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.
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 \textit{primitive recursion} principle

\textbf{Theorem (Primitive Recursion Theorem for lists)}

\[
\vdash \forall e \; f. \; \exists! fn. \; (fn [] = e) \land \\
(\forall n. \; fn (h::t) = f \; h \; t \; (fn \; t))
\]
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)
Based on

Theorem (Wellfounded Recursion theorem)

\[ \vdash \text{WF } R \land (f = \text{WFREC } R \ M) \Rightarrow \exists f. \forall x. f(x) = M (f \mid_{Rx}) \ x \]

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

**Definition (FLAT)**

\[ |- (\text{FLAT} \; \text{[]} = \text{[]}) \land \]
\[ (\text{FLAT} (\text{[]}::\text{rst}) = \text{FLAT} \; \text{rst}) \land \]
\[ (\text{FLAT} ((\text{h::t)}::\text{rst}) = \text{h} :: \text{FLAT} (\text{t::rst})) \]

**Theorem (FLAT induction)**

\[ |- !P. \; P \; \text{[]} \land \]
\[ (!\text{rst}. \; P \; \text{rst} ==> P (\text{[]}::\text{rst})) \land \]
\[ (!h \; t \; \text{rst}. \; P (\text{t::rst}) ==> P ((\text{h::t)}::\text{rst})) \]
\[ ==> \]
\[ !\text{list}. \; P \; \text{list} \]
Typical exercise

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

\[ \alpha \to \alpha \text{ list} \]

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

\[
\text{DFSp} : (\alpha \to \alpha \text{ list}) \to (\alpha \to \beta \to \beta) \to \alpha \text{ list} \to \alpha \text{ list} \to \beta \to \beta
\]

\[
\text{DFSp } G \ f \ \text{seen } [] \ \text{acc} = \text{acc}
\]

\[
\text{DFSp } G \ f \ \text{seen } (h :: t) \ \text{acc} =
\begin{align*}
\text{if } & \text{mem } h \ \text{seen} \\
\text{then } & \text{DFSp } G \ f \ \text{seen } t \ \text{acc} \\
\text{else } & \text{DFSp } G \ f \ (h :: \ \text{seen}) \\
& (G \ h \ ++ \ t) \\
& (f \ h \ \text{acc})
\end{align*}
\]
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
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)?
The fold will always terminate given a finite set of reachable nodes. How to define reachability?

**Definition (Reachability)**

\[
\begin{align*}
R_G x y &\equiv \text{mem } y \ (G \ x) \\
\text{reach}_G &\equiv \text{RTC } R_G \\
\text{reachlist}_G \ nodes \ y &\equiv \ \exists x. \ \text{mem } x \ nodes \ \land \ \text{reach}_G x y
\end{align*}
\]

Thus, we want to constrain \textbf{DFSp} by finiteness of nodes reachable from root nodes of graph.
Total DFS

\[\text{DFS} : (\alpha \to \alpha \text{ list}) \to (\alpha \to \beta \to \beta) \to \alpha \text{ list} \to \alpha \text{ list} \to \beta \to \beta\]

\[\text{DFS } G \ f \ \text{seen to}_\text{visit} \ \text{acc} =
\begin{align*}
\quad & \text{if } \text{Finite} \ (\text{reachlist}_G \ \text{to}_\text{visit}) \\
\quad & \text{then case } \text{to}_\text{visit} \\
\quad & \quad \text{of } [] \Rightarrow \text{acc} \\
\quad & \quad | \ (h :: t) \Rightarrow \\
\quad & \quad \quad \text{if } \text{mem} \ h \ \text{seen} \\
\quad & \quad \quad \text{then } \text{DFS } G \ f \ \text{seen } t \ \text{acc} \\
\quad & \quad \quad \text{else } \text{DFS } G \ f \ (h :: \ \text{seen}) \\
\quad & \quad \quad \qquad (G \ h \ +\ + \ t) \\
\quad & \quad \quad \qquad (f \ h \ \text{acc}) \\
\quad & \quad \text{else } \text{ARB}
\end{align*}\]
\[ \vdash \text{DFS } G \ f \ \text{seen} \ \ [\ ] \ \text{acc} = \text{acc} \]

\textbf{Finite} (reachlist}_G (h :: t))
\[ \vdash \text{DFS } G \ f \ \text{seen} \ (h :: t) \ \text{acc} = \]
\[ \quad \text{if mem} \ h \ \text{seen} \]
\[ \quad \text{then DFS } G \ f \ \text{seen} \ t \ \text{acc} \]
\[ \quad \text{else DFS } G \ f \ (h :: \text{seen}) \]
\[ \quad \ (G \ h :: t) \]
\[ \quad \ (f \ h \ \text{acc}) \]
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**

\[(G, f, seen', to_visit', acc') \prec (G, f, seen, to_visit, acc)\]

iff

\[\left(\|\text{reachlist}_G to\_visit' \setminus \text{ListToSet}\ seen'\|, \text{length}\ to\_visit'\right) \prec_{\text{lex}}\]

\[\left(\|\text{reachlist}_G to\_visit \setminus \text{ListToSet}\ seen\|, \text{length}\ to\_visit\right)\]

- 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 \textit{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 \textit{seen} list was previously reachable the size of the calculated set decreases.
DFS induction

\[ \forall P. \begin{cases} \forall G \; f \; s \; h \; t \; a. \\ \quad P \; G \; f \; s \; [\; ] \; a \land \\ \quad \left[ \begin{array}{l} \text{(Finite \; (reachlist}_G (h :: t)) \land \text{mem} \; h \; s \Rightarrow P \; G \; f \; s \; t \; a) \land \\ \text{(Finite \; (reachlist}_G (h :: t)) \land \neg \text{mem} \; h \; s \\ \Rightarrow P \; G \; f \; (h :: s) \; (G \; h \; +++ \; t) \; (f \; h \; a)) \\ \Rightarrow P \; G \; f \; s \; (h :: t) \; a \end{array} \right] \end{cases} \Rightarrow \forall v \; v_1 \; v_2 \; v_3 \; v_4. \; P \; v \; v_1 \; v_2 \; v_3 \; v_4 \]
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 \texttt{cons} as the folding function given to \texttt{DFS}, so that the returned list is just the visited nodes.
**DFS** with folding function $f$ is equal to gathering all the visited nodes and then folding $f$ over the resulting list.

