Compilation Mechanized Metatheory Challenge

Adam Chlipala (`adamc@cs.berkeley.edu`)

September 11, 2006

1 Motivation

This is a challenge problem that came up in a project that I'm working on, and that I had to think about for quite a while to get reasonably right, even with all of the data that's come out of the submitted solutions to the first POPLmark challenge. The problem falls under the general category of reasoning about compilers, and I think it has the right combination of smallness and interestingness to serve as a good benchmark for this kind of task.

The first POPLmark challenge problem "hardcodes" a number of decisions about how the object language and its semantics should be represented. Some mailing list participants have argued that it's important to be flexible about representations when you think about translating informal proofs into mechanized proofs. This challenge is stated in that spirit, attempting to be just precise enough that it's clear what the challenge involves. Coming up with patterns for effective formal representation is a first-class part of the challenge.

It's also seemed like none of the submissions to the first challenge really have the property of being especially human-readable. I've tried to meet the lofty goal of readability in my own solution to this new challenge, and I hope that anyone else proposing a solution can try to do the same.

The basic progression of this challenge is this:

- Define a simply-typed lambda calculus extended with product and sum types.
- Formalize its static and dynamic semantics.
- Prove correctness theorems about some simple object-language programs, to see how easy it is to reason about your semantics with your chosen proof tool.
- Define an imperative lambda calculus for infinite, untyped heaps.
- Formalize its static and dynamic semantics.
- Try some more small correctness proof tasks for the imperative language.

- Define a compiler from the functional source language into the imperative target language, where aggregate data types must be represented using the untyped heap.
- Prove that your compiler is semantics-preserving.

2 The Source Language: $\lambda^{\times+}$

2.1 Syntax

The source language (called $\lambda^{\times+}$) is simply-typed lambda calculus plus product and sum types, with one restriction to avoid making the compilation task too hard: the types of "data" are segregated from types in general, so that we can think about compiling "data" into an untyped heap-based representation while leaving "code" alone. This means that there's no need to implement closure conversion, and the challenge is already interesting enough without that.

$$\begin{array}{rrcl}
\text{Data types} & \sigma & ::= & \mathbb{N} \mid \sigma \times \sigma \mid \sigma + \sigma \\
\text{Types} & \tau & ::= & \sigma \mid \tau \to \tau \\
\\
\text{Variables} & x & & \\
\text{Constants} & n & \in & \mathbb{N} \\
\text{Terms} & e & ::= & x \mid \lambda x : \tau^{I}.\ e \mid e\ e \\
 & & & \mid\ n \\
 & & & \mid\ (e, e) \mid \pi_1 e \mid \pi_2 e \\
 & & & \mid\ \mathsf{inl}_\sigma\ e \mid \mathsf{inr}_\sigma\ e \mid (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}_\sigma\ x \Rightarrow e \mid \mathsf{inr}_\sigma\ x \Rightarrow e)
\end{array}$$

Challenge 1. *Formalize the abstract syntax of* $\lambda^{\times+}$.

2.2 Static Semantics

Here's a standard static semantics that I don't think needs any explanation. A formalization of this semantics needn't be in the form of typing judgments separate from the object language syntax; my solution uses dependently-typed ASTs to combine syntax and static semantics.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{}{\Gamma \vdash n : \mathbb{N}}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash (e_1, e_2) : \sigma_1 \times \sigma_2} \quad \frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash \pi_1 e : \sigma_1} \quad \frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash \pi_2 e : \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma_1}{\Gamma \vdash \mathsf{inl}_{\sigma_2}\ e : \sigma_1 + \sigma_2} \quad \frac{\Gamma \vdash e : \sigma_2}{\Gamma \vdash \mathsf{inr}_{\sigma_1}\ e : \sigma_1 + \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma_1 + \sigma_2 \quad \Gamma, x_1 : \sigma_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \sigma_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}_{\sigma_2}\ x_1 \Rightarrow e_1 \mid \mathsf{inr}_{\sigma_1}\ x_2 \Rightarrow e_2 : \tau}$$

Challenge 2. *Formalize the static semantics of* $\lambda^{\times+}$.

