Big-step semantics for the strong λ-calculus

Nathanaël Courant

November 2, 2020

Motivation

- Convertibility testing: an integral part of Coq typechecking
- Some proofs need a lot of computation
- To check convertibility, easiest is to compute *strong* normal form

Flavours of λ-calculus

- Common part of λ-calculus: $(\lambda x.t_1)\ t_2 \rightarrow t_1[x := t_2]$.
- Difference between flavours : free variables and handling of λ.
  - Weak: no free variables, no reduction under λ,
  - Open: free variables but no reduction under λ,
  - Strong: reduction under λ.
- Most programming languages: weak reduction.

Normal forms of strong λ-calculus

- For a normal form: no λ applied to an argument.
- Separate *inert* terms from the rest.

$$r ::= i \mid \lambda x.r$$
$$i ::= x \mid i\ r$$

Strong call-by-name

- Key idea: to compute the normal form of $t_1\ t_2$, if $t_1 \to^* \lambda x.t_3$, we don't need the normal form of t_1.
- Two modes:
  - For $\Downarrow_{\mathtt{d}}$, values are normal forms, $i \mid \lambda x.r$.
  - For $\Downarrow_{\mathtt{s}}$, values are $i \mid \lambda x.t$.
- Difference between the two modes: in mode $\Downarrow_{\mathtt{s}}$, if the result is a λ, don't reduce it to normal form.

Strong call-by-name

- f is $\mathtt{s}$ or $\mathtt{d}$
- Values: $i \mid \lambda x.r$ for $\Downarrow_{\mathtt{d}}$, $i \mid \lambda x.t$ for $\Downarrow_{\mathtt{s}}$.

$$\textsc{Var}\quad \frac{}{x \Downarrow_f x}$$

$$\textsc{Lam-S}\quad \frac{}{\lambda x.t \Downarrow_{\mathtt{s}} \lambda x.t} \qquad \textsc{Lam-D}\quad \frac{t \Downarrow_{\mathtt{d}} r}{\lambda x.t \Downarrow_{\mathtt{d}} \lambda x.r}$$

$$\textsc{App-}\lambda\quad \frac{t_1 \Downarrow_{\mathtt{s}} \lambda x.t_3 \qquad t_3[x := t_2] \Downarrow_f v}{t_1\ t_2 \Downarrow_f v} \qquad \textsc{App-I}\quad \frac{t_1 \Downarrow_{\mathtt{s}} i \qquad t_2 \Downarrow_{\mathtt{d}} r}{t_1\ t_2 \Downarrow_f i\ r}$$

Environment semantics

- Replace substitutions with an environment
- Values in environment are either free variable or thunks
- Values for $\Downarrow_{\mathtt{s}}$ are inert terms or closures

$$\textsc{Var-F}\ \frac{e(x) = y}{e \vdash x \Downarrow_f y} \qquad \textsc{Var-C}\ \frac{e(x) = (t, e') \qquad e' \vdash t \Downarrow_f v}{e \vdash x \Downarrow_f v} \qquad \textsc{Lam-S}\ \frac{}{e \vdash \lambda x.t \Downarrow_{\mathtt{s}} (\lambda x.t, e)}$$

$$\textsc{Lam-D}\ \frac{e + x \mapsto x \vdash t \Downarrow_{\mathtt{d}} r}{e \vdash \lambda x.t \Downarrow_{\mathtt{d}} \lambda x.r} \qquad \textsc{App-}\lambda\ \frac{e \vdash t_1 \Downarrow_{\mathtt{s}} (e', \lambda x.t_3) \qquad e' + x \mapsto (t_2, e) \vdash t_3 \Downarrow_f v}{e \vdash t_1\ t_2 \Downarrow_f v}$$

$$\textsc{App-I}\ \frac{e \vdash t_1 \Downarrow_{\mathtt{s}} i \qquad e \vdash t_2 \Downarrow_{\mathtt{d}} r}{e \vdash t_1\ t_2 \Downarrow_f i\ r}$$

Strong call-by-name λ-calculus

From call by name to call by need

Coq formalisation

Compiling the strong lambda-calculus

Call-by-need as memoizing call-by-name

- Two ways to see call-by-need: lazy call-by-value, or memoizing call-by-name
- Here: we have a call-by-name semantics, memoize it
- Problem: two evaluation modes, $\Downarrow_{\mathtt{s}}$ and $\Downarrow_{\mathtt{d}}$
- Don't memoize independently:
  `let` a `= very_long_computation () in` $\lambda x.a$

