FP in HOL

The story so far

Konrad Slind

School of Computing, University of Utah

November 20, 2006

Higher Order Logic

- Logic built on top of typed lambda calculus
- Originally due to Church (1940s)
- First implemented by Gordon (early 1980s), by adapting LCF implementation of Milner and colleagues
- Now we have HOL-4, HOL-Light, ProofPower, and Isabelle/HOL, all vital systems
- Basically a kind of typed set theory that builds in functions
- Not clear that HOL has anything to do with FP.

Basic FP in HOL

- 1980s
- Gordon's initial work developed some basic types (numbers, pairs, lists) sufficient to do hardware verification examples.
- Melham (thesis) implemented a package for definition of inductive datatypes
- Each such definition provided induction and a so-called *primitive recursion* principle

Theorem (Primitive Recursion Theorem for lists)

```
|- !e f. ?!fn. (fn [] = e) /\
              (!n. fn (h::t) = f h t (fn t))
```

Basic FP in HOL

- Illustrates standard methodology for HOL developments : do not extend logic with new axioms, but instead derive tools on top that use inference to mechanize general theorems
- For example,

Definition (Prim. Rec. FLAT)

```
|- (FLAT [] = []) /\
   (FLAT(h::t) = h ++ FLAT t)
```

  is introduced by constructing appropriate *e* and *f* instances for the P.R. theorem and then deriving the specified equations
- Time consuming and possibly slow,
- *BUT* cuts down on soundness bugs, and gives a nice assurance story
- Reasoning about prim. rec. functions over inductive datatypes supported by custom induction principles.

1990s

- Good start
- BUT, from the point of view of FP'ers this is an impoverished setting in which to write programs:
  - Only very simple patterns
  - Only very restricted kinds of recursion
  - Numerous other irritations
- In 1990s systems emerged that dealt with some of these problems
- Fourman's **LAMBDA** system (defunct)
- **TFL**
  - Complex patterns
  - Arbitrary recursion (termination proofs required)
  - Per-function induction principles (following Boyer and Moore)

TFL

- Based on

Theorem (Wellfounded Recursion theorem)

$$\vdash \textbf{WF}\ R \wedge (f = \textbf{WFREC}\ R\ M) \Rightarrow \exists f.\forall x.\ f(x) = M\ (f\ |_{Rx})\ x$$

- Works by instantiating and manipulating WF Rec. thm (proved in OL)
- A parameterized implementation, instantiated to HOL-4 and Isabelle/HOL
- Handles deep patterns, *e.g.*Okasaki-style Red-Black trees
- Deals well with mutually recursive functions
- Deals with nested recursive functions, but not well (since improved by Matthews and Krstic, and recently by Krauss)

FLAT again

The following version of **FLAT** has more complex patterns and also needs a termination proof in order to be admitted.

Definition (FLAT)

```
|- (FLAT [] = []) /\
   (FLAT([]::rst) = FLAT rst) /\
   (FLAT ((h::t)::rst) = h :: FLAT (t::rst))
```

Theorem (FLAT induction)

```
|- !P. P []  /\
      (!rst. P rst ==> P ([]::rst)) /\
      (!h t rst. P (t::rst) ==> P((h::t)::rst))
     ==>
     !list. P list
```

Typical exercise

Depth first *fold* with graph represented as a function of type

$$\alpha \to \alpha\ \textbf{list}$$

which takes a node and delivers the children of the node.

$\textbf{DFSp} : (\alpha \to \alpha\ \textbf{list}) \to (\alpha \to \beta \to \beta) \to \alpha\ \textbf{list} \to \alpha\ \textbf{list} \to \beta \to \beta$

$\textbf{DFSp}\ G\ f\ seen\ [\,]\ acc = acc$
$\textbf{DFSp}\ G\ f\ seen\ (h :: t)\ acc =$
$\quad \texttt{if}\ \textbf{mem}\ h\ seen$
$\qquad \texttt{then}\ \textbf{DFSp}\ G\ f\ seen\ t\ acc$
$\qquad \texttt{else}\ \textbf{DFSp}\ G\ f\ (h :: seen)$
$\qquad\qquad\qquad\qquad (G\ h \mathbin{++} t)$
$\qquad\qquad\qquad\qquad (f\ h\ acc)$

