Dependent open terms and the evaluation contexts that bind them

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Rutgers University
ccshan@cs.rutgers.edu

How should we represent open terms and their binding contexts, especially in HOAS, and especially when one binding may depend on another? We came to this problem when formalizing the small-step semantics of staging so as to combine staging with effects in a sound type system. We describe our Twelf solution and call for a more elegant and natural representation.

Code generation is the most promising approach in high-performance computing (as in SPIRAL [5]) and high-assurance embedded programming (as in Hume [3]). Staged languages such as MetaOCaml are an attractive way to express such code generation. Code generation techniques like let insertion require either programming in CPS, which is cumbersome, or using effects such as state or delimited control, which risks scope extrusion: no staged language today has a type system that is sound with effects.

Yukiyoshi Kameyama and us are studying the combination of staging and effects. The most straightforward way to formalize languages with effects (especially delimited control) is small-step semantics with evaluation contexts [7]. With staging, our redexes may be open and our evaluation contexts may be binding. We show a simple example in Scheme, MetaOCaml, and Taha and Nielsen's idealization λ^α [6] of MetaOCaml.

(1) Scheme: `‘(lambda (x) ,(id ‘x))`
MetaOCaml: `.<fun x -> .~(id .<x>.)>.`
λ^α: $\langle \lambda x^\alpha : \text{int}.\sim(\text{id}\,\langle x\rangle^\alpha)\rangle^\alpha$

In λ^α, a bracket $\langle \ldots \rangle$ is labeled with a *classifier* such as α or β, and a variable or typing judgment is labeled with a *level*, or a sequence of classifiers. This program decomposes into the open redex

(2) Scheme: `(id ‘x)`
MetaOCaml: `id .<x>.`
λ^α: $\text{id}\,\langle x\rangle^\alpha$

and the evaluation context

(3) Scheme: `‘(lambda (x) ,[])`
MetaOCaml: `.<fun x -> .~[]>.`
λ^α: $\langle \lambda x^\alpha : \text{int}.\sim[\,]\rangle^\alpha$,

which binds the staged variable `x` or x. Further, the program below illustrates that an evaluation context may contain several bindings.

(4) Scheme:
```
‘(lambda (x)
   ‘(lambda (y) ,,(id ‘‘y)))
```
MetaOCaml:
```
.<fun x ->
  .<fun y -> .~(.~(id .<.<y>.>.))>.>.
```
λ^α: $\langle \lambda x^\alpha : \text{int}.(\beta)\langle \lambda y^{\alpha\beta} : \text{int}.\sim\sim(\text{id}\,\langle\langle y\rangle^\beta\rangle^\alpha)\rangle^\beta\rangle^\alpha$

Here the λ^α expression $(\beta)\ldots$ binds the classifier β. Hence the λ^α evaluation context $\langle \lambda x^\alpha : \text{int}.(\beta)\langle \lambda y^{\alpha\beta} : \text{int}.\sim\sim[\,]\rangle^\beta\rangle^\alpha$ binds the variable $x^\alpha : \text{int}$, the classifier β, and the variable $y^{\alpha\beta} : \text{int}$.

Because the label $\alpha\beta$ for y above *depends* on the bound classifier β, we face conflicting demands from evaluation and binding when we try to formalize evaluation contexts as an LF type. On one hand, in order for an evaluation context to be a defunctionalized continuation of a CPS evaluator or one-small-step reducer [2], we should represent it inside-out, with the part next to the hole most accessible. On the other hand, in order for β to bind into $y^{\alpha\beta} : \text{int}$ above, we should represent the evaluation context outside-in, with the program root most accessible.

Our mechanization of λ^α represents evaluation contexts outside-in. We thus take advantage of HOAS to ensure α-conversion invariance in a staged language [6], at the cost of obscuring the connection to defunctionalized continuations. We define in LF

1. the type `exp` of a closed expression (program) `E`,
2. the type `gs` of a typing environment `G`,
3. the type `oquals G` of an open level `A` bound by `G`,
4. the type `oexp A` of an open expression `O` at `A` bound by `G`,
5. the type `ctxb A` of a context `C` whose hole is at `A` and bound by `G`, and finally
6. the type `zip-up O C E` of plugging `O` into `C` to yield `E`.

We convinced Twelf that `%mode zip-up +O +C -E` is total. This totality suggests that these types adequately represent ordered dependent sequences of bindings, be they needed by an expression or provided by a context. These definitions let us specify the first small-step semantics for staging.

The challenge remains to represent contexts inside-out while expressing its binding structure, in particular how the continuation of a staged evaluator may "bind off" a later-stage variable. Possibly relevant are dependent types in CPS [1] and contexts in LF [4].

[1] Barthe, Gilles, John Hatcliff, and Morten Heine B. Sørensen. 1999. CPS translations and applications: The cube and beyond. *Higher-Order and Symbolic Computation* 12(2):125–170.

[2] Danvy, Olivier. 2004. On evaluation contexts, continuations, and the rest of the computation. In *CW'04: Proceedings of the 4th ACM SIGPLAN continuations workshop*, ed. Hayo Thielecke. Tech. Rep. CSR-04-1, School of Computer Science, University of Birmingham.

[3] Hammond, Kevin, and Greg Michaelson. 2003. Hume: A domain-specific language for real-time embedded systems. In *Proceedings of GPCE 2003: 2nd international conference on generative programming and component engineering*, ed. Frank Pfenning and Yannis Smaragdakis, 37–56. Lecture Notes in Computer Science 2830, Berlin: Springer-Verlag.

[4] Nanevski, Aleksandar, Frank Pfenning, and Brigitte Pientka. 2007. Contextual modal type theory. *Transactions on Computational Logic*. In press.

[5] Püschel, Markus, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE special issue on program generation, optimization, and adaptation* 93(2):232–275.

[6] Taha, Walid, and Michael Florentin Nielsen. 2003. Environment classifiers. In *POPL '03: Conference record of the annual ACM symposium on principles of programming languages*, 26–37. New York: ACM Press.

[7] Wright, Andrew K., and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* 115(1):38–94.