Recursive function definition for types with binders

Michael Norrish
Michael.Norrish@nicta.com.au

National ICT Australia

17 September 2004

Outline

Theorem-proving, redux

1. Find your type:
   - When proving Fermat's Last Theorem, HOL provides the type of natural numbers ($\mathbb{N}$)
   - When verifying a hardware design, the (new) type for the system state-space needs to be specified (tuple of registers, memory ...)

Theorem-proving, redux

1. Find your type:
   - When proving Fermat's Last Theorem, HOL provides the type of natural numbers ($\mathbb{N}$)
   - When verifying a hardware design, the (new) type for the system state-space needs to be specified (tuple of registers, memory . . .)
2. Define functions over the type:
   - Define gcd over $\mathbb{N}^2$
   - Define a transition relation over the hardware state-space

Theorem-proving, redux

1. Find your type:
   - When proving Fermat's Last Theorem, HOL provides the type of natural numbers ($\mathbb{N}$)
   - When verifying a hardware design, the (new) type for the system state-space needs to be specified (tuple of registers, memory . . .)
2. Define functions over the type:
   - Define gcd over $\mathbb{N}^2$
   - Define a transition relation over the hardware state-space
3. Prove theorems!
   - . . .
   - Prove safety, liveness . . .

Theorem-proving, redux

1. Find your type:
   - When proving Fermat's Last Theorem, HOL provides the type of natural numbers ($\mathbb{N}$)
   - When verifying a hardware design, the (new) type for the system state-space needs to be specified (tuple of registers, memory . . .)
2. Define functions over the type:
   - Define gcd over $\mathbb{N}^2$
   - Define a transition relation over the hardware state-space
3. Prove theorems!
   - . . .
   - Prove safety, liveness . . .

This talk is about step 2: function definition.

Inductive types, recursive functions

- Given the type of lists, want to define a (primitive) recursive function such as foldl, with definition

$$\begin{aligned} \mathsf{foldl}\ f\ x\ [\,] &= x \\ \mathsf{foldl}\ f\ x\ (e :: t) &= \mathsf{foldl}\ f\ (f(e,x))\ t \end{aligned}$$

- How can such a definition be allowed?

Recursion theorems

For lists:

$$\begin{aligned} \vdash \quad & \forall n\, c.\ \exists h. \\ & \quad h\ [\,] = n\ \wedge \\ & \quad \forall e\ t.\ \ h\ (e :: t) = c\ (h\ \ t)\ \ e\ \ t \end{aligned}$$

- n is the value when the function (h) is applied to an empty list
- c is the value when the function is applied to a "cons". c can compute its answer with reference to
  - the head element of the list (e)
  - the rest of the list (t)
  - the result of the recursive call of h applied to t

Demonstrating the existence of foldl

Begin with the recursion theorem

$$\vdash \quad \forall n\ c.\ \exists h.\ h\ [\,] = n\ \wedge\ \forall e\ t.\ \ h\ (e :: t) = c\ \ (h\ \ t)\ \ e\ \ t$$

Demonstrating the existence of foldl

Begin with the recursion theorem

$$\vdash \quad \forall n\ c.\ \exists h.\ h\ [\,] = n\ \wedge\ \forall e\ t.\ h\ (e :: t) = c\ (h\ t)\ e\ t$$

- Take n to be $(\lambda f\ x.\ x)$

Demonstrating the existence of foldl

Begin with the recursion theorem

$$\vdash \quad \forall c.\ \exists h.\ \ h\ [\,] = (\lambda f\ x.\ \ x)\ \wedge\ \ \forall e\ t.\ \ h\ (e :: t) = c\ (h\ \ t)\ \ e\ \ t$$

- Take n to be $(\lambda f\ x.\ \ x)$

Demonstrating the existence of foldl

Begin with the recursion theorem

$$\vdash \quad \forall c.\ \exists h.\quad h\ [\,] = (\lambda f\ x.\ \ x)\ \wedge \quad \forall e\ t.\ \ h\ (e :: t) = c\ \ (h\ \ t)\ \ e\ \ t$$

