Mechanising and Using Nominal Recursion Principles

Michael Norrish
Michael.Norrish@nicta.com.au
National ICT Australia
6 February 2007
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 . . . )
Find your type:
1. When proving Fermat’s Last Theorem, HOL provides the type of natural numbers ($\mathbb{N}$).
2. When verifying a hardware design, the (new) type for the system state-space needs to be specified (tuple of registers, memory . . .)

Define functions over the type:
3. Define $\text{gcd}$ over $\mathbb{N}^2$
4. Define a transition relation over the hardware state-space
Mechanised 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 $\text{gcd}$ over $\mathbb{N}^2$
   - Define a transition relation over the hardware state-space

3. Prove theorems!
   - . . .
   - Prove safety, liveness . . .
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 . . . )

Define functions over the type:
- Define $\text{gcd}$ over $\mathbb{N}^2$
- Define a transition relation over the hardware state-space

Prove theorems!
- . . .
- Prove safety, liveness . . .

This talk is about step 2: function definition
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 . . . )

Define functions over the type:
- Define $\text{gcd}$ over $\mathbb{N}^2$
- Define a transition relation over the hardware state-space

Prove theorems!
- . . .
- Prove safety, liveness . . .

This talk is about step 2: function definition (for binders, in HOL)
Given the type of lists, want to define a (primitive) recursive function such as \texttt{foldl}, with definition

\[
\text{foldl } f \times [] = x \\
\text{foldl } f \times (e :: t) = \text{foldl } f (f(e, x)) t
\]

How can such a definition be allowed?
Recursion theorems

For lists:

\[ \vdash \forall n \ c. \ \exists h. \]
\[ h \ [] = n \ \land \]
\[ \forall e \ t. \ h \ (e :: t) = c \ (h \ t) \ e \ t \]

- \( n \) 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 \(t\)

- Not quite the categorical initiality theorem.
Demonstrating the existence of foldl

Begin with the recursion theorem

\[ \vdash \forall n \ c. \ \exists h. \]
\[ h \ [ ] = n \land \]
\[ \forall e \ t. \ h (e :: t) = c (h t) e t \]
Demonstrating the existence of foldl

Begin with the recursion theorem

\[ \vdash \forall n \ c. \ \exists h. \]
\[ h \ [\ ] = n \ \land \]
\[ \forall e \ t. \ h \ (e :: t) = c \ (h \ t) \ e \ t \]

- Take \( n \) to be \((\lambda f \ x. \ x)\)
Begin with the recursion theorem

\[ \vdash \forall c. \exists h. \]
\[ h \; [] = (\lambda f \; x. \; x) \land \]
\[ \forall e \; t. \; h \; (e :: t) = c \; (h \; t) \; e \; t \]

- Take \( n \) to be \((\lambda f \; x. \; x)\)
Demonstrating the existence of foldl

Begin with the recursion theorem

\[ \forall c. \exists h. \]
\[ h \;[] = (\lambda f \; x. \; x) \land \]
\[ \forall e \; t. \; h \;(e :: t) = c \;(h \;t) \;e\;t \]

- 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

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

- 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

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

- Take \( n \) to be \((\lambda f \; x. \; x)\)
- Take \( c \) to be \((\lambda r \; e \; t \; f \; x. \; r \; f \; (f(e, x)))\)
- \( \beta \)-reduce
Demonstrating the existence of foldl

Begin with the recursion theorem

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

- Take \( n \) to be \((\lambda f \; x. \; x)\)
- Take \( c \) to be \((\lambda r \; e \; t \; f \; x. \; r \; f \; (f(e, x)))\)
- \( \beta \)-reduce
Demonstrating the existence of foldl

Begin with the recursion theorem

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

- Take \( n \) to be \((\lambda f \ x. \ x)\)
- Take \( c \) to be \((\lambda r \ e \ t \ f \ x. \ r \ f \ (f(e, x)))\)
- \( \beta \)-reduce
- Use extensionality to handle \( \lambda \)s 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

$$\vdash \forall \ldots f_i \ldots \exists h. \ldots \land \\forall \ldots x_j \ldots r_k. h \left( C_i(\ldots x_j, \ldots r_k) \right) = f_i \left( h \ r_k \right) \ldots x_j \ldots r_k \land \ldots$$

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 $(h \ r_k)$
The type representing the syntax of $\lambda$-terms will have constructors:

- **VAR**: $\text{string} \rightarrow \text{term}$
- **APP**: $\text{term} \rightarrow (\text{term} \rightarrow \text{term})$
- **LAM**: $\text{string} \rightarrow (\text{term} \rightarrow \text{term})$

