may analyse it with $\mathbb{N}$-**case**:

$$?:\ \forall A \,|\, \mathsf{Type}.\ \forall xs\!:\!\mathsf{Vect}\ A\ (\mathsf{s}\ 0).\Phi \prec xs\ \rightarrow\ A$$
$$?:\ \forall A \,|\, \mathsf{Type}.\ \forall m \,|\, \mathbb{N}. \forall xs : \mathsf{Vect}\ A\ (\mathsf{s}\ (\mathsf{s}\ m)).\Phi \prec xs\ \rightarrow\ A$$

For the 'singleton' subgoal, **case** on xs delivers the head element we need; for longer vectors, **case** exposes a tail for which the equations constraining recursion are satisfied. We avoid looking two steps down the vector because we know its length.

7.3 no confusion

Generalizing **injectivity** and **conflict** to dependent datatypes requires more subtlety than the methods used for simple types by Cornes and Terrasse [2] in Coq, which I adapted for Lego [7]. In particular, we can no longer construct 'predecessor' functions to show **injectivity**: there are no obvious candidates for the dummy values in the unimportant cases.

My approach here is to compute, for any pair of terms, the elimination rule which is appropriate when the two are equal: **injectivity** for like constructors and **conflict** for unlike. When both terms are constructor-headed, this '**no-conf-thm**' function will choose the right theorem:

$$\mathsf{Vect}\text{-}\mathbf{no\text{-}conf\text{-}thm}\ :\ \forall A \,|\, \mathsf{Type}.\ \forall m \,|\, \mathbb{N}.\ \forall xs\!:\!\mathsf{Vect}\ A\ m.\ \forall n \,|\, \mathbb{N}.\ \forall ys\!:\!\mathsf{Vect}\ A\ m.\ \mathsf{Type}$$

$$\begin{array}{llll}
\mathsf{Vect}\text{-}\mathbf{no\text{-}conf\text{-}thm} & \mathsf{vnil} & \mathsf{vnil} & \mapsto \forall \Phi\!:\!\mathsf{Type}.\ \Phi\ \rightarrow\ \Phi \\
\mathsf{Vect}\text{-}\mathbf{no\text{-}conf\text{-}thm} & \mathsf{vnil} & (\mathsf{vcons}\ y\ ys) & \mapsto \forall \Phi\!:\!\mathsf{Type}.\ \Phi \\
\mathsf{Vect}\text{-}\mathbf{no\text{-}conf\text{-}thm} & (\mathsf{vcons}\ x\ xs) & \mathsf{vnil} & \mapsto \forall \Phi\!:\!\mathsf{Type}.\ \Phi \\
\mathsf{Vect}\text{-}\mathbf{no\text{-}conf\text{-}thm} & (\mathsf{vcons}_m\ x\ xs) & (\mathsf{vcons}_n\ y\ ys) & \mapsto \\
\multicolumn{4}{l}{\quad \forall \Phi\!:\!\mathsf{Type}.\ (m = n\ \rightarrow\ x = y\ \rightarrow\ xs = ys\ \rightarrow\ \Phi)\ \rightarrow\ \Phi}
\end{array}$$

We can clearly construct Vect-**no-conf-thm** with two applications of Vect-**case**. Now we can prove Vect-**no-confusion**, which states:

$$?:\ \forall A \,|\, \mathsf{Type}.\ \forall n \,|\, \mathbb{N}.\ \forall xs, ys : \mathsf{Vect}\ A\ n.\ xs = ys\ \rightarrow\ \mathsf{Vect}\text{-}\mathbf{no\text{-}conf\text{-}thm}\ xs\ ys$$

We need only consider vectors of the same type, as `Unify` only eliminates homogeneous equations. We may thus attack $xs = ys$ with **=-elim**, leaving the 'diagonal':

$$?:\ \forall A\,|\,\mathsf{Type}.\ \forall n\,|\,\mathbb{N}.\ \forall xs\,{:}\,\mathsf{Vect}\ A\ n.\mathsf{Vect}\text{-}\mathbf{no\text{-}conf\text{-}thm}\ xs\ xs$$

