
Why Dependent Types Matter

James McKinna, University of St.Andrews

`james.mckinna@st-andrews.ac.uk`

ACM POPL'06, Charleston SC 2006-01-13

Based on joint work with: McBride, Brady, Altenkirch

Thanks to: Simon PJ and the POPL'06 committee

Why *Sexy* Types Matter

Why Types Matter

Hardly need to speak to this issue for *this* audience, but . . .

- types furnish a *model* of your (eventual) implementation
- they “prevent certain kinds of execution error”
- they provide access control to resources
- they may help the language implementor
- they specify *properties* (however weakly)

An analogy based on Curry-Howard

No matter how weak your type system, we can intuitively read it like this:

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

and

typechecking the program *proves* the theorem

Type soundness theorems strengthen this intuition

well-typed programs don't go wrong

Logical substructure

The underlying 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 type soundness, this is OK.

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

Going further

This picture seems very modest!

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*

I: Pay-as-you-go specification

Progressive refinement of a specification: sorting

The limit of HM-polymorphism. . . [Haskell type classes, SPJ & al. 1993]

$[A] \rightarrow [A]$ — list to list

$\text{Ord } A \Rightarrow [A] \rightarrow [A]$ — A admits an order

but we might want a bit more, e.g.

$\text{Ord } A \Rightarrow [A]_n \rightarrow [A]_n$ — sorting preserves length

where $[A]_n$ represents lists of length n .

We might even like to be more explicit, and write

$\text{Ord } A \Rightarrow (n : \text{Nat}) \Rightarrow [A]_n \rightarrow [A]_n$

indicating A and n are only needed at *compile-time*

Extending Hindley-Milner (roughly 1998–present)

... extensions to existing languages have made this (relatively) easy:

- DML [Xi& Pfenning]: arithmetic constraints (and oracle checker) are *annotations* on SML programs; ATS [Xi & al] add a whole programming logic layer
- Haskell: “faking it” type dependency [McBride; many others]; use type level *proxies* to stand for the (static) uses of n ; compute with this “data” via type class inference, Prolog-style;
- GADTs [Hinze& Cheney; Xi *et al*; SPJ *et al*...]: specify the *non-uniformity* of the dependency of $[A]_n$ on n in data *constructors*;
- Ω mega (Sheard& Pasalic): reify type-level proxies for data at the *kind* layer, with first-class *functions* definable over them

Impressive as all this may be

All of this effort has been directed at

- increasing the expressive strength of the type system to follow a path of increasing the specification expressivity, while
- retaining evolutionary compatibility with existing practice
- Haskell/ghc: ‘the laboratory for advanced type hackery’
- ... and in particular, variations on the phase distinction
 - compile-time “pure” vs. run-time “impure”, or effectful
 - erasure semantics for DML, Ω mega, GHC+GADTs+...

Phase distinction: the event horizon for type systems

there seems to be a fundamental problem with keeping static and dynamic layers apart:

you cannot say that the thing you construct is
related to the input you started with

for others, there seems to be a fundamental problem with mixing static and dynamic layers:

types might get passed at run-time

Keep going!

- $\text{Ord } A \Rightarrow [A] \rightarrow [A]$
- $\text{Ord } A \Rightarrow (n : \text{Nat}) \Rightarrow [A]_n \rightarrow [A]_n$
- $(\leq : \text{Ord } A) \Rightarrow (n : \text{Nat}) \Rightarrow [A]_n \rightarrow [A]_n^{\leq}$

where $[A]_n^{\leq}$ is a type of \leq -sorted lists (of length n): need to represent *evidence* for being ordered

- $(\leq : \text{Ord } A) \Rightarrow (as : [A]) \rightarrow [A]_{\sim as}^{\leq}$

where $[A]_{\sim as}^{\leq}$ is a type of \leq -sorted lists which are a *permutation* of the *input* list as : you need not only propositional information, but how it relates eventual *run-time* objects.

