# Defining types with binders in HOL

Michael Norrish
Michael.Norrish@nicta.com.au

National ICT Australia

24 April 2005

# Outline

Introduction
- Type definition, traditionally
- Dealing with $\alpha$-equivalence
- Summary

Model Choices
- Untyped $\lambda$-calculus model
- Weak HOAS type as model
- Quotient of algebraic type as model

Conclusions

# 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 1: type definition, for types with binders.

# Inductive Types

- User specifies the desired type by giving some sort of algebraic signature.
- For example, lists:

$$\alpha\ \mathsf{list} = \mathsf{nil} \ \mid\ \mathsf{cons}(\alpha, \alpha\ \mathsf{list})$$

- For example, trees:

$$\alpha\ \mathsf{tree} = \mathsf{Lf} \ \mid\ \mathsf{Node}(\alpha, \alpha\ \mathsf{tree}, \alpha\ \mathsf{tree})$$

- System responds by defining the type, with new constants corresponding to the desired "constructors", and a *recursion theorem*.

# 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 $t$

# Recursion Theorems, Generally

The basic desirable form for a recursion theorem is

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

Where

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

# Specifying Types with Binders

Assume from the outset that we are working with quotients of algebraic types (rather than co-algebraic ones).

Need to be able to specify

- which constructors are binders
- the unit of binding
- which units are bound by which constructors

Good notation for many variations still to be devised.

# Example Types

Untyped $\lambda$-calculus: Constructed by `LAM`, `VAR`, `APP`. `LAM` is a binder over its first argument (a string). Arguments to `VAR` are potentially bound.

$$(\lambda x.\ x\,y) \equiv_\alpha (\lambda z.\ z\,y)$$

Typed $\lambda$-calculus *à la* HOL: Constructed by `LAM`, `VAR`, `APP`, `CON`. `LAM` is a binder over `string`-`type` pairs. Arguments to `VAR` are potentially bound.

$$(\lambda(x:\alpha).\ (x:\alpha\rightarrow\alpha)\,(x:\alpha)) \ \equiv_\alpha \ (\lambda(y:\alpha).\ (x:\alpha\rightarrow\alpha)\,(y:\alpha))$$

Note: though the names ($x$ and $y$) are allowed to vary, and the type ($\alpha$) is not, the type still helps determine what is bound.

# More Examples

Barendregt's $\Lambda'^*$ (weighted vars): Constructed by `LAM`, `LAM`$_n$, `APP`, `VAR`. `LAM` and `LAM`$_n$ are binders over strings. String arguments to `VAR` are potentially bound, weight arguments are not.

$$(\lambda_3 x.\ x^4\, y^1)\, z^5 \equiv_\alpha (\lambda_3 z.\ z^4\, y^1)\, z^5 \qquad (\rightarrow_\beta z^5\, y^1)$$

System F: Two types (one of types, one of terms). Types constructed by $\forall$, `tyVAR`, $\rightarrow$. $\forall$ binds strings under `tyVAR`. Terms constructed by `LAM`, `APP`, `tyAPP`, `VAR` and $\Lambda$. `LAM` binds over strings under `VAR`, $\Lambda$ binds over strings under `tyVAR` (under `VAR`).

$$(\Lambda\alpha.\ \lambda x : (\forall\beta.\ \alpha \rightarrow \beta).\ x) \equiv_\alpha (\Lambda\beta.\ \lambda y : (\forall\gamma.\ \beta \rightarrow \gamma).\ y)$$

# Recursion Theorems for Types with Binders

- Recursion theorems in a 'natural' style are possible for types with binders
- 'Natural' means that the recursion theorems
  - look like theorems for normal algebraic types,
  - allow functions to refer to bound variable names
  - automatically include necessary side conditions on name-avoidance:

$$v \neq x \wedge v \notin \mathtt{FV}(M) \;\Rightarrow\; (\mathtt{LAM}\;\; v\;\; M)[x \mapsto N] = \mathtt{LAM}\;\; v\;\; (M[x \mapsto N])$$

- I proved the recursion theorem for untyped $\lambda$-calculus by hand—we need an automatic tool

# The Type Definition Problem Once More

Given: a specification of type(s) with binders (in a notation yet to be devised)

Prove: (1) the type exists, defining constructors etc.

