Mechanizing multilevel metatheory with control effects

Yukiyoshi Kameyama — University of Tsukuba
Oleg Kiselyov
Chung-chieh Shan — Rutgers University

WMM5 – Workshop on Mechanizing Metatheory
September 25, 2010

Outline

We begin with the introduction of the programming language/calculus whose meta-theory we wish to mechanize. The calculus is the familiar call-by-value lambda-calculus with a non-trivial combination of two features, which we illustrate with sample reductions.

Staging

$$\begin{array}{lll} & (\lambda x.\,\langle 30 + {\sim}x\rangle)\ \langle 12\rangle & \\ \rightsquigarrow & \langle 30 + {\sim}\langle 12\rangle\rangle & \\ \rightsquigarrow & \langle 30 + 12\rangle & \text{value} \end{array}$$

$\langle 30 + 12\rangle$	‘Compiled program’
$\langle\langle 30 + 12\rangle\rangle$	Compiled compiler

The first feature is staging. One may think of brackets $\langle \cdot \rangle$ as Lisp's quasi-quote and escape $\sim\cdot$ as an unquote. Code expressions without escapes are values. The first reduction is just a beta substitution of a value. We then do splicing, obtaining the code value $\langle 30 + 12 \rangle$. It is the value. One may think of it as a compiled program. In the future life, when the compiled code is run, we get a well-known value. We can have more than one future life: we can program not only code generators but generators of generators, etc.

Delimited Control

$$
\begin{array}{rc}
 & \{42\} \\
\rightsquigarrow & 42 \\
 & \{1 + \text{出}(\lambda k.\, 14 * k2)\} \\
\equiv & \{1 + \underline{(\lambda k.\, 14 * k2)\square}\} \\
\rightsquigarrow & \{\underline{(\lambda k.\, 14 * k2)(1 + \square)}\} \\
\rightsquigarrow & (\lambda k.\, 14 * k2)(\lambda y.\, 1 + y) \\
\rightsquigarrow & 14 * (\lambda y.\, (1 + y))2 \\
\rightsquigarrow & 42
\end{array}
$$

$\{\cdot\}$ marks the “outside”

The next feature of our language is delimited control, expressed by a special form $\{\cdot\}$ and a higher-order constant 出. One may think of reset as OCaml's `try` block, 出 sort of as a raising of an exception. If an expression raises no exception (for example, it is a value), then enclosing it into reset has no effect.

If the evaluation of an expression comes across a shift application, it creates a bubble, consisting of shift's argument and a context, initially empty. On the figure the bubble is notated with the underline. The bubble propagates up devouring its context. Eventually the bubble comes across reset, which pricks the bubble. It takes the accumulated context from bubble's stomach, converts it to a function and feeds to the argument of shift, which has been carried along. It is a run-time error if the bubble does not find a reset.

Delimited Control

$$
\begin{array}{rc}
 & \{42\} \\
\rightsquigarrow & 42 \\
 & \{1 + 出(\lambda k.\, 14 * k2)\} \\
\equiv & \{1 + \underline{(\lambda k.\, 14 * k2)\square}\} \\
\rightsquigarrow & \{\underline{(\lambda k.\, 14 * k2)(1 + \square)}\} \\
\rightsquigarrow & (\lambda k.\, 14 * k2)(\lambda y.\, 1 + y) \\
\rightsquigarrow & 14 * (\lambda y.\, (1 + y))2 \\
\rightsquigarrow & 42 \\
= & 14 * \{1 + 2\}
\end{array}
$$

$\{\cdot\}$ marks the "outside"

The end result looks like the whole body of the shift application gets replaced by 2, with the multiplication by 14 got “outside”. Reset marks the “outside”.

Staging and Delimited Control

Inserting `let`, `assert`, `if`, `import`,... outside

$$
\begin{array}{rl}
 & \{\langle\, \mathit{expensive}\ x + {\sim}(\ 出 \lambda k \langle \mathsf{assert}\ x \neq 0; {\sim}(k\langle 1/x\rangle)\ \rangle\ )\ \rangle\} \\
\rightsquigarrow^{+} & \langle \mathsf{assert}\ x \neq 0; \mathit{expensive}\ x + 1/x \rangle
\end{array}
$$

Hopefully the example gave a hint why combining delimited control with staging is useful. We can write code generators, which, deep in the midst of generating a sub-expression, could insert `let`, `import`, `if` and other statements somewhere “outside”.

