Programming with dependent types: passing fad or useful tool?

Xavier Leroy

INRIA Paris-Rocquencourt

IFIP WG 2.8, 2009-06

Dependent types

In a very general sense: all frameworks enabling programmers to

- Write functional programs;
- State logical properties about them;
- Prove these properties with machine assistance.

Examples: most proof assistants (HOL, Isabelle/HOL, Coq, Agda, ...)

Unquestionably a Very Very Good Thing.

Dependent types

In a narrower sense: all frameworks enabling programmers to

- Include logical propositions within data and function types;
- Include proof terms within data and functions.

Foundations: Martin-Löf's type theory.

Examples: Coq, Agda, Epigram (general); Dependent ML (restricted).

This talk: an experience report on using / not using dependent types when programming and verifying functional programs in Coq.

Dependent function types in Coq

Functions can take proof terms as arguments. . .

```
div: forall (a: Z) (b: Z), b <> 0 -> Z.
```

This function must be called with 3 arguments: 2 integers `a`, `b` and a proof that `b <> 0`.

Dependent data types in Coq

The "subset" type: `{ x : T | P x }`

Data of this type are pairs of an `x` of type `T` and a proof that the proposition `P x` holds. (With `P : T -> Prop`.)

```
proj1_sig: {x: T | P x} -> T
proj2_sig: forall (p: {x: T | P x}), P (proj1_sig x)
```

Examples:

```
Definition Zstar : Type := { x : Z | x <> 0 }.
Definition Zplus : Type := { x : Z | x >= 0 }.
```

Dependent data types in Coq

More generally: dependent record types.

```
Record cfg: Type := mk_cfg {
  graph: nat -> option instruction;
  entrypoint: nat;
  lastnode: nat;
  entrypoint_exists: graph entrypoint <> None;
  graph_finite: forall n, n > lastnode -> graph n = None
}
```

Dependent data types in Coq

The primitive notion: inductive definitions where constructors receive dependent function types.

```
Inductive sig (A: Type) (P: A -> Prop) : Type :=
  | exist:
      forall (x: A), P x -> sig A P.

Definition proj1_sig (A: Type) (P: A -> Prop) (s: sig A P) : A :=
  match s with exist x p => x end.
```

Putting all together

The general shape of a function with precondition P and postcondition Q:

$$\texttt{forall}\ (x_1 : A_1)\ \ldots\ (x_n : A_n),\ P\ x_1\ \ldots\ x_n \rightarrow \{y : B \mid Q\ x_1\ \ldots\ x_n\ y\}$$

Example:

```
divrem: forall (a b: Z), b > 0 ->
        { qr: Z * Z | 0 <= snd(qr) < b
                      /\ a = b * fst(qr) + snd(qr) }
```

Using dependently-typed functions

The hard way: write proof terms by hand.

```
Lemma square_nonzero_pos:
  forall (y: Z), y <> 0 -> y * y > 0.
Proof.
  (* interactive proof *)
Qed.

Definition f (x: Z) (y: Z) (nonzero: y <> 0) : Z :=
  fst (proj1_sig (divrem x (y*y) (square_nonzero_pos y nonzero))).
```

Using dependently-typed functions

The easier way: Matthieu Sozeau's `Program` mechanism.

```
Program Definition f (x: Z) (y: Z) (nonzero: y <> 0) : Z :=
  fst (divrem x (y*y) _ ).

Next Obligation.
  (* interactive proof of
        Z -> forall y : Z, y <> 0 -> y * y > 0 *)
Qed.
```

My practical experience

Dependent types work great to automatically propagate invariants

- Attached to data structures (standard);
- In conjunction with monads (new!).

In most other cases, plain functions + separate theorems about them are generally more convenient.

Attaching invariants to data structures

The example of AVL trees:

```
Inductive tree: Type :=
  | Leaf: tree
  | Node: tree -> A -> tree -> tree.

Inductive bst: tree -> Prop := ...
  (* to be a binary search tree *)

Inductive avl: tree -> Prop := ...
  (* to be balanced according to the AVL criterion *)
```

Attaching invariants to data structures

Need to prove that all base operations over trees preserve the `bst` and `avl` invariants:

```
Definition add (x: A) (t: tree) : tree := ...

Lemma add_invariant:
  forall x t, bst t /\ avl t -> bst (add x t) /\ avl (add x t).
```

Problem: users must also prove that their functions using the base operations preserves these invariants. Without strong proof automation, this entails a lot of manual proof.