Prove: (2) the type's recursion theorem

(Have this for the untyped $\lambda$-calculus.)

# Probable Shape of Destination

I expect to define first order types:

- total constructors
- no exotic terms
- classical, simply-typed, higher-order logic setting

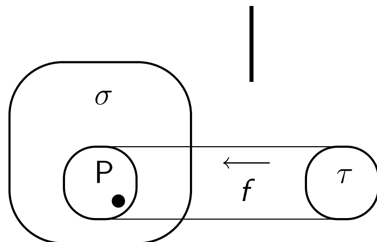
This choice looks promising because of work done so far:

- Mechanised proofs
- A function definition principle for untyped $\lambda$-calculus

# Proving Types Exist in HOL

Rule of type definition (introducing the type $\tau$ to name the set P):

$$\frac{\vdash \exists x{:}\sigma.\, \mathrm{P}\ x}{\begin{array}{l}\vdash \exists f{:}\ \tau \to \sigma. \\ \qquad \forall a_1\ a_2.\, (f\ a_1 = f\ a_2) \Rightarrow (a_1 = a_2) \wedge \\ \qquad \forall\, r.\, \mathrm{P}\ r = (\exists\, a.\, r = f\ a)\end{array}}$$



# After Type Existence Comes Recursion Theorem

It is not enough to prove the existence of the new type.

A type is crippled for logical reasoning if it doesn't have a recursion theorem. (Having to define functions as relations is *not* an answer.)

We know what the recursion theorems for types with binders should look like.

The question is: *which model makes it easiest to get that recursion theorem out?*

# Possible Models for Types with Binders

Four possibilities (in order of declining appeal):

- The existing untyped $\lambda$-calculus type
- A quotient over the corresponding algebraic type
- A subset of a weak HOAS type
- A type using de Bruijn indices

The de Bruijn solution lies behind the construction (by Gordon & Melham) of the existing untyped $\lambda$-calculus type.

(It's sufficiently scary that I won't explore it further.)

# The Untyped $\lambda$-calculus Model

Track record:

- Used it as the basis for the definition of
  - Barendregt's $\Lambda'$ type
  - the type of $\mathsf{F}_{<:}$ types (for PoplMark)
- But didn't prove recursion theorems for either type

Advantage:

- Model type + recursion theorem exist already
- Similar to (efficient) HOL type definition technology

Disadvantage:

- May not have enough flexibility to allow varying binding flavours

# The Polymorphic `CON` Constructor

- The untyped $\lambda$-calculus type includes a `CON` constructor.
- Thus, `term` is actually $(\alpha)$`term`, with `CON` : $\alpha \rightarrow \alpha$ `term`
- The polymorphism makes it easy to model new types that refer to other, existing types (numbers, strings &c)

# Defining $\Lambda'$—1

$$
\begin{array}{rcll}
\Lambda' & ::= & v & \text{(variable)} \\
 & | & \Lambda' \, \Lambda' & \text{(application)} \\
 & | & (\lambda v .\, \Lambda') & \text{(abstraction)} \\
 & | & (\lambda_n \, v .\, \Lambda') \, \Lambda' & \text{(labelled redex)}
\end{array}
$$

- this is a simple type that is very close to the model
- there are four constructors, two of which are binders.
- the labelled redex form calls for a constructor of four arguments: label ($\in \mathbb{N}$), bound variable, body of abstraction, right hand argument to redex

# Defining $\Lambda'$—2

Inductively specify subset of $\Lambda$:

$$\frac{}{\mathsf{ok}(v)} \qquad \frac{\mathsf{ok}(M) \quad \mathsf{ok}(N)}{\mathsf{ok}(M\ N)}$$

$$\frac{\mathsf{ok}(M)}{\mathsf{ok}(\lambda v.\ M)} \qquad \frac{\mathsf{ok}(M) \quad \mathsf{ok}(N)}{\mathsf{ok}((\mathtt{CON}(n))\ (\lambda v.\ M)\ N)}$$

Use type definition principle to create copy of this subset as new type, $\Lambda'$, with $f$ injective from $\Lambda'$ to $\Lambda$.