- Take n to be $(\lambda f\ x.\ \ x)$
- Take c to be $(\lambda r\ e\ t\ f\ x.\ \ r\ \ f\ \ (f(e,x)))$

Demonstrating the existence of foldl

Begin with the recursion theorem

$$\vdash \quad \exists h. \quad h\ [\,] = (\lambda f\ x.\ \ x)\ \wedge \quad \forall e\ t.\ \ h\ (e :: t) = (\lambda r\ e\ t\ f\ x.\ \ r\ \ f\ \ (f(e,x)))\ (h\ \ t)\ \ e\ \ t$$

- Take n to be $(\lambda f\ x.\ \ x)$
- Take c to be $(\lambda r\ e\ t\ f\ x.\ \ r\ \ f\ \ (f(e,x)))$

Demonstrating the existence of foldl

Begin with the recursion theorem

$$\vdash \quad \exists h. \quad h\ [\,] = (\lambda f\ x.\ \ x)\ \wedge \quad \forall e\ t.\ \ h\ (e :: t) = \underset{\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots}{(\lambda r\ e\ t\ f\ x.\ \ r\ \ f\ \ (f(e,x)))\ (h\ \ t)\ \ e\ \ t}$$

- Take n to be $(\lambda f\ x.\ \ x)$
- Take c to be $(\lambda r\ e\ t\ f\ x.\ \ r\ \ f\ \ (f(e,x)))$
- β-reduce

Demonstrating the existence of foldl

Begin with the recursion theorem

$$\vdash \quad \exists h.\; h\;[\,] = (\lambda f\;x.\;\; x) \;\wedge\; \forall e\;t.\;\; h\;(e :: t) = (\lambda f\;x.\;\; h\;t\;f\;\;(f(e,x)))$$

- Take n to be $(\lambda f\;x.\;\; x)$
- Take c to be $(\lambda r\;e\;t\;f\;x.\;\; r\;\;f\;\;(f(e,x)))$
- β-reduce

Demonstrating the existence of foldl

Begin with the recursion theorem

$$\vdash \quad \exists h. \quad \forall f\ x.\ \ h\ [\,]\ f\ \ x = x\ \ \wedge \quad \forall e\ t\ f\ x.\ \ h\ (e :: t)\ \ f\ \ x = h\ \ t\ \ f\ \ (f(e,x)))$$

- Take n to be $(\lambda f\ x.\ \ x)$
- Take c to be $(\lambda r\ e\ t\ f\ x.\ \ r\ \ f\ \ (f(e,x)))$
- β-reduce
- Use extensionality to handle λs on the right

Types need recursion theorems

- It's easy to provide recursion theorems for standard algebraic types (lists, trees, &c)
- Basic desirable form is

$$
\begin{array}{l}
\vdash \forall \ldots \color{red}{f_i} \ldots\ \exists h. \\
\qquad \cdots \quad \wedge \\
\qquad \forall \ldots x_j \ldots r_k. \\
\qquad\quad h\ (\color{red}{C_i}(\ldots x_j, \ldots r_k)) = \color{red}{f_i}\ (h\ \ r_k)\ \ \ldots\ \ x_j\ \ \ldots\ \ r_k\ \ \wedge \\
\qquad \cdots
\end{array}
$$

Where

- x_j is a non-recursive parameter to constructor $\color{red}{C_i}$
- r_k is a recursive parameter to the same constructor
- $\color{red}{f_i}$ gets access to x_j, r_k, and the result of recursive call $(h\ \ r_k)$

α-equivalence

- The type representing the syntax of λ-terms will have constructors:

$$\begin{array}{lcl} \mathtt{VAR} & : & \mathit{string} \to \mathtt{term} \\ \mathtt{APP} & : & \mathtt{term} \to (\mathtt{term} \to \mathtt{term}) \\ \mathtt{LAM} & : & \mathit{string} \to (\mathtt{term} \to \mathtt{term}) \end{array}$$

- Add α-equivalence: *"the choice of variable name doesn't matter"*:
  - LAM x x is "the same" as LAM y y
- On *raw* syntax, α-equivalence ($\equiv_\alpha$) captures "the same"
- At level of interest, $\equiv_\alpha$ is just $=$

Recursion theorem for types with α-equivalence

- The recursion theorem for the type "should" have the `LAM`-clause:

$$h\ (\mathtt{LAM}\ v\ t) = lam\ v\ t\ (h\ t)$$

- But this would allow unsound definition of

$$bogus\ (\mathtt{LAM}\ v\ t) = v$$

- Side-conditions will be required!