Depth-first search

- Folds function *f* over directed, possibly cyclic graph
- Applies *f* to each node and accumulating parameter *acc*
- By instantiating *f* can get map, search, max, filter, *etc* functions for such graphs
- Perfectly acceptable functional program

Depth-first search

- The functional representation of the graph allows infinite graphs
- Makes the function partial (if graph has an infinite number of reachable nodes)
- For example $\lambda x.[x + 1]$
- HOL only supports total functions so **DFSp** wouldn't be admitted
- How to repair (totalize)?

Depth-first search repaired

The fold will always terminate given a finite set of reachable nodes. How to define reachability?

Definition (Reachability)

$$\mathbf{R}_G\ x\ y \equiv \mathbf{mem}\ y\ (G\ x)$$
$$\mathbf{reach}_G \equiv \mathbf{RTC}\ R_G$$
$$\mathbf{reachlist}_G\ nodes\ y \equiv \exists x.\ \mathbf{mem}\ x\ nodes \wedge \mathbf{reach}_G\ x\ y$$

Thus, we want to constrain **DFSp** by finiteness of nodes reachable from root nodes of graph.

Total DFS

$$\textbf{DFS} : (\alpha \to \alpha\ \textbf{list}) \to (\alpha \to \beta \to \beta) \to \alpha\ \textbf{list} \to \alpha\ \textbf{list} \to \beta \to \beta$$

```
DFS G f seen to_visit acc =
  if Finite (reachlist_G to_visit)
     then case to_visit
            of [] => acc
             | (h :: t) =>
                 if mem h seen
                    then DFS G f seen t acc
                    else DFS G f (h :: seen)
                                 (G h ++ t)
                                 (f h acc)
     else ARB
```

Nicer Presentation

$$\vdash \mathbf{DFS}\ G\ f\ seen\ \mathbf{[\,]}\ acc = acc$$

$$\begin{array}{l}
\mathbf{Finite}\ (\mathbf{reachlist}_G\ (h :: t)) \\
\quad \vdash \quad \mathbf{DFS}\ G\ f\ seen\ (h :: t)\ acc = \\
\qquad\qquad \texttt{if}\ \mathbf{mem}\ h\ seen \\
\qquad\qquad\quad \texttt{then}\ \mathbf{DFS}\ G\ f\ seen\ t\ acc \\
\qquad\qquad\quad \texttt{else}\ \mathbf{DFS}\ G\ f\ (h :: seen) \\
\qquad\qquad\qquad\qquad\qquad (G\ h \mathbin{++} t) \\
\qquad\qquad\qquad\qquad\qquad (f\ h\ acc)
\end{array}$$

Termination

- In first recursive call, list of seen nodes doesn't change, but nodes still to visit shrinks
- In second recursive call, seen nodes gets bigger and nodes to visit can increase in size
- So simple measures don't work.
- Recall that the set of reachable nodes is finite.
- Idea. Let $\prec$ be the lexicographic combination of the number of reachable nodes not yet seen and the number of nodes in *to_visit*.

Termination relation

Definition

$$(G, f, seen', to\_visit', acc') \prec (G, f, seen, to\_visit, acc)$$
$$\text{iff}$$
$$(\|\mathbf{reachlist}_G\ to\_visit' \setminus \mathbf{ListToSet}\ seen'\|, \mathbf{length}\ to\_visit') <_{lex} (\|\mathbf{reachlist}_G\ to\_visit \setminus \mathbf{ListToSet}\ seen\|, \mathbf{length}\ to\_visit)$$

- In first recursive call h has been previously seen, so the set of unseen reachable nodes does not change, and t is smaller than $h :: t$.
- In the second call, h is added to the *seen* list, and all of the nodes reachable from the children of h are also reachable from h itself. Thus the set of reachable nodes gets no additions, and since the addition to the *seen* list was previously reachable the size of the calculated set decreases.