Define constructors. For example,

$$\begin{array}{lcl} \mathtt{APP}_{\Lambda'}\ M\ N & = & f^{-1}((f(M))\ (f(N))) \\ \mathtt{LAMn}\ n\ v\ M\ N & = & f^{-1}((\mathtt{CON}(n))\ (\lambda v.\ f(M))\ (f(N))) \end{array}$$

# Functions Over $\Lambda'$?

- Substitution in $\Lambda'$ corresponds to substitution in the representation
- Term size in $\Lambda'$ plausibly corresponds to term size in the representation
- I defined the $\varphi$ function (reduces all labelled redexes) by well-founded recursion (ugh!)

I *think* it should be possible to define a generic recursion theorem, but it would be hard....

# A better way?

Recall

$$\texttt{LAMn}\ n\ v\ M\ N = f^{-1}((\texttt{CON}(n))\ (\lambda v.\ f(M))\ (f(N)))$$

In $\Lambda'$, one constructor, LAMn is represented by four constructors in $\Lambda$ (two applications, one CON and one abstraction).

A straightforward recursion in $\Lambda'$ would have to be modelled by a very convoluted recursion in $\Lambda$.

But if the underlying type ($\alpha\,\texttt{term}$) had one constructor:

$$\texttt{C} : \alpha \times \texttt{string}^* \times (\texttt{string} \times \alpha\,\texttt{term})^* \times (\alpha\,\texttt{term})^* \rightarrow \alpha\,\texttt{term}$$

all of $\Lambda'$'s constructors could be modelled by C.

# More on the Super-Constructor, C

$$\boxed{\mathtt{C} : \alpha \times \mathtt{string}^* \times (\mathtt{string} \times \alpha\,\mathtt{term})^* \times (\alpha\,\mathtt{term})^* \to \alpha\,\mathtt{term}}$$

For $\Lambda'$, use $\alpha = 1 + \mathbb{N}$, and

$$\begin{array}{lcl}
f(\mathtt{VAR}_{\Lambda'}\ s) & = & \mathtt{C}(\mathsf{inl}\langle\rangle, [s], \epsilon, \epsilon) \\
f(\mathtt{APP}_{\Lambda'}\ M\ N) & = & \mathtt{C}(\mathsf{inl}\langle\rangle, \epsilon, \epsilon, [f(M), f(N)]) \\
f(\mathtt{LAM}_{\Lambda'}\ v\ M) & = & \mathtt{C}(\mathsf{inl}\langle\rangle, \epsilon, [(v, f(M))], \epsilon) \\
f(\mathtt{LAMn}\ n\ v\ M\ N) & = & \mathtt{C}(\mathsf{inr}(n), \epsilon, [(v, f(M))], [f(N)])
\end{array}$$

with $f : \Lambda' \to \Lambda$

- Recursion theorem should drop out much more readily.
- Allows any finitely branching structure.
- Enforces strings (and only strings) as bound objects.