2.3 Dynamic Semantics

Here's a standard call-by-value big-step operational semantics for $\lambda^{\times+}$. A formalization needn't use operational semantics at all; my solution uses denotational semantics.

$$\frac{}{\lambda x : \tau.\ e \Downarrow \lambda x : \tau.\ e} \quad \frac{e_1 \Downarrow \lambda x : \tau.\ e \quad e_2 \Downarrow e_2' \quad e[x \mapsto e_2'] \Downarrow e'}{e_1\ e_2 \Downarrow e'}$$

$$\frac{}{n \Downarrow n}$$

$$\frac{e_1 \Downarrow e_1' \quad e_2 \Downarrow e_2'}{(e_1, e_2) \Downarrow (e_1', e_2')} \quad \frac{e \Downarrow (e_1, e_2)}{\pi_1 e \Downarrow e_1} \quad \frac{e \Downarrow (e_1, e_2)}{\pi_2 e \Downarrow e_2}$$

$$\frac{e \Downarrow e'}{\mathsf{inl}_\sigma\ e \Downarrow \mathsf{inl}_\sigma\ e'} \quad \frac{e \Downarrow e'}{\mathsf{inr}_\sigma\ e \Downarrow \mathsf{inr}_\sigma\ e'}$$

$$\frac{e \Downarrow \mathsf{inl}_{\sigma_2}\ e' \quad e_1[x_1 \mapsto e'] \Downarrow e''}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}_{\sigma_2}\ x_1 \Rightarrow e_1 \mid \mathsf{inr}_{\sigma_1}\ x_2 \Rightarrow e_2 \Downarrow e''} \quad \frac{e \Downarrow \mathsf{inr}_{\sigma_1}\ e' \quad e_2[x_2 \mapsto e'] \Downarrow e''}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}_{\sigma_2}\ x_1 \Rightarrow e_1 \mid \mathsf{inr}_{\sigma_1}\ x_2 \Rightarrow e_2 \Downarrow e''}$$

Challenge 3. *Formalize the dynamic semantics of* $\lambda^{\times+}$.

Challenge 4 (termination of evaluation)**.** *Prove that your formalization of* $\Downarrow$*:*

- *Is a function*
- *Is a* total *function when the domain is restricted to well-typed closed terms*

This challenge is a piece of work in itself for most formalization techniques used in solutions to the first POPLmark challenge, so it would be reasonable to skip it in favor of the more interesting parts to come. However, my solution (based on denotational semantics) inherits these theorems “for free” because it's given as a translation into a strongly normalizing language, so I thought I'd include this for comparison purposes.

2.4 Correctness Proof Challenges

Challenge 5 ("snd"). *For any $n_1, n_2 \in \mathbb{N}$, $(\lambda x : \mathbb{N}.\ \lambda y : \mathbb{N}.\ y)\ n_1\ n_2 \Downarrow n_2$.*

Challenge 6 ("dup"). *For any $n \in \mathbb{N}$, $\pi_1((\lambda x : \mathbb{N}.\ (x, x))\ n) \Downarrow n$.*

Challenge 7 ("JM_case"). *For any e such that $\cdot \vdash e : \mathbb{N} + \mathbb{N}$, $(\lambda x : \mathbb{N} + \mathbb{N}.\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}_{\mathbb{N}}\ y \Rightarrow 42 \mid \mathsf{inr}_{\mathbb{N}}\ z \Rightarrow 42)\ e \Downarrow 42$.*

3 Target Language: λ^I

3.1 Syntax

The target language (called λ^I) is simply-typed lambda calculus plus imperative access to an infinite, untyped heap. The heap is addressed by natural numbers and stores only natural numbers, which we've been including all along as our base type. The imperative expressions $!e^I$, $e^I := e^I$, and $e^I ; e^I$ are modeled on ML syntax, though here we are dealing with untyped references, and the "ref" type is actually just $\mathbb{N}$. There is no equivalent to the ML "ref" expression, as individual programs are responsible for their own allocation and memory management.

Additionally, we add the natural number successor operation S, because we're going to need it to do the arithmetic behind manual memory management; and an if expression that tests naturals, since we've lost case.

