Real World Binding Structures

Susmit Sarkar
with Peter Sewell and Francesco Zappa Nardelli
Paradigm Binding

- Single binders

\[ \text{exp} ::= X \]

\[ \mid \lambda X . \text{exp} \quad \text{bind } X \text{ in } \text{exp} \]

\[ \mid \text{exp} \ \text{exp}' \]
Paradigm Binding

- Single binders
  \[ exp ::= X \]
  \[ \mid \lambda X . exp \quad \text{bind } X \text{ in } exp \]
  \[ \mid exp \ exp' \]

- Lots of work on representations
  - deBruijn
  - HOAS
  - Locally nameless
  - Nominal
  - ...
How about: Patterns?

- Many binders

\[
\text{let } (x, y) = z \text{ in } x \ y
\]
How about: Patterns?

- Many binders

```
let ( x , y ) = z in x y
```

```
exp ::= X
| ( exp , exp' )
| let pat = exp in exp' bind b(pat) in exp'

pat ::= X
| _
| ( pat , pat' )
```
How about: Patterns?

- Many binders

\[
\text{let } (x, y) = z \text{ in } x \ y
\]

\[
\begin{align*}
\text{exp} & \ ::= \ X \\
& \quad | \ (\ \text{exp}, \ \text{exp}') \\
& \quad | \ \text{let} \ \text{pat} = \ \text{exp} \ \text{in} \ \text{exp}' \\
& \quad | \ \text{bind} \ b(\text{pat}) \ \text{in} \ \text{exp}' \\
\text{pat} & \ ::= \ X \\
& \quad | \ _ \\
& \quad | \ (\ \text{pat}, \ \text{pat}') \\
& \quad | \ b = b(\text{pat}) \cup b(\text{pat}')
\end{align*}
\]
How about: Let rec?

- Binding one variable in multiple scopes

\[
\text{letrec } x = (x, y) \text{ in } (x, y)
\]
How about: Let rec?

- Binding one variable in multiple scopes

\[
\text{letrec } x = (x, y) \text{ in } (x, y)
\]

\[
\begin{align*}
exp & \ ::= \ x \\
& \mid () \\
& \mid (\ exp \ , \ exp' \ ) \\
& \mid \text{let rec } x = \ exp \ \text{in } exp' \\
& \hspace{1em} \text{bind } x \ \text{in } exp \\
& \hspace{1em} \text{bind } x \ \text{in } exp'
\end{align*}
\]
How about: Or-patterns?

- A variable does not have a binding occurrence

```plaintext
let
  ( (None, Some x) || (Some x, None) ) = w
in
  ( x, x )
```
How about: Dependent Patterns?

- Binding within binders

```
let
  val [X <: top, x : X] = w
in
  [ X, (x, y) ]
```
This work

- A language for binding structures
- What does it mean, mathematically?
- *What does it really mean, mechanically?*
Bindspec language annotations

element, \( e \ ::= \)

\( \mid \) \text{terminal} \\
\( \mid \) \text{metavar} \\
\( \mid \) \text{nonterm} \\

prod, \( p \ ::= \)

\( \mid \mid \text{element}_1 \ldots \text{element}_m \ ::= \ ::= \ ::= \text{prodname} ( + \text{bs}_1 \ldots \text{bs}_n + ) \)

bindspec, \( bs \ ::= \)

\( \mid \text{bind mse in nonterm} \)
\( \mid \ldots \)
Metavariable set expressions

- Bind arbitrary sets of metavariables in declared nonterminals

\[ \text{metavar\_set\_expression}, \text{mse ::=} \]

- \{\} \quad \text{Empty}
- \text{metavar} \quad \text{Singleton}
- mse union mse' \quad \text{Union}
- auxfn (nonterm) \quad \text{Auxiliary function}
Auxiliary Functions

- Collect some particular set of metavariables
- User-defined, primitive recursive functions
- Annotation of bindspec language

\[
bindspec, \ bs ::= \\
| \ldots \ \\
| auxfn = mse
\]
Example: Multiple Letrec

\[
\begin{align*}
\text{exp} & \ ::= \ X \\
& \quad \mid \ \text{let rec } \text{lrbs in exp} \quad (+ \text{ bind } b(\text{lrbs}) \text{ in } \text{lrbs }+) \\
& \quad \quad (+ \text{ bind } b(\text{lrbs}) \text{ in exp }+)
\end{align*}
\]

\[
\begin{align*}
\text{lrb} & \ ::= \ X \ \text{pat} = \ \text{exp} \quad (+ \ b = X \ +) \\
& \quad \quad (+ \text{ bind } bpat(\text{pat}) \text{ in exp }+)
\end{align*}
\]

\[
\begin{align*}
\text{lrbs} & \ ::= \ \text{lrb} \quad (+ \ b = b(\text{lrb}) \ +) \\
& \quad \mid \ \text{lrb and lrbs} \quad (+ \ b = b(\text{lrb}) \cup b(\text{lrbs}) \ +)
\end{align*}
\]

\[
\begin{align*}
\text{pat} & \ ::= \ X \quad (+ \ bpat = X \ +) \\
& \quad \mid \ ( \ \text{pat} , \ \text{pat}' \ ) \quad (+ \ bpat = bpat(\text{pat}) \cup bpat(\text{pat}') \ +)
\end{align*}
\]
What does it mean?