DFS induction

$$
\begin{array}{l}
\forall P.\\
\left(\begin{array}{l}
\forall G\ f\ s\ h\ t\ a.\\
\quad P\ G\ f\ s\ [\,]\ a \land\\
\quad \left(\begin{array}{l}
(\textbf{Finite}\ (\textbf{reachlist}_G\ (h :: t)) \land \textbf{mem}\ h\ s \Rightarrow P\ G\ f\ s\ t\ a) \land\\
(\textbf{Finite}\ (\textbf{reachlist}_G\ (h :: t)) \land \neg\textbf{mem}\ h\ s\\
\qquad \Rightarrow P\ G\ f\ (h :: s)\ (G\ h +\!\!+\ t)\ (f\ h\ a))\\
\Rightarrow P\ G\ f\ s\ (h :: t)\ a)
\end{array}\right)
\end{array}\right)\\
\Rightarrow\\
\forall v\ v_1\ v_2\ v_3\ v_4.\ P\ v\ v_1\ v_2\ v_3\ v_4
\end{array}
$$

Correctness

What does it mean for this fold on graphs to be correct?

- All reachable nodes are visited
- No unreachable nodes are visited
- No reachable node is visited twice

How, though, do we capture the notion of *visits*?

- We capture this notion by using **cons** as the folding function given to **DFS**, so that the returned list is just the visited nodes.

Visits

DFS with folding function f is equal to gathering all the visited nodes and then folding f over the resulting list.

Theorem (DFS Fold)

$$\begin{array}{l} \textbf{Finite}\ (\textbf{reachlist}_G\ \mathit{to\_visit}) \Rightarrow \\ \quad \textbf{DFS}\ G\ f\ \mathit{seen}\ \mathit{to\_visit}\ \mathit{acc} \\ \qquad = \\ \quad \textbf{foldr}\ f\ \mathit{acc}\ (\textbf{DFS}\ G\ \textbf{cons}\ \mathit{seen}\ \mathit{to\_visit}\ [\,]) \end{array}$$

Correctness

With this understanding, it suffices to prove that the invocation **DFS** G **cons** *seen to_visit* **[ ]** contains no duplicate entries, contains each node reachable from *to_visit*, and contains no nodes not so reachable. The first property is

Theorem (DFS Distinct)

$$\textbf{Finite}\ (\textbf{reachlist}_G\ \mathit{to\_visit}) \Rightarrow \textbf{all\_distinct}\ (\textbf{DFS}\ G\ \textbf{cons}\ \mathit{seen}\ \mathit{to\_visit}\ [\,])$$

and the other two are phrased as

Theorem (DFS Reach)

$$\textbf{Finite}\ (\textbf{reachlist}_G\ \mathit{to\_visit}) \Rightarrow \forall x.\ \textbf{reachlist}_G\ \mathit{to\_visit}\ x \Leftrightarrow \textbf{mem}\ x\ (\textbf{DFS}\ G\ \textbf{cons}\ [\,]\ \mathit{to\_visit}\ [\,])$$

Summary

- Possible to go on and instantiate the various parameters of DFS to get various simplifications
  - Simpler constraint assuring (but not characterizing) termination : finite number of parent nodes in graph
  - DFS with adjacency lists

Adjacency lists

- An adjacency list

$$A : (\alpha \times \alpha\ \textbf{list})\ \textbf{list}$$

gives a listing of nodes alongside their children.

- **toGraph** converts an adjacency list into a graph.