# Untyped $\lambda$-calculus Model—Summary

- Use existing type with binders (probably type with "super-combinator")
- Encode new type as inductive subset of existing type (easy)
- Lift recursion theorem (easy; see existing HOL datatype package)

- Super-combinator type and recursion theorem still have to be established manually

# The Weak HOAS type as model

Track record:

- None: (stupid) implementation reasons make it a pain to do in HOL

Advantage:

- Can use existing proof to generate recursion theorem
- Should allow variation in binding structure

Disadvantage:

- Requires definition of some supporting constants (including substitution) over the HOAS type

# Using (Weak) Higher-Order Abstract Syntax

- Pretend we are defining $\Lambda$ for the first time
- The following are constructors for a reasonable algebraic type

$$\begin{array}{lcl} \texttt{VAR} & : & \texttt{string} \to \alpha\,\texttt{term} \\ \texttt{APP} & : & \alpha\,\texttt{term} \to \alpha\,\texttt{term} \to \alpha\,\texttt{term} \\ \texttt{LAM'} & : & (\texttt{string} \to \alpha\,\texttt{term}) \to \alpha\,\texttt{term} \end{array}$$

- It's not great as a model for $\Lambda$ because of the exotic terms possible in `LAM'`'s function space.
- But the exotic terms can be avoided by restricting the functions allowed:

$$\mathsf{ok}(f) \;\equiv\; \exists v\, t.\; f = (\lambda u.\; t[v \mapsto u])$$

(ignoring the fact that we don't necessarily know how to define substitution on this type properly)

# The Recursion Theorem for the HOAS Model

The abstraction clause of the recursion theorem for the HOAS type will look like:

$$h(\texttt{LAM'}\ f) = lam(\lambda s.\ h(f(s)))$$

for some $lam : (\texttt{string} \to \alpha) \to \alpha$.

If $f$ has been restricted to avoid exoticness, then it will be of the form $(\lambda u.\ t[v \mapsto u])$, and the above becomes

$$h(\texttt{LAM'}\ f) = lam(\lambda s.\ h(t[v \mapsto s]))$$

with `LAM'` $f$ the "same as" `LAM v t`, the above is exactly the `LAM`-clause from the Gordon-Melham recursion theorem.

## The GM Recursion Theorem

- The GM recursion theorem has this awkward

$$h\ (\texttt{LAM}\ v\ t) = lam\ (\lambda s.\ h\ (t[v \mapsto s]))$$

  clause
- But this clause can be turned into the much more reasonable

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

  if you pick up side-conditions on *lam* (is a finitely-supported function &c)
- This proof could be replayed for each new type.

# Weak HOAS Type as Model—Summary

- Define the new type in “HOAS” style (easy, in Isabelle)
- Carve out non-exotic subset, lose higher-order nature
  - Use `LAM` $v\ \ t$ = `LAM'` $(\lambda s.\ t[v \mapsto s])$
  - Assume we have substitution already defined on HOAS type
- Gain Gordon-Melham style recursion theorem (easy)
- Transform this into nice version, replaying TPHOLs’04 proof (easyish)
- Done!

# Quotienting

Track Record:

- People have done this by hand a number of times
    - Peter Homeier has done it all (recursion theorem included) in HOL

Advantage:

- Conceptually simple, allows variation in binding structure

Disadvantage

- Quotienting and associated proof may make this inefficient

# Using Quotienting to Define a New Type

- Again, pretend we are defining $\Lambda$ for the first time
- Define obvious algebraic type:

$$
\begin{array}{lcl}
\texttt{VAR} & : & \texttt{string} \rightarrow \alpha\,\texttt{term} \\
\texttt{APP} & : & \alpha\,\texttt{term} \rightarrow \alpha\,\texttt{term} \rightarrow \alpha\,\texttt{term} \\
\texttt{LAM} & : & \texttt{string} \rightarrow \alpha\,\texttt{term} \rightarrow \alpha\,\texttt{term}
\end{array}
$$

- Define $\alpha$-equivalence, free variables, name permutation
- Take quotient
- Prove recursion theorem for quotient
  - (This is the hard bit)

