A Binding Bestiary

This note collects a variety of binding forms found in the wild, together with a few artificial examples.

There seems to be no obvious upper bound for the desirable expressiveness of a notion of binding algebra – it is always possible to invent a more wacky example – so our focus here is on exploring the limits of what people actually want to use. Any additions would be very welcome!

One might use these to understand and compare proposals for binding syntax – to see how they would be dealt with in the various obvious candidates (Twelf/HOAS, De Bruijn/Coq, Nominal datatypes/Isabelle-HOL,...), and to assess the level of "encoding noise" involved.

For most of them they are described here using the metalanguage processed by the ott prototype. Examples flagged [*] cannot be expressed in that as it stands.

We think we need to be able to express 1–19, perhaps 20, and are not concerned with 21–22.

First, a series of ML-style let binders, of increasing fanciness. Weirder things follow afterwards.

1) Single binders - simple lambda calculus

$$
\begin{array}{l}
\mathsf{sort}\ \mathit{termvar} \\
\mathsf{var}\ X\ ::\ \mathit{termvar} \\
\\
\begin{array}{lcll}
\mathit{exp} & ::= & & \\
 & & X & \\
 & | & \lambda\ X\ .\ \mathit{exp} & \mathsf{bind}\ X\ \mathsf{in}\ \mathit{exp} \\
 & | & \mathit{exp}\ \mathit{exp}' &
\end{array}
\end{array}
$$

Examples: $x\ \lambda\ {\color{red}x}\ .\ {\color{red}x}\ y$ and $\lambda\ {\color{red}x}\ .\ {\color{red}x}\ \lambda\ {\color{green}x}\ .\ \lambda\ {\color{blue}y}\ .\ {\color{green}x}\ {\color{blue}y}$

2) Pattern binders - lambda calculus with pairs and pair patterns

$$
\begin{array}{l}
\mathsf{sort}\ \mathit{termvar} \\
\mathsf{var}\ X\ ::\ \mathit{termvar} \\
\\
\begin{array}{lcll}
\mathit{exp} & ::= & & \\
 & & X & \\
 & | & \lambda\ X\ .\ \mathit{exp} & \mathsf{bind}\ X\ \mathsf{in}\ \mathit{exp} \\
 & | & \mathit{exp}\ \mathit{exp}' & \\
 & | & (\ \mathit{exp}\ ,\ \mathit{exp}'\ ) & \\
 & | & \mathbf{let}\ \mathit{pat}\ =\ \mathit{exp}\ \mathbf{in}\ \mathit{exp}' & \mathsf{bind}\ \mathit{binders}(\mathit{pat})\ \mathsf{in}\ \mathit{exp}' \\
\\
\mathit{pat} & ::= & & \\
 & & X & \mathit{binders} = X \\
 & | & \_ & \mathit{binders} = \{\} \\
 & | & (\ \mathit{pat}\ ,\ \mathit{pat}'\ ) & \mathit{binders} = \mathit{binders}(\mathit{pat}) \cup \mathit{binders}(\mathit{pat}') \\
 & & & \mathsf{names}(\mathit{binders}(\mathit{pat}))\ \#\ \mathsf{names}(\mathit{binders}(\mathit{pat}'))
\end{array}
\end{array}
$$

Example: $\mathbf{let}\ (\ {\color{red}x}\ ,\ {\color{green}y}\ )\ =\ z\ \mathbf{in}\ {\color{red}x}\ {\color{green}y}$ with its pat subterm $(\ x\ ,\ y\ )$

Here we use an auxiliary 'binders' to collect the binding occurrences of a pattern ('binders' is not a keyword, and some examples need more than one auxiliary.).

The names(binders(pat)) denotes the set of names at those occurrences in pat.

There is a potential conflict between multiple occurrences of the same identifier in a pattern. Informally, we usually impose a condition that the identifiers are all distinct. Whether that is built into the definition of abstract syntax up to alpha varies (it need not be involved in the definition of alpha equivalence - instead it just defines well-formedness predicates, on both raw and quotiented terms).

