EPIGRAM by example

dr. James McKinna, HG 02.540

james@cs.ru.nl

Grondslagen, November 29, 2007

Ism: Conor McBride, Edwin Brady

Dank aan: ‘proefkonijn’ Dan Synek

Themes

- (Certified) Programming... is algorithmic problem-solving... is (interactive; human-guided, machine-supported) proof search;
- Kernel language design
- Inductive families of types
- Enabling idea: generalised “elimination with a motive”
- What about “real programs”: *infinite* or *interactive* computation?

An informal Curry-Howard for programming languages

No matter how weak the type system, we can intuitively interpret it like this:

> the *type* of your program is a *theorem* asserting
> how it will behave

and

> *typechecking* the program *proves* the theorem

[So typechecking is *automated theorem proving*, and programmers can shed the burden of justifying (‘proving’) the behaviour of their programs.]

Type soundness theorems strengthen this intuition

> well-typed programs don’t go wrong

Logical substructure

The underlying (meta-)logic of these theorems and proofs had better be

- *sound* — so you don't talk nonsense
- *expressive* — so you can say what you mean
- *adequate* — so what you say is what you *really* mean

For (statement and proof of) type soundness theorems, this is OK.

For the types of programs themselves, (relative) inexpressivity and non-termination make each of these more problematic.

Going further

Why can't we say

well-typed programs go as specified?

Why can't we expect more

- a more expressive type system, giving better specifications
- a *total* logic, so that we lose the uncertainty of '. . . run forever without blocking. . . '
- while retaining programming as we know it?

Holy Grail: *correctness by design*

A Kernel language design: EPIGRAM(1)

- Inductive families of types
- Function definition: type signatures and implicit syntax
- Generalised elimination: “by” rule <=
- Allowable recursive calls
- Pattern guards/matching intermediate computations: only implemented in Agda2!
- Semantics given by elaboration into (raw) type theory

Inductive families of types

- Index information enforces stronger (A)DT invariants;
- Type-safe meta-programming for free;
- Control structures (can be) reified as *data*;
- Standard ADT programming techniques not available?

Examples: indexing with real data

Peano-Dedekind naturals

$$\underline{\text{data}} \quad \frac{}{\mathsf{Nat} : \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{0} : \mathsf{Nat}} \quad \frac{n : \mathsf{Nat}}{\mathsf{S}\, n : \mathsf{Nat}}$$

Also: …booleans, polymorphic lists…

Polymorphic recursion [Bird & Paterson, Altenkirch & Reus]

$$\underline{\text{data}} \quad \frac{n : \mathsf{Nat}}{\mathsf{Lam}\, n : \star} \quad \underline{\text{where}} \quad \frac{v : \mathsf{Var}\, n}{\mathsf{var}\, v : \mathsf{Lam}\, n} \quad \frac{e : \mathsf{Lam}\, (\mathsf{S}\, n)}{\mathsf{lam}\, e : \mathsf{Lam}\, n} \quad \frac{f, e : \mathsf{Lam}\, n}{\mathsf{app} f e : \mathsf{Lam}\, n}$$

Inference-rule notation suppresses:

- notational noise: quantification, qualification, arrows
- *implicit syntax* (Pollack): arguments which can be inferred by usage

GADT-like examples

Bounded numbers

$$\underline{\text{data}} \quad \frac{n \;:\; \mathsf{Nat}}{\mathsf{Fin}\, n \;:\; \star} \quad \underline{\text{where}} \quad \frac{}{0_n \;:\; \mathsf{Fin}\, \mathsf{S}\, n} \quad \frac{i \;:\; \mathsf{Fin}\, n}{\mathsf{S}_n\, i \;:\; \mathsf{Fin}\, (\mathsf{S}\, n)}$$

Vectors (lists with length)

$$\underline{\text{data}} \quad \frac{A \;:\; \star \quad n \;:\; \mathsf{Nat}}{\mathsf{Vec}\, A\, n \;:\; \star} \quad \underline{\text{where}} \quad \frac{}{[\,]_A \;:\; \mathsf{Vec}\, A\, 0} \quad \frac{v \;:\; A \quad vs \;:\; \mathsf{Vec}\, A\, n}{v ::_n vs \;:\; \mathsf{Vec}\, A\, (\mathsf{S}\, n)}$$

(NB. lengths are correlated with corresponding constructors)

Hence also $m \times n$ Matrices

We get bounds-safe lookup and matrix transpose etc. without tears

Classical Abstract Datatypes

Balanced trees as an intermediate data structure for sorting:

$$\underline{\text{data}} \quad \frac{c : \mathsf{Col} \quad h : \mathsf{Nat}}{\mathsf{RBT}\; c\, h \; : \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{Bleaf} \; : \; \mathsf{RBT}\;\mathsf{B}\;\mathsf{0}}$$

$$\frac{a : A;\; l : \mathsf{RBT}\; lc\, h;\; r : \mathsf{RBT}\; rc\, h}{\mathsf{Bnode}\; a\; l\; r \; : \; \mathsf{RBT}\;\mathsf{B}\;(\mathsf{S}\; h)} \quad \frac{a : A;\; l, r : \mathsf{RBT}\;\mathsf{B}\; h}{\mathsf{Rnode}\; a\; l\; r \; : \; \mathsf{RBT}\;\mathsf{R}\; h}$$

Note: the invariant here is tightly specified; no wiggle room!

