

Views for Recursion

'The view from the left', JFP, to appear

James McKinna and Conor McBride

Computer-Assisted Reasoning Group

Dept. Computer Science, University of Durham

{J. H. McKinna, C. T. McBride}@durham.ac.uk

Workshop on Termination Proofs, Hindås, 14–15 November, 2002.

Outline of talk

1. A general picture
2. Case analysis and recursion principles
 - Immediately admissible
 - User-defined; evidence
3. A language to support programming in type theory: EPIGRAM
4. An example: **gcd**
 - case analysis: **compare**
 - termination: **plusRec**
5. Bove-Capretta accessibility
6. Further work
7. Conclusions

A general picture

- Work in/on top of a total type theory:
 - choose good types for programs
 - basic theory as a target for elaboration . . .
 - . . . from a higher-level notation
- Separate out case analysis from recursion/termination in the basic theory
- Support, in the higher-level notation, *user-defined* case analysis and recursion principles: identify the structure you would like data to possess, and build the corresponding *views*
- Develop *reusable* instances of each

A language of types

- Luo's UTT (extended with local definitions, implicit syntax ...)
- heterogeneous equality (Streicher, Altenkirch, McBride—'John Major')
- inductive families, presented as

$$\text{data } \frac{\vec{t} : \vec{T}}{\mathbf{D} \vec{t} : \star} \quad \text{where } \frac{\Delta_1}{\mathbf{c}_1 \Delta_1 : \mathbf{D} \vec{s}_1} \quad \dots \quad \frac{\Delta_n}{\mathbf{c}_n \Delta_n : \mathbf{D} \vec{s}_n}$$

giving rise to standard Martin-Löf elimination constants **D-elim** and corresponding ι -reductions.

Recursion, patterns and guardedness

The standard case analysis principle for an inductive family \mathbf{D} ,

$$\mathbf{D}\text{-case} : \forall \vec{t}; x : \mathbf{D} \vec{t}. \forall P : \forall \vec{t}; x : \mathbf{D} \vec{t}. \star .$$

$$\forall m_1 : \forall \Delta_1. P (\mathbf{c}_1 \vec{s}_1). \dots \forall m_n : \forall \Delta_n. P (\mathbf{c}_n \vec{s}_n). P x$$

is always available, while a general form of recursion principle,

$$\mathbf{D}\text{-Frec} : \forall \vec{t}; x : \mathbf{D} \vec{t}. \forall P : \forall \vec{t}; x : \mathbf{D} \vec{t}. \star .$$

$$(\forall \vec{t}; y : \mathbf{D} \vec{t}. \mathbf{F}(P) y \rightarrow P y) \rightarrow P x$$

may be admissible according to the form of the predicate transformer \mathbf{F} . The following instances of $\mathbf{D}\text{-Frec}$ are *always* admissible, using $\mathbf{D}\text{-elim}$:

primitive recursion recursive calls on the immediate subterms

$$\mathbf{F}_0(P)(\mathbf{c}_i \vec{s}_i) \simeq \times_j (P s_{ij})$$

structurally smaller recursive calls on the constructor-guarded sub-patterns

$$\mathbf{F}_1(P)(\mathbf{c}_i \vec{s}_i) \simeq \times_j ((\mathbf{F}_1(P)) s_{ij}) \times \times_j (P s_{ij})$$

User-defined recursion schemes

The functional \mathbf{F} describes how recursive calls are *collected*; what user-defined collection functionals \mathbf{F} give rise to admissible notions of recursion?

An interesting class is given by an already-defined function \mathbf{f} :

$$\mathbf{F}_{\mathbf{f}}(P) x \simeq \forall \vec{y}. (x = \mathbf{f} \vec{y}) \rightarrow P y_j$$

and generalizations thereof. We will use an example of this below.

Question for what \mathbf{f} does this yield admissible recursive calls? (Walther, Abel, Jones *et al.* ...)

A higher-level notation: EPIGRAM

We will develop programs as sequences of top-level datatype declarations and function definitions,

$$\underline{\text{let}} \quad \frac{\Delta}{\mathbf{f} \Delta : T} \quad p$$

where programs p explain how calls to \mathbf{f} should be executed:

- either ‘by’ (\Leftarrow) invoking an eliminator; this generalizes the usual application of **D-elim** in proof by induction/definition by primitive recursion;
- or ‘with’ ($|$) the result of an intermediate computation added to the data under scrutiny; this generalizes Peyton-Jones’ pattern guards;
- 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.

EPIGRAM is currently given a semantics by elaboration into the basic type theory.