This language is designed to share characteristics with an intermediate language used in the last stages of a compiler for a functional language.

$$\begin{array}{rrcl}
\text{Data types} & \sigma^I & ::= & 1 \mid \mathbb{N} \\
\text{Types} & \tau^I & ::= & \sigma^I \mid \tau^I \to \tau^I \\
\\
\text{Variables} & x & & \\
\text{Constants} & n & \in & \mathbb{N} \\
\text{Terms} & e^I & ::= & x \mid \lambda x : \tau^I.\ e^I \mid e^I\ e^I \\
 & & & \mid () \\
 & & & \mid\ n \mid \mathsf{S}\ e^I \\
 & & & \mid !e^I \mid e^I := e^I \mid e^I ; e^I \\
 & & & \mid \mathsf{if}\ e^I\ \mathsf{then}\ e^I\ \mathsf{else}\ e^I
\end{array}$$

Challenge 8. *Formalize the abstract syntax of λ^I.*

3.2 Static Semantics

The static semantics for λ^I is again entirely standard.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash^I x : \tau} \qquad \frac{\Gamma, x : \tau_1^I \vdash^I e^I : \tau_2^I}{\Gamma \vdash^I \lambda x : \tau_1^I.\ e^I : \tau_1^I \to \tau_2^I} \qquad \frac{\Gamma \vdash^I e_1^I : \tau_1^I \to \tau_2^I \quad \Gamma \vdash^I e_2^I : \tau_1^I}{\Gamma \vdash^I e_1^I\ e_2^I : \tau_2^I}$$

$$\frac{}{\Gamma \vdash^I () : 1}$$

$$\frac{}{\Gamma \vdash^I n : \mathbb{N}} \quad \frac{\Gamma \vdash^I e^I : \mathbb{N}}{\Gamma \vdash^I \mathsf{S}\ e^I : \mathbb{N}}$$

$$\frac{\Gamma \vdash^I e^I : \mathbb{N}}{\Gamma \vdash^I !e^I : \mathbb{N}} \quad \frac{\Gamma \vdash^I e_1^I : \mathbb{N} \quad \Gamma \vdash^I e_2^I : \mathbb{N}}{\Gamma \vdash^I e_1^I := e_2^I : 1} \quad \frac{\Gamma \vdash^I e_1^I : 1 \quad \Gamma \vdash^I e_2^I : \tau^I}{\Gamma \vdash^I e_1^I ; e_2^I : \tau^I}$$

$$\frac{\Gamma \vdash^I e^I : \mathbb{N} \quad \Gamma \vdash^I e_1^I : \tau^I \quad \Gamma \vdash^I e_2^I : \tau^I}{\Gamma \vdash^I \mathsf{if}\ e^I\ \mathsf{then}\ e_1^I\ \mathsf{else}\ e_2^I : \tau^I}$$

Challenge 9. *Formalize the static semantics of* λ^I.

3.3 Dynamic Semantics

We have another standard call-by-value big-step operational semantics for λ^I. Since we're working with an imperative language, the semantics is defined over pairs of memories and terms.

$$\frac{}{(m, \lambda x : \tau^I .\ e^I) \Downarrow^I (m, \lambda x : \tau^I .\ e^I)}$$

$$\frac{(m, e_1^I) \Downarrow^I (m', \lambda x : \tau^I .\ e^I) \quad (m', e_2^I) \Downarrow^I (m'', e_3^I) \quad (m'', e^I[x \mapsto e_3^I]) \Downarrow^I (m''', e_4^I)}{(m, e_1^I\ e_2^I) \Downarrow^I (m''', e_4^I)}$$

$$\frac{}{(m, ()) \Downarrow^I (m, ())}$$

$$\frac{}{(m, n) \Downarrow^I (m, n)} \quad \frac{(m, e^I) \Downarrow^I (m', n)}{(m, \mathsf{S}\ e^I) \Downarrow^I (m', n+1)}$$

