Mechanizing multilevel metatheory with control effects

Yukiyoshi Kameyama
University of Tsukuba
kameyama@acm.org

Oleg Kiselyov
FNMOC
oleg@pobox.com

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

Abstract

We have mechanized the type soundness proof for the first sound multilevel calculus with control effects. The calculus (an extension of [3]) lets us write direct-style generators that memoize open code. Our mechanization overcomes two challenges: first, to intrinsically encode an object calculus whose typing judgements involve non-trivial type functions; second, to represent open code and especially evaluation contexts containing variable binders. These challenges and the necessary small-step operational semantics recommend multilevel calculi with effects as a benchmark of mechanized metatheory.

1. Multilevel calculus with control effects

Our calculus $\lambda^{\oslash}$, Figure 1, extends the multilevel calculus $\lambda^{\bigcirc}$ [1] with delimited control operators. It is a call-by-value λ-calculus with integers, addition, pairs, fixpoint and the conditional, as usual. Expressions, values and contexts are annotated with a non-negative integer superscript denoting the level. (We may drop the superscript if it can be inferred.) Level 0 stands for the present stage, at which evaluation takes place. Future-stage computations, or "code", are built with the operations bracket $\langle e^{(i+1)} \rangle$ (the analogue of quasiquotation in Lisp) and escape $\sim e^i$ (the analogue of `unquote`). (These operations are called `next` and `prev` in $\lambda^{\bigcirc}$.) The calculus has the delimited control operator $\{e\}$ (pronounced "reset") and the higher-order constant 出 (pronounced "shift"). Whether an expression is a value depends on its level [4]. The calculus $\lambda^{\oslash}$ extends [3] to multiple future-stage levels.

The operational semantics, Figure 2, is *small-step*, as needed to express delimited control. (We elide the standard reductions for the pair projections, etc.) The captured continuation is built one frame F^i at a time, as the "bubble" created by the application of 出 percolates up [2]. The operational semantics and the context formation rule $C^j ::= C^i[\lambda x. D^{ij}]$ pose the first challenge: evaluation may occur under a future-stage binder λx, and the evaluation context C^i may contain binders. Therefore, we can build *open* code values, which contain free variables bound by the context. By capturing and removing a part of the context, the control operators could therefore remove variable binders from the context and thus produce code with unbound variables. The risk of such errors is why adding effects to a multilevel calculus is tricky.

Our calculus prevents such *scope extrusion* errors by restricting control effects to within the scope of a future-stage binder. Such a restriction still lets us express the standard benchmark problems of code generation [3]. To make sure that such a run-time restriction does not cause the evaluation to get stuck, we impose a type-and-effect system; Figure 3 shows the crucial parts. Control effects at each level are tracked with the help of an answer type. A typing judgment $\Gamma \vdash e : \tau \,;\, T_i$ for a level-i expression e includes the answer-type sequence T_i of length $i+1$. The arrow $\tau \to \tau'/\tau_0$ and code $\langle \tau/\tau_0 \rangle$ types are annotated with the answer type τ_0 describing the effect that may occur when applying the function or executing the code.

$$
\begin{aligned}
C^0\big[(\lambda x.\, e)\, v^0\big] &\leadsto C^0\big[e[x := v^0]\big] && (\beta_v)\\
C^0\big[\{v^0\}\big] &\leadsto C^0\big[v^0\big] && (\{\})\\
C^1\big[\sim\langle v^1\rangle\big] &\leadsto C^1\big[v^1\big] && (\sim)\\
C^0\big[\{\text{出} v^0\}\big] &\leadsto C^0\big[\{v^0(\lambda y.\, y)\}\big] &&\\
C^i\big[F^i[\sim^i(\text{出} v^0)]\big] &\leadsto C^i\big[\sim^i(\text{出}(\lambda k.\, v^0(\lambda y.\, \{k\langle F^i[\sim^i y]\rangle^i\})))\big] &&\\
C^i\big[\langle \sim^{i+1}(\text{出} v^0)\rangle\big] &\leadsto C^i\big[\sim^i(\text{出} v^0)\big] &&\\
C^i\big[\lambda x.\, \sim^i(\text{出} v^0)\big] &\leadsto C^i\big[\lambda x.\, \sim^i\{\text{出} v^0\}\big] \quad \text{where } i \geq 1 &&
\end{aligned}
$$

Figure 2. Operational semantics: small-step reduction $e \leadsto e'$. Here $\langle\ \rangle^i$ and $\sim^i$ stand for i levels of brackets and escapes; $i \geq 0$. On the right, y and k are fresh.