3) Multiple bindings in a single production - function let with an explicit argument

sort $termvar$
var X :: $termvar$

exp	::=		
		X	
	$\vert$	$\lambda\ X\ .\ exp$	bind X in exp
	$\vert$	$exp\ exp'$	
	$\vert$	$(\ exp\ ,\ exp'\ )$	
	$\vert$	$\mathbf{let}\ X\ pat\ =\ exp\ \mathbf{in}\ exp'$	bind $binders(pat)$ in exp bind X in exp' $\mathsf{names}(X)\ \#\ \mathsf{names}(binders(pat))$
pat	::=		
		X	$binders = X$
	$\vert$	$\_$	$binders = \{\}$
	$\vert$	$(\ pat\ ,\ pat'\ )$	$binders = binders(pat) \cup binders(pat')$ $\mathsf{names}(binders(pat))\ \#\ \mathsf{names}(binders(pat'))$

Examples: $\lambda\ x\ .\ \mathbf{let}\ f\ (\ x\ ,\ y\ )\ =\ x\ \mathbf{in}\ f\ (\ x\ ,\ y\ )$ and $\lambda\ x\ .\ \lambda\ y\ .\ y\ x$.

Here (not much of a diff from the previous one) a single production has two independent bind clauses, binding different binders in different subterms.

4) List forms in patterns - function let with explicit arguments, and tuple patterns [*]

```
exp ::= ... | let X pat1 .. patn = exp in exp'
```

or

```
pat ::= ... | (pat1,..,patn)  n>=0
```

Typically one would formalise with new syntactic categories for the list forms, but .. forms could be supported more directly (cf the EBNF examples later).

5) Recursive binders - single letrec

sort $termvar$
var X :: $termvar$

exp	::=		
		X	
	$\vert$	$(\ exp\ ,\ exp'\ )$	
	$\vert$	$\mathbf{letrec}\ X\ =\ exp\ \mathbf{in}\ exp'$	bind X in exp bind X in exp'

Here the scope of a binder is two distinct subterms.

Example: $\mathbf{letrec}\ X\ =\ (\ X\ ,\ Y\ )\ \mathbf{in}\ (\ X\ ,\ Y\ )$

6) Recursive binders - single letrec with explicit argument

sort $\mathit{termvar}$
var X :: $\mathit{termvar}$

exp	::=		
		X	
	|	$(\ \mathit{exp}\ ,\ \mathit{exp}'\ )$	
	|	$\textbf{letrec}\ X\ \mathit{pat}\ =\ \mathit{exp}\ \textbf{in}\ \mathit{exp}'$	bind X in exp bind X in exp' bind $b(\mathit{pat})$ in exp names(X) # names$(b(\mathit{pat}))$
pat	::=		
		X	$b = X$
	|	$(\ \mathit{pat}\ ,\ \mathit{pat}'\ )$	$b = b(\mathit{pat}) \cup b(\mathit{pat}')$ names$(b(\mathit{pat}))$ # names$(b(\mathit{pat}'))$

Example: $\textbf{letrec}\ f\ (\ x\ ,\ y\ )\ =\ (\ f\ ,\ (\ a\ ,\ (\ x\ ,\ y\ )\ )\ )\ \textbf{in}\ (\ f\ ,\ (\ b\ ,\ (\ x\ ,\ y\ )\ )\ )$

Here there is a potential conflict between the X and pat binders, which could resolve - as here - by requiring them to be distinct. It's perhaps more intuitive to have the pat scope shadow the X scope in exp if pat contains any Xs, by introducing an intermediate syntactic category for the X pat = exp form (as below).

7) Multiple recursive binders - multiple letrec

sort $\mathit{termvar}$
var X :: $\mathit{termvar}$

exp	::=		
		X	
	|	$()$	
	|	$(\ \mathit{exp}\ ,\ \mathit{exp}'\ )$	
	|	$\textbf{letrec}\ \mathit{lrbs}\ \textbf{in}\ \mathit{exp}$	bind $b(\mathit{lrbs})$ in lrbs bind $b(\mathit{lrbs})$ in exp distinctnames$(b(\mathit{lrbs}))$
lrb	::=		
		$X\ =\ \mathit{exp}$	$b = X$
lrbs	::=		
		lrb	$b = b(\mathit{lrb})$
	|	$\mathit{lrb}\ \textbf{and}\ \mathit{lrbs}$	$b = b(\mathit{lrb}) \cup b(\mathit{lrbs})$

Example: $\textbf{letrec}\ f\ =\ (\ g\ ,\ (\ f\ ,\ x\ )\ )\ \textbf{and}\ g\ =\ (\ g\ ,\ (\ f\ ,\ x\ )\ )\ \textbf{in}\ (\ f\ ,\ g\ )$ with its subterm $f\ =\ (\ g\ ,\ (\ f\ ,\ x\ )\ )$

Just like (5), though the Xs on the left of a lrbs should usually all be distinct.

