
Real World Binding Structures

Susmit Sarkar

with Peter Sewell and Francesco Zappa Nardelli

Paradigm Binding

- Single binders

$$\begin{array}{l} \textit{exp} ::= X \\ \quad | \lambda X . \textit{exp} \quad \text{bind } X \text{ in } \textit{exp} \\ \quad | \textit{exp} \textit{exp}' \end{array}$$

Paradigm Binding

- Single binders

$$\begin{array}{l} \textit{exp} ::= X \\ \quad | \lambda X . \textit{exp} \quad \text{bind } X \text{ in } \textit{exp} \\ \quad | \textit{exp} \textit{exp}' \end{array}$$

- 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

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

<i>exp</i>	::=	<i>X</i>	
		(<i>exp</i> , <i>exp'</i>)	
		let <i>pat</i> = <i>exp</i> in <i>exp'</i>	bind <i>b(pat)</i> in <i>exp'</i>
<i>pat</i>	::=	<i>X</i>	<i>b</i> = <i>X</i>
		–	<i>b</i> = { }
		(<i>pat</i> , <i>pat'</i>)	<i>b</i> = <i>b(pat)</i> ∪ <i>b(pat')</i>

How about: Let rec?

- Binding one variable in multiple scopes

letrec $x = (x, y)$ in (x, y)

How about: Or-patterns?

- A variable does not have a binding occurrence

let

((None , Some x)
|| (Some x , None)) = w

in

(x , x)

How about: Dependent Patterns?

- Binding within binders

let

val $[X <: \text{top}, x : X] = w$

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 ::=

| *terminal*

| *metavar*

| *nonterm*

prod, p ::=

| | *element*₁ .. *element*_m :: :: *prodname* (+ *bs*₁ .. *bs*_n +)

bindspec, bs ::=

| **bind** *mse* **in** *nonterm*

| ...

Metavariable set expressions

- Bind arbitrary sets of metavariables in declared nonterminals

metavar_set_expression, mse ::=

$\{\}$	Empty
<i>metavar</i>	Singleton
<i>mse union mse'</i>	Union
<i>auxfn (nonterm)</i>	Auxiliary function

Auxiliary Functions

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

bindspec, bs ::=

| ...

| *auxfn = mse*

Example: Multiple Letrec

exp	$::=$	X	
		let rec $lrbs$ in exp	(+ bind $b(lrbs)$ in $lrbs$ +) (+ bind $b(lrbs)$ in exp +)
lrb	$::=$	X $pat = exp$	(+ $b = X$ +) (+ bind $bpat(pat)$ in exp +)
$lrbs$	$::=$	lrb	(+ $b = b(lrb)$ +)
		lrb and $lrbs$	(+ $b = b(lrb) \cup b(lrbs)$ +)
pat	$::=$	X	(+ $bpat = X$ +)
		(pat , pat')	(+ $bpat = bpat(pat) \cup bpat(pat')$ +)

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

let rec $f\ x = g\ (x - 1)$

and $g\ x = f\ x + h\ x$

and $h\ x = 0$

in $(g\ 5)$

Alpha-equivalence classes

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

let rec $f\ x = g\ (x - 1)$

and $g\ x = f\ x + h\ x$

and $h\ x = 0$

in $(g\ 5)$

- Alpha-equivalence is equivalence upto 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

$$\begin{aligned} exp & ::= \text{let rec } lrbs \text{ in } exp && (+ \text{ bind } b(lrbs) \text{ in } lrbs +) \\ & && (+ \text{ bind } b(lrbs) \text{ in } exp +) \end{aligned}$$

- Collect relevant occurrences of variables and relate them

$$\begin{aligned} & \text{let rec } f \ x = f \ (x - 1) \\ & \text{in } f \ 4 \end{aligned}$$

Calculating closed PER

- Calculated by induction on term structure

- Case: bind mse in nt annotation

$$\begin{aligned} exp & ::= \text{let rec } lrbs \text{ in } exp && (+ \text{ bind } b(lrbs) \text{ in } lrbs +) \\ & && (+ \text{ bind } b(lrbs) \text{ in } exp +) \end{aligned}$$

- Collect relevant occurrences of variables and relate them

$$\begin{aligned} & \text{let rec } f \ x = f \ (x - 1) \\ & \text{in } f \ 4 \end{aligned}$$

- 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?

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

Inexpressible binding

- Binding non-terminals in non-terminals

```
let x : bool = e  
in ( x : bool, x : int )
```

- Note: It is handled in the implementation with concrete atoms
- First match patterns
 - First occurrence of variable in pattern is binding, others bound