Suppose we are generating this code fragment, adding two expressions. We have already generated the first expression and are about to generate the second one. The second expression turns out to be the reciprocal of x, which fails if x is zero. But it fails after we spend a lot of time on the expensive computation. We can write the generator for the second summand using shift. We obtain the desired result: we check for non-zero *x before* embarking on the expensive computation.

Staging and Delimited Control

Inserting `let`, `assert`, `if`, `import`,... outside

$$\{\langle\ \mathit{expensive}\ x + \sim(\ 出 \lambda k \langle \mathsf{assert}\ x \neq 0; \sim(k\langle 1/x\rangle)\ \rangle\ )\ \rangle\}$$

$$\leadsto^{+} \quad \langle \mathsf{assert}\ x \neq 0; \mathit{expensive}\ x + 1/x\rangle$$

$$\langle \lambda x \sim($$ Binding context

$$\{\langle\ \mathit{expensive}\ x +$$ Open code

$$\sim(\ 出 \lambda k \langle \mathsf{assert}\ x \neq 0; \sim(k\langle 1/x\rangle)\ \rangle\ )\ \rangle\}$$

$$)\rangle$$

$$\leadsto^{+} \quad \langle \lambda x\ \mathsf{assert}\ x \neq 0; \mathit{expensive}\ x + 1/x\rangle$$

You may be wondering about *x*. Is it a free variable? Actually, it is. Here is our code in context. The evaluation context is *binding*, and the the evaluated code is `open`. That is the major challenge posed by staging.

Outline

Calculus: $\lambda^{\bigcirc}$ with control effects

▶ **Application**

Problems

Solution and Mechanization

Conclusions

Code Generation in HPC

ATLAS

used in MAPLE, MATLAB, Mathematica, Octave, Absoft Pro Fortran, GSL, LAPACK, Scilab, . . .

SPIRAL

Why do we care about code generation? One application is high-performance computing. On modern superscalar architectures with multiple levels of caches, nobody except some Intel employees can predict the performance. If we aim at the highest performance, the only choice is to generate several candidates, evaluate then on sample data and pick the fastest. These famous generators of highest-performance numerical code (used by everybody) do exactly that.

We certainly want some of that fame. Mainly, we want to claim that we generate correct code. These famous code generators want to claim that to. It is very difficult to see that they do indeed generate the correct code – even to the authors of these packages.

Writing Code Generators is Hard

“As you have seen, this note and the protocols it describes have plenty of room for improvement. Now, as the end-user of this function, you may have a naturally strong and negative reaction to these crude mechanisms, tempting you to send messages decrying my lack of humanity, decency, and legal parentage to the atlas or developer mailing lists. . . . So, the proper bitch format involves

- *First thanking me for spending time in hell getting things to their present crude state*
- Then, supplying your constructive ideas”

`math-atlas.sourceforge.net/devel/atlas_contrib/`
R. Clint Whaley: User contribution to ATLAS.

Here is a good quote, from the concluding section of this document, written by the ATLAS developer Clint Whaley. As you can see, he is well aware that ATLAS leaves much room for improvement. He also says that even getting this far has been very difficult.

So, our goal is to generate correct code without going to hell.

Outline

Calculus: $\lambda^{\bigcirc}$ with control effects

Application

▶ **Problems**

Solution and Mechanization

Conclusions

Scope Extrusion and its Prevention

$$
\begin{array}{rl}
 & \{\langle\ (\lambda x.\,{\sim}(\text{出}\,\lambda k.\,\langle x\rangle))\ 1\ \rangle\} \\
\rightsquigarrow^{+} & \langle x_3\rangle
\end{array}
$$

Combining staging and delimited control is non-trivial

Let's start with confessions. ATLAS generated C code using printf; it couldn't even statically assure that the parentheses match. By using a staged language, we are certain that the generated code is well-formed. Delimited control lets us conveniently and modularly write generators that seemingly require several generation passes. But adding delimited control to a staged language is non-trivial, because it is possible to screw up. On the slide you see generating body of a future-stage lambda. The generator aborts with $\langle x \rangle$. The result is nonsense: a code value with an unbound variable. One can indeed obtain such a value in MetaOCaml, for example, exactly as shown on the slide. This is bad.