Definition (toGraph)

$$\begin{array}{l} \textbf{toGraph}\ al\ n = \\ \quad \texttt{case}\ \textbf{filter}\ (\lambda(x,\_).\ (x = n))\ al \\ \qquad \texttt{of}\ [\,] \to [\,] \\ \qquad\ \mid (\_,x) :: t \to x \end{array}$$

- Can then prove that DFS terminates when called on any graph derived from an adjacency list

Unconstrained DFS

$$
\begin{array}{l}
\vdash \quad \mathbf{DFS}\ (\mathbf{toGraph}\ A)\ f\ \mathit{seen}\ [\,]\ \mathit{acc} = \mathit{acc} \\
\quad\ \ \mathbf{DFS}\ (\mathbf{toGraph}\ A)\ f\ \mathit{seen}\ (h :: t)\ \mathit{acc} = \\
\qquad\qquad \texttt{if}\ \mathbf{mem}\ h\ \mathit{seen} \\
\qquad\qquad\quad \texttt{then}\ \mathbf{DFS}\ (\mathbf{toGraph}\ A)\ f\ \mathit{seen}\ t\ \mathit{acc} \\
\qquad\qquad\quad \texttt{else}\ \mathbf{DFS}\ (\mathbf{toGraph}\ A)\ f\ (h :: \mathit{seen}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad ((\mathbf{toGraph}\ A)\ h +\!\!+\ t) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (f\ h\ \mathit{acc})
\end{array}
$$

Summary

- Fun tutorial study
- Formalization challenges (partiality, termination, *visits*, ...)
- Need to do math in order to work with such programs (*e.g.*, reachability)
- Possibly of future use; could be added to a library

Current state of affairs

- Programming total functions over inductive datatypes is pretty well handled in most proof systems
- Could always be improved, of course
- Isabelle/HOL has a nice development of domain theory for applications that need it.
- But domains make life more complicated (lifting)
- Support for lazy datatypes and functions over them, not using domains, was pioneered by John Matthews, but is still not mechanized well in any HOL implementation.
- HOL systems have only simple types, and there doesn't seem to be much momentum for supporting more expressive type systems.

Critique and a Response

- Functions essentially trapped inside the formal system ("Case studies are boring")
- Sterile environment?
- But ACL2 community has shown that breaking free of the proof system is possible
- Emitting formal programs into outside world has many benefits
- Other systems have followed: PVS, Isabelle/HOL, HOL-4 all provide export for formal programs
- Coq has a variety of solutions, both internal and external
- Other systems (*e.g.*, Matlab) also export programs and/or hardware

What then?

- Observation: programs are exported to the metalanguage
- Lisp for ACL2, ML for Isabelle/HOL and HOL
- Nice research project: export to mainstream languages like Java or even C
- However, current program export facilities exploit the fact that the conceptual gap between the formal program and the host PL is small
- But what if we want to export to Java, C, or even hardware?
- End up re-capitulating phases of compilation
- But then the small gap gets ever wider ...

Compilation-by-proof

- Our current research investigates ways to
  - specify functional programs as mathematics
  - prove correctness properties at the mathematics level
  - translate to assembly or hardware
  - translation **done by proof**, so result is guaranteed to return the correct answers.
- Amounts to *compilation of logic functions*, inside the logic
- Two approaches to providing this:
  - Verified compiler. This is what is done traditionally
  - Translation validation. Recent alternative proposed by Pnueli

Compilation-by-proof

- Have built two prototype TV compilers for a very simple functional language
  - hardware (with Mike Gordon)
  - ARM assembly (with Owens, Li, Tuerk)
- Work is still very much in progress
- Target example: Elliptic Curve Crypto (relatively efficient replacement for RSA)
  - Formal theory of elliptic curves (on top of finite field theory)
  - Define recursive functions that implement, *e.g.*, addition of points on elliptic curves
  - Compile these to ARM assembly
  - formal ARM model in HOL-4

One approach

- Try to do as much as possible by source-to-source translations.
- Start by translating to combinator form, then to ANF (administrative normal form)
- These end up being *semantic* versions of the standard syntax bashing done in CPS translation
- Register allocation done by standard graph-colouring algorithm. Used to deliver an α-convertible version (nice trick from Jason Hickey)
- Maintenance of equality, by proof, from starting program
- That's the front end