# Defining Algebraic Level Functions

- Quotient will be with respect to notion of $\alpha$-equivalence
- So, must define $\alpha$-equivalence on algebraic terms
- Recursion theorem proof also requires name permutation on algebraic terms:

$$\begin{array}{rcl} (x\,y)\cdot(\texttt{VAR}\;\;s) & = & \texttt{VAR}\;(\texttt{swapstr}\;\;x\;\;y\;\;s) \\ (x\,y)\cdot(\texttt{APP}\;\;M\;\;N) & = & \texttt{APP}\;((x\,y)\cdot M)\;((x\,y)\cdot N) \\ (x\,y)\cdot(\texttt{LAM}\;\;v\;\;M) & = & \texttt{LAM}\;(\texttt{swapstr}\;\;x\;\;y\;\;v)\;((x\,y)\cdot M) \end{array}$$

- Free variables is also primitive recursive

# Defining $\alpha$-equivalence

Given notion of permutation and swapping, use following inductive relation:

$$\frac{}{\mathtt{VAR}(s) \equiv_\alpha \mathtt{VAR}(s)} \qquad \frac{M_1 \equiv_\alpha M_2 \quad N_1 \equiv_\alpha N_2}{\mathtt{APP}\ M_1\ N_1 \equiv_\alpha \mathtt{APP}\ M_2\ N_2}$$

$$\frac{M \equiv_\alpha M'}{\mathtt{LAM}\ v\ M \equiv_\alpha \mathtt{LAM}\ v\ M'}$$

$$\frac{u \neq v \quad u \notin \mathsf{fv}(N) \quad M \equiv_\alpha (u\,v) \cdot N}{\mathtt{LAM}\ u\ M \equiv_\alpha \mathtt{LAM}\ v\ N}$$

# Proving Recursion Theorem from Scratch

- Andy Pitts has shown me how to do this (now partially mechanised)
- Result is a recursion theorem that is "natural" (looks similar to earlier TPHOLs version)

Proof:

- Use recursion theorem of underlying algebraic type to define a new function at that level (magic here)
- Show that this function "respects" $\alpha$-equivalence, that $(t_1 \equiv_\alpha t_2) \Rightarrow f(t_1) = f(t_2)$
- Lift to quotiented level

- Generic proof gives rise to recursion theorem

# Lots of Nominal Theory Required

- This approach requires a great deal of supporting nominal theory (notions of permutation, support &c).
- In pure, simply-typed HOL, these notions become excessively parameterised:
  - For example, permutation on function spaces requires explicit reference to permutation actions on domain and range:

$$\mathtt{fnpm}\ d\ r\ \pi\ f = \lambda x.\ r\ \pi\ (f\ (d\ \pi^{-1}\ x))$$

- Isabelle-style overloaded constants (and axiomatic type classes) might make this underlying theory more palatable
- Some (probably unnecessary) flexibility lost

# Quotienting—Summary

- Generate fresh algebraic type for each new type (easy)
- Define small suite of nominal theory constants on this type (easy)
- Quotient (easy)
- Prove recursion theorem (involved)

- Done!

# The Efficiency Contest

And the winner is:

# The Efficiency Contest

And the winner is:

*Existing type with binders + recursion theorem*

# The Efficiency Contest

And the winner is:

*Existing type with binders + recursion theorem*

- Only one new type (the type the user is interested in) is defined
- The transfer of the recursion theorem from the existing type is straightforward (no inductions required)

(Might use quotienting approach to manually generate representation type and its recursion theorem.)

# The Flexibility Contest

And the winner is:

# The Flexibility Contest

And the winner is:

*Quotienting*

(Weak HOAS gets honourable mention but looks logically fraught.)

# The Flexibility Contest

And the winner is:

*Quotienting*

(Weak HOAS gets honourable mention but looks logically fraught.)

- The system builds everything from the ground up, so any manner of binding weirdness should be possible

# Future Work

The Happy HOL Hacker in me loves this stuff!

- Am working on various examples by hand
- Want to attack full PoplMark problem
- Must design notation for binding flavours

General, well-tested technology for definition of types with binders in HOL is probably at least a year away.

General Principle: Implementations need to support proofs of real theorems. All of PoplMark might be a bit much, but Church-Rosser isn't enough.