Explicit dependency on inputs increases with specification strength.

we want types which vary with
data

How far can you go? how much will it cost?

The main theme of this talk is to describe a notation

- which affords all of the above typings
- ... and much, much, **much** more besides
- and relate it to the current state-of-the-art

II: Inductive families

Dependent families of types [Martin-Löf 1971]

The key device we exploit to achieve this is the idea of a

dependent family of types $F : T \rightarrow \star$

a function on type $T : \star$ which returns *types* $F t : \star$ given $t : T$.

Allow *arbitrary* T as the domain of variation (not just \star itself)

Then F behaves like a *predicate* on T

Quantification \forall, \exists given by type constructors $\Pi, \Sigma \dots$ so we have typed programs and logic with explicit proofs

An important class of datatypes arise by considering *inductively-defined* F [Dybjer 1991].

A uniform generalisation of GADTs and related notions

Consider the spectrum of possible instances $T\vec{a}$ occurring in source and target types of term constructors and functions:

Hindley-Milner \vec{a} can be type variables only; uniform choice over all constructors of a datatype; function instances $T\vec{\tau}$ similarly uniform

polymorphic recursion non-uniform instances in *source* types for constructors, on a per-constructor basis

GADTs non-uniform instances in source and *target* types: \vec{a} may be *arbitrary* type expressions (necessarily in type-constructor form; no type-level functions)

Ω mega \vec{a} may be *arbitrary* type expressions (*not* necessarily in type-constructor form)

Examples: indexing with real data

Peano-Dedekind naturals

data $\frac{}{\text{Nat} : \star}$ where $\frac{}{0 : \text{Nat}}$ $\frac{n : \text{Nat}}{S\ n : \text{Nat}}$

Polymorphic recursion [Bird & Paterson, Altenkirch & Reus]

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

Inference-rule notation suppresses:

- notational noise: quantification, qualification, arrows
- arguments which can be inferred by usage

GADT-like examples

Bounded numbers

data $\frac{n : \text{Nat}}{\text{Fin } n : \star}$ where $\frac{}{0_n : \text{Fin } S \ n}$ $\frac{i : \text{Fin } n}{S_n \ i : \text{Fin } (S \ n)}$

Vectors (lists with length)

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

(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:

$$\begin{array}{l} \text{data} \quad \frac{c : \text{Col} \quad h : \text{Nat}}{\text{RBT } c \ h : \star} \quad \text{where} \quad \overline{\text{Bleaf} : \text{RBT } B \ 0} \\ \frac{a : A ; l : \text{RBT } l \ c \ h ; r : \text{RBT } r \ c \ h}{\text{Bnode } a \ l \ r : \text{RBT } B \ (\text{S } h)} \quad \frac{a : A ; l, r : \text{RBT } B \ h}{\text{Rnode } a \ l \ r : \text{RBT } R \ h} \end{array}$$

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

Slogan:

smart constructors are just constructors

Also: AVL trees [AVL 1962], sized-balanced trees [Adams 1993], ...

Sorted lists

Need an informative type family $\text{Order } a b$ of *evidence* for ordering on A .

A type of sorted lists with respect to this ordering, *within bounds* a, b ,

data $\frac{a, b : A}{\text{SList } a b : \star}$

where $\frac{p : \text{Order } a b}{\text{SNil } p : \text{SList } a b} \quad \frac{p : \text{Order } a b ; bs : \text{SList } b c}{\text{Scons}_p b bs : \text{SList } a c}$

Note: merging only requires upper or lower bounds, but to append we need upper and lower bounds

Test proposition: $\frac{as : \text{SList } a b}{\text{ordered } as : \text{Order } a b}$ needs transitivity of Order

Adding permutations

Permutation \sim is just another family of evidence

- a congruence for the list constructors
- append becomes commutative $as ++ bs \sim bs ++ as$

