Specifying real-world binding structures

Susmit Sarkar* Peter Sewell* Francesco Zappa Nardelli+

*University of Cambridge +INRIA Rocquencourt

Representing binding structures is a central challenge in mechanizing the theory of programming languages. Two facets of the problem should be distinguished: on the one hand, one needs a specification language for express the complex binding structures found in realistic programming languages, while on the other hand one needs an implementation technique to represent such structures in proof assistants. Most research effort to date has focused on the second issue, limiting to the case in which a single variable binds in a single subterm. Notable exceptions are FreshOCaml [3] and Cαml [1], but neither provides a complete solution (expressive enough to deal with the variety of binding found in the wild but also simple and intuitive).

We argue that lightweight annotations of productions in the syntax grammar are sufficient to specifiy complex binding structures, including multiple `let recs`, OCaml's or-patterns, dependent records, and even Join Calculus patterns. The first annotation bind *mse* in *ntr* lets one specify that the variables defined in the set *mse* bind in nonterminals of the production. Here *mse* ranges over sets of metavariables, which can be specified explicitly or by auxiliary functions, as in the Ocaml or-pattern language fragment below.

$$\begin{array}{lcll}
\mathit{exp} & ::= & x & \\
 & | & (\ \mathit{exp}\ ,\ \mathit{exp}'\ ) & \\
 & | & \textbf{let}\ \mathit{pat}\ =\ \mathit{exp}\ \textbf{in}\ \mathit{exp}' & \mathsf{bind}\ b(\mathit{pat})\ \mathsf{in}\ \mathit{exp}' \\
\mathit{pat} & ::= & & \\
 & | & (\ \mathit{pat}\ ||\ \mathit{pat}'\ ) & b = b(\mathit{pat}) \cup b(\mathit{pat}') \\
 & | & (\ \mathit{pat}\ ,\ \mathit{pat}'\ ) & b = b(\mathit{pat}) \cup b(\mathit{pat}') \\
 & | & \textbf{Some}\ x & b = x \\
 & | & \textbf{None} & b = \{\}
\end{array}$$

This specifies that the linked occurences of x below must vary together:

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

We will explain the process of giving meaning to binding specifications by considering a fragment of a grammar for System $\mathrm{F}_{<:}$ for contexts and judgments. (In this particular development we choose to have the variables in the Γ of a judgment bind in the judgment body.)

$$\begin{array}{ll}
\Gamma ::= & \\
\quad |\ \varnothing & \mathsf{Tdom} = \{\} \\
 & \mathsf{tdom} = \{\} \\
\quad |\ \Gamma,\ X{<:}T & \mathsf{Tdom} = \mathsf{Tdom}(\Gamma) \cup X \\
 & \mathsf{tdom} = \{\} \\
 & \mathsf{bind}\ \mathsf{Tdom}(\Gamma)\ \mathsf{in}\ T \\
\quad |\ \Gamma,\ x{:}T & \mathsf{Tdom} = \mathsf{Tdom}(\Gamma) \\
 & \mathsf{tdom} = \mathsf{tdom}(\Gamma) \cup x \\
 & \mathsf{bind}\ \mathsf{Tdom}(\Gamma)\ \mathsf{in}\ T \\
 & \\
\mathit{Judgments} ::= & \ldots \\
\quad |\ \Gamma \vdash t : T & \mathsf{bind}\ \mathsf{Tdom}(\Gamma)\ \mathsf{in}\ t \\
 & \mathsf{bind}\ \mathsf{Tdom}(\Gamma)\ \mathsf{in}\ T \\
 & \mathsf{bind}\ \mathsf{tdom}(\Gamma)\ \mathsf{in}\ t
\end{array}$$

The key building block for our definitions is that of sets of variable occurences that are meant to alpha-vary together. Thus for example, the linked occurences of concrete variables must vary together in the following abstract syntax term for a context.

$$\Gamma = X{<:}\mathit{Top},\ Y{<:}X{\to}X,\ x{:}X,\ y{:}Y$$

We call these sets the *open binding sets*, and we can give a compositional calculation of them over the structure of the term by looking at the binding specifications. They are not always allowed to actually vary, such as in the example context above. The variables bound in the context can only be alpha-varied when placed in a judgment, such as in

$$X{<:}\mathit{Top},\ Y{<:}X{\to}X,\ x{:}X,\ y{:}Y \vdash x : X$$

This phenomenon typically occurs when some of the binding structure is visible, but more binding will occur in higher levels of the abstract syntax tree.

When we can syntactically decide at a bind declaration point that none of the variables will be picked out at a binding site, we can seal the sets of variable occurence, which we call *closed binding set*, such as in the judgment form. The notion of alpha-equivalence is then the equivalence class of all terms which differ only in the identity of the concrete variable within the closed binding sets. We give a definition of capture-avoiding substitution, and prove that it respects this expanded notion of alpha-equivalence. Correctness of our definitions cannot be defined in general, as there is no widely accepted other notion of non-single-variable binding, but it can be shown in specific cases — we do so for the important special case of the untyped lambda calculus.

The Ott tool [2] implements this language of binding specifications by translating them into what we call a *fully concrete* representation within various proof assistants. In this representation, abstract syntax terms may contain concrete variables, and terms are not considered up to alpha-equivalence. The (Ott-generated) substitution functions assume that the substituted term is always closed. A wide variety of programming language theory, including a formalization of OCaml without objects or modules, can still be done in this restricted setting, since typical languages do not perform reductions under binders. We prove that under suitable sanity conditions, the operational definition of substitution within Ott coincides with the definition of substitution respecting alpha-equivalence above. The proof is on paper, and partly formalized within Isabelle.

This fully concrete representation, is, however, clearly not sufficient for some examples. An important problem for future work is to develop good proof assistant implementations that respect alpha equivalence in general, not just for closed substitutions. One could envisage doing this indirectly, by translating into languages with single binders, or directly, e.g. perhaps by generalising a locally nameless or nominal representation.

References

[1] François Pottier. An overview of Cαml. In *ACM Workshop on ML, ENTCS 148(2)*, pages 27–52, March 2006.

[2] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *ICFP*, 2007.

[3] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. ICFP*, 2003.