Scope Extrusion and its Prevention

$$\begin{aligned} & \{\langle\, (\lambda x.\, \sim(\{\langle \sim(\text{出}\,\lambda k.\,\langle x\rangle)\rangle\})\; 1\,\rangle\} \\ \rightsquigarrow^{+}\; & \langle(\lambda x_3.\, x_3)\; 1\rangle \end{aligned}$$

Combining staging and delimited control is non-trivial

The work with Kameyama-san and Chung-chieh Shan suggested a restriction: let us assume that each future-stage lambda contains an implicit reset. The slide shows the second-level lambda; the third-level lambda will have two implicit resets, etc. Effects are allowed when building code, but they are restricted in scope. We have verified that the restriction is not severe and we can build a lot of practically interested generators. But is it always sound? That's where the formalization and mechanization comes in. BTW, the answer is yes.

Type and effect system

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

$$\frac{\Gamma, \langle x : \tau \rangle^i \ \vdash e : \tau' \ ; \langle \tau' / \tau_i' \rangle^{[i]}, \tau_i'}{\Gamma \vdash (\lambda x.\, e) : \tau \rightarrow \tau' / \tau_i' \ ; T_i}$$

Type functions

$$\tau^{[i]} = \tau^{\langle i \rangle}, \tau^{\langle i-1 \rangle}, \ldots, \tau^{\langle 1 \rangle}$$

$$\tau^{\langle 1 \rangle} = \tau$$

$$\tau^{\langle i+1 \rangle} = \langle \tau^{\langle i \rangle} / \tau^{\langle i \rangle} \rangle$$

We have already illustrated the interesting parts of the dynamic semantics. We now show interesting bits of the type-and-effect system, ensuring that well-typed expressions do not get stuck. The effect annotation takes the form of an answer-type, the type of the exception thrown by shift, if you will. We effect-annotate arrows and code types; actually, each level of the code. That's why we have a stack on answer-types, for each level. Judgments too bear annotations, the stack of answer-types. Remember that implicit reset we put under lambdas? That is the reason the answer-type of a future-stage lambda is so involved, with several type-functions (T_i in the conclusion, is a type function too, which is a function producing a fresh sequence). And we have to encode all this in mechanization!

Outline

Challenges of mechanization

1. Typing rules with complex (inductive) type functions
2. Open code and binding context
3. Small-step semantics

Intrinsic Encoding

```
exp: tp -> atp -> type.
+ : 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).


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

Expressions are well-typed by construction
Twelf checks and infers object types for us.

We use the intrinsic encoding. Expressions are represented by the LF family `exp`; as we see, the representation of object expressions includes the object type and the stack of answer-types. It is not possible to represent an ill-typed expression. Expression constructors show not only the syntax of our language, but also encode the type system. The first rule says that the plus operator takes two integer expressions at the same level and produces an integer expression at the same level. The level is arbitrary.

I did not show the typing rules for brackets and escapes, but we can read them directly off the Twelf code. We see the bracket takes an expression at a higher level and produces an expression at the lower level, tucking the answer-type in. The escape does the reverse.

And here is the scaring-looking rule from the slide, for a future stage lambda. The last two lines show the standard higher-order abstract syntax representation of object-level lambdas. The first line contains the type functions I've been talking about, which compute the resulting answer-type sequence `AR` and the answer-type sequence for lambda's body, `A`.

Theorems proved

1. Values are answer-type polymorphic at level 0
2. Expression decomposition lemma
3. Subject reduction
4. Progress

We proved that present-stage values are answer-type polymorphic, that each expression is either a value, a continuation bubble, or decomposable into an evaluation context and a pre-redex. Mainly, we proved that reductions preserve types (subject reduction) and that a well-typed non-value can be reduced (progress).

Progress Theorem