$$\frac{(m, e^I) \Downarrow^I (m', n)}{(m, !e^I) \Downarrow^I (m', \mathsf{sel}\ (m', n))} \quad \frac{(m, e_1^I) \Downarrow^I (m', n_1) \quad (m', e_2^I) \Downarrow^I (m'', n_2)}{(m, e_1^I := e_2^I) \Downarrow^I (\mathsf{upd}\ (m'', n_1, n_2), ())}$$

$$\frac{(m, e_1^I) \Downarrow^I (m', ()) \quad (m', e_2^I) \Downarrow (m'', e^I)}{(m, e_1^I ; e_2^I) \Downarrow^I (m'', e^I)}$$

$$\frac{(m, e) \Downarrow^I (m', n) \quad n \neq 0 \quad (m', e_1) \Downarrow (m'', e^I)}{(m, \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \Downarrow^I (m'', e^I)} \quad \frac{(m, e) \Downarrow^I (m', 0) \quad (m', e_2) \Downarrow (m'', e^I)}{(m, \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \Downarrow^I (m'', e^I)}$$

Challenge 10. *Formalize the dynamic semantics of* λ^I*.*

Challenge 11 (termination of evaluation)**.** *Prove that your formalization of* $\Downarrow^I$*:*

- *Is a function*
- *Is a* total *function when the domain is restricted to well-typed closed terms*

3.4 Correctness Proof Challenges

Challenge 12 ("read_back")**.** *For any memory* m *and* $n \in \mathbb{N}$*,* $(m, (\lambda x : \mathbb{N}.\ 0 := x; !0)\ n) \Downarrow (m', n)$ *for some* m'*.*

Challenge 13 ("scribbler")**.** *For any memory* m *and* $n \in \mathbb{N}$*,* $(m, (\lambda x : \mathbb{N}.\ x := 0; \mathsf{S}\ x := 1; !x)\ n) \Downarrow (m', 0)$ *for some* m'*.*

4 Compilation

Challenge 14. *Define two compilation functions, both denoted with the overloaded syntax* $\lfloor \cdot \rfloor$*. For any* τ*,* $\lfloor \tau \rfloor$ *should be a* λ^I *type that is equivalent to* τ *in some appropriate sense. For any* e*,* $\lfloor e \rfloor$ *should be a* λ^I *term that is equivalent to* e *in some appropriate sense.*

I'm intentionally leaving the compilation and representation strategies up to the implementer, but I also give a definition of the compilation strategy from my solution in Appendix A.

Challenge 15. *Prove that both your compilation functions are total.*

Challenge 16. *Prove that, for any* e *and* τ *such that* $\cdot \vdash e : \tau$*, it follows that* $\cdot \vdash^I \lfloor e \rfloor : \lfloor \tau \rfloor$*.*

Challenge 17. *Prove that your term compilation is semantics-preserving.*

Saying what exactly this last challenge means is already getting into details of the formalization that ought to be up to the implementer. One approach would be to define a relation $e \simeq^m e^I$ which means that source term e and target term e^I are equivalent in the context of memory m. The theorem statement is then: For any e, τ, and v such that $\cdot \vdash e : \tau$ and $e \Downarrow v$, and for any memory m, we have $(m, \lfloor e \rfloor) \Downarrow (m', v^I)$ for some m' and v^I such that $v \simeq^{m'} v^I$.

Just to provide a point of comparison, here's a concrete version of a subcase of the last challenge:

Challenge 18. *For any* e *and* n *such that* $\cdot \vdash e : \mathbb{N}$ *and* $e \Downarrow n$*, and for any memory* m*, we have* $(m, \lfloor e \rfloor) \Downarrow (m', n)$ *for some* m'*.*

Of course, just this property isn't enough to show that a compilation respects the spirit of the challenge. For instance, a trivial translation branches on the type of the input. For type $\mathbb{N}$, it just evaluates the expression down to a constant and returns that. All other types are compiled to 1, so the translation just returns () for expressions of those types. A "real" translation should work more or less like we expect a "real" compiler to work.