To record that a sorted list carries a permutation of a given input list ls ,
add an extra index **SList** $a\ c\ ls$

- in the nil case, we just take $ls = []$
- in the cons case, we just have to say where the head occurred

$$\frac{p : \text{Order } a\ b ;\ bs : \text{SList } b\ c\ ls ;\ q : ls \sim ms ++ ns}{\text{Scons}_p\ b\ bs\ q : \text{SList } a\ c\ (ms ++ (b :: ns))}$$

Binary Search Trees

Actual data carried at nodes yield a finite interval in the ordering, while the null leaves provide the proofs of the pairwise ordering:

$$\begin{array}{l} \text{data} \quad \frac{a, b : A}{\text{BST } a \ b : \star} \quad \text{where} \\ \frac{p : \text{Order } a \ b}{\text{BSTleaf } p : \text{SList } a \ b} \quad \frac{lt : \text{BST } a \ b ; rt : \text{BST } b \ c}{\text{BSTnode } lt \ b \ rt : \text{BST } a \ c} \end{array}$$

Now, to define a good insert function, we also pass in two proofs that the inserted value is within the bounds; splitting arising from the comparison gives us the correct proof to pass into recursive calls.

Adding permutations to a BST

To record that a BST carries a permutation, add an extra index $\text{BST } a \ c \ ls$

- in the leaf case, we just take $ls = []$

- in the node case

$$\frac{lt : \text{BST } a \ b \ ls; \ rt : \text{BST } b \ c \ rs; \ p : ls ++ rs \sim as ++ bs}{\text{BSTnode } lt \ b \ rt \ p : \text{BST } a \ c \ (as ++ (b :: bs))}$$

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]

$$\text{data } \frac{n : \text{Nat}}{\text{Bin } n : \star} \text{ where } \frac{}{\mathbf{B}_0 : \text{Bin } 0} \quad \frac{b : \text{Bin } n}{\mathbf{B}_{S0} b : \text{Bin } (2n)}$$
$$\frac{}{\mathbf{B}_1 : \text{Bin } 1} \quad \frac{b : \text{Bin } n}{\mathbf{B}_{S1} b : \text{Bin } (2n + 1)}$$

can easily be generalised to consider

- positional notation $\text{Num } D n$ with respect to an arbitrary set of digits D ; then can correctly specify arithmetic
 $\otimes :: \text{Num } D n \Rightarrow \text{Num } D n \Rightarrow \text{Num } D (m \times n)$
- explicit size bounds on the digits, and on the words over them

Bounded integers; branching on overflow

Obvious function $| - | : \text{Fin } n \rightarrow \text{Nat}$

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

data $\frac{b, n : \text{Nat}}{\text{Bounded } b \ n : \star}$

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

Obvious function $\text{bounded } b \ n : \text{Bounded } b \ n$

Now, case analysis on values of $\text{bounded } b \ n$ gives an informative *view*

[Wadler 1987; McBride-McKinna 2004] of numbers. Slogan:

smarter types deserve smarter eliminators

The type-safe evaluator: universes

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

$\text{TyExp} = \text{nat} \mid \dots$ with $\text{Val nat} = \text{Nat}$ 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 $\text{val} : \text{Val } T \rightarrow \text{Exp } T$
- the type of the evaluator is the *statement* of type preservation:
 $\text{eval} : \text{Exp } T \rightarrow \text{Val } T$

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

Correct compilers and interpreters by design

Can straightforwardly extend the simple evaluator example to include

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

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

String recognition with regexps (sketch)

Firstly define the type $\text{Lang } r$ indexed over a regexp r

$$\begin{array}{l} \text{data } \frac{r : \text{Regexp } A}{\text{Lang } r : \star} \quad \text{where } \frac{}{\epsilon : \text{Lang } 1} \\ \\ \frac{p_1 : \text{Lang } r_1 \quad p_2 : \text{Lang } r_2}{p_1 \bullet p_2 : \text{Lang } r_1 r_2} \quad \frac{a : A \quad p : \text{Lang } r}{a.p : \text{Lang } a.r} \dots \text{etc.} \end{array}$$

NB. $\text{Lang } 0$ is an *empty* type: no constructor is declared for it. cf. $\text{Fin } 0$.