Now Vect-**case** on xs will leave us with trivial **injectivity** goals. This is fortunate, as the **conflict** theorems chosen by Vect-**no-conf-thm** are too good to be true. 'Dummy values' do not arise—each subgoal is specifically adapted to its constructor:

$$\begin{array}{l} ?:\ \forall A\,{:}\,\mathsf{Type}.\ \forall \Phi\,{:}\,\mathsf{Type}.\ \Phi\ \to\ \Phi \\ ?:\ \forall A\,{:}\,\mathsf{Type}.\ \forall n\,{:}\,\mathbb{N}.\ \forall x\,{:}\,A.\ \forall xs\,{:}\,\mathsf{Vect}\ A\ n. \\ \quad\ \forall \Phi\,{:}\,\mathsf{Type}.\ (n = n\ \to\ x = x\ \to\ xs = xs\ \to\ \Phi)\ \to\ \Phi \end{array}$$

We can make Vect-**no-confusion** target an equation over $\mathsf{Vect}\ A\ n$ and apply it with `BasicElim`, but only because `BasicElim` identifies the motive variable after targetting has enabled the computation which makes it appear.

7.4 no cycles

The remaining theorem we need states that any goal follows from $x = t$ when x is guarded by constructors in t. The computed 'collector' $\Phi \prec t$ exposes Φ for the guarded subterms of t. We may thus express 'x is not a proper subterm of t' by

$$\begin{array}{l} s \neq t\ \mapsto\ s = t\ \to\ \forall \Phi\,{:}\,\mathsf{Type}.\ \Phi \\ x \not\prec t\ \mapsto\ (x\ \neq) \prec t \\ (\text{with } x \not\preceq t\ \mapsto\ (x\ \neq) \preceq t) \end{array}$$

If x appears guarded in t, then $x \not\prec t$ reduces to a product containing $x \neq x$, from which anything follows. Correspondingly, Tree-**no-cycles** states that

$$?:\ \forall x, t\,{:}\,\mathsf{Tree}.\ x = t\ \to\ x \not\prec t$$

`Unify` turns t into x. Induction on x gives a trivial base case and an unfolding step:

$$\begin{array}{lcl} s, t & : & \mathsf{Tree} \\ Hs & : & s \not\prec s \\ Ht & : & t \not\prec t \end{array} \qquad ?:\ \begin{array}{l} \quad(\mathsf{node}\ s\ t) \not\preceq s \\ \times\,(\mathsf{node}\ s\ t) \not\preceq t \end{array}$$

Each branch of this product follows from the corresponding hypothesis, but only with the aid of a cunning generalization: let us prove

$$?:\ \forall x, s, t\,{:}\,\mathsf{Tree}.\ s \not\prec x\ \to\ \mathsf{node}\ s\ t \not\preceq z$$

and its analogue for t. The computational behaviour of $(s\ \neq) \prec x$ suggests that we employ induction on x. The base case,

$$?:\ \forall s, t\,{:}\,\mathsf{Tree}.\ 1\ \to\ (1\ \times\ \mathsf{node}\ s\ t \neq \mathsf{leaf})$$

follows by **conflict**. The step case unfolds as follows—the arrows give the proof:

$$\begin{array}{lcl} l, r & : & \mathsf{Tree} \\ Hl & : & \boxed{s \not\prec l} \to \boxed{\mathsf{node}\ s\ t \not\preceq l} \\ Hr & : & \boxed{s \not\prec r} \to \boxed{\mathsf{node}\ s\ t \not\preceq r} \end{array} \quad \begin{array}{l} H\ :\ (\boxed{s \not\prec l} \times \boxed{s \neq l}) \\ \quad\ \ \times\,(\boxed{s \not\prec r} \times \boxed{s \neq r}) \\ \qquad\qquad \xrightarrow{\textbf{injectivity}} \end{array} \quad \begin{array}{l} ?:\ (\boxed{\mathsf{node}\ s\ t \not\preceq l} \\ \quad\ \times\,\boxed{\mathsf{node}\ s\ t \not\preceq r}) \\ \quad\ \times\,\boxed{\begin{array}{l}\mathsf{node}\ s\ t \\ \neq \mathsf{node}\ l\ r\end{array}} \end{array}$$

`BasicElim` cannot be used to apply Tree-**no-cycles**: for a given t containing x, $x \not\prec t$ reduces not to a single fully targetted elimination rule in the style of **-no-confusion**, but to a product of elimination rules. `BasicElim`, as specified above, is not smart enough to root out the proof of $x = x \;\rightarrow\; \forall \Phi\!:\!\mathsf{Type}.\,\Phi$. However, the path to this proof is determined exactly by the position of x in t, so it is not difficult to implement this step of `Unify` separately.