Dependent types to the rescue

An internal implementation using plain data structures:

```
Inductive raw_tree: Type := ...
Inductive bst: raw_tree -> Prop := ...
Inductive avl: raw_tree -> Prop := ...
Definition raw_add (x: A) (t: raw_tree) : raw_tree := ...
Lemma raw_add_invariant:
  forall x t, bst t /\ avl t ->
              bst (raw_add x t) /\ avl (raw_add x t).
```

An external interface using a subset type, guaranteeing that the invariant always holds in well-typed user code:

```
Definition tree : Type := { t: raw_tree | bst t /\ avl t }.
Definition add (x: A) (t: tree) : tree :=
  match t with exist rt INV =>
       exist (raw_add x rt) (raw_add_invariant x rt INV)
  end.
```

Attaching invariants to monadic computations

Example: incremental construction of a control-flow graph by successive additions of nodes. A job for the state monad!

```
Record cfg : Type := mk_cfg {
    graph: nat -> option instr;
    nextnode: nat;
    wf: forall n, n >= nextnode -> graph n = Node }.

Definition mon (A: Type) : Type := cfg -> A * cfg.

Definition ret (A: Type) (x: A) : mon A :=
  fun s => (x, s).
Definition bind (A B: Type) (x: mon A) (f: A -> mon B): mon B :=
  fun s => let (r, s') := x s in f r s'.
Program Definition add (i: instr) : mon nat :=
  fun s => (nextnode s,
            mk_cfg (update (graph s) (nextnode s) (Some i))
                   (nextnode s + 1) _).
```

Monotone evolution of the state

Crucial property: nodes are added to the CFG, but existing nodes are never modified.

```
Definition cfg_incl (s1 s2: cfg) : Prop :=
  nextnode s1 <= nextnode s2
  /\ forall n i, graph s1 n = Some i -> graph s2 n = Some i.
```

Easy to prove that `ret`, `bind`, `add` satisfy this property:

```
Lemma add_incr:
  forall s i n s', add i s = (n, s') -> cfg_incl s s'.
```

But users need to prove that similar properties hold for all the functions they define using this monad …

Dependent types to the rescue

Attach an invariant to the monad (new!):

```
Definition mon (A: Type) : Type :=
  forall (s: cfg), { r: A * cfg | cfg_incl s (snd r) }
```

The definitions of the monad operations include some proofs (for `ret` and `bind`: reflexivity and transitivity of `cfg_incl`, respectively).

Then, the `cfg_incl` property comes for free for all user code written with this monad!

What doesn't work well with dependent types

Issue 1: Where to put preconditions?

```
div: forall (a b: Z), b <> 0 -> Z

div: Z -> { b: Z | b <> 0 } -> Z
```

No best choice between these two presentations.

What doesn't work well with dependent types

Issue 2: what properties shoud be attached to the result of a function? what properties should be stated separately?

Extreme example: very basic functions such as list append have a huge number of properties of interest

```
app nil l = l
app l nil = l
app (app l1 l2) l3 = app l1 (app l2 l3)
app l1 l2 = l2 -> l1 = nil
rev (app l1 l2) = app (rev l2) (rev l1)
length (app l1 l2) = length l1 + length l2
In x (app l1 l2) <-> In x l1 \/ In x l2
...
```

If we were to give a dependent type to app, which of these should be attached to the result?
(Assuming they can be attached at all – not true for associativity, e.g.)

What properties should be attached to the result of a general-purpose function?

Sensible answer: none!

Almost sensible answer: an inductive predicate describing the recursion pattern of the function.

```
Inductive p_app: list A -> list A -> list A -> Prop :=
  | p_app_nil: forall l, p_app nil l l
  | p_app_cons: forall l1 l2 l3 a,
      p_app l1 l2 l3 -> p_app (a :: l1) l2 (a :: l3).
```

```
Fixpoint app (l1 l2: list A): { l | p_app l1 l2 l } := ...
```

Enables replacing some reasoning over the function app by reasoning over the inductive predicate p_app. For app, nothing is gained. Sometimes useful for more complex recursion patterns, though.

Summary

Dependent types are a “niche” feature with a couple of convenient uses:

- Propagating data invariants attached to data structures.
- Propagating input-output invariants through monadic computations.

(Note that proof automation more powerful than Coq’s could, in principle, achieve the same propagation without dependent types.)

In all other cases, I believe it’s just more effective to write plain ML/Haskell-style data structures and functions, and separately state and prove properties about them.

Some open questions

Any other good "design patterns" for dependent types?

Any other good examples of dependently-typed monads?