Note that the "bind b(lrbs) in lrbs" binds in all parts of the lrbs; there's nothing saying "bind only in the right-hand sides". That might seem strange at first sight, but we think it's not a problem. In fact, once you've quotiented by alpha, the binding occurrence has no special status.

8) Multiple recursive binders - multiple letrec with multiple clauses for each function (prompted by James Cheney's MERLIN talk)

For example something like this:

```
let rec  f ((),y)   = g (y,y,())
```

```
      | f (y,z)    = g (y,(),z)
    and g ((),y,z) = f (y,z)
      | g (x,y,z)  = f ((),())
in ...
```

where each block defines a function (f and g), with potentially many clauses, but each function is defined by at most one block, and each block consists only of clauses for that function.

sort *termvar*
var X :: *termvar*

exp	::=		
		X	
	|	$()$	
	|	$(\ exp\ ,\ exp'\ )$	
	|	$(\ exp\ )$	
	|	$exp\ exp'$	
	|	**let rec** $lrbs$ **in** exp	bind $b(lrbs)$ in $lrbs$ bind $b(lrbs)$ in exp
$fnclause$	::=		
		$X\ pat\ =\ exp$	$b = X$ bind $bpat(pat)$ in exp $\mathsf{names}(X) \# \mathsf{names}(bpat(pat))$
$fnclauses$	::=		
		$fnclause$	$b = b(fnclause)$ $bheads = b(fnclause)$
	|	$fnclause \mid\mid fnclauses$	$b = b(fnclause) \cup b(fnclauses)$ $bheads = b(fnclause)$ $\mathsf{names}(b(fnclause))\ =\ \mathsf{names}(b(fnclauses))$
lrb	::=		
		$fnclauses$	$b = b(fnclauses)$ $bheads = bheads(fnclauses)$
$lrbs$	::=		
		lrb	$b = b(lrb)$ $bheads = bheads(lrb)$
	|	lrb **and** $lrbs$	$b = b(lrb) \cup b(lrbs)$ $bheads = bheads(lrb) \cup bheads(lrbs)$ $\mathsf{names}(bheads(lrb)) \# \mathsf{names}(bheads(lrbs))$
pat	::=		
		X	$bpat = X$
	|	$()$	$bpat = \{\}$
	|	$(\ pat\ ,\ pat'\ )$	$bpat = bpat(pat) \cup bpat(pat')$

Here: b collects the recursive binders - all occurrences of f,g etc bheads collects the first of each of these - the first f, the first g, etc - to state the (*) distinctness condition. This is not used to define binding. bpat collects the binders of a pattern

For example **let rec** $f\ x\ =\ g\ (\ f\ x\ ) \mid\mid f\ (\ a\ ,\ b\ )\ =\ g\ (\ f\ (\ a\ ,\ x\ )\ )$ **and** $g\ y\ =\ g\ (\ f\ x\ )$ **in** $(\ f\ ,\ g\ )$, with its subterm $f\ x\ =\ g\ (\ f\ x\ )$.

At present this doesn't exclude

let rec x () = () **and** y x = x **in** (x , y)

and neither does OCaml 3.07+2, but neither identify the x's: `- : (unit -> unit) * ('a -> 'a) = (<fun>, <fun>)`.

Depending on the exact definition of alpha one might have all the x's above alpha-vary together, which would be wrong. Our definition of alpha gives the intended binding, because bpat is not propagated outside the fnclause production.

If we did want to impose distinctness, how would we say it? In letrec lrbs in exp we have, for all pat occurring in lrbs,

```
names(b(lrbs)) intersect names(bpat(pat))
```

Add set-of-sets of occurrences to the auxiliaries?