8 Elimination with Abstraction

The traditional way to reason about a recursively defined function is to use the inductions on its arguments which allow it to reduce: this amounts to simulatimg of the recursive structure by which it was constructed in the first place. Functional abstraction allows us to synthesize programs in a highly compositional manner, but if we must always analyse programs at the level of data, the scalability of our technology will be seriously limited.

An alternative is to work at the level of the *relations* induced by recursive definitions. For example, $+$ induces a three-place relation of form $x + y = z$:

$$\frac{}{\mathsf{0} + y = y} \qquad \frac{z + y = z}{(\mathsf{s}\,x) + y = (\mathsf{s}\,z)}$$

$$\Phi \;:\; \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathsf{Type}$$

$$+\text{-}\mathbf{elim}\;\frac{\Phi\,\mathsf{0}\,y\,y \qquad \begin{array}{c} x + y = z \quad \Phi\,x\,y\,z \\ \cdots\cdots\cdots\cdots \\ \Phi\,(\mathsf{s}\,x)\,y\,(\mathsf{s}\,z) \end{array}}{\forall x, y, z\!:\!\mathbb{N}.\;\boxed{x + y} = z \;\rightarrow\; \Phi\,x\,y\,z}$$

The introduction rules abstract the recursive calls as premises: they follow by substituting these premises. The elimination rule is exactly the corresponding relation induction principle: it follows from the same combination of **recursion** and **case** analyses by which $+$ was constructed. I shall explain the purpose of the box shortly.

$+$-**elim** eliminates *equations* of the form $x + y = z$. Given such an equation, we can use `BasicElim`, but what if there is no such equation? Consider

$$?\;:\;\forall a, b, c\!:\!\mathbb{N}.\;(a + b) + c = a + (b + c)$$

For any of these $+$'s, we can transform the goal to introduce an equation, then eliminate, reduce and unify, leaving easy subgoals:

$$\begin{array}{rll}
\text{transformed} & ?:\forall a, b, c, z\!:\!\mathbb{N}. & a + b = z \;\rightarrow\; z + c = a + (b + c) \\[1ex]
\text{base case} & ?:\quad \forall y, c\!:\!\mathbb{N}. & y + c = y + c \\
\text{step case} & ?:\forall x, y, c\!:\!\mathbb{N}. & (x + y) + c = \quad x + (y + c) \;\rightarrow \\
 & & \mathsf{s}\,(x + y) + c = \mathsf{s}\,(x + (y + c))
\end{array}$$

This is essentially the standard proof that $+$ is associative, but, it avoids the choice of 'a good induction on data': the derived rule gives by design exactly the computation we need. Similar techniques in proving properties of inductively defined functions appear in James McKinna's thesis [10].

Let us build some technology to facilitate this way of working with functions. We could write a tactic, `Abst` $e[\vec{y}]$, abstracting the occurrences of $e[\vec{y}]$ from the goal:

$$\text{before} \quad ? : \forall \vec{y}\!:\!\vec{Y}.\, P[\vec{y}, e[\vec{y}]]$$

$$\text{after} \quad ? : \forall \vec{y}\!:\!\vec{Y}.\, \forall x.\, e[\vec{y}] = x \;\rightarrow\; P[\vec{y}, x]$$

Of course, $P[\vec{y}, x]$ may not be well-typed[6], but when it is, `Abst` then `BasicElim` on the equation does what we want. The box around $x + y$ in **+-elim** indicates that the rule *targets* expressions of form $x+y$, but *eliminates* equations $x+y = z$, hence an abstraction is to precede elimination. Let us call this extended tactic `AbstElim`.