A A Sample Compilation Strategy

Below, I'll use the abbreviation:

$$\mathsf{let}\ x \ = \ e_1 \ \mathsf{in}\ e_2$$

to stand for:

$$(\lambda x : \tau^I .\ e_2)\ e_1$$

The general idea in this strategy is that memory location 0 is reserved to store a heap limit pointer, such that a new allocation involves picking this pointer as the address of the new object and then incrementing the pointer with the new object's length (which happens always to be 2 here). Products are simply stored in adjacent memory cells, while sums are represented like products, where the first cell is a tag (with 0 signifying inl and 1 signifying inr) and the second cell holds the data value (of tag-dependent type).

To avoid having to reason about data pointers not pointing to the 0 cell reserved for the limit pointer, I store data pointers as the *predecessors* of "real" addresses. A pointer is always incremented with S before it is dereferenced.

$$
\begin{array}{rcl}
\lfloor\sigma\rfloor & = & \mathbb{N} \\
\lfloor\tau_1 \to \tau_2\rfloor & = & \lfloor\tau_1\rfloor \to \lfloor\tau_2\rfloor \\
\\
\lfloor x\rfloor & = & x \\
\lfloor\lambda x : \tau.\ e\rfloor & = & \lambda x : \lfloor\tau\rfloor.\ \lfloor e\rfloor \\
\lfloor e_1\ e_2\rfloor & = & \lfloor e_1\rfloor\ \lfloor e_2\rfloor \\
\\
\lfloor n\rfloor & = & n \\
\\
\lfloor(e_1, e_2)\rfloor & = & \mathsf{let}\ \mathit{limit}\ = !0\ \mathsf{in} \\
 & & \mathsf{let}\ v_1\ =\ \lfloor e_1\rfloor\ \mathsf{in} \\
 & & \mathsf{let}\ v_2\ =\ \lfloor e_2\rfloor\ \mathsf{in} \\
 & & \mathsf{S}\ \mathit{limit} := v_1; \mathsf{S}\ (\mathsf{S}\ \mathit{limit}) := v_2; 0 := \mathsf{S}\ (\mathsf{S}\ \mathit{limit}) \\
\lfloor\pi_1 e\rfloor & = & !(\mathsf{S}\ \lfloor e\rfloor) \\
\lfloor\pi_2 e\rfloor & = & !(\mathsf{S}\ (\mathsf{S}\ \lfloor e\rfloor)) \\
\\
\lfloor\mathsf{inl}_\sigma\ e\rfloor & = & \mathsf{let}\ \mathit{limit}\ = !0\ \mathsf{in} \\
 & & \mathsf{let}\ v\ =\ \lfloor e\rfloor\ \mathsf{in} \\
 & & \mathsf{S}\ \mathit{limit} := 0; \mathsf{S}\ (\mathsf{S}\ \mathit{limit}) := v; 0 := \mathsf{S}\ (\mathsf{S}\ \mathit{limit}) \\
\lfloor\mathsf{inr}_\sigma\ e\rfloor & = & \mathsf{let}\ \mathit{limit}\ = !0\ \mathsf{in} \\
 & & \mathsf{let}\ v\ =\ \lfloor e\rfloor\ \mathsf{in} \\
 & & \mathsf{S}\ \mathit{limit} := 1; \mathsf{S}\ (\mathsf{S}\ \mathit{limit}) := v; 0 := \mathsf{S}\ (\mathsf{S}\ \mathit{limit}) \\
\\
\lfloor\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}_{\sigma_2}\ x_1 \Rightarrow e_1 & = & \mathsf{let}\ v\ =\ \lfloor e\rfloor\ \mathsf{in} \\
\mid \mathsf{inr}_{\sigma_1}\ x_2 \Rightarrow e_2\rfloor & & \mathsf{let}\ v_1\ =\ \lfloor e_1\rfloor\ \mathsf{in} \\
 & & \mathsf{let}\ v_2\ =\ \lfloor e_2\rfloor\ \mathsf{in} \\
 & & \mathsf{if}\ !(\mathsf{S}\ v)\ \mathsf{then}\ v_2[x_2 \mapsto !(\mathsf{S}\ (\mathsf{S}\ v))]\ \mathsf{else}\ v_1[x_1 \mapsto !(\mathsf{S}\ (\mathsf{S}\ v))]
\end{array}
$$