9) Let sequence, with each binding in the next [*?]

sort $termvar$
var X :: $termvar$

exp	::=		
		X	
	|	()	
	|	0	
	|	1	
	|	2	
	|	3	
	|	exp + exp'	
	|	**let** $lets$ **in** exp	bind $b(lets)$ in exp distinctnames($b(lets)$)
$lets$	::=		
		$alet$	$b = b(alet)$
	|	$alet$ **and** $lets$	$b = b(alet) \cup b(lets)$ bind $b(alet)$ in $lets$
$alet$	::=		
		X = exp	$b = X$

For example: **let** x = y **and** y = x **in** x + y

Note that here it would be nice not to require distinctness, eg to admit

```
let x=0 and x=1+x and x=2+x in x
```

You might regard this as syntactic sugar for iterated single lets, but suppose you wanted to express it directly, with a grammar

sort $termvar$
var X :: $termvar$

exp	::=		
		X	
	|	$()$	
	|	0	
	|	1	
	|	2	
	|	3	
	|	$exp\ +\ exp'$	
	|	**let** $lets$ **in** exp	bind $b(lets)$ in exp
$lets$	::=		
		$alet$	$b = b(alet)$
	|	$alet$ **and** $lets$	$b = b(alet) \cup b(lets)$ bind $b(alet)$ in $lets$
$alet$	::=		
		$X\ =\ exp$	$b = X$

At present the standard semantics doesn't support this, identifying all the x's in

let $x\ =\ 0\ +\ x$ **and** $x\ =\ 1\ +\ x$ **and** $x\ =\ 2\ +\ x$ **and** $x\ =\ 3\ +\ x$ **in** 0

and in its lets subterm $x\ =\ 0\ +\ x$ **and** $x\ =\ 1\ +\ x$ **and** $x\ =\ 2\ +\ x$ **and** $x\ =\ 3\ +\ x$

It's possible that the definition could be benignly changed to not do this. With the `variant_c_x_semantics` switch the x's in the lets subterm are all unequated (except the last two, which is an artifact of the fact that the grammar clause for lets ::= alet does not have an annotation (+ bind b(alet) in nothing +)). However, in the full term they are all identified again - by exactly the mechanism that means that or-patterns and join patterns work correctly.

10) Dependent record patterns

For concreteness, this is loosely based on the Pict 4.1 grammar. (here we use multiple sorts of identifiers, and do not have ¡empty¿ productions).

sort $typevar$
sort $termvar$
sort $LABEL$
var X :: $typevar$
var x :: $termvar$
var $Label$ :: $LABEL$

$Proc$	::=		
		x	
	$\vert$	$[\ X\ ,\ Proc\ ]$	
	$\vert$	$(\ Proc\ ,\ Proc'\ )$	
	$\vert$	**let** Dec **in** $Proc$	bind $b1(Dec)$ in $Proc$
Dec	::=		
		val $Pat\ =\ Val$	bind $b1(Pat)$ in Val
			$b1 = b1(Pat)$
Pat	::=		
		$x\ :\ Type$	$b1 = x$
	$\vert$	$[\,]$	$b1 = \{\}$
	$\vert$	$[\ Pats\ ]$	$b1 = b1(Pats)$
$Pats$	::=		
		$Label\ =\ FieldPat$	$b1 = b1(FieldPat)$
	$\vert$	$Label\ =\ FieldPat\ Pats$	$b1 = b1(FieldPat) \cup b1(Pats)$
			bind $b2(FieldPat)$ in $Pats$
			names$(b1(FieldPat))$ # names$(b1(Pats))$
$FieldPat$	::=		
		Pat	$b1 = b1(Pat)$
			$b2 = \{\}$
	$\vert$	$\#\ X\ <\ Type$	$b1 = X$
			$b2 = X$
$Type$	::=		
		int	
	$\vert$	**unit**	
	$\vert$	**top**	
	$\vert$	$Type\ *\ Type'$	
	$\vert$	X	
Val	::=		
		$()$	
	$\vert$	x	

b1 collects all the binders of a complex pattern

b2 collects just the binders that bind to the right of a particular (type) field

(several b1 and b2 definition clauses might be omitted if the default-union rule is used, though we would then want to give the types of b1:Pat,Pats,FieldPat and b2:FieldPat explicitly somewhere)

For example, consider

let val $[\ l1\ =\ \#\ X\ <\ $ **top** $\ l2\ =\ x\ :\ X\ ]\ =\ w$ **in** $[\ X\ ,\ (\ x\ ,\ y\ )\ ]$

and its Dec subterm

$\textbf{val}\ [\ \mathit{l1}\ =\ \#\ X\ <\ \textbf{top}\ \mathit{l2}\ =\ x\ :\ X\ ]\ =\ w$

and the Dec

$\textbf{val}\ [\ \mathit{l1}\ =\ \#\ X\ <\ \textbf{top}\ \mathit{l2}\ =\ [\ \mathit{l2a}\ =\ x\ :\ X\ \mathit{l2b}\ =\ \#\ Y\ <\ \textbf{top}\ ]\ \mathit{l3}\ =\ y\ :\ X\ *\ Y\ ]\ =\ w$

where the Y in l3 is free.