Types	$\tau ::= \mathsf{int} \mid \tau \to \tau'/\tau_0 \mid \langle \tau/\tau_0 \rangle \mid (\tau, \tau')$
Answer-type sequences	$T_i ::= \tau_0, \ldots, \tau_i$
Judgments	$\Gamma \vdash e : \tau \,;\, T_i$
Environments	$\Gamma ::= [] \mid \Gamma, \langle x : \tau \rangle^i$

$$
\frac{\Gamma, \langle x:\tau\rangle^i \vdash e : \tau' \,;\, \langle \tau'/\tau_i' \rangle^{\langle i \rangle}, \langle \tau'/\tau_i' \rangle^{\langle i-1 \rangle}, \ldots, \langle \tau'/\tau_i' \rangle^{\langle 1 \rangle}, \tau_i'}{\Gamma \vdash (\lambda x.\, e) : \tau \to \tau'/\tau_i' \,;\, T_i}
$$

$$
\frac{\Gamma \vdash e : \tau_i \,;\, T_{i-1}, \tau_i}{\Gamma \vdash \{e\} : \tau_i \,;\, T_{i-1}, \tau_i'}
\qquad
\frac{\Gamma \vdash e : \tau \,;\, T_i, \tau_{i+1}}{\Gamma \vdash \langle e \rangle : \langle \tau/\tau_{i+1} \rangle \,;\, T_i}
\qquad
\frac{\Gamma \vdash e : \langle \tau/\tau_{i+1} \rangle \,;\, T_i}{\Gamma \vdash {\sim} e : \tau \,;\, T_i, \tau_{i+1}}
$$

Figure 3. The type system of $\lambda^{\oslash}$ and selected typing rules. The notation $\tau^{\langle i \rangle}$ is inductively defined by $\tau^{\langle 1 \rangle} = \tau$, $\tau^{\langle i+1 \rangle} = \langle \tau^{\langle i \rangle} / \tau^{\langle i \rangle} \rangle$.

The most interesting typing rule, the first one in Figure 3, is for future-stage abstraction. A level-i λ restricts the scope of control effects at levels 0 through $i-1$ (inclusive). This restriction explains the quite involved answer-type sequence for the body of the λ.

2. Intrinsic encoding into LF

We use intrinsic encoding[1] to embed $\lambda^{\oslash}$ in Twelf. The expressions of $\lambda^{\oslash}$ are represented by the LF type family of the signature `exp: tp -> atp -> type`. This type family is parameterized by the $\lambda^{\oslash}$ type `tp` and by the answer-type sequence `atp`, with constructors `at0: tp -> atp` and `at: tp -> atp -> atp`. The length of the answer-type sequence is the level of the expression. Given

[1] `http://twelf.plparty.org/wiki/Intrinsic_encoding`

$$
\begin{array}{lll}
\text{Variables} & x,y,z,f,k & \\
\text{Expressions} & e ::= n \mid e+e \mid \lambda x.\,e \mid \mathsf{fix} \mid ee \mid (e,e) \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{ifz}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \text{出} \mid \{e\} \mid \langle e\rangle \mid {\sim}e \mid x & \\
\text{Values} & v^i ::= n \mid \mathsf{fix} \mid (v^i,v^i) \mid \mathsf{fst} \mid \mathsf{snd} \mid \text{出} \mid \langle v^{i+1}\rangle \mid x \qquad v^0 \mathrel{+}= \lambda x.\,e & \\
 & v^i \mathrel{+}= v^i+v^i \mid \lambda x.\,v^i \mid v^i v^i \mid \mathsf{ifz}\ v^i\ \mathsf{then}\ v^i\ \mathsf{else}\ v^i \mid \{v^i\} & \text{when } i \geq 1 \\
 & v^i \mathrel{+}= {\sim}v^{i-1} & \text{when } i \geq 2 \\
\text{Frames} & F^i ::= \square+e \mid v^i+\square \mid \square e \mid v^i\square \mid (\square,e) \mid (v^i,\square) \mid \mathsf{ifz}\ \square\ \mathsf{then}\ e\ \mathsf{else}\ e & \\
 & F^i \mathrel{+}= \mathsf{ifz}\ v^i\ \mathsf{then}\ \square\ \mathsf{else}\ e \mid \mathsf{ifz}\ v^i\ \mathsf{then}\ v^i\ \mathsf{else}\ \square \mid \{\square\} & \text{when } i \geq 1 \\
\text{Delimited contexts} & D^{ij} ::= D^{ij}[F^j] \mid D^{i(j+1)}[{\sim}\square] \qquad D^{ii} \mathrel{+}= \square & \\
 & D^{ij} \mathrel{+}= D^{i(j-1)}[\langle\square\rangle] & \text{when } j \geq 1 \\
\text{Contexts} & C^j ::= D^{0j} \mid C^0[\{D^{0j}\}] \mid C^i[\lambda x.\,D^{ij}] & \text{when } i \geq 1
\end{array}
$$