```
eval : {E: exp T A} non-value E -> exp T A -> type.
%mode eval +E1 +NV -E2.
```

We see from the type of eval that it is a witness of progress: given an expression that is not a value, it returns another, reduced, expression – *of the same type*.

The same eval is also the interpreter for our calculus, which we used to run test examples.

Progress Theorem

```
eval : {E: exp T A} non-value E -> exp T A -> type.
%mode eval +E1 +NV -E2.
ev-+: eval ((n N1) + (n N2)) (nv-+ _ _) (n N3)
      <- plus N1 N2 N3.
ev-+C1: eval (E1 + E2) (nv-+C1 NV1) (E1' + E2)
      <- eval E1 NV1 E1'.
ev-+R1: eval ((deru TS C E) + E2) nv-+R1
             (deru TS ([x] Down (Up (C x) + E2)) E)
      <- ts-coercions TS Down Up.
```

The implementation of eval is the encoding of dynamic semantics. Here, for example, are the reduction rules for addition. Now Twelf makes sure that each and every reduction rule satisfies the type for eval, that is, reductions preserve types. We can't even write a reduction rule that does not preserve a type!

Progress Theorem

```
eval : {E: exp T A} non-value E -> exp T A -> type.
%mode eval +E1 +NV -E2.


ev-l+C: eval (l+ LC P E) (nv-l+C NV) (l+ LC P E')
     <- {x:arg T (1 _)} eval (E x) (NV x) (E' x).

%block bl-ev : some {T:tp}{N:nat} block {e:arg T (1 N)}.

%worlds (bl-ev) (eval _ _ _).
```

After a few more lines, we get to the end of the proof. Recall our first challenge: evaluating open code and representing binding contexts. LF worlds and hypothetical reasoning made the challenge easy to meet. But I have to show the reduction rule for the future lambda. As you can see, we use hypothetical reasoning to “assume” the expression bound to a variable and evaluate the body of lambda based on that assumption.

These two lines, the block and the world declarations, confirm that do evaluate open code, which may contain many future-level bindings.

Progress Theorem

```
eval : {E: exp T A} non-value E -> exp T A -> type.
%mode eval +E1 +NV -E2.


ev-l+C: eval (l+ LC P E) (nv-l+C NV) (l+ LC P E')
      <- {x:arg T (1 _)} eval (E x) (NV x) (E' x).

%block bl-ev : some {T:tp}{N:nat} block {e:arg T (1 N)}.

%worlds (bl-ev) (eval _ _ _).

%total {E} (eval E _ _).
```

The final line asserts the `eval` type family is total: all expressions that are non-values can be reduced, in final time, to some expression. So, we have type soundness.

Outline

Calculus: $\lambda^{\bigcirc}$ with control effects

Application

Problems

Solution and Mechanization

▶ **Conclusions**

Conclusions

staging $\vee$ shift $=$ trouble $\vee$ fun

Mechanization of the type soundness proof for the first sound multilevel calculus with control effects

- Mechanization was (relatively) easy and even fun
- Intrinsic encoding works despite complex typing rules
- LF worlds and hypothetical reasoning help with open code and binding contexts
  (cf. type checking)
- Multi-level calculi with effects make a good benchmark

`http://okmij.org/ftp/formalizations/`

Combining staging and delimited control is interesting and non-trivial. We have mechanized the type soundness proof for the first sound multilevel calculus with control effects.
It was enjoyable as far mechanization could go. Intrinsic encoding works out even in the case like ours with complex typing rules involving type functions.
In fact, type checking, which is a form of evaluation, routinely deals with open code and binding contexts. We do type-check under lambda. It is that evaluation normally doesn't occur under lambda. So, staged evaluation does look like type checking in this respect.
The complete Twelf development along with several examples is available at the shown URL.

Open questions

- Capturing a genuine *binding* context?
- Representing a binding context inside-out?
- Status of Twelf development?