**Theorem (DFS Fold)**

$$\text{Finite } (\text{reachlist}_G \text{ to\_visit}) \Rightarrow \text{DFS } G \ f \ \text{seen to\_visit} \ \text{acc} = \text{foldr} \ f \ \text{acc} \ (\text{DFS } G \ \text{cons} \ \text{seen to\_visit} \ [\ ])$$
Correctness

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

**Theorem (DFS Distinct)**

\[
\text{Finite } (\text{reachlist}_G \text{ to\_visit}) \\
\Rightarrow \text{all\_distinct } (\text{DFS G cons seen to\_visit [ ]})
\]

and the other two are phrased as

**Theorem (DFS Reach)**

\[
\text{Finite } (\text{reachlist}_G \text{ to\_visit}) \Rightarrow \\
\forall x. \text{reachlist}_G \text{ to\_visit } x \\
\iff \\
\text{mem } x (\text{DFS G cons [ ] to\_visit [ ]})
\]
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

\[ A : (\alpha \times \alpha \text{ list}) \text{ list} \]

gives a listing of nodes alongside their children.
- **toGraph** converts an adjacency list into a graph.

**Definition (toGraph)**

\[
\text{toGraph } al \ n = \\
\quad \text{case } \text{filter} \ (\lambda(x,\_). \ (x = n)) \ al \\
\quad \text{of } [] \rightarrow [] \\
\quad | (\_, x) :: t \rightarrow x
\]

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

\[\begin{align*}
\vdash & \text{DFS} \ (\text{toGraph} \ A) \ f \ seen \ [\ ] \ acc = acc \\
& \text{DFS} \ (\text{toGraph} \ A) \ f \ seen \ (h :: t) \ acc = \\
& \quad \text{if mem} \ h \ seen \\
& \quad \text{then DFS} \ (\text{toGraph} \ A) \ f \ seen \ t \ acc \\
& \quad \text{else DFS} \ (\text{toGraph} \ A) \ f \ (h :: seen) \\
& \quad (\text{toGraph} \ A) \ h \ ++ \ t \\
& \quad (f \ h \ acc)
\end{align*}\]
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
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
 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 ...
Our current research investigates ways to
  - specify functional programs as mathematics
  - prove correctness properties at the mathematics level
  - translate to assembly or hardware
  - translation \textbf{done by proof}, so result is guaranteed to return
    the correct answers.

Amounts to \emph{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
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 $\alpha$-convertible version (nice trick from Jason Hickey)
- Maintenance of equality, by proof, from starting program
- That’s the front end
\[ \text{Rounds}(n, (y, z), (k_0, k_1, k_2, k_3), s) = \]
\[ \text{if } n = 0 \text{ then } ((y, z), (k_0, k_1, k_2, k_3), s) \]
\[ \text{else } \text{Rounds}(n - 1, w, \]
\[ \text{let } s' = s + 2654435769w \text{ in} \]
\[ \text{let } y' = y + \text{ShiftXor}(z, s', k_0, k_1) \]
\[ \text{in } ((y', z + \text{ShiftXor}(y', s', k_2, k_3)), (k_0, k_1, k_2, k_3), s') \]
After Front-end processing

\[ \vdash \text{Rounds}(r_0, (r_8, r_5), (r_4, r_3, r_2, r_6), r_7) = \]
\[ \quad \begin{array}{l}
\text{let } v_9 = (\text{op } \not=) (r_0, 0w) \\
in \text{ if } v_9 \text{ then } ((r_8, r_5), (r_4, r_3, r_2, r_6), r_7) \\
\quad \text{else let } m_2 = (\text{op } \not-) (r_0, 1w) \text{ in} \\
\quad \quad \begin{array}{l}
\text{let } m_4 = (\text{op } \not+) (r_7, 2654435769w) \text{ in}
\text{let } r_1 = \text{ShiftXor}(r_5, m_4, r_4, r_3) \text{ in}
\text{let } r_9 = (\text{op } \not+) (r_8, r_1) \text{ in}
\text{let } r_1 = \text{ShiftXor}(r_9, m_4, r_2, r_6) \text{ in}
\text{let } r_1 = (\text{op } \not+) (r_5, r_1) \text{ in}
\text{let } ((m_5, m_3), (m_1, m_0, m_6, r_1), r_0) =
\quad \quad \text{Rounds}(m_2, (r_9, r_1), (r_4, r_3, r_2, r_6), m_4) \\
\text{in } ((m_5, m_3), (m_1, m_0, m_6, r_1), r_0) 
\end{array}
\end{array} \]
Back end

- Back end proof *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.
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
TFL supports recursion schemes, by allowing free variables in rhs, for example

**Definition (While-loops)**

\[
\text{While } s = \text{if } B \ s \ \text{then } \text{While } (C \ s) \ \text{else } s.
\]

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