A homogeneous equational law, $\forall \vec{x}\!:\!\vec{X}.\, s[\vec{x}] = t[\vec{x}]$ gives a derived rule via **=-elim**:

$$\begin{array}{l} \vec{x} \;:\; \vec{X} \\ \Phi \;:\; T[\vec{x}] \;\rightarrow\; \mathsf{Type} \end{array}$$

$$\frac{\Phi\; t[\vec{x}]}{\forall z\!:\!T[\vec{x}].\, \boxed{s[\vec{x}]} = z \;\rightarrow\; \Phi\; z}$$

`AbstElim` with such a rule rewrites by the law: targetting allows us to select which term to rewrite, provided unification can infer the $\vec{x}$. Commonplace one-step rewriting can be implemented by a simple wrapper for `AbstElim`.

The recursive structure of a function is not always its key characteristic. However, there is nothing to stop us deriving more useful properties. Consider, for example, the equality test for natural numbers:

$$\mathbb{N}\textbf{-eq} \;:\; \mathbb{N} \;\rightarrow\; \mathbb{N} \;\rightarrow\; \mathsf{Bool}$$

$$\begin{array}{llll} \mathbb{N}\textbf{-eq} & 0 & 0 & \mapsto \mathsf{true} \\ \mathbb{N}\textbf{-eq} & 0 & (\mathsf{s}\, y) & \mapsto \mathsf{false} \\ \mathbb{N}\textbf{-eq} & (\mathsf{s}\, x) & 0 & \mapsto \mathsf{false} \\ \mathbb{N}\textbf{-eq} & (\mathsf{s}\, x) & (\mathsf{s}\, y) & \mapsto \mathbb{N}\textbf{-eq}\; x\; y \end{array}$$

The obvious induction principle is like this (you can guess the cases for false):

$$\Phi \;:\; \mathbb{N} \;\rightarrow\; \mathbb{N} \;\rightarrow\; \mathsf{Bool} \;\rightarrow\; \mathsf{Type}$$

$$\mathbb{N}\textbf{-eq-elim}\; \frac{\Phi\; 0\; 0\; \mathsf{true} \qquad \cdots \qquad \begin{array}{c} \mathbb{N}\textbf{-eq}\; x\; y = b \quad \Phi\; x\; y\; b \\ \vdots \\ \Phi\; (\mathsf{s}\, x)\; (\mathsf{s}\, y)\; b \end{array}}{\forall x, y\!:\!\mathbb{N}.\, \forall b\!:\!\mathsf{Bool}.\, \boxed{\mathbb{N}\textbf{-eq}\; x\; y} = b \;\rightarrow\; \Phi\; x\; y\; b}$$

Suppose, however, we are trying to prove a goal like

$$? :\; \forall x, y\!:\!\mathbb{N}.\; P[\text{if } (\mathbb{N}\textbf{-eq}\; x\; y) \text{ then } t[x, y] \text{ else } e[x, y]]$$

$\mathbb{N}$**-eq-elim** $x\, y$ is not very helpful: it analyses the inputs to the test, but in the step case, we learn nothing about the output. We might prefer an *inversion* principle:

$$\mathbb{N}\textbf{-eq-inv}\; \frac{\Phi\; x\; x\; \mathsf{true} \qquad \begin{array}{c} x \neq y \\ \vdots \\ \Phi\; x\; y\; \mathsf{false} \end{array}}{\forall x, y\!:\!\mathbb{N}.\, \forall b\!:\!\mathsf{Bool}.\, \boxed{\mathbb{N}\textbf{-eq}\; x\; y} = b \;\rightarrow\; \Phi\; x\; y\; b}$$

[6] This problem already arises with rewriting tactics, and it deserves closer attention.

Applying this rule to our goal with `AbstElim` always instantiates the result of the test with a constructor, allowing the 'if' to reduce, yielding

$$\begin{array}{l} ? : \forall x\!:\!\mathbb{N}.\ P[t[x,x]] \\ ? : \forall x,y\!:\!\mathbb{N}.\ x \neq y \ \to\ P[e[x,y]] \end{array}$$

It is easy to prove $\mathbb{N}$**-eq-inv** from $\mathbb{N}$**-eq-elim**: the details are in my thesis, but the key technique is demonstrated in the next section. `AbstElim` makes the inversion principle a much more useful characterization than a theorem like

$$\forall x,y\!:\!\mathbb{N}.\ (\mathbb{N}\textbf{-eq}\ x\ y) = \mathsf{true} \Leftrightarrow x = y$$

9 Derived Structure for Datatypes

As long ago as 1987 [12], Phil Wadler proposed a mechanism to allow a type (not necessarily inductive) an alternative constructor presentation or *view*, given mappings between old and new. This permits pattern matching programs over the view, regardless of the underlying representation, overcoming a key drawback of abstract datatypes. We can achieve a similar effect by deriving elimination rules.