An example program: gcd

The following is nearly an EPIGRAM program.

$$\begin{array}{l} \text{let } \frac{m, n : \mathbb{N}}{\text{gcd } m \ n : \mathbb{N}} \\ \text{gcd } 0 \quad (sy) \quad \mapsto \quad sy \\ \text{gcd } (sx) \quad (sx + sy) \mapsto \text{gcd } (sx) \ (sy) \\ \text{gcd } x \quad x \quad \mapsto \quad x \\ \text{gcd } (sx) \quad 0 \quad \mapsto \quad sx \\ \text{gcd } (sy + sx) \quad (sy) \quad \mapsto \quad \text{gcd } (sx) \ (sy) \end{array}$$

Remarks

- recognizably “correct”, if we interpret \mapsto as definitional equality
- need to justify the various case analyses;
 - this is *not* pattern matching on the defined symbol $+$
- need to justify the recursive calls to **gcd**.

Derived case analysis

The ‘patterns’ exhibited in the code for **gcd** arise from the following derivable case analysis principle (\dagger) for pairs of natural numbers $m : \mathbb{N}, n : \mathbb{N}$:

$$\begin{aligned} & \forall P : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \star. \\ & (\forall x, y. P \quad x \quad (x + sy)) \rightarrow \\ & (\forall x. P \quad x \quad x) \rightarrow \\ & (\forall x, y. P (y + sx) \quad y) \rightarrow \\ & P \quad m \quad n \end{aligned}$$

If we can prove such a thing, we could apply it to reduce the problem of computing **gcd** $m \ n$ to the three simpler patterns, which themselves decompose further by pattern matching on x , resp. y .

Comparison of natural numbers, I

The principle (\dagger) is just the *non-dependent* eliminator for the following inductive family:

$$\begin{array}{l} \text{data} \quad \frac{x, y : \mathbb{N}}{\text{Compare } x \ y} \quad \text{where} \quad \frac{}{\text{It } x \ y : \text{Compare } x \ (x + \mathbf{s}y)} \\ \frac{}{\text{eq } x : \text{Compare } x \ x} \\ \frac{}{\text{gt } x \ y : \text{Compare } (y + \mathbf{s}x) \ y} \end{array}$$

We extend EPIGRAM with a syntax

view c

which takes any element c of an inductive family, and elaborates to (the corresponding instance of) its non-dependent eliminator; if we have

$c : \text{Compare } m \ n$, then

view $c : (\dagger)$

Comparison of natural numbers, II

The family **Compare** expresses precisely the pattern decomposition used in **gcd**. As a proposition, $\forall m, n : \mathbb{N}. \text{Compare } m \ n$ expresses the trichotomy law for $<_{\mathbb{N}}$, and we can prove it by writing a program:

let $\text{compare } m \ n : \text{Compare } m \ n$

compare $0 \ 0 \ \mapsto \text{eq } 0$

compare $0 \ (s\ n) \ \mapsto \text{lt } 0 \ n$

compare $(s\ m) \ 0 \ \mapsto \text{gt } m \ 0$

compare $(s\ m) \ (s\ n) \ \Leftarrow \text{view } \text{compare } m \ n$

compare $(s\ x) \ (s(x + s\ y)) \ \mapsto \text{lt } (s\ x) \ y$

compare $(s\ x) \ (s\ x) \ \mapsto \text{eq } (s\ x)$

compare $(s(y + s\ x)) \ (s\ y) \ \mapsto \text{gt } x \ (s\ y)$

compare simultaneously performs testing *and subtraction!*

The **gcd** function, I: its case analysis

We may now put back the missing case analysis for **gcd**:

$$\mathbf{gcd} \quad m \quad n \quad \Leftarrow \text{view } \mathbf{compare} \ m \ n$$

$$\mathbf{gcd} \quad x \quad (x + \mathbf{sy}) \quad \Leftarrow \mathbb{N}\text{-}\mathbf{case} \ x$$

$$\mathbf{gcd} \quad \mathbf{0} \quad (\mathbf{sy}) \quad \mapsto \mathbf{sy}$$

$$\mathbf{gcd} \quad (\mathbf{sx}) \quad (\mathbf{sx} + \mathbf{sy}) \quad \mapsto \mathbf{gcd} \ (\mathbf{sx}) \ (\mathbf{sy})$$

$$\mathbf{gcd} \quad x \quad x \quad \mapsto x$$

$$\mathbf{gcd} \quad (y + \mathbf{sx}) \quad y \quad \Leftarrow \mathbb{N}\text{-}\mathbf{case} \ y$$

$$\mathbf{gcd} \quad (\mathbf{sx}) \quad \mathbf{0} \quad \mapsto \mathbf{sx}$$

$$\mathbf{gcd} \ (\mathbf{sy} + \mathbf{sx}) \ (\mathbf{sy}) \quad \mapsto \mathbf{gcd} \ (\mathbf{sx}) \ (\mathbf{sy})$$

Remarks

- by convention, we may suppress the two appeals to $\mathbb{N}\text{-}\mathbf{case}$;
- evidence for the non-trivial decomposition appears *once*

The gcd function, II: its termination analysis

The recursive calls $\text{gcd}(sx)(sy)$ arising from the case analyses of m, n as (sx) , $(sx + sy)$, respectively $(sy + sx)$, sy , will be legitimised by appeal to a user-defined recursor, which explains why the subterm (sy) is guarded in $(sx + sy)$. We may define the type of such a recursor as follows:

$$\frac{\forall n. (\forall a, b. (n = s(a + b)) \rightarrow P b) \rightarrow P n}{\forall n. P n}$$

This type is the non-dependent elimination rule for the following family **plusRec**:

$$\text{data } \frac{n : \mathbb{N}}{\text{PlusRec } n} \quad \text{where } \frac{R : \forall a, b. (n = s(a + b)) \rightarrow \text{PlusRec } b}{\text{plusRec } R : \text{PlusRec } n}$$

To use it, we must show that the family covers \mathbb{N} ; the proof,

$\text{let } \frac{}{\text{plusrec } n : \text{PlusRec } n} \dots$, is a variant of the usual proof of the well-foundedness of $<_{\mathbb{N}}$.