The Gordon-Melham type and its recursion theorem

- Gordon & Melham (1996) provide a type of λ-terms
- Represents α-equivalent terms, satisfying $\texttt{LAM}\ x\ x = \texttt{LAM}\ y\ y$
- Defines substitution, e.g., $M[v \mapsto N]$
- Has recursion theorem, but $\texttt{LAM}$ clause is

$$\begin{aligned} h\ (\texttt{LAM}\ v\ t)\ &= \\ &\mathit{lam}\ (\lambda y.\ h\ (t[v \mapsto \texttt{VAR}(y)]))\ (\lambda y.\ t[v \mapsto \texttt{VAR}(y)]) \end{aligned}$$

- *lam* gets no access to v, and access to body and recursion result is via functions that perform substitutions

Building on the Gordon-Melham type

I will transform the Bad Clause

$$h\ (\mathtt{LAM}\ v\ t) \ = \ lam\ (\lambda y.\ h\ (t[v \mapsto \mathtt{VAR}(y)]))\ (\lambda y.\ t[v \mapsto \mathtt{VAR}(y)])$$

into the Good Clause

$$h\ (\mathtt{LAM}\ v\ t) = lam\ (h\ t)\ v\ t$$

while still preventing

$$bogus\ (\mathtt{LAM}\ v\ t) = v$$

through appropriate side-conditions

Motivating examples

- Some direct references to bound variable names and abstraction bodies are legitimate.
- If the range of the function is a simple type
  - Calculating term size:

```
size (CON k)   = 1
size (VAR s)   = 1
size (APP t u) = 1 + size t + size u
size (LAM v t) = 1 + size t
```

  - Is a term in β-normal form:

```
bnf (CON k)   = T
bnf (VAR s)   = T
bnf (APP t u) = ¬ is_lam t ∧ bnf t ∧ bnf u
bnf (LAM v t) = bnf t
```

Another motivating example

Referring to the bound variable is the easiest way to express η-normal form:

```
enf (CON k) = T
enf (VAR s) = T
enf (APP t u) = enf t ∧ enf u
enf (LAM v t) = enf t ∧
                (is_app t ∧ rand t = VAR v ⇒
                 v ∈ FV (rator t))
```

$((\lambda x.\ M\,x) \to_\eta M$ if $x \notin \mathtt{FV}(M))$

Substitutions *vs.* permutations

- α-equivalence often expressed in terms of substitution:

$$(\lambda x.\ M) \equiv_\alpha (\lambda y.\ M[x \mapsto y])$$

(where $y \notin \texttt{FV}(M)$)

- But substitutions are awful to work with
  - Theorems typically hedged by side-conditions on freshness of variables, e.g., Barendregt's Lemma 2.1.16:

$$x \neq y \land x \notin \texttt{FV}(L) \Rightarrow$$
$$(M[x \mapsto N])[y \mapsto L] = (M[y \mapsto L])[x \mapsto N[y \mapsto L]]$$

Permutations

- Pitts & Gabbay suggest permutations a better choice than substitutions
- $(x\, y) \cdot M$ represents the action of swapping names x and y throughout M
- If $y \notin \mathtt{FV}(M)$, then $(\lambda x.\ M) \equiv_\alpha (\lambda y.\ ((x\, y) \cdot M))$

Permutations

- Pitts & Gabbay suggest permutations a better choice than substitutions
- $(x\,y) \cdot M$ represents the action of swapping names x and y throughout M
- If $y \notin \mathtt{FV}(M)$, then $(\lambda x.\ M) \equiv_\alpha (\lambda y.\ ((x\,y) \cdot M))$
- And permutations have great properties