I suggested earlier that an elimination rule for a datatype D, with a motive indexed over D, induces a notion of pattern matching for D. Consequently, a derived elimination rule induces a derived notion of pattern matching. For example, given the function **vsnoc** which attaches an element to the end of a vector, we can prove

$$\begin{array}{l} A\ :\ \mathsf{Type} \\ \Phi\ :\ \forall n\,|\,\mathbb{N}.\ \mathsf{Vect}\ A\ n\ \to\ \mathsf{Type} \end{array}$$

$$\mathsf{Vect}\textbf{-snoc-elim}\ \frac{\Phi\ \mathsf{vnil} \qquad \begin{array}{c}\Phi\ xs \quad x : A \\ \vdots \\ \Phi\ (\textbf{vsnoc}\ xs\ x)\end{array}}{\forall n\,|\,\mathbb{N}.\ \forall xs\!:\!\mathsf{Vect}\ A\ n.\ \Phi\ xs}$$

This gives an alternative to 'destructor functions' such as the 'last element' operation described earlier. Case analysis with respect to **vsnoc** has at least two advantages over the destructor:

- The pattern (**vsnoc** $xs\ x$) clearly shows the decomposition into 'last' and 'all but last'.
- The derived notion of 'bigger' given by **vsnoc** yields a derived notion of 'structurally smaller', legitimizing recursive calls.

Of course, comparing the lengths in their types, xs is clearly smaller than ($\textbf{vsnoc}xsx$), but there are plenty of derived notions of 'smaller' which are not so obvious. For example, y is clearly smaller than $\mathsf{s}\,(x+y)$. We can derive the corresponding recursion principle for $\mathbb{N}$:

$$\Phi\ :\ \mathbb{N}\ \to\ \mathsf{Type}$$

$$\mathbb{N}\textbf{-plus-rec}\ \frac{\forall n\!:\!\mathbb{N}.\ (\forall x,y\!:\!\mathbb{N}.\ n = \mathsf{s}\,(x+y)\ \to\ \Phi\ y)\ \to\ \Phi\ n}{\forall n\!:\!\mathbb{N}.\ \Phi\ n}$$

Of course, this says the same thing as *well-founded* induction for $<$, but what is important is the way that it says it, namely 'if n is *matched* to $(\mathsf{s}\,(x+y))$, then a recursive call on y is legitimate'. $\mathbb{N}$**-plus-rec** does not explain how to find such a match; it just installs a hypothesis collector in the style of **recursion** rules—the proof is similar.

To make patterns with $+$, we need a derived notion of case analysis. Many are possible, but this one which compares two numbers, showing their difference via $+$:

$$\Phi \;:\; \mathbb{N} \to \mathbb{N} \to \mathsf{Type}$$

$$\mathbb{N}\textbf{-compare}\ \frac{\Phi\,(y+(\mathsf{s}\,x))\,y \qquad \Phi\,x\,x \qquad \Phi\,x\,(x+(\mathsf{s}\,y))}{\forall x,y\!:\!\mathbb{N}.\ \Phi\,x\,y}$$

This rule splits the (x,y)-plane into three regions: below, on, and above the diagonal $x=y$. Its proof illustrates a key technique (also used to prove $\mathbb{N}$**-eq-inv**): induction on x and **case** on y, but allowing Φ to vary inside the induction motive. The base cases (positive x-axis, origin, positive y-axis) fit neatly into the three regions covered by the rule's premises. It is the step case which is subtle: we do not yet know the region in which the point $(\mathsf{s}\,x,\mathsf{s}\,y)$ lies. However, the inductive hypothesis, Hxy, is a fully targetted elimination rule which locates (x,y)! `BasicElim` with Hxy reduces the goal to instances of the corresponding premises.