Figure 1. Values and contexts of $\lambda^{\oslash}$. We write `+=` to add alternatives to a preceding BNF rule.

below is a sample of `exp` constructors: addition, bracket `^`, escape `~`, 出, reset at the present `?` and future `?+` levels, present- `l` and future-stage `l+` abstractions.

```
+    : exp int A -> exp int A -> exp int A.
^    : exp T1 (at Ta A) -> exp (& T1 Ta) A.
~    : exp (& T1 Ta) A  -> exp T1 (at Ta A).
de   : exp (arr (arr (arr T Ta Ta) Ta Ta) T Ta) A.

?    : exp T (at0 T)  -> exp T (at0 _).
?+   : exp T (at T A) -> exp T (at _ A).

l    : (arg T1 0 -> exp T2 (at0 T2a)) ->
       exp (arr T1 T2 T2a) (at0 _).

l+   : l+-cnt N (& T2 T2a) A -> polyA N AR
       -> (arg T1 N -> exp T2 (at T2a A))
       -> exp (arr T1 T2 T2a) AR.
```

Because expressions are annotated with their $\lambda^{\oslash}$ types, these definitions encode not only the syntax of $\lambda^{\oslash}$ but also its type system (cf. Figure 3). The notation `& T Ta` stands for the code type $\langle\tau/\tau_a\rangle$ and `arr T1 T2 Ta` is the arrow type with the answer type `Ta`.

The first challenge is encoding abstractions of $\lambda^{\oslash}$. Since the calculus is call-by-value, bound variables are substituted by values, which are answer-type polymorphic. It is enough therefore to annotate a bound variable, beside its type, with its level rather than the full answer-type sequence. The type family `arg: tp -> nat -> type` is such a representation for bound variables. The main challenge comes from the complexity of the typing rule for the future-stage abstraction, Figure 3. We have to encode the non-trivial type computations of that rule as part of the `l+`-expression. One such computation is determining the answer-type sequence for the abstraction's body, using the inductive function $\tau^{\langle i\rangle}$. We define an auxiliary family `l+-cnt` to represent this computation. The type family `polyA N AR` indexes the abstraction by a sequence `AR` of `N` fresh answer types.

The second challenge is representing open code and binding evaluation contexts, both arising from the evaluation under a future-stage λ. LF worlds and hypothetical reasoning make the challenge easy to meet. Since we use higher-order abstract syntax for $\lambda^{\oslash}$ binders, the body of a `l+` is a function of the type `arg T1 N -> exp T2 (at T2a A)`. To evaluate that body, we hypothesize an LF term `{x:arg T (1 _)}` standing for the bound variable, pass that term to the body of the function, and evaluate the resulting `exp`. Thus we represent the evaluation context of $\lambda^{\oslash}$ as LF evaluation context, and the $\lambda^{\oslash}$ bindings in that context as components of the LF world.

The advantage of the intrinsic encoding is that all $\lambda^{\oslash}$ expressions we can enter in Twelf are well-typed by construction, and the types are inferred by Twelf. The latter property saves us from writing our own type checker.

We have mechanized the proofs of the following (meta)theorems of $\lambda^{\oslash}$:

1. values are answer-type polymorphic at level 0;
2. each expression is either a value, a continuation bubble, or decomposable into an evaluation context and a pre-redex;
3. reductions preserve types (subject reduction);
4. a well-typed non-value can be reduced (progress).

The complete Twelf development along with several examples is available at `http://okmij.org/ftp/Computation/staging/README.dr`.

3. Open questions

We are working on extending $\lambda^{\oslash}$, dropping the restriction on control effects, so to permit moving code past the binders (for example, moving loop-invariant code out of the `for`-loop body). The context captured by a control operator may now include binders. How to represent such contexts?

We have used the bubble-up operational semantics for control operators, which builds the captured continuation one frame at a time. Proving bi-simulation with the CPS-transformed code is greatly facilitated by the semantics that captures the prefix of the current continuation in one step. For such a one-fell-swoop capture, representing contexts inside-out is most appropriate. Alas, it is not known how to represent binding contexts (contexts with binders) in the inside-out fashion.

References

[1] Davies, Rowan. 1996. A temporal logic approach to binding-time analysis. In *LICS*, 184–195.

[2] Felleisen, Matthias, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. 1986. Reasoning with continuations. In *Proceedings of the 1st symposium on logic in computer science*, 131–141.

[3] Kameyama, Yukiyoshi, Oleg Kiselyov, and Chung-chieh Shan. 2009. Shifting the stage: Staging with delimited control. In *Proceedings of the 2009 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, ed. Germán Puebla and Germán Vidal, 111–120. New York: ACM Press.

[4] Taha, Walid, and Michael Florentin Nielsen. 2003. Environment classifiers. In *POPL*, 26–37.