University of Cambridge
Department of Computer Science and Technology
Supervision 1 – Michaelmas 2018
Semantics of Programming Languages

Guidelines:

- When drawing a derivation tree, do not draw it all at once. Split it into multiple derivation trees and name different parts of it.
- Clearly state what rules you are applying.
- In your code, stay consistent with the formatting and write paragraph-like comments that are easy to read.
- Make sure your code compiles with either `Poly/ML` or `MLton` if you're using Standard ML, `ocamlc/ocamlopt` from `opam` if you're using OCaml and `OpenJDK` if you're using Java.
- If possible, use Standard ML over the alternatives **(not required)**.
- And above all, be precise and unambiguous with your statements. Do not use terminology that can be interpreted in different ways depending on the context such as *strongly typed* or *weakly typed*. Be precise and elaborate what you mean with the terminology you do use.
- Unless explicitly stated otherwise, the definition of L1 referred to in a given question is the original definition from the course notes, not a modified version from any of the other questions.
- Clearly structure your proofs. Number each statement in the proof that you intend to use later on. Make sure you don't shadow any variables in your proofs.
- In this supervision sheet, we use the λ notation to denote anonymous functions, `fn`, in ML. This comes from λ-calculus.[1]

Q.1) Consider the proof of determinacy in the lecture notes. What rule could be added such that determinacy does not hold anymore? Explain exactly where the proof would not hold.

Q.2) Prove type-safety of L1 (Hint: this is essentially a lemma of two theorems proven in the lecture notes!).

[1]https://en.wikipedia.org/wiki/Lambda_calculus

Q.3) Prove uniqueness of typing for L1.

Q.4) In the lecture notes, the progress theorem in L1 is stated as follows:

Theorem 1 (Progress) *If* $\Gamma \vdash e : \tau$ *and* $dom(\Gamma) \subseteq dom(s)$ *then either* e *is a value or* $\exists e', s'.\langle e, s\rangle \longrightarrow \langle e', s'\rangle$.

Suppose that we omit the condition $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$. Could we still prove progress? If so, present a full, rigorous proof of progress without the condition. Otherwise, elaborate exactly where the proof would fail and present a simple example L1 program which type checks and yet gets stuck.

Q.5) Function typing is more subtle than it might seem at first.

1. Functions in L2 require explicit type annotation of their binders, unlike functions in (say) SML. When are these type annotations used? What does their presence simplify?
2. L2 without `while` or store operations is strongly normalizing; that is, all well-typed programs written in this fragment of L2 will terminate. However, in lambda calculus, $(\lambda x.x\ x)\ (\lambda x.x\ x)$ is a term which β-reduces to itself (i.e., it is an "infinite loop").[2] We can write this term in L2 syntax and evaluation of this term would not get stuck; so, why is this not a well-typed L2 program?

Q.6) The implementation of L2 uses De Bruijn indices[3] and its substitution function for computing $[e/x]e'$ considers only the case of *closed* e.

1. Why is this not a problem for the existing implementation? (That is, what feature of the semantics guarantees that we never need to consider substitution with open e?)
2. Write a function to turn the De Bruijn notation into one with named binders, inventing names as you see fit.
3. Consider generalizing substitution to work under a common prefix of binders, so that, for example, one can interpret the meta-language term $\lambda f.[(\lambda z.z\ f)/x](\lambda y.f\ x\ y)$ arising from β reduction *under the binder* in the λ-calculus term $\lambda f.(\lambda x.\lambda y.f\ x\ y)\ (\lambda z.z\ f)$. (In De Bruijn notation, that's $\lambda(\lambda\lambda v_2\ v_1\ v_0)\ (\lambda v_0\ v_1)$ and the result is $\lambda\lambda v_1\ (\lambda v_0\ v_2)\ v_0$.) If you get this to work, pat yourself on the back; if not, try to have identified the basic pieces you would need and don't sweat the details.

Q.7) *Continuation-machine semantics.* Consider the small-step evaluation of the L2 program

$$(\lambda x_0.x_0)\ ((\lambda x_1.x_1)\ ((\lambda x_2.x_2)\ (\cdots((\lambda x_n.x_n)\ y)\cdots))).$$

[2]This expression is also a good place to start thinking about "Y combinators" which permit more general recursion in a λ-calculus without special syntax for recursive definitions.

[3]For all their attractiveness, both theoretical and practical, De Bruijn indices are widely understood to be cylon detectors (`https://mazzo.li/epilogue/index.html%3Fp=773.html`).