Add $\alpha$-equivalence: “the choice of variable name doesn’t matter”:

- LAM $x \times x$ is “the same” as LAM $y \times y$
- On raw syntax, $\alpha$-equivalence ($\equiv_\alpha$) captures “the same”
- At level of interest, $\equiv_\alpha$ is just $=$
The recursion theorem for the type “should” have the LAM-clause:

\[ h (\text{LAM } v \ t) = \text{lm } v \ t \ (h \ t) \]

But this would allow unsound definition of

\[ \text{bogus } (\text{LAM } v \ t) = v \]

Side-conditions will be required!
Outline

1. Introduction
   - Function definition, traditionally
   - Problems with binders

2. Function definition with binders
   - Motivating examples
   - Permutations
   - Building a Type with Binders
   - Proving a Recursion Principle

3. Using a Recursion Principle

4. Conclusion
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:
    
    \[
    \begin{align*}
    \text{size} \ (\text{VAR} \ s) &= 1 \\
    \text{size} \ (\text{APP} \ t \ u) &= 1 + \text{size} \ t + \text{size} \ u \\
    \text{size} \ (\text{LAM} \ v \ t) &= 1 + \text{size} \ t
    \end{align*}
    \]
  
- Is a term in $\beta$-normal form:
  
  \[
  \begin{align*}
  \text{bnf} \ (\text{VAR} \ s) &= T \\
  \text{bnf} \ (\text{APP} \ t \ u) &= \neg \text{is\_lam} \ t \wedge \text{bnf} \ t \wedge \text{bnf} \ u \\
  \text{bnf} \ (\text{LAM} \ v \ t) &= \text{bnf} \ t
  \end{align*}
  \]
Another (simple) example

Referring to the bound variable is the easiest way to express $\eta$-normal form:

\begin{align*}
enf (\text{VAR } s) &= T \\
enf (\text{APP } t \ u) &= enf t \land enf u \\
enf (\text{LAM } v \ t) &= enf t \land \\
& \quad (\text{is_app } t \land \text{rand } t = \text{VAR } v \Rightarrow \\
& \quad \quad v \in FV (\text{rator } t))
\end{align*}

$$((\lambda x. \ M x) \rightarrow_\eta M \text{ if } x \not\in FV(M))$$
Substitutions vs. permutations

- $\alpha$-equivalence often expressed in terms of substitution:

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

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

- 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:

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

\[(x \, y) \cdot (\text{LAM } \nu \, M) = \text{LAM} \, ((x \, y) \cdot \nu) \, ((x \, y) \cdot M)\]

We want this to be true...
Defining the λ-Calculus—Barendregt

\[ x \in \Lambda \]
\[ M \in \Lambda \]
\[ (\lambda x. M) \in \Lambda \]
\[ M \in \Lambda \quad N \in \Lambda \]
\[ (M \, N) \in \Lambda \]

and

1. **Identify two terms if each can be transformed to the other by a renaming of its bound variables.**
2. **Consider a λ-term as a representative of its equivalence class**
3. **Interpret substitution \( M[x \mapsto N] \) as an operation on the equivalence classes of \( M \) and \( N \)**
Establishing the underlying algebraic type is easy:

```
Hol_datatype' preterm = var of string
           | app of preterm # preterm
           | lam of string # preterm
```

Defining the quotient is trickier (and diverges from Barendregt...)

What’s Wrong with Barendregt’s Approach

(And all the other classical presentations...) 

The standard approach is to quotient with respect to a notion of $\alpha$-equivalence on algebraic syntax.

The standard definition of $\alpha$-equivalence is in terms of substitution.

Capture-avoiding Substitution is awful to define over algebraic terms!
Permutation to the Rescue

Luckily, permutations are enough to define $\alpha$-equivalence.

...and permutation is easy to define.