- There is no notion of binding occurrence
  - Recall: binders collected by user-defined auxfns
What does it mean?

- There is no notion of binding occurrence
  - Recall: binders collected by user-defined auxfns
- Let us think about alpha-equivalence classes
Alpha-equivalence classes

- Concrete variables that must all vary together
- Relate by partial equivalence relations of occurrence of variables
Alpha-equivalence classes

- Concrete variables that must all vary together
- Relate by partial equivalence relations of occurrence of variables

\[
\begin{align*}
\text{let rec } & \quad f \ x = \ g \ ( \ x - 1 ) \\
\text{and } & \quad g \ x = \ f \ x + h \ x \\
\text{and } & \quad h \ x = 0 \\
\end{align*}
\]

\[\text{in } ( g \ 5 )\]
Alpha-equivalence classes

- Concrete variables that must all vary together
- Relate by partial equivalence relations of occurrence of variables

\[
\text{let rec } f \ x = g (x - 1) \\
\text{and } g \ x = f \ x + h \ x \\
\text{and } h \ x = 0 \\
\text{in } (g \ 5)
\]

- Alpha-equivalence is equivalence up to identity of these concrete variables
Calculating closed PER

- Calculated by induction on term structure
Calculating closed PER

- Calculated by induction on term structure
- Case: bind $mse$ in $nt$ annotation

\[
exp ::= \text{let rec } lrbs \text{ in } exp \quad (+ \text{ bind } b(lrbs) \text{ in } lrbs +)
\]

\[
(+ \text{ bind } b(lrbs) \text{ in } exp +)
\]
Calculating closed PER

- Calculated by induction on term structure
- Case: \( \text{bind } mse \text{ in } nt \text{ annotation} \)

\[
exp ::= \text{let rec } lrbs \text{ in } exp \quad (+ \text{ bind } b(lrbs) \text{ in } lrbs +)
\]

\[
(+ \text{ bind } b(lrbs) \text{ in } exp +)
\]

- Collect relevant occurrences of variables and relate them

\[
\text{let rec } f \ x = f \ ( x - 1 )
\]

\[
\text{in } f \ 4
\]
Calculating closed PER

- Calculated by induction on term structure
- Case: bind $mse$ in $nt$ annotation
  
  \[
  \text{exp} \quad ::= \quad \text{let rec } lrbs \text{ in exp} \quad (+ \text{ bind } b(lrbs) \text{ in } lrbs +) \\
  \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (+ \text{ bind } b(lrbs) \text{ in } \text{exp} +)
  \]

- Collect relevant occurrences of variables and relate them
  
  \[
  \text{let rec } f \; x = f \; (x - 1) \\
  \text{in } f \; 4
  \]

- Seal the equivalence relation of all such variables (forget its identity)…
Open PER

… but not always!

Consider when there is binding within binding

```
let

val [X <: top, x : X ] = w

in [ X, ... ]
```
Open PER

…but not always!

Consider when there is binding within binding

\[ [X <: \text{top}, x : X] \]

Cannot forget the concrete variable (more binding possible)

Syntactically analyze when safe to seal
Well-formed Substitution

- Defined over our alpha-equivalence classes
- Must avoid capture (PER’s undisturbed)
- When substituting closed terms, cheap solution possible
  - Check for equality when descending binders
  - Clearly not what you want to use in general
What does it *Really* Mean?

- Proof assistant representations
- Translations to a proper alpha-equivalent representation: deBruijn, HOAS, locally nameless, nominal…
- Not clear how to translate the entire language
The way forward?

- Simple cases are easy
  - Single binders in one or more terms
The way forward?

- Simple cases are easy
  - Single binders in one or more terms
- Translate (almost) everything to single binders?
  - Possibly, cases without nested binding
The way forward?

- Simple cases are easy
  - Single binders in one or more terms
- Translate (almost) everything to single binders?
  - Possibly, cases without nested binding
- …without loss of expressiveness?
The way forward?

- Simple cases are easy
  - Single binders in one or more terms
- Translate (almost) everything to single binders?
  - Possibly, cases without nested binding
- ... without loss of expressiveness?
- ... making idiomatic proofs possible?
Related work

- Much work on single binders
- Rich binding specifications: FreshML, Cαml
  - Cαml: similar goals, but different expressivities
  - Alpha-equivalence classes coincides on large subset
  - Multiple auxiliary functions, or multiple binding occurrences, in Cαml?
  - Bind only in some subterms in Ott bindspec?
Current and future work

- Mechanized rich theory of binding (mini-Ott in Ott)
- Showed correspondence with usual notions in simple cases
- Define a notion of correctness (aka adequacy)
- Want: a translation to a practical representation
Thank you!

http://www.cl.cam.ac.uk/~pes20/ott
Inexpressible binding

- Binding non-terminals in non-terminals

\[
\text{let } x : \text{bool} = e \\
\text{in ( } x : \text{bool}, x : \text{int} \ )
\]

- Note: It is handled in the implementation with concrete atoms

- First match patterns
  - First occurrence of variable in pattern is binding, others bound