Note how often we end up repeating the same set of justifications, e.g. the "app2" rule, just to get back to the place we just did a reduction, only to make another small step and then do it all over again.[4] Let's address this by keeping better track of what we already know.

1. We will need a notion of a "one-hole context" of an evaluation: the outer expression in which the expression being reduced is contained. The grammar for such contexts is

$$C[\bullet] ::= (\bullet\ op\ e) \mid (v\ op\ \bullet) \mid (l\ :=\ \bullet) \mid (\bullet\ ;\ e) \mid (\texttt{if}\ \bullet\ \texttt{then}\ e\ \texttt{else}\ e) \mid \ldots$$

   Note the similarity between the possibilties here and the "purely structural" reduction rules; write out the rest of the grammar rule.

2. Introduce a new judgement form, $\langle s, e, \kappa\rangle \longrightarrow \langle s', e', \kappa'\rangle$ in which κ is a sequence of contexts $(C[\bullet]; C[\bullet]; \ldots)$. The inference rules for this form will mimic the existing reduction rules, but with an interesting twist. For examples,

$$\frac{}{\langle s, v, (k[\bullet]; \kappa)\rangle \longrightarrow \langle s, k[v], \kappa\rangle}\ \text{val} \qquad \frac{}{\langle s, !l, \kappa\rangle \longrightarrow \langle s, s(l), \kappa\rangle}\ \text{deref}$$

$$\frac{\neg(e_1\ \text{value})}{\langle s, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \kappa\rangle \longrightarrow \langle s, e_1, (\texttt{if}\ \bullet\ \texttt{then}\ e_2\ \texttt{else}\ e_3; \kappa)\rangle}\ \text{if3}$$

   Where by $k[v]$ we mean the context $k[\bullet]$ with its $\bullet$ replaced with v, which will be an *expression*: if $k[\bullet] = (v\ op\ \bullet)$ then $k[w] = (v\ op\ w)$. Write out a representative sample of the rest of the rules, including "fn", the CBV value-value application rule. What is the twist (where are the $\longrightarrow$s)?

3. Write (a representative sample of) an interpreter in ML for these semantics. How does the twist translate into implementation? Is that useful?

4. Suppose we *add* the following rule to the definition of $\longrightarrow$; does the provable set of judgements change?[5]

$$\frac{}{\langle s, !l, (k; \kappa)\rangle \longrightarrow \langle s, k[s(l)], \kappa\rangle}\ \text{deref2}$$

5. Prove, or at least consider how you would prove, the equivalence of these semantics and the small-step semantics given in lecture.

6. **Mind-bending extra thoughts**: Introduce sequences of contexts as new *values*, denoted $\bar{\kappa}$, and a pair of primitive operations, `callcc` and `resume`, defined thus:

$$\frac{}{\langle s, \texttt{callcc}\ e, \kappa\rangle \longrightarrow \langle s, e, (\bullet\ \bar{\kappa}; \kappa)\rangle}\ \text{callcc} \qquad \frac{}{\langle s, \texttt{resume}\ \bar{\kappa}\ e, \kappa'\rangle \longrightarrow \langle s, e, \kappa\rangle}\ \text{resume-v}$$

$$\frac{\neg(e_1\ \text{value})}{\langle s, \texttt{resume}\ e_1\ e_2, \kappa'\rangle \longrightarrow \langle s, e_1, (\texttt{resume}\ \bullet\ e_2; \kappa)\rangle}\ \text{resume-e}$$

[4] Because it takes k uses of "app2" to reach the k^{th} application and the reduction removes only one application from the expression, it takes $O(n^2)$ total rule applications to show that this program $\longrightarrow^* y$. `https://accidentallyquadratic.tumblr.com/`

[5] Inference rules which do not change the set of provable judgements, even if they add completely different proof *trees*, are called "admissible."

Given an initial context (sequence) κ, what is the evaluation of `callcc` $(\lambda k.k)$? of `callcc` $(\lambda k.$`resume` $k\ e)$? of $(\lambda n.$`resume` $n\ n)$ (`callcc` $(\lambda k.k)$)? In light of the last answer, what do you think

$$(\lambda n.(\lambda i.\texttt{if}\ i \geq 0\ \texttt{then}\ l_{\mathrm{i}} := i{+}(-1); l_{\mathrm{a}} := i{+}!l_{\mathrm{a}}; \texttt{resume}\ n\ n\ \texttt{else}\ !l_{\mathrm{a}})\ !l_{\mathrm{i}})\ (\texttt{callcc}\ (\lambda k.k))$$

evaluates to if, initially, $l_{\mathrm{i}} \mapsto 3$ and $l_{\mathrm{a}} \mapsto 0$?