The wonderful properties of permutations

- Permutations can cancel out

$$(x\,y) \cdot ((x\,y) \cdot M) = M$$

- Permutations commute with just about everything
  - Themselves:

$$(x\,y) \cdot ((u\,v) \cdot M) = (((x\,y) \cdot u)\,((x\,y) \cdot v)) \cdot ((x\,y) \cdot M)$$

  - and substitutions:

$$(x\,y) \cdot (N[v \mapsto M]) = ((x\,y) \cdot N)[((x\,y) \cdot v) \mapsto ((x\,y) \cdot M)]$$

- And these equations are side-condition free!

Permutations—they're great

- One last property of permutation:

$$(x\,y) \cdot (\lambda v.\ M) = (\lambda((x\,y) \cdot v).\ ((x\,y) \cdot M))$$

Permutations—they're great

- One last property of permutation:

$$(x\,y)\cdot(\lambda v.\ M) = (\lambda((x\,y)\cdot v).\ ((x\,y)\cdot M))$$

- And permutation on λ-terms can be defined using the Gordon-Melham recursion theorem.

Getting from Bad to Good—I

Have access to two function-terms in the LAM-clause of Bad. One is

$$(\lambda y.\ h\ \ (t[v \mapsto \mathtt{VAR}(y)]))$$

Can apply both functions to a "fresh" variable z. The above turns into

- $h\ \ (t[v \mapsto \mathtt{VAR}(z)])$

Getting from Bad to Good—I

Have access to two function-terms in the LAM-clause of Bad. One is

$$(\lambda y.\ h\ \ (t[v \mapsto \mathtt{VAR}(y)]))$$

Can apply both functions to a "fresh" variable z. The above turns into

- $h\ \ (t[v \mapsto \mathtt{VAR}(z)])$; into
- $h\ \ ((z\ v) \cdot t)$

Getting from Bad to Good—I

Have access to two function-terms in the LAM-clause of Bad. One is

$$(\lambda y.\ h\ (t[v \mapsto \mathtt{VAR}(y)]))$$

Can apply both functions to a "fresh" variable z. The above turns into

- $h\ (t[v \mapsto \mathtt{VAR}(z)])$; into
- $h\ ((z\ v) \cdot t)$

Similarly, $(\lambda y.\ t[v \mapsto \mathtt{VAR}(y)])$ turns into $(z\ v) \cdot t$

Getting from Bad to Good—II

LAM-clause has become

$$h\ (\mathtt{LAM}\ \ v\ \ t) =$$
$$\quad \text{let } z = \langle \text{a ``fresh'' name} \rangle \text{ in}$$
$$\quad\quad lam\ \ (h\ \ ((z\ v) \cdot t))\ \ ((z\ v) \cdot v)\ \ ((z\ v) \cdot t)$$

Getting from Bad to Good—II

LAM-clause has become

$$h\ (\texttt{LAM}\ \ v\ \ t) = \\ \quad \text{let } z = \langle \text{a ``fresh'' name} \rangle \text{ in} \\ \quad\quad lam\ (h\ ((z\,v)\cdot t))\ ((z\,v)\cdot v)\ ((z\,v)\cdot t)$$

- By induction, $h\ ((x\,y)\cdot t) = (x\,y)\cdot(h\ \ t)$

Getting from Bad to Good—II

`LAM`-clause has become

$$
\begin{array}{l}
h\ (\texttt{LAM}\ \ v\ \ t) = \\
\quad \text{let } z = \langle \text{a “fresh” name} \rangle \ \text{ in} \\
\quad\quad \mathit{lam}\ \ (\underset{\dots\dots\dots}{(z\,v)\cdot(h\ \ t)})\ \ ((z\,v)\cdot v)\ \ ((z\,v)\cdot t)
\end{array}
$$