11) OCaml or-patterns. From the manual:

The pattern pattern1 — pattern2 represents the logical "or" of the two patterns pattern1 and pattern2. A value matches pattern1 — pattern2 either if it matches pattern1 or if it matches pattern2. The two sub-patterns pattern1 and pattern2 must bind exactly the same identifiers to values having the same types. Matching is performed from left to right. More precisely, in case some value v matches pattern1 — pattern2, the bindings performed are those of pattern1 when v matches pattern1. Otherwise, value v matches pattern2 whose bindings are performed.

For our binding specifications to capture this we might add equality constraints on name sets, eg

```
pattern ::= ...
         | (pattern1 | pattern2)    b = b(pattern1) union b(pattern2)
                                    names(b(pattern1)) = names(b(pattern2))
```

Note that in the constraint the names(b(pattern1)) and names(b(pattern2)) denote the sets of identifiers, not the underlying sets of occurrences of identifiers.

In the b = b(pattern1) union b(pattern2) clause we mean the union of the sets of occurrences, though (as usual), to ensure they alpha convert together. For example,

```
  let f ((None,Some x)|(Some x,None)) = x in f (None,Some 2);;
=alpha
  let f ((None,Some y)|(Some y,None)) = y in f (None,Some 2);;
```

Think this is ok for deeply nested or and non-or patterns, eg:

sort $termvar$
var x :: $termvar$

exp	::=		
		x	
	$\vert$	$(\ exp\ ,\ exp'\ )$	
	$\vert$	$\textbf{let}\ pat\ =\ exp\ \textbf{in}\ exp'$	bind $b(pat)$ in exp'
pat	::=		
		$(\ pat\ ,\ pat'\ )$	$b = b(pat) \cup b(pat')$ names$(b(pat))$ # names$(b(pat'))$
	$\vert$	$(\ pat \mid\mid pat'\ )$	$b = b(pat) \cup b(pat')$ names$(b(pat))$ = names$(b(pat'))$
	$\vert$	$\textbf{Some}\ x$	$b = x$
	$\vert$	$\textbf{None}$	$b = \{\}$

$\textbf{let}\ (\ (\ \textbf{None}\ ,\ \textbf{Some}\ x\ ) \mid\mid (\ \textbf{Some}\ x\ ,\ \textbf{None}\ )\ )\ =\ w\ \textbf{in}\ (\ x\ ,\ x\ )$

12) Join calculus

Join calculus definitions have several interesting aspects. Here is a raw syntax extracted from the JoCaml manual of January 8, 2001, with binding spec made up by PS.

sort $names$
var $name$ :: $names$

$process$	::=		
		$declaration$ **in** $process$	bind $b(declaration)$ in $process$
	|	0	
	|	$name\ expression$	
	|	$process \mid\mid process'$	
$declaration$	::=		
		let def $automata\_definition$	$b = b(automata\_definition)$ bind $b(automata\_definition)$ in $automata\_definition$
$automata\_definition$	::=		
		$automaton$	$b = b(automaton)$
	|	$automaton$ **and** $automata\_definition$	$b = b(automaton) \cup b(automata\_definition)$ names$(b(automaton))$ # names$(b(automata_definiti$
$automaton$	::=		
		$join\_pattern\ =\ process$	$b = b(join\_pattern)$ bind $b(join\_pattern)$ in $process$ bind $b2(join\_pattern)$ in $process$
	|	$join\_pattern\ =\ process$ **or** $automaton$	$b = b(join\_pattern) \cup b(automaton)$ bind $b2(join\_pattern)$ in $process$
$join\_pattern$	::=		
		$channel\_decl$	$b = b(channel\_decl)$ $b2 = b2(channel\_decl)$
	|	$channel\_decl \mid\mid join\_pattern$	$b = b(channel\_decl) \cup b(join\_pattern)$ $b2 = b2(channel\_decl) \cup b2(join\_pattern)$ names$(b2(channel_decl))$ # names$(b2(join_pattern))$
$channel\_decl$	::=		
		$name\ OCaml\_pattern$	$b = name$ $b2 = bindings(OCaml\_pattern)$
$expression$	::=		
		$name$	
	|	$(\ expression\ ,\ expression'\ )$	
$OCaml\_pattern$	::=		
		$name$	$bindings = name$
	|	$(\ name\ )$	$bindings = name$
	|	$()$	$bindings = \{\}$
	|	$(\ OCaml\_pattern\ ,\ OCaml\_pattern'\ )$	$bindings = bindings(OCaml\_pattern) \cup bindings($ names$(bindings(OCaml_pattern))$ # names$(binding$

Note:

- it would be rather nicer to give the raw grammar in an extended BNF (as the JoCaml definition does), with optional clauses in [..] The binding specification language would need to follow suit.