Slogan:

smart constructors are just constructors

Also: AVL trees [A-V,L 1962], *etc.* . . .

Indexing with respect to a defined function

a more informative type of binary numbers, indexed with respect to their decoding cf. singleton types [Harper, Xi, Sheard]

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}}{\mathsf{Bin}\, n : \star}\ \underline{\text{where}} \quad \frac{}{\mathsf{B}_0 : \mathsf{Bin} 0} \quad \frac{b : \mathsf{Bin}\, n}{\mathsf{B}_{\mathsf{S}0}\, b : \mathsf{Bin}(2n)}$$

$$\frac{}{\mathsf{B}_1 : \mathsf{Bin} 1} \quad \frac{b : \mathsf{Bin}\, n}{\mathsf{B}_{\mathsf{S}1}\, b : \mathsf{Bin}(2n+1)}$$

can easily be generalised to consider

- positional notation $\mathsf{Num} D n$ with respect to an arbitrary set of digits D; then can correctly specify arithmetic

  $\otimes :: \mathsf{Num} D n \Rightarrow \mathsf{Num} D n \Rightarrow \mathsf{Num} D (m \times n)$

- explicit size bounds on the digits, and on the words over them

Bounded integers; branching on overflow

Obvious function $|-| \; : \; \mathsf{Fin}\, n \rightarrow \mathsf{Nat}$

Gives rise to a family over $b, n \; : \; \mathsf{Nat}$ expressing “small integer” property

$$\underline{\text{data}} \quad \frac{b, n \; : \; \mathsf{Nat}}{\mathsf{Bounded}\; b\; n \; : \; \star}$$

$$\underline{\text{where}} \quad \frac{i \; : \; \mathsf{Fin}\; b}{\mathsf{Small}\; i \; : \; \mathsf{Bounded}\; b\; |i|} \qquad \frac{b, k \; : \; \mathsf{Nat}}{\mathsf{Large}\; b\; k \; : \; \mathsf{Bounded}\; b\; (k + b)}$$

Obvious function $\mathsf{bounded}\; b\; n \; : \; \mathsf{Bounded}\; b\; n$

Now, case analysis on values of $\mathsf{bounded}\; b\; n$ gives an informative *view* [Wadler 1987; McBride-McKinna 2004] of numbers. Slogan:

smarter types deserve smarter eliminators

A type-safe evaluator: universes

A *universe* is given by a type TyExp of (type-)*names*, and a decoding function (a *recursive* family) $\mathsf{Val} : \mathsf{TyExp} \to \star$, e.g.

$\mathsf{TyExp} = \mathsf{nat} | \cdots$ with $\mathsf{Val}\,\mathsf{nat} = \mathsf{Nat}$ etc.

Well-typed evaluator example. . . with a twist

- use of type names means we separate out host language types from object language (but can take $T = \star$ for GADT-style)
- value constructor $\mathsf{val} : \mathsf{Val}\ T \to \mathsf{Exp}\ T$
- the type of the evaluator is the *statement* of type preservation: $\mathsf{eval} : \mathsf{Exp}\ T \to \mathsf{Val}\ T$

cf. intensional polymorphism [Morrison et al., Harper et al., Weirich et al.]

Type-safe meta-programming

Can straighforwardly extend the simple evaluator example to include

- stack type (name)s $\mathsf{StkTyExp}$: just *lists* of TyExp
- well-typed stacks $\mathsf{Stk}\, S$ indexed wrt $S : \mathsf{StkTyExp}$
- family of code fragments $c : \mathsf{Code}\, S\, S'$ indexed wrt $S, S' : \mathsf{StkTyExp}$
- compiler generates code to push a value:
  $\mathsf{compile} : \mathsf{Exp}\, T\; T \to \mathsf{Code}\, S\, (T :: S)$
- interpreter for code: $\dfrac{c : \mathsf{Code}\, S\, S' \;;\; s : \mathsf{Stk}\, S}{\mathsf{exec}\, c\, s : \mathsf{Stk}\, S'}$

Stack-safety for free by decorating the program you (McCarthy) first thought of.

Control is data

- Continuation-passing style emphasises this point;
- Can redo Hutton-Wright “Calculating an exceptional Interpreter” (what about termination?);
- Classical ADT operations: “break invariant; update ; repair” programming pattern needs some help: *zippers* (RBTs again)
- McCarthy’s idea of recursion-induction rehabilitated: computation traces are first-class data (there’s much more to say about this topic)

Elimination with a motive and its generalisation

- Programming with (sub-) families can be (used to be) painful;
- Raw induction/elimination rules are too clumsy;
- Need for equational constraints (Clark completion);
- Type shape of elimination is what matters. . .
- “Non-standard” recursion or case analysis is OK. . . provided it is supported by evidence
- Can this be done in COQ?

Prospectus

Now what?

EPIGRAM(2): a new type theory and implementation

What about computational effects?

What about applications?

What about *infinite* or *interactive* computation?

- Hancock/Setzer/Hyvernat: use Petersson/Synek trees
- Uustalu/Capretta/Altenkirch/McBride/. . . : finally sort out coinduction/corecursion properly, with nice syntax?
- Transaction models: memory, TCP/IP, http, ITasks?

Vragen?