- By induction, $h\ \ ((x\,y)\cdot t) = (x\,y)\cdot(h\ \ t)$

Getting from Bad to Good—II

LAM-clause has become

$$h\ (\mathtt{LAM}\ \ v\ \ t) = $$
$$\quad \text{let } z = \langle \text{a “fresh” name} \rangle \ \text{ in}$$
$$\qquad lam\ \ ((z\, v) \cdot (h\ \ t))\ \ ((z\, v) \cdot v)\ \ ((z\, v) \cdot t)$$

- By induction, $h\ \ ((x\, y) \cdot t) = (x\, y) \cdot (h\ \ t)$
- By side-condition, permutations commute with *lam*

Getting from Bad to Good—II

LAM-clause has become

$$
\begin{array}{l}
h\ (\mathtt{LAM}\ \ v\ \ t) = \\
\qquad \text{let } z = \langle \text{a ``fresh'' name} \rangle \text{ in} \\
\qquad\quad (z\,v) \cdot (\mathit{lam}\ (h\ \ t)\ \ v\ \ t)
\end{array}
$$

- By induction, $h\ ((x\,y)\cdot t) = (x\,y)\cdot (h\ \ t)$
- By side-condition, permutations commute with *lam*

Getting from Bad to Good—II

LAM-clause has become

$$h\ (\mathtt{LAM}\ v\ t) = \\ \quad \text{let } z = \langle\text{a ``fresh'' name}\rangle \text{ in} \\ \quad\quad (z\ v) \cdot (lam\ (h\ t)\ v\ t)$$

- By induction, $h\ ((x\ y) \cdot t) = (x\ y) \cdot (h\ t)$
- By side-condition, permutations commute with *lam*
- If z and v don't occur in M, then $(z\ v) \cdot M = M$.
  By side-condition, *lam* and h don't produce results with extra free names, so
  - z is not in $(lam\ \ldots)$; and
  - v is not in $\mathtt{FV}(\mathtt{LAM}\ v\ t)$, so v is not in $(lam\ \ldots)$ either

Getting from Bad to Good—II

LAM-clause has become

$$
\begin{array}{l}
h\ (\mathtt{LAM}\ \ v\ \ t) = \\
\qquad \text{let } z = \langle \text{a ``fresh'' name} \rangle \text{ in} \\
\qquad\quad lam\ \ (h\ \ t)\ \ v\ \ t
\end{array}
$$

- By induction, $h\ ((x\,y) \cdot t) = (x\,y) \cdot (h\ \ t)$
- By side-condition, permutations commute with *lam*
- If z and v don't occur in M, then $(z\,v) \cdot M = M$.
  By side-condition, *lam* and h don't produce results with extra free names, so
  - z is not in $(lam\ \ \ldots)$; and
  - v is not in $\mathtt{FV}(\mathtt{LAM}\ \ v\ \ t)$, so v is not in $(lam\ \ \ldots)$ either

Getting from Bad to Good—II

LAM-clause has become

$$h\ (\mathtt{LAM}\ \ v\ \ t) =$$

$$lam\ \ (h\ \ t)\ \ v\ \ t$$

- By induction, $h\ \ ((x\,y) \cdot t) = (x\,y) \cdot (h\ \ t)$
- By side-condition, permutations commute with *lam*
- If z and v don't occur in M, then $(z\,v) \cdot M = M$.
  By side-condition, *lam* and h don't produce results with extra free names, so
  - z is not in $(lam\ \ \ldots)$; and
  - v is not in $\mathtt{FV}(\mathtt{LAM}\ \ v\ \ t)$, so v is not in $(lam\ \ \ldots)$ either
- let $z = \ldots$ in has empty scope

From Bad to Good—summary

- Two additional properties of h:
  - $h\ ((x\,y)\cdot t) = (x\,y)\cdot(h\ \ t)$
  - $\mathtt{FV}(h\ \ t) \subseteq \mathtt{FV}(t)$

  Both proved by induction.

From Bad to Good—summary

- Two additional properties of h:
  - $h\ ((x\,y) \cdot t) = (x\,y) \cdot (h\ \ t)$
  - $\mathtt{FV}(h\ \ t) \subseteq \mathtt{FV}(t)$

  Both proved by induction.
- Side-conditions embody these restrictions for *lam*, *app*, *con* and *var*.