- the different or-clauses of an automaton and —-clauses of a join-pattern do not necessarily have distinct binders. For example,

let def x () || x () $=$ a (x , y) **or** x () || y () $=$ b (x , y) **in** c (x , (y , z))

with two is just fine, binding x and y in P, Q, and R. This is alpha equivalent to

let def x' () || x' () $=$ a (x' , y') **or** x' () || y' () $=$ b (x' , y') **in** c (x' , (y' , z))

but not to

let def x' () || x' () $=$ a (x' , y') **or** x'' () || y' () $=$ b (x'' , y') **in** c (x' , (y' , z))

- the identifiers within the collection of OCaml-patterns in a join pattern, on the other hand, presumably should all be distinct, and should be distinct from all the names. For example,

```
let def c(x) | d(x) = P in R
```

`and`

```
let def x(x) = P in R
```

should not be allowed, whereas

let def c (x) || d (y) $=$ p (c , (d , (x , y))) **or** c (x) $=$ q (c , (d , x)) **in** r (c , d)

should.

13) Multiple binding sorts (and the POPLmark example)

In languages with multiple name sorts, eg of type and term names, we want to ensure that a binder of one sort does not bind occurrences of another. For example, we might write

```
let f = Lambda X:Type => lambda ((x:X),(f:X->X)) => f x in ...
```

but

```
let f = Lambda x:Type => lambda ((x:x),(f:x->x)) => f x in ...
```

should either be forbidden or it should be understood that the x type binder binds only the occurrences of x in type positions.

The Fsub-with-records example illustrates this

sort $typevar$
sort $termvar$
sort $label$
var X :: $typevar$
var x :: $termvar$
var l :: $label$

T	$::=$		
		X	
	$\mid$	$\mathbf{Top}$	
	$\mid$	$T \ \rightarrow \ T'$	
	$\mid$	$\forall\, X \ <: \ T \ . \ T'$	bind X in T'
	$\mid$	$\{\,\}$	
	$\mid$	$\{ \ T\_recbody \ \}$	
$T\_recbody$	$::=$		
		$l \ : \ T$	
	$\mid$	$l \ : \ T \ , \ T\_recbody$	
t	$::=$		
		x	
	$\mid$	$\lambda\, x \ : \ T \ . \ t$	bind x in t
	$\mid$	$t \ t'$	
	$\mid$	$\lambda\, X \ <: \ T \ . \ t$	bind X in t
	$\mid$	$t \ [\ T \ ]$	
	$\mid$	$\{\,\}$	
	$\mid$	$\{ \ t\_recbody \ \}$	
	$\mid$	$t \ . \ l$	
	$\mid$	$\mathbf{let} \ p \ = \ t \ \mathbf{in} \ t'$	bind $bo(p)$ in t'
$t\_recbody$	$::=$		
		$l \ = \ t$	
	$\mid$	$l \ = \ t \ , \ t\_recbody$	
p	$::=$		
		$x \ : \ T$	$bo = x$
	$\mid$	$\{\,\}$	$bo = \{\}$
	$\mid$	$\{ \ p\_recbody \ \}$	$bo = bo(p\_recbody)$
$p\_recbody$	$::=$		
		$l \ = \ p$	$bo = bo(p)$
	$\mid$	$l \ = \ p \ , \ p\_recbody$	$bo = bo(p) \cup bo(p\_recbody)$
G	$::=$		
		$\mathbf{empty}$	$dom = \{\}$
	$\mid$	$G \ , \ X \ <: \ T$	$dom = dom(G) \cup X$
			names$(dom(G))$ # names(X)
	$\mid$	$G \ , \ x \ : \ T$	$dom = dom(G) \cup x$
			names$(dom(G))$ # names(x)
J	$::=$		
		$G \ \vdash \ T \ <: \ T'$	
	$\mid$	$G \ \vdash \ t \ : \ T$	
	$\mid$	$t \ \longrightarrow \ t'$	
Gb	$::=$		
		$\mathbf{empty}$	$dom = \{\}$
	$\mid$	$Gb \ , \ X \ <: \ T$	$dom = dom(Gb) \cup X$
			bind $dom(Gb)$ in T
			names$(dom(Gb))$ # names(X)
	$\mid$	$Gb \ , \ x \ : \ T$	$dom = dom(Gb) \cup x$
			bind $dom(Gb)$ in T
			names$(dom(Gb))$ # names(x)
Jb	$::=$		
		$Gb \ \vdash \ T \ <: \ T'$	bind $dom(Gb)$ in T
			bind $dom(Gb)$ in T'
	$\mid$	$Gb \ \vdash \ t \ : \ T$	bind $dom(Gb)$ in t
			bind $dom(Gb)$ in T

