OPLSS 2022 - Abstract Machines and Classical Realizability - Paul Dowen

Brian Christian, Parisa Fathololumi, Hannah Leung, Joshua Turcotti

June 24, 2022

Motivational Contrast: Why is anyone interested in abstract machines?

Direct (Operational) Semantics

- Defines *meaning* for *source code*, returns *value* or *behavior*
- Makes it easy to reason about *high-level properties*
- Harder to understand *low-level cost* of programs

Abstract Machines

- Definition inside *theoretical* machine, abstracts some details, but is closer to a real machine
- Easier to reason about *cost*
- Harder to reason about *high-level properties*

But! The goal is to be able to reason about high-level properties in abstract machines anyways - making them the best of both worlds. For example, *type safety* is a property that at first seems easier to reason about in a direct operational semantics, but will be shown to be possible to model nicely in an abstract machine as well.

The Simply-Typed λ Calculus

We will now proceed to give the syntax, reduction steps $\mapsto$, typing rules, and statements of type safetly for the *Simply Typed λ Calculus*

Grammar

$$M, N ::= x \mid MN \mid \lambda x.M \mid \texttt{True} \mid \texttt{False} \mid \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$
$$A, B ::= \texttt{Bool} \mid (A \to B)$$
$$E ::= [] \mid \texttt{if}\ E\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \mid E\ N$$

The first entry in this grammar gives us the language of terms M and N for the λ calculus. The second entry defines *types*, which in this version of the STLC are only Bools as primitive types and functions $(A \to B)$ between any other two types A, B. The third entry defines *evaluation contexts* - expressions containing *holes* that may be filled in by arbitrary other expressions.

Reductions

$$(\lambda x.M)N \mapsto M[N/x] \quad (\beta_{\to})$$
$$\texttt{if True then}\ N_1\ \texttt{else}\ N_2 \mapsto N_1 \quad (\beta_{\texttt{Bool}_1})$$
$$\texttt{if False then}\ N_2\ \texttt{else}\ N_2 \mapsto N_2 \quad (\beta_{\texttt{Bool}_2})$$
$$\frac{M \mapsto N}{E[m] \mapsto E[n]}$$

It's important to remember that these reductions do not provide in themselves a sufficiently performative scheme for evaluation in practice, such as in a realistic compiler. The heart of this issue is rule $\beta_{\to}$ - which cannot actually perform the rewriting indicated by the rule.

These reduction rules constitute what is known as a *small step* operational semantics, alternatively, a *big step* operational semantics could be written down that would specify the end state reached by any given term in a single judgment, but this is more difficult to reason about.

Typing Rules

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{\Gamma : M : \texttt{Bool} \qquad \Gamma \vdash N_1 : A \qquad \Gamma \vdash N_2 : A}{\Gamma \vdash \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 : A} \qquad \frac{}{\Gamma \vdash \texttt{True} : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash \texttt{False} : \texttt{Bool}}$$

Type Safety

Lemma 1 (Progress)**.** *If* $\cdot \vdash M : \texttt{Bool}$*, then either* M *is "done" (it's* True *or* False*) or* $\exists M' : M \mapsto M'$

Lemma 2 (Preservation). *If $M \mapsto M'$ and $\Gamma \vdash M : A$, then $\Gamma \vdash M' : A$*

Theorem 1 (Type Safety). *If $\cdot \vdash M : \texttt{Bool}$, then whenever $M \mapsto ... \mapsto M'$ and $\nexists M'' : M' \mapsto M''$, then M' is a final value (i.e.* `True` *or* `False`*).*

Type Safety is also commonly phrased as "Well-types programs can't go wrong", this clearly lacks formality.

Reduction Theory

We now introduce two new bits of syntax: C for *any* context (placing *holes* anywhere in the term language, as opposed to the definition of E which places *holes* only in two places), and $\rightarrow$ (overloaded with the types syntax) for a possibly nondeterministic relation representing *human free will*, and generalizing the deterministic $\mapsto$ relation. This new $\rightarrow$ relation is derivable by the following rule describing the process of identifiting a small step from the original semantics, and combining it with an arbitrary context.

$$\textsc{Compatibility}\quad \frac{M \mapsto N}{C[M] \rightarrow C[n]}$$

We also now define the $\twoheadrightarrow$ relation (reflexive, transitive closure of $\rightarrow$) by the following rules:

$$\textsc{Lifting}\quad \frac{M \rightarrow N}{M \twoheadrightarrow N} \qquad \textsc{Reflexivity}\quad \frac{}{M \twoheadrightarrow M} \qquad \textsc{Transitivity}\quad \frac{M \twoheadrightarrow M' \qquad M' \twoheadrightarrow M''}{M \twoheadrightarrow M''}$$

This completes the reduction theory.