Define:

\[
\begin{align*}
\text{perm } \pi (\text{var } v) &= \text{var } (\pi \cdot v) \\
\text{perm } \pi (\text{app } m \ n) &= \text{app } (\text{perm } \pi \ m) \ (\text{perm } \pi \ n) \\
\text{perm } \pi (\text{lam } v \ m) &= \text{lam } (\pi \cdot v) \ (\text{perm } \pi \ m)
\end{align*}
\]
This is **not** the notion of free variables

Define\`\`
\[ \text{allatoms (var } v) = \{v\} \]
\[ \text{allatoms (app } m \; n) = \text{allatoms } m \cup \text{allatoms } n \]
\[ \text{allatoms (lam } v \; m) = \text{allatoms } m \cup \{v\} \]
\`\`

Further, this notion **does** make sense because we’re still at the level of the algebraic, unquotiented type.
\(\alpha\)-Equivalence

An inductive relation, immediately mechanisable:

\[
\begin{align*}
\text{var } v &\equiv_\alpha \text{var } v \\
\text{app } m_1 &\equiv_\alpha m_2 & n_1 &\equiv_\alpha n_2 \\
\text{app } m_1 &\equiv_\alpha \text{app } m_2 & n_1 &\equiv_\alpha n_2
\end{align*}
\]

\[
\begin{align*}
z \notin \text{allatoms}(m_1) \cup \text{allatoms}(m_2) \cup \{v_1, v_2\} \\
\text{perm } (z \, v_1) &\equiv_\alpha \text{perm } (z \, v_2) \\
\text{lam } v_1 &\equiv_\alpha \text{lam } v_2
\end{align*}
\]

Proofs of reflexivity, symmetry and transitivity are reasonably straightforward.
Performing the Quotient

Notions that are to be lifted into the quotiented type need to be shown to respect $\alpha$-equivalence. [Cue Hard Words about substitution…]

For example, $fv$ (free variables)

$$m \equiv_\alpha n \Rightarrow fv\ m = fv\ n$$

In contrast, allatoms has no such result.

Preterm constructors ($var$, $app$ and $lam$) also need “respectfulness results”.
Quotienting Itself

- Well-understood technology
- Used in systems to construct $\mathbb{Z}$ and $\mathbb{R}$, for example
- Establishes
  - new type (term)
  - new functions on new type (FV, LAM, \ldots)
  - results about new functions on new types (FV’s characterisation, perm’s characterisation)

For example, we have ("for free")

\[
\begin{align*}
FV (\text{VAR } s) &= \{s\} \\
FV (\text{APP } M N) &= FV M \cup FV N \\
FV (\text{LAM } v M) &= FV M - \{v\}
\end{align*}
\]
In particular, we have (a lifted result from preterm)

$$\nu \notin \text{FV}(M) \implies \text{LAM } u \ M = \text{LAM } \nu \ (\text{perm } (u \ \nu) \ M)$$
Now We Have a Type of $\lambda$-Terms

In particular, we have (a lifted result from preterm)

$$\nu \notin \mathrm{FV}(M) \Rightarrow \quad \lambda u \ M = \lambda \nu \ (\mathrm{perm} \ (u \ \nu) \ M)$$

To use nominal syntax:

$$\nu \not\equiv M \Rightarrow \quad \lambda u \ M = \lambda \nu \ ((u \ \nu) \cdot M)$$
Our new type (term) does not have a recursion theorem.

The old type (preterm) had a recursion theorem, but it can’t lift.

We do want a recursion theorem, because we want to define substitution for term.
Outline

1. **Introduction**
   - Function definition, traditionally
   - Problems with binders

2. **Function definition with binders**
   - Motivating examples
   - Permutations
   - Building a Type with Binders
   - Proving a Recursion Principle

3. **Using a Recursion Principle**

4. **Conclusion**
We want to be able to define a function with characterising equations:

\[(\text{VAR } \nu)[x \mapsto N] = \text{if } x = \nu \text{ then } N \text{ else } \text{VAR } \nu\]

\[(\text{APP } M_1 \ M_2)[x \mapsto N] = \text{APP } (M_1[x \mapsto N]) \ (M_2[x \mapsto N])\]

\[(\text{LAM } \nu \ M)[x \mapsto N] = \text{LAM } \nu \ (M[x \mapsto N])\]

[as long as \(x \neq \nu, \nu \not\in \text{FV}(N)\)]
The following (sketch of the core idea of a) proof is:

- The “elegant one” of the two proofs in Pitts (JACM, 2006).
- **Not** the basis for the nominal-isabelle package
- **Probably** inadequate on weirder binding signatures
- Mechanised, but not automated, by me in HOL.
Basic idea:

- Prove a recursion principle that is respectful of $\alpha$-equivalence at the preterm level
- Lift this result through the quotienting process
Recursion for Preterms

Have (initiality, BTW) for type of preterm

\[ \exists f. \]

\[
\begin{align*}
    f \ (\text{var} \ v) &= vr \ v \\
    f \ (\text{app} \ m \ n) &= ap \ (f \ m) \ (f \ n) \\
    f \ (\text{lam} \ v \ m) &= lm \ v \ (f \ m)
\end{align*}
\]
Recursion for Preterms

Have (initiality, BTW) for type of preterm

\[ \exists f. \]

\[
f (\text{var } v) = vr v
\]
\[
f (\text{app } m n) = ap (f m) (f n)
\]
\[
f (\text{lam } v m) = lm v (f m)
\]

One approach to defining substitution at this preterm level is to actually define simultaneous substitution, so that a new substitution can be added to rename the bound variable away from possible clashes.