From Bad to Good—summary

- Two additional properties of h:
  - $h\ ((x\,y)\cdot t) = (x\,y)\cdot(h\ \ t)$
  - $\mathtt{FV}(h\ \ t) \subseteq \mathtt{FV}(t)$

  Both proved by induction.
- Side-conditions embody these restrictions for *lam*, *app*, *con* and *var*.
- Result type must support notion of permutation and FV constant (subject to characterising constraints)

Parameters

- Remember the foldl example: the result was a function
  (foldl $[\,] = (\lambda f\ x.\ x)$)
- The new recursion theorem places permutation and FV constraints on each "helper" (*lam*, *app* &c.):
  - Permutation for functions is easy (given permutation actions for domain and range)
  - Constrained generation of free variables *is* a problem.
- FV constraint for *lam* is

$$\mathtt{FV}(t') \subseteq \mathtt{FV}(t) \;\Rightarrow\; \mathtt{FV}(lam\ \ t'\ \ v\ \ t) \subseteq \mathtt{FV}(\mathtt{LAM}\ \ v\ \ t)$$

What are the "free variables" of a function (of type $\mathtt{term} \to \mathtt{term}$, say)?

Parameters (continued)

- Rather than force functions to support notion of free variables, make parameter explicit:
  - When no (interesting) parameters, original recursion theorem is derivable by setting parameter type to `unit`
  - Multiple parameters can be combined into one tuple
- Free variable constraint for `LAM`-clause becomes:

$$\begin{array}{l} \mathtt{FV}(t') \subseteq \mathtt{FV}(t) \quad \Rightarrow \\ \quad \mathtt{FV}(\mathit{lam}\ \ t'\ \ v\ \ t\ \ p) \subseteq \mathtt{FV}(p) \cup \mathtt{FV}(\mathtt{LAM}\ \ v\ \ t) \end{array}$$

- Previously

$$(z\ v) \cdot \mathit{lam} \ldots = \mathit{lam} \ldots$$

  because $v \notin \mathtt{FV}(\mathit{lam} \ldots)$ and z fresh
- Now also need $v \notin \mathtt{FV}(p)$ and $\forall p.\ \ \mathit{finite}(\mathtt{FV}(p))$

Parameter restrictions

- Parameter restrictions lead to side-conditions on equations
- For example, substitution's LAM-clause might be

$$sub\ \ M\ \ u\ \ (\mathtt{LAM}\ \ v\ \ t) = \mathtt{LAM}\ \ v\ \ (sub\ \ M\ \ u\ \ t)$$

- To have this work, v must avoid the free variables of the parameters:

$$\begin{array}{l} v \notin \mathtt{FV}(M) \land v \neq u \;\Rightarrow \\ \quad sub\ \ M\ \ u\ \ (\mathtt{LAM}\ \ v\ \ t) = \mathtt{LAM}\ \ v\ \ (sub\ \ M\ \ u\ \ t) \end{array}$$

Notes on the implementation

On top of usual formula manipulation, need

- An internal database, suggesting permutation and `FV` functions for range and parameter types
- Ability to try multiple options
  - Always try "null" permutation-`FV` option
- Ability to discharge side-conditions

Extensions

- Handle multiple domain types
- Handle parameters automatically
- (Harder) Automatically derive recursion theorem for new types

Conclusions

- It *is* possible to define functions in a *natural* style over a type of α-equivalent terms

Conclusions

- It *is* possible to define functions in a *natural* style over a type of α-equivalent terms
    - . . . and to do this in classical HOL

Conclusions

- It *is* possible to define functions in a *natural* style over a type of α-equivalent terms
  - ...and to do this in classical HOL
- My recursion theorem embodies the fact of this possibility

Conclusions

- It *is* possible to define functions in a *natural* style over a type of α-equivalent terms
  - ...and to do this in classical HOL
- My recursion theorem embodies the fact of this possibility
- The side-conditions enforce the reasonableness of possible definitions