$$\begin{array}{ll}
x,y & :\mathbb{N} \\
Hxy & :\forall\Phi'\!:\!\mathbb{N}\to\mathbb{N}\to\mathsf{Type}. \\
& \quad(\forall x,y\!:\!\mathbb{N}.\ \Phi'\,(y+(\mathsf{s}\,x))\,y) \to \\
& \quad(\forall x\!:\!\mathbb{N}.\ \Phi'\,x\,x) \to \\
& \quad(\forall x,y\!:\!\mathbb{N}.\ \Phi'\,x\,(x+(\mathsf{s}\,y))) \to \\
& \quad\Phi'\,x\,y
\end{array}
\qquad
\begin{array}{l}
\Phi \;:\; \mathbb{N}\to\mathbb{N}\to\mathsf{Type} \\
\boxed{\begin{array}{ll}
\phi_b & :\ \forall x,y\!:\!\mathbb{N}.\ \Phi\,(y+(\mathsf{s}\,x))\,y \\
\phi_d & :\ \forall x\!:\!\mathbb{N}.\ \Phi\,x\,x \\
\phi_a & :\ \forall x,y\!:\!\mathbb{N}.\ \Phi\,x\,(x+(\mathsf{s}\,y))
\end{array}} \\
\qquad\qquad\downarrow
\end{array}$$

$$?:\Phi\,(\mathsf{s}\,x)\,(\mathsf{s}\,y) \xleftarrow[\texttt{BasicElim}\ Hxy;\ \texttt{Unify};]{} \boxed{\begin{array}{l}
?:\ \forall x,y\!:\!\mathbb{N}.\ \Phi\,(\mathsf{s}\,(y+(\mathsf{s}\,x)))\,(\mathsf{s}\,y) \\
?:\ \forall x\!:\!\mathbb{N}.\ \Phi\,(\mathsf{s}\,x)\,(\mathsf{s}\,x) \\
?:\ \forall x,y\!:\!\mathbb{N}.\ \Phi\,(\mathsf{s}\,x)\,(\mathsf{s}\,(x+(\mathsf{s}\,y)))
\end{array}}$$

An example—$\mathbb{N}$**-plus-rec** and $\mathbb{N}$**-compare** allow us to write Euclid's algorithm:

$$\mathbf{Gcd} \mapsto \lambda x,y\!:\!\mathbb{N}.\ \mathbb{N}$$

$$?:\ \forall x,y\!:\!\mathbb{N}.\ \mathbf{Gcd}\,x\,y$$

Eliminating x and y in turn with $\mathbb{N}$**-plus-rec** installs 'hypothesis collectors' for recursive calls on lexicographically smaller pairs of numbers. Now let us analyse the arguments with $\mathbb{N}$**-compare** and $\mathbb{N}$**-case**—here are the generated patterns:

$$\begin{array}{lllll}
 & & \mathbf{Gcd}\,(y+(\mathsf{s}\,x))\,y & \xleftarrow[\mathbb{N}\textbf{-case on } y]{} & \boxed{\begin{array}{l}\mathbf{Gcd}\ (\mathsf{s}\,x)\ \ 0 \\ \mathbf{Gcd}\,(\mathsf{s}\,(y+(\mathsf{s}\,x)))\,(\mathsf{s}\,y)\end{array}} \\
\mathbf{Gcd}\,x\,y & \xleftarrow[\mathbb{N}\textbf{-compare}]{} & \mathbf{Gcd}\ x\quad x & \longleftarrow & \mathbf{Gcd}\ x\quad x \\
 & & \mathbf{Gcd}\,x\,(x+(\mathsf{s}\,y)) & \xleftarrow[\mathbb{N}\textbf{-case on } x]{} & \boxed{\begin{array}{l}\mathbf{Gcd}\ 0\ \ (\mathsf{s}\,y) \\ \mathbf{Gcd}\,(\mathsf{s}\,x)\,(\mathsf{s}\,(x+(\mathsf{s}\,y)))\end{array}}
\end{array}$$

For the 'diagonal' case, we return x; for the 0 cases, we return $(\mathsf{s}\,x)$ and $(\mathsf{s}\,y)$ respectively. The remaining cases are solved by appeal to the hypothesis collectors, which legitimize the recursive calls producing $\mathbf{Gcd}\,(\mathsf{s}\,x)\,(\mathsf{s}\,y)$.