Here we have three sorts of names (but no lexical distinction between them); bo(pattern) only collects the term names of a pattern; and in Lambda X¡:T.term the X binds throughout the term, including in any types in patterns it may contain. Potential monsters such as

$\lambda\ y\ <:\ \mathit{Top}\ .\ \mathbf{let}\ x\ :\ X\ =\ y\ \mathbf{in}\ x$

(with the y binding and bound) are excluded only by the sort distinction, which ensures that the two y's are different.

We could add another auxiliary and conditions to ensure that the labels in a record are distinct.

One could have binding specs that make explicit use of the sorts (as the new Fresh does), eg

```
...
  | let pattern = term in term'       bind termvar(pattern) in term'
```

If you have the machinery for defining arbitrary name-occurrence auxiliaries (such as the bo here) it's not clear that this is useful, though. But having multiple sorts is - particularly when you come to concrete terms. When we say "bind MSE in NN" that really means "bind all occurrences of identifiers in positions of the corresponding sort (as in MSE) in NN".

For judgements, one might have the domain of a type environment G binding in the remainder of the judgement (as in Jbinding) or not (as in J). Note that we are not restricting auxiliaries (eg dom(..)) to be sets of occurrences of variables of the same sort. Example:

:J $\mathbf{empty}\ ,\ X\ <:\ \mathit{Top}\ ,\ Y\ <:\ X\ \rightarrow\ X\ ,\ x\ :\ X\ ,\ y\ :\ Y \vdash\ y\ x\ :\ X$

:Jb $\mathbf{empty}\ ,\ X\ <:\ \mathit{Top}\ ,\ Y\ <:\ X\ \rightarrow\ X\ ,\ x\ :\ X\ ,\ y\ :\ Y \vdash\ y\ x\ :\ X$

In the latter case the Gb has a non-trivial o:

:Gb $\mathbf{empty}\ ,\ X\ <:\ \mathit{Top}\ ,\ Y\ <:\ X\ \rightarrow\ X\ ,\ x\ :\ X\ ,\ y\ :\ Y$

14) Scoping without binding

Labels, ML constructor names, and so on. Various classes of identifiers have scopes, and are subject to distinctness conditions, but do not alpha-vary.

Whether this is something one wants to address in the abstract syntax is unclear, but the distinctness conditions we use elsewhere perhaps would suffice. (Though if you introduce occurrence auxiliaries just for that, that are not identifying binders, the definition of alpha equivalence should not pay attention to them.)

15) Forbidding shadowing [?]

Java local declarations are not permitted if they would shadow. This is maybe best treated as a distinctness condition, but with or without binding?

16) Store [*, but should]

```
store ::=   location -->_{finite,partial} value

config ::=  store; expr   dom(store) binds in store
                          dom(store) binds in expr
```

The binding here is just like letrecs - the only interesting thing is that the syntax is not free, but either:

- with finite partial function spaces and dom() provided as primitive, or - subject to associative, commutative, idempotency equations and with a condition saying each location occurs at most once on the left.

In Acute we had configurations with both a store typing and a store, together with running processes. Really, the store typing and store should simultaneously bind (the identifiers in their domains, which should be identical) in the store range and the processes. (In the actual definition we had neither bind, as that seemed a bit baroque.)

17) Internal/external names in module systems

ML-style module system semantics often use both 'external' names, which don't alpha-vary, and 'internal' names, which do. For example, in

```
module M = struct
             type tt_t = int
             val  xx_x : t = 3
           end
let y = M.xx
```

the tt and xx are external names, used in 'dot notation' projections in the scope of the definition of M, whereas the t and x are binders, binding in the suffix of the structure.