Define $\frac{p : \text{Lang } r ; s : \text{String } A}{\text{unparse } p \ s : \text{String } A}$ such that $\text{unparse } (\epsilon) \ s = s$,

$\text{unparse } (p_1 \bullet p_2) \ s = \text{unparse } p_1 (\text{unparse } p_2 \ s)$ etc.

Consider the family $\text{Recog } r \ s$ of *partially successful matches* with constructor

$$\frac{p : \text{Lang } r \quad s : \text{String } A}{\text{consume } p \ s : \text{Recog } r (\text{unparse } p \ s)}$$

which captures successful parsing of *an initial segment* of the input string.

Function definitions

Each family D comes with the “obvious” case analysis and structural induction/recursion principles

We insist, at this stage, on *total* function definitions. So we insist on explicit evidence for termination: apply structural recursion.

But constraining the source and target types for functions makes that easier!

(Can make it harder: don't get to loosen the invariant)

Views give us a way to transcend constructor-based pattern analysis

Examples

Vectors: $\text{tail} : \text{Vec } A (\mathbf{S } n) \rightarrow \text{Vec } A n$ only applies in the $v::vs$ case

Typed stacks: a stack of type $\text{Stk } (\mathbf{nat}:: S)$ has a Nat at the top

Bounded numbers: view of a pair of numbers gives *ordering* and *subtraction* all-in-one

But there are some wrinkles:

- insertion in a red-black tree: what's the height of the result? Possible solution: use the types to re-model insertion via *zippers*
- the empty sorted list, the empty set as a BST, as opposed to empty BSTleaves used as “spacers” in the ordering

Learning by testing

Constructors tell you more about types

Constrained types of functions tell you more about the possible constructors in case analysis

Case analysis on (intermediate) function calls tell you more about. . .

. . . the inputs to the function!

- When $(\text{bounded } b \ n) = \text{Small } i$, then we know (and so does the typechecker!!!) that $n = |i|$.
- When $(\text{recog } r \ s) = \text{consume } p \ s'$, then $s = \text{unparse } p \ s'$.

Some programming patterns?

decorate — transform — undecorate

complex control becomes reified as complex type family

the (imperative) ADT pattern

establish invariant — break — transform — repair

invariant

doesn't survive entirely unscathed

establish invariant — classify possible broken states in
the transform steps — transform — extract invariant from
end state

smart constructors for smarter
types

smarter eliminators

learning by testing

III: EPIGRAM

Core design considerations

Give programmers full access to inductive families, not just GADTs

But: overcome the heartache when programming with them in LEGO, Coq

Insist on type signatures for functions, but then write “untyped” bodies

Case analysis, “pattern matching” notation, should be available and recognisable, rather than hidden behind a tactic layer

Curry-Howard interpretation should be retained (trivially), but not follow the model of “prove-then-extract” of Coq, and way before it, NuPRL

Interactive, typechecker-assisted, programming cf. ALF, Agda

What do you get?

Inductive families, more or less as you see them

Function definition:

- let plus function signature plus program body
- 'return' => defines RHS
- invoking an analysis principle <=: for `case`, `rec`, `view`
- that's it! (our original definition allowed analysis of intermediate computations; on the to-do list!)

Proving theorems about programs

The language is for *programming* . . .