We can do something similar with permutations.
Recursion for Preterms (with permutations)

Accompany every recursion with an accumulating permutation:

$$\exists f.
\begin{align*}
  f (\text{var } v) \pi &= vr (\pi(v)) \\
  f (\text{app } m n) \pi &= ap (f m \pi) (f n \pi) \\
  f (\text{lam } v m) \pi &= ???
\end{align*}$$
Accompany every recursion with an accumulating permutation:

\[ \exists f. \]

\[
f \ (\text{var } v) \ \pi = vr \ (\pi(v))
\]

\[
f \ (\text{app } m \ n) \ \pi = ap \ (f \ m \ \pi) \ (f \ n \ \pi)
\]

\[
f \ (\text{lam } v \ m) \ \pi = ???
\]

In substitution on preterms case, \text{lam} clause might look like

\[
\text{ssub} \ (\text{lam } v \ m) \ \sigma = \text{lam } z \ (\text{ssub} \ m \ \sigma')
\]

where \( z \) is some fresh variable (deterministically chosen), and

\[
\sigma' = (v \mapsto z) :: \sigma
\]
Recursion for Preterms (with permutations)

So, analogously

\[ \exists f. \]

\[
\begin{align*}
    f (\text{var } v) \pi &= vr (\pi(v)) \\
    f (\text{app } m n) \pi &= ap (f m \pi) (f n \pi) \\
    f (\text{lam } v m) \pi &= \text{fresh} (\lambda z. \text{lm } z (f m (\pi \circ (z v))))
\end{align*}
\]

In substitution on preterms case, \text{lam} clause might look like

\[
\text{ssub} (\text{lam } v m) \sigma = \text{lam } z (\text{ssub } m \sigma')
\]

where \( z \) is some fresh variable (deterministically chosen), and

\[
\sigma' = (v \mapsto z) :: \sigma
\]
What is “fresh”?  

Type:

$$\text{fresh} : (\text{string} \rightarrow \alpha) \rightarrow \alpha$$

An expression

$$\text{fresh} \ (\lambda z. \ldots)$$

picks a name fresh for the argument (a function), and feeds it to the function.

If the function ($$\lambda z. \ldots$$) is well-behaved, then it will behave identically on all fresh inputs.

$$\text{Well-behaved}(f) \equiv \forall v. \ v \# f \Rightarrow v \# (f \ v)$$
(This is where all the real work is.)

- Prove that the support of the $f$ is the union of the supports for functions $vr$, $ap$, $lm$
- Prove $f \left( \pi_2 \cdot t \right) \pi_1 = f \left( t \left( \pi_1 \circ \pi_2 \right) \right)$
- Prove $t_1 \equiv_\alpha t_2 \Rightarrow f \left( t_1 \pi = f \left( t_2 \pi \right) \right)$

Lift!
Accumulated Side-conditions

The range of the function must be a nominal set. (Must support a notion of permutation.)
Accumulated Side-conditions

The range of the function must be a **nominal set**. (Must support a notion of permutation.)

Auxiliary functions $vr$, $ap$, and $lm$ must have **finite support**.
Accumulated Side-conditions

The range of the function must be a **nominal set**. (Must support a notion of permutation.)

Auxiliary functions $vr$, $ap$, and $lm$ must have **finite support**.

The $lm$ function must be **well-behaved**:

$$\forall v \, x. \ v \not\in A \Rightarrow v \not\in (lm \ v \ x)$$

where $A$ is the union of the supports of the $vr$, $ap$, and $lm$. 
\langle side conditions \rangle
\Rightarrow
\exists f.
\begin{align*}
f \ (\text{VAR } v) &= vr \ v \\
f \ (\text{APP } M \ N) &= ap \ (f \ M) \ (f \ N) \\
f \ (\text{LAM } v \ M) &= lm \ v \ (f \ M) \quad \text{[as long as } v \# A]\end{align*}

where $A$ is the union of the supports of the $vr$, $ap$, and $lm$. 
1 Introduction
   - Function definition, traditionally
   - Problems with binders

2 Function definition with binders
   - Motivating examples
   - Permutations
   - Building a Type with Binders
   - Proving a Recursion Principle

3 Using a Recursion Principle

4 Conclusion
Applying the Principle—Easy Cases

The first motivating examples (size, bnf, and enf) all had range types (\(\mathbb{N}\) and bool) where all values have empty support.