Here the t and x are just conventional binders, and this lies in the notes1 definition. One can also have a combined form, in two flavours, as in "Names with auxiliary data".

18) Binding specs in grammars of contexts

(eg the lambda-r example)

```
let x=e in _   .  e'
```

Generally our contexts are concrete gadgets, at least on the path to the hole, but one could do things differently.

19) Type environments and inference rules

Nothing very new here, in fact, but there are several choices as to what binding you have, and very different encodings in different provers.

Type envs can bind internally and in the other parts of judgements or not - matter of taste; one should allow either. This is shown in the POPLmark example above.

Type envs can be either on the left or the right - a stylistic choice only:

```
E ::= empty
    | X,E                  X bind in E
    | x:T,E
```

or (on the right)

```
E ::= empty         b={}
    | E,X           b=b(E) union X
    | E,x:T         b=b(E)              b(E) bind in T
```

(and sometimes , is associative).

Have to do a type formation judgement - depending on the choices above, either:

```
J ::= E |- T Type       b(E) bind in T
    | E |- ok
```

or

```
J ::= E |- T Type
    | E |- ok



             E |- ok
             E |- T Type     E |- ok
-----------  --------------  --------------
empty |- ok  E,x:T ok        E,X ok
```

Question: where do we impose distinctness of names in an E. We could say

```
names(b(E)) intersect names(X)) = {}
names(b(E)) intersect names(x)) = {}
```

in the two productions of the E grammar, or we could say

```
distinct(names(b(E))
```

in one or both productions of the J grammar, or we could say

```
x notin dom(E)
X notin dom(E)
```

in the ok-ness typing rules, or we could have built in that to the definition of , (in which case it's not a matter for us, it's just something the proof assistant knows about).

Note that we might be using distinct(names(...)) for non-binders, eg as here with the "E don't bind" choice.

20) Names with auxiliary data [*, not clear whether or not this should be supported]

(from Michael Norrish) HOL and Isabelle implement types of terms where the variables are stored with their types. Thus

```
\ (x:num). (x:num) + f (x:bool)
```

is a valid term. The (x:bool) is not bound.

In some sense, the combination of "x" and "num" is the binding unit, but when you alpha convert, you are only given licence to change the "x", not the type. The above is thus alpha-equal to

```
\ (y:num).  (y:num) + f (x:bool)
```

Similar binding was used in the Acute definition for module external/internal name pairs. Module names were of the form `MM_M` where MM is an external name (non binding) and M is an internal (subject to binding, but only for occurrences associated with the same external name). Keeping both parts was needed to support rebinding. As far as I recall the alternative approach, of having the M be a simple binder, was technically sufficient but seemed less intuitive.

Perhaps we should generalise sorting to support this, allowing arbitrary term structure in sorts - though if we allow names (or, worse, names and binding) in sorts things would be more complex.

Examples which we don't think we need to express

21) First-match patterns [*] [not something we want to do]

Occasionally one has patterns in which the first occurrence of an x is a binding occurrence and later occurrences are equality-patterns. This cropped up in a composite-event language (Richard Hayton, Cambridge). Stephanie Weirich mentioned something in Perl?? Olin Shivers ICFP talk had binding dependent on control flow.

When things get this wierd, maybe one would just be using an environment semantics in any case, and so not need syntax up to alpha.

22) Brian's triplet [not a natural example]

Overlapping scopes that are not included in each other, eg

sort $termvar$
var $X\ Y\ Z$ :: $termvar$

$$
\begin{array}{lcll}
P & ::= & & \\
 & & \textbf{WeirdBind}\ X\ ,\ Y\ \textbf{in}\ (\ P\ ,\ P'\ ,\ P''\ ) & \textsf{bind}\ X\ \textsf{in}\ P \\
 & & & \textsf{bind}\ X\ \textsf{in}\ P' \\
 & & & \textsf{bind}\ Y\ \textsf{in}\ P' \\
 & & & \textsf{bind}\ Y\ \textsf{in}\ P'' \\
 & | & X & \\
 & | & (\ P\ ,\ P'\ ) &
\end{array}
$$

$\textbf{WeirdBind}\ X\ ,\ Y\ \textbf{in}\ (\ (\ X\ ,\ Y\ )\ ,\ (\ X\ ,\ Y\ )\ ,\ (\ X\ ,\ Y\ )\ )$

We don't know a natural example like this (is there one?), but it can be specified in this metalanguage.