TEA

$$
\begin{array}{l}
\vdash \textbf{Rounds}\ (n,(y,z),(k_0,k_1,k_2,k_3),s) = \\
\quad \texttt{if}\ n = 0w\ \texttt{then}\ ((y,z),(k_0,k_1,k_2,k_3),s) \\
\quad \texttt{else}\ \textbf{Rounds}\ (n - 1w, \\
\qquad\qquad\qquad \texttt{let}\ s' = s + 2654435769w\ \texttt{in} \\
\qquad\qquad\qquad \texttt{let}\ y' = y + \textbf{ShiftXor}(z,s',k_0,k_1) \\
\qquad\qquad\qquad \texttt{in}\ ((y',z + \textbf{ShiftXor}(y',s',k_2,k_3)),(k_0,k_1,k_2,k_3),s')
\end{array}
$$

After Front-end processing

$$
\begin{array}{l}
\vdash \mathbf{Rounds}(r_0,(r_8,r_5),(r_4,r_3,r_2,r_6),r_7) = \\
\quad \texttt{let } v_9 = (\mathbf{op}\ =)\ (r_0, 0w) \\
\quad \texttt{in if } v_9 \texttt{ then } ((r_8,r_5),(r_4,r_3,r_2,r_6),r_7) \\
\quad\quad \texttt{else let } m_2 = (\mathbf{op}\ -)\ (r_0, 1w) \texttt{ in} \\
\quad\quad\quad \texttt{let } m_4 = (\mathbf{op}\ +)\ (r_7, 2654435769w) \texttt{ in} \\
\quad\quad\quad \texttt{let } r_1 = \mathbf{ShiftXor}\ (r_5, m_4, r_4, r_3) \texttt{ in} \\
\quad\quad\quad \texttt{let } r_9 = (\mathbf{op}\ +)\ (r_8, r_1) \texttt{ in} \\
\quad\quad\quad \texttt{let } r_1 = \mathbf{ShiftXor}\ (r_9, m_4, r_2, r_6) \texttt{ in} \\
\quad\quad\quad \texttt{let } r_1 = (\mathbf{op}\ +)\ (r_5, r_1) \texttt{ in} \\
\quad\quad\quad \texttt{let } ((m_5,m_3),(m_1,m_0,m_6,r_1),r_0) = \\
\quad\quad\quad\quad \mathbf{Rounds}\ (m_2,(r_9,r_1),(r_4,r_3,r_2,r_6),m_4) \\
\quad\quad\quad \texttt{in } ((m_5,m_3),(m_1,m_0,m_6,r_1),r_0)
\end{array}
$$

Back end

- Back end poof *synthesizes* a counterpart function to the front end function. Then a tactic is executed to show the two are equal.
- In general, the backend is pretty conventional compiler verification technology
- We *synthesize* the IL semantics from the ARM semantics, rather than relating two operational semantics
- Main difficulty is dealing with memory
- In particular, function call is hard.

Summary

- To be done in near future: defunctionalization to support higher order
- Not mentioned: work on translating FP by proof to hardware
- Early days in this area, but I think it's quite exciting

Recursion Schemes

- **TFL** supports *recursion schemes*, by allowing free variables in rhs, for example

Definition (While-loops)

$$\textbf{While}\ s = \texttt{if}\ B\ s\ \texttt{then}\ \textbf{While}\ (C\ s)\ \texttt{else}\ s.$$

- **While** can also be defined directly
- Connection with FP: Lewis, Shields, Meijer, Launchbury, *Implicit parameters: Dynamic scoping with static types*

Polytypism

- Polytypism (type-indexed functions) is becoming a basic FP tool
- Applications in logic:
  - Termination proofs
  - Normalization by Evaluation
  - Translation between representations (mapping to binary format, to SAT, to LISP, *etc*)
- In HOL systems, two ways to support it:
  - A polytypic function *f* is represented by a meta-level function parameterized by a P.R. theorem (HOL-4).
  - Explicit definitions over type structure (Isabelle/HOL).

Type Classes

- Actually, not so recent ...
- Used extensively in Isabelle/HOL, but not the other HOL systems
- Supports some abstract algebra and number theory hierarchies
- Recent work from CMU translates type classes to ML functors, offering a way to map formal developments from Isabelle/HOL to HOL-Light

THE END