This makes the well-behavedness of the \(lm\) function easy

\[ \forall v \ x. \ v \not\in A \Rightarrow v \not\in (lm \ v \ x) \]

But even enf is not entirely trivial...
LAM-clause of enf’s desired definition:

\[
\text{enf (LAM } v \ t) = \text{enf } t \land \\
(\text{is_app } t \land \text{rand } t = \text{VAR } v \Rightarrow \\
v \in \text{FV (reator } t))
\]

The problem is that \( t \) occurs multiple times not wrapped in a (recursive) call to \( \text{enf} \).
We want to change

\[ lm : \text{string} \rightarrow \alpha \rightarrow \alpha \]

to

\[ lm : \text{string} \rightarrow \alpha \rightarrow \text{term} \rightarrow \alpha \]

Do this by instantiating the \( \alpha \) of original theorem to \( \alpha \times \text{term} \).

Now \( f \) is of type \( \text{term} \rightarrow (\alpha \times \text{term}) \)

Make \( f \) return its argument as the second component of the pair; auxiliaries refer to this when needing to use original value.

Finish by proving (by induction on \( t \)) that second component really is same as input.
When working with “labelled λ-calculus” ([Barendregt, §11]), have a new “labelled redex” form

\[(\lambda_i v. M)N\]

(Similar to “let \( v = N \) in \( M \)”, ignoring label \( i \).)

What should the well-behaved side-condition be for this form?
Well-behavedness for Redexes

Let \( lt \) be of type

\[
\text{string} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha
\]
Well-behavedness for Redexes

Let $lt$ be of type

$$\text{string} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

One candidate for well-behaved is

$$\forall v \ M \ N. \ v \not\in A \Rightarrow v \not\in (lt \ v \ M \ N)$$

but this is unreasonable because the $N$ term might contribute a $v$ of its own to the result.
Let \( lt \) be of type

\[
\text{string} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha
\]

Another candidate for well-behaved is

\[
\forall v \; M \; N. \; v \# A \land v \# N \Rightarrow v \# (lt \; v \; M \; N)
\]
Well-behavedness for Redexes

Let $lt$ be of type

$$\text{string} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

Another candidate for well-behaved is

$$\forall v \ M \ N. \ v \not\# A \land v \not\# N \implies v \not\# (lt \ v \ M \ N)$$

This requires the LET clause to look like

$$f \ (\text{LET} \ v \ M \ N) = lt \ v \ (f \ M) \ (f \ N) \quad [\text{as long as } v \not\# N, \ v \not\# A]$$

This is plausible, but ugly (why have to rename away from free variables in the unrelated $N$?)
Well-behavedness for Redexes

Let \( lt \) be of type

\[
\text{string} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha
\]

Another candidate for well-behaved is

\[
\forall v \ M. \ v \not\# A \Rightarrow v \not\# (lt \ v \ M)
\]

(Partial application of \( lt! \))
Well-behavedness for Redexes

Let $lt$ be of type

$$\text{string} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

Another candidate for well-behaved is

$$\forall v \ M. \ v \not\in A \implies v \not\in (lt \ v \ M)$$

(Partial application of $lt$!)

The LET clause looks “right”: 

$$f \ (\text{LET} \ v \ M \ N) = lt \ v \ (f \ M) \ (f \ N) \quad \text{[as long as } v \not\in A\text{]}$$

but calculating support for function spaces is painful.
Conclusions

- It *is* possible to define functions in a *natural* style over a type of \(\alpha\)-equivalent terms
Conclusions

- It *is* possible to define functions in a *natural* style over a type of \( \alpha \)-equivalent terms
  - \ldots and to do this in classical HOL
Conclusions

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

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

- It *is* possible to define functions in a *natural* style over a type of \( \alpha \)-equivalent terms
  - ...and to do this in classical HOL
- The recursion theorem embodies the fact of this possibility
- The side-conditions enforce the reasonableness of possible definitions
- Coming up with right expression of the side-conditions is still future work
It is possible to define functions in a *natural* style over a type of $\alpha$-equivalent terms
  - ...and to do this in classical HOL

The recursion theorem embodies the fact of this possibility

The side-conditions enforce the reasonableness of possible definitions

Coming up with right expression of the side-conditions is still future work
  - They need to be as easy to discharge as possible