Equational Theory

We now define an *equational theory* - centered around the new operator $=$ on terms in the STLC. It has the expected rules, in addition to three other η rules that are necessary to fill out the theory.

$$\textsc{Compatibility}\quad \frac{M = N}{C[M] = C[N]} \qquad \textsc{Lifting}\quad \frac{M \mapsto N}{M = N} \qquad \textsc{Symmetry}\quad \frac{M = M'}{M' = M} \qquad \textsc{Reflexivity}\quad \frac{}{M = M}$$

$$\textsc{Transitivity}\quad \frac{M = M' \qquad M' = M''}{M = M''} \qquad \lambda x.(Mx) = M \quad (\eta_{\rightarrow})$$

$$\texttt{if}\ M\ \texttt{then True else False} = M \quad (\eta_{\texttt{Bool}})$$

$$E[\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2] = \texttt{if}\ M\ \texttt{then}\ E[N_1]\ \texttt{else}\ E[N_2] \quad (\eta_{\texttt{Bool}})$$

This completes the equational theory.

Abstract Machines

Abstract machines codify the process of attempting to apply reductions to terms in the STLC. They are comprised of steps that fall into two categories: *refocusing* steps that attempt to identify reducible expressions (i.e. *redexes*), and *reduction* steps that rewrite those redexes.

Abstract Machine-specific Syntax

To specifiy our abstract machine, we will retain the entry M to describe terms in the STLC, replace the E entry in the original grammar with a new one representing "continuations", and add a c entry to represent the overall state of the machine.

$$
\begin{aligned}
E &::= \alpha \mid N \cdot E \mid \texttt{if then } c_1 \texttt{ else } c_2 \\
c &::= \langle M \mid E \rangle
\end{aligned}
$$

Note that in this E entry, the α corresponds to the holes [] from the prior STLC grammar.

We can now specify the two kinds of steps allowable by our machine:

Refocusing Steps

$$
\begin{aligned}
\langle M\ N \mid E \rangle &\mapsto \langle M \mid N \cdot E \rangle \quad (\mu_{\to}) \\
\langle \texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 \mid E \rangle &\mapsto \langle M \mid \texttt{if then } \langle N_1 | E \rangle \texttt{ else } \langle N_2 | E \rangle \rangle \quad (\mu_{\texttt{Bool}})
\end{aligned}
$$

Reduction Steps

$$
\begin{aligned}
\langle \lambda x.M \mid N \cdot E \rangle &\mapsto \langle M[N/x] \mid E \rangle \quad (\beta_{\to}) \\
\langle \texttt{True} \mid \texttt{if then } c_1 \texttt{ else } c_2 \rangle &\mapsto c_1 \quad (\beta_{\texttt{Bool}_1}) \\
\langle \texttt{False} \mid \texttt{if then } c_1 \texttt{ else } c_2 \rangle &\mapsto c_2 \quad (\beta_{\texttt{Bool}_2})
\end{aligned}
$$

A Better Abstract Machine

Unfortunately, there is some redundancy in these rules for our abstract machine. Imagine if we treat the symbol α in the continuation language as a *hole*, admitting a rule like:

$$
\langle M\ N \mid \alpha \rangle \mapsto \langle M \mid N \cdot \alpha \rangle
$$

We formalize this with the binding operator μ, which, for example, can encode the following:

$$M\ N := \mu\alpha.\langle M \mid N \cdot \alpha\rangle$$
$$\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 := \mu\alpha.\langle M \mid \texttt{if then}\ \langle N_1 \mid \alpha\rangle\ \texttt{else}\ \langle N_2 \mid \alpha\rangle\rangle$$

We can now rewrite our machine to have a *single* refocussing step phrased in terms of μ.

$$\langle \mu_a.c \mid E\rangle \mapsto c[E/\alpha] \quad (\mu)$$

We also add another compatibility rule for command steps:

$$\textsc{Compatibility}\ \frac{c \mapsto c'}{C[c] \mapsto C[c']}$$

And we can now rephrase our three η rules as well to use this more general μ binder.

$$\lambda x.\mu\alpha.\langle M \mid x \cdot \alpha\rangle = M : A \to B \quad (\eta_\to)$$
$$\texttt{if then}\ \langle \texttt{True} \mid E\rangle\ \texttt{else}\ \langle \texttt{False} \mid E\rangle = E : \texttt{Bool} \quad (\eta_{\texttt{Bool}})$$
$$\mu\alpha.\langle M \mid \alpha\rangle = M \quad (\eta_\mu)$$

And finally we could rewrite our expression grammer as:

$$M, N ::= x \mid \lambda x.M \mid \texttt{True} \mid \texttt{False} \mid \mu\alpha.c$$

This new abstract machine is more easily extensible and easier to reason about.

Exercises

1. Let $\texttt{and} = \lambda x.\lambda y.\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else False}$. Show $(\lambda y.\texttt{and True}\ y) \twoheadrightarrow \lambda y.y$.
2. Let $\texttt{not} = \lambda x.\texttt{if}\ x\ \texttt{then False else}\ True$. Show $(\lambda x.\texttt{not}(\texttt{not}\ x)) = \lambda x.x$.
3. Write a compile function $[\![M]\!]$ of STLC that produces a term of the final abstract machine language described in this lecture. Example semantics: $[\![\lambda x.M]\!] = \lambda x.[\![M]\!]$.
4. (bonus) Also redo 1 and 2 for the compiled versions of `and` and `not`.