The gcd function, III: in its full glory

gcd $m \quad n \quad \Leftarrow \underline{\text{view}} \text{ plusrec } m$

gcd $m \quad n \quad \Leftarrow \underline{\text{view}} \text{ plusrec } n$

gcd $m \quad n \quad \Leftarrow \underline{\text{view}} \text{ compare } m \ n$

gcd $x \quad (x + sy) \quad \Leftarrow \mathbb{N}\text{-case } x$

gcd $0 \quad (sy) \quad \mapsto sy$

gcd $(sx) \quad (sx + sy) \quad \mapsto \text{gcd } (sx) \ (sy)$

gcd $x \quad x \quad \mapsto x$

gcd $(y + sx) \quad y \quad \Leftarrow \mathbb{N}\text{-case } y$

gcd $(sx) \quad 0 \quad \mapsto sx$

gcd $(sy + sx) \quad (sy) \quad \mapsto \text{gcd } (sx) \ (sy)$

Remarks

- five lines of ‘program’; five lines of ‘evidence’

The view-construct

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

$$\underline{\text{data}} \frac{t : T}{\text{View-}T t : \star} \quad \underline{\text{where}} \quad \dots \frac{\Delta_j}{c_j \Delta_j : \text{View-}T p_j} \quad \dots$$

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

$$\underline{\text{let}} \quad \frac{}{\text{view-}T t : \text{View-}T t} \quad \dots$$

The view may then be invoked in a program using the ‘by’ construct,

$$lhs \Leftarrow \underline{\text{view}} \text{view-}T t \{p\}$$

Elaboration of views: non-recursive case

The point of the view family $\text{View} \dashv T$ is that we do not care about case analysis on its elements, except inasmuch as it sheds light on the possible “patterns” p_j . Accordingly, we use the *non-dependent* elimination rule for the family:

$$\Gamma \Vdash v \triangleright v' : \mathbf{D} \vec{t}$$

$$\Gamma \vdash \mathbf{D}\text{-case} v' : \forall P' : \forall \vec{t}. \mathbf{D} \vec{t} \rightarrow \star \dots (\forall \Delta_i. P'_{\vec{s}_i} (\mathbf{c}_i \Delta_i)) \rightarrow \dots \rightarrow P' v'$$

$$\Gamma \Vdash \underline{\text{view}} v \triangleright \lambda P : \forall \vec{t}. \star . \mathbf{D}\text{-case} v' (\lambda \vec{t}. \lambda _ : \mathbf{D} \vec{t}. P \vec{t})$$

$$: \forall P : \forall \vec{t}. \star . \dots (\forall \Delta_i. P \vec{s}_i) \rightarrow \dots \rightarrow P \vec{t}$$

Remark Easily extends to the recursive case, but notation is worse still!

Bove-Capretta views

It should not be surprising that viewing data through its Bove-Capretta accessibility predicate now provides a means to legitimize recursive calls: for quicksort,

$$\underline{\text{let}} \frac{l : \text{List } \mathbb{N}}{\text{quicksort } l : \text{List } \mathbb{N}}$$

$$\text{quicksort } l \quad \Leftarrow \quad \underline{\text{view}} \text{ allQsAcc } l$$

$$\text{quicksort } l \quad \Leftarrow \quad \text{List } \text{-case } l$$

$$\text{quicksort } [] \quad \mapsto \quad []$$

$$\text{quicksort } (a : as) \quad \mapsto \quad \text{quicksort } (\text{filter } (< x) as) ++ \\ a :: \text{quicksort } (\text{filter } (\geq x) as)$$

The (non-dependent) recursion principle for the family `qsAcc` yields the required recursive calls; the proof `view allQsAcc l` allows us to apply it.

Further work

- Lots of it!
- Identifying (classes of) admissible recursion schemes
- Views of functions: 'Macarthy-Painter' induction and Bove-Capretta
- ...

Conclusions

- Dependent types allow you to be precise about what you want
- Non-dependent elimination on a family supports the idea of *views*
- Get for free what you can have for free; everything else needs a proof!
- Importance of concise, declarative notation; such proofs should be easy to write, and easier to exploit
- EPIGRAM codes up a lot of experience in how to obtain what you want; you have to believe the elaboration algorithm!
- Now we have to learn to program this way!