Our program is not justified by an external argument relating $<$ to subtraction 'on the right'. We have analysed the data declaratively 'on the left' in terms of $+$, and employed a *structural* recursion. This derived structure gets its operational semantics not from a clever matcher, but by executing $\mathbb{N}$-**plus-rec** and $\mathbb{N}$-**compare**.

10 Conclusion and Further Work

Many of the ideas underlying the tactics and techniques in this paper are rooted in folklore: using equational constraints for induction on instantiated relations is hardly novel, but there is a gap between folklore and an implementable general purpose tactic—a gap now bridged. No current theorem prover provides a tactic comparable in flexibility to `AbstElim`. The concise and powerful style of characterization it supports for data, relations and functions could, I believe, optimize large developments considerably for many people, as it has for me.

Along the way, I have given a new 'John Major' definition of equality, adequate to express systems of equations in the presence of dependency. I have also supplied proofs for the '**no confusion**' and '**no cycles**' properties of constructors which extend to dependent datatypes, underpinning `Unify`—an update of the unification tactic from [7]. I hope these technical contributions will prove useful.

However, the story is far from over: there are a number of ways in which this technology could be improved and extended. For example, the current separation of `AbstElim` into `Abst` and `BasicElim` is suboptimal: it introduces an equation only to eliminate it—with a little thought, it should be possible to deliver the same analysis directly. There is also no reason why the basic elimination behaviour for functions should not be generated automatically for each definition. Just as with datatypes themselves, this could be split into a **case** rule, capturing the function's pattern analysis, and a **recursion** rule, capturing its termination structure.

To my mind, though, the most important potential benefit from good elimination technology is the declarative power it gives to programming, especially with dependent types. Here case analysis not only determines control flow, but also refines the type information. The separation of **case** from **recursion**, the characterization of functions and the support for derived notions of structure all have a rôle: a function over some $T\,(f\,x)$, depending on a computed index, may well terminate by the native or derived structure of T, but its analysis should probably examine f.

`AbstElim` and `Unify` enable us to construct these new programs interactively. Nevertheless, it seems desirable to have a high-level term language in which elimination rules (a special kind of dependently typed function) can be defined, then invoked explicitly, delivering derived patterns on the left-hand side of a program. By deriving new elimination rules from old, we have the potential to add significant declarative power to the languages of programming and proof, without any need to extend the underlying operational semantics of the λ-calculus with inductive types.

Acknowledgements

This work would not have been possible without a considerable inheritance of technology from the Coq project, in particular from Cristina Cornes. Much of the detail was worked out under the supervision of Healfdene Goguen and Rod Burstall, to whom I also owe a debt of gratitude. However, it was James McKinna who planted the seeds which grew into this work, and his grant (UK EPSRC GR/N 24988/01) which continues to support it: this paper is for him.

References

1. L'Équipe Coq. The Coq Proof Assistant Reference Manual. paulllac.inria.fr/coq/doc/main.html, Apr 2001.
2. Cristina Cornes and Delphine Terrasse. Inverting Inductive Predicates in Coq. In *Types for Proofs and Programs, '95*, volume 1158 of *LNCS*. Springer-Verlag, 1995.
3. N.G. de Bruijn. Telescopic Mappings in Typed Lambda-Calculus. *Information and Computation*, 91:189–204, 1991.
4. E. Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybyer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs, '94*, volume 1158 of *LNCS*, pages 39–59. Springer-Verlag, 1994.
5. E. Giménez. Structural recursive definitions in type theory. In *Proceedings of ICALP '98*, LNCS 1443. Springer-Verlag, July 1998.
6. Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, May 1992.
7. Conor McBride. Inverting inductively defined relations in LEGO. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs, '96*, volume 1512 of *LNCS*, pages 236–253. Springer-Verlag, 1998.
8. Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.
9. Conor McBride. First-Order Unification by Structural Recursion. Submitted to the Journal of Functional Programming, February 2001.
10. J. McKinna. *Deliverables: A Categorical Approach to Program Development in Type Theory*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.
11. Alan Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12:23–41, 1965.
12. P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL'87*. ACM, 1987.