. . . but theorem proving is not so very different:

- certified evidence for more interesting test properties
- is also available. . . by writing programs

Speculation: most theorems that programmers actually want to prove about programs

- are about invariant preservation
- so arise as possible refined typings of those programs
- or, fall into a category of “prove once for hygiene/sanity’s sake; then throw away”

Weird stuff

Dependent types bring weird stuff with them

- *non-structural* computational equality \simeq on types
- equational reasoning involved in type checking: equality as a *type family*
- the design space of types for programs is much bigger (and unfamiliar)

Much of the work in the original language definition is about encapsulating weird stuff to do with equality

(NB. we use a subtly different equality to that in Coq, so dependent case analysis/inversion is a lot easier in our setting, without the troublesome loss of equations one expects to find experienced when using Coq).

Equality: computational and propositional

Typing function application

- Equality on types, induced by computation in terms occurring as indices;
- To typecheck $f : (x : S) \rightarrow T$ applied to $e : S'$ we need to check $S \simeq S'$, and then conclude that $fe : [x \mapsto e]T$

Using equality to support typechecking

- given a *proof* $p : S = S'$, then we can construct $p^*e : S$
- hence $f(p^*e) : [x \mapsto e]T$, without further computation

Equality is *very special*; lots of it hidden away to ensure well-typedness.

Ongoing: observational type theory [Altenkirch, McBride]

Case analysis

To construct a program of type: $\text{Vec } A \ (\mathbf{S } k) \rightarrow \text{Vec } A \ k$

... it suffices to do so for: $\text{Vec } A \ n \rightarrow (p : n = (\mathbf{S } k)) \rightarrow \text{Vec } A \ k$

case gives us programs of type: $(vs : \text{Vec } A \ n) \rightarrow T \ A \ n \ vs$

provided we have programs of type $T \ A \ \mathbf{0} \ []$ and

$$(m : \text{Nat}) \rightarrow (v : A) \rightarrow (vs : \text{Vec } A \ m) \rightarrow T \ A \ (\mathbf{S } m) \ (v :: vs)$$

[McKinna & McBride, JFP 2004; McBride 2000, McBride 1995]

cf. “dependent inversion” in Coq [Cornes, Terrasse 1995];

“dependent pattern matching” in ALF [Coquand, 1992]

Beyond case analysis

case isn't special!

Any term of the right type shape can be exploited in this way.

- structural recursion/induction
- well-founded recursion
- domain predicates [Bove, Capretta]
- our take on Wadlerian views
- even the Y combinator!

[McKinna-McBride 2004] for *all* the gory details

Semantics

Focussing on total fragment yields very strong properties for

- compile-time language
- and for run-time [Brady]
- crucial that subterms (e.g. proofs of equations) be removable
- need them to be total!

New twists on the phase distinction [omitted]

Pragmatics

EPIGRAM represents both

- One implementation of dependently-typed programming, unlike any other so far
- Another kind of experiment with our ideas about declarative programming

Sapir-Whorf hypothesis:

language influences thought

We started with interactive, type directed, theorem proving [thanks to Rod Burstall, Randy Pollack]. Dependent types directly support this activity

HCI/cognitive dimensions issues: esp. *premature commitment*

www.e-pig.org

What you get

an archive of

- papers
- documentation, tutorials
- implementations:
 - ghc plus XEmacs (!)
 - stand-alone typechecker [Chapman]
 - G-machine back-end [Brady]
- ... and the inevitable blog

What you don't get

- A language for hardened C++ games programmers!
- No module system: programming from first principles is very painful, so examples in our library tend to showcase small examples
- No royal road to designing and choosing types for a given problem
 - How to combine e.g. red-black trees with the BST invariant
 - More generally, where do you draw the line between types and precondition for functions?
 - One reason to be interested in interactive development
- The sorting (but mergesort is in our paper) or regexp examples (yet!)

Lots to do to get it to a state beyond an experimental platform.

please join in!

IV: Extensions

no time or space to consider
these!

Extensions

Implementation extensions

Language extensions:

- Coinductive types: partiality is an effect [Capretta]
- Generalising the I/O monad [Hancock, Setzer, Hyvernati]

www.e-pig.org

www.dcs.st-and.ac.uk/~james

Questions?