Relation between shallow and deep evaluation

- $e \vdash t \Downarrow_{\mathtt{s}} i$ iff $e \vdash t \Downarrow_{\mathtt{d}} i$
- If $e \vdash t_1 \Downarrow_{\mathtt{s}} \lambda x.t_2$ and $e \vdash \lambda x.t_2 \Downarrow_{\mathtt{d}} r$, then $e \vdash t_1 \Downarrow_{\mathtt{d}} r$
- Can compute result of $\Downarrow_{\mathtt{d}}$ from result of $\Downarrow_{\mathtt{s}}$
- In the other direction: remember $(\lambda x.t, e)$ before reducing under the λ

Call-by-need semantics for the strong λ-calculus

- Mutable memory for memoization
- Environment contains memory locations
- Possible values in memory:
  - Unevaluated thunks (t, e)
  - Inert terms i (including free variables)
  - Closures $(\lambda x.t, e)$
  - Closures with normal form $(\lambda x.t, e, \lambda x.r)$
- Result of $\Downarrow_{\mathtt{s}}$: $i \mid (\lambda x.t, e)$
- Result of $\Downarrow_{\mathtt{d}}$: $i \mid (\lambda x.t, e, \lambda x.r)$
- Extract normal form from result of $\Downarrow_{\mathtt{d}}$: $\mathtt{nf}\ i = i$, $\mathtt{nf}\ (\lambda x.t, e, \lambda x.r) = \lambda x.r$

Call-by-need semantics for the strong λ-calculus

- Applications mostly unchanged
- Allocate a new memory location for the newly unevaluated thunk

$$\textsc{App-}\lambda\quad \frac{a \notin m_2 \qquad \begin{array}{c} e, m_1 \vdash t_1 \Downarrow_{\mathtt{s}} (e', \lambda x.t_3), m_2 \\ e' + x \mapsto a, m_2 + a \mapsto (t_2, e) \vdash t_3 \Downarrow_f v, m_3 \end{array}}{e, m_1 \vdash t_1\ t_2 \Downarrow_f v, m_3}$$

$$\textsc{App-I}\quad \frac{e, m_1 \vdash t_1 \Downarrow_{\mathtt{s}} i, m_2 \qquad e, m_2 \vdash t_2 \Downarrow_{\mathtt{d}} r, m_3}{e, m_1 \vdash t_1\ t_2 \Downarrow_f i\ (\mathtt{nf}\ r), m_3}$$

Call-by-need semantics for the strong λ-calculus

- Deep evaluation of λ-abstractions now return the closure as well

$$\textsc{Lam-S}\quad \frac{}{e, m \vdash \lambda x.t \Downarrow_{\mathtt{s}} (\lambda x.t, e), m}$$

$$\textsc{Lam-D}\quad \frac{a \notin m_1 \qquad e + x \mapsto a, m_1 + a \mapsto x \vdash t \Downarrow_{\mathtt{d}} r, m_2}{e, m_1 \vdash \lambda x.t \Downarrow_{\mathtt{d}} (\lambda x.t, e, \lambda x.(\mathtt{nf}\ r)), m_2}$$

Call-by-need semantics for the strong λ-calculus

- If the variable refers to an unevaluated thunk, evaluate it and store the result
- If it refers to an inert term, then it is the result

$$\textsc{Var-Thunk}\ \frac{e(x) = a \qquad m_1(a) = (t, e') \qquad e', m_1 \vdash t \Downarrow_f v, m_2}{e, m_1 \vdash x \Downarrow_f v, m_2[a := v]}$$

$$\textsc{Var-I}\ \frac{e(x) = a \qquad m(a) = i}{e, m \vdash x \Downarrow_f i, m}$$

Call-by-need semantics for the strong λ-calculus

If the variable refers to a closure with normal form:

- In deep mode, return it
- In shallow mode, extract the closure from it

$$\textsc{Var-DS}\quad \frac{e(x) = a \qquad m(a) = (\lambda x.t, e', \lambda x.r)}{e, m \vdash x \Downarrow_{\mathtt{s}} (\lambda x.t, e'), m}$$

$$\textsc{Var-DD}\quad \frac{e(x) = a \qquad m(a) = (\lambda x.t, e', \lambda x.r)}{e, m \vdash x \Downarrow_{\mathtt{d}} (\lambda x.t, e', \lambda x.r), m}$$

Call-by-need semantics for the strong λ-calculus

If the variable refers to a closure without normal form:

- In shallow mode, return it
- In deep mode, compute the normal form, and update

$$\frac{e(x) = a \qquad m(a) = (\lambda x.t, e')}{e, m \vdash x \Downarrow_{\mathtt{s}} (\lambda x.t, e'), m}\ \textsc{Var-SS}$$

$$\frac{e(x) = a \qquad m_1(a) = (\lambda x.t, e') \qquad e', m_1 \vdash \lambda x.t \Downarrow_{\mathtt{d}} v, m_2}{e, m_1 \vdash x \Downarrow_{\mathtt{s}} v, m_2[a := v]}\ \textsc{Var-SD}$$

Properties

- No reduction under a λ-abstraction before applying it
- Efficient: complexity bilinear in the number of β-steps and the size of the initial term (conjecture)

Strong call-by-name λ-calculus

From call by name to call by need

Coq formalisation

Compiling the strong lambda-calculus

Contents of formalisation

- Proof of consistency of all semantics with β-reduction
- Extensions to support constructors and (shallow) pattern matching
- No proof of preservation of errors, preservation of divergence only for the first semantics

Pretty-big-step semantics

- Semantics using pretty-big-step instead of big-step
- Extended terms: $\hat{t} ::= t \mid \mathsf{app}_2(v, t_2) \mid \mathsf{app}_3(i, v) \mid \ldots$

$$\textsc{App-}\lambda\quad \frac{t_1 \Downarrow_{\mathtt{s}} \lambda x.t_3 \qquad t_3[x := t_2] \Downarrow_f v}{t_1\ t_2 \Downarrow_f v} \qquad\qquad \textsc{App-I}\quad \frac{t_1 \Downarrow_{\mathtt{s}} i \qquad t_2 \Downarrow_{\mathtt{d}} r}{t_1\ t_2 \Downarrow_f i\ r}$$

$$\Downarrow$$

$$\textsc{App}\quad \frac{t_1 \Downarrow_{\mathtt{s}} v_1 \qquad \mathsf{app}_2(v_1, t_2) \Downarrow_f v_2}{t_1\ t_2 \Downarrow_f v_2} \qquad\qquad \textsc{App-}\lambda\quad \frac{t_3[x := t_2] \Downarrow_f v}{\mathsf{app}_2(\lambda x.t_3, t_2) \Downarrow_f v}$$

$$\textsc{App-I}\quad \frac{t_2 \Downarrow_{\mathtt{d}} r \qquad \mathsf{app}_3(i, r) \Downarrow_f v}{\mathsf{app}_2(i, t_2) \Downarrow_f v} \qquad\qquad \textsc{App-3}\quad \frac{}{\mathsf{app}_3(i, r) \Downarrow_f i\ r}$$

Advantages of pretty-big-step

- Easy to express divergence using the same semantics
- De-duplication
- Makes the execution order explicit

The proof

- Current size: $\approx$ 6k lines
- De Bruijn indices for input terms, named variables for outputs (sharing)
- Small library for proving stronger induction principles easily

Strong call-by-name λ-calculus

From call by name to call by need

Coq formalisation

Compiling the strong lambda-calculus

Objectives

- Objective: compile to strict, weak language like OCaml (easy to compile further)
- Performance objective: efficient on weak, strict computations
- Assume we're not limited by the OCaml runtime: code pointers allowed anywhere, ability to mutate tags
- Ongoing work

Performance-sensitive code

- Function application: should be fast
- Application of inert terms: can be slow (number of applications $\leq$ size of result)
- Minimize cost of repetitive use of lazy value

Compilation scheme

- Each function, argument of a function, etc. is compiled into two code pointers (shallow and deep)
- Function application is compiled to function application
- Need to encode lazy and inert terms

Closure layout

Layout of a closure:

header
shallow code pointer
deep code pointer
0
environment

or

header
shallow code pointer
deep code pointer
normal form
environment

Handling inert terms

- For an inert term i, $i\ t$ should evaluate t to normal form r and return $i\ r$
- `accumulate` t evaluates t to normal form r, and returns a identical block with $i\ r$ instead of i

header (tag = 0)
`accumulate`
`accumulate`
i

Handling laziness

- Thunks are represented by a function that will evaluate the thunk before applying it to the argument
- Modify the block in place to put a forward block instead, which delegates the application
- *Assuming we can modify the OCaml GC:* can contract forward blocks

header (tag = 1)
`lazy_shallow`
`lazy_deep`
shallow code pointer
deep code pointer
environment

header (tag = 2)
`forward_shallow`
`forward_deep`
v

Another way?

- Modify the semantics: deep reduction is always shallow reduction followed by a deepening phase
- Only one code pointer needed in every block

Further questions

- Can OCaml efficient n-ary application be used?
- Is it efficient? (Objective: speed comparable to `native_compute`)
- Can we do it without modifying the OCaml runtime?

Future work

- Experiment with performance
- Prove the compilation in Coq
- Write a convertibility test