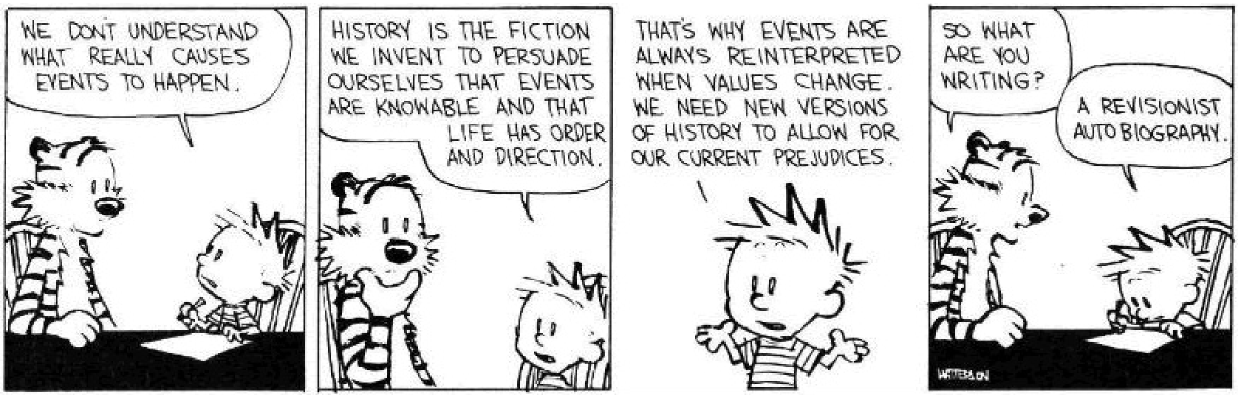
Meeting 12: Functions



Announcements

- Homework 2 due Friday at 6:00pm (48 hour automatic extensions because of snow day)
- Lab part build and testing infrastructure is optional.
- HW1 Feedback today (scheduler on moodle)

Questions

1) 1.4 – whole loop

2) ETPS – implementation

Assignment #2: Language Design and Implementation

CSCI 5535 / ECEN 5533: Fundamentals of Programming Languages

Spring 2018: Due Friday, February 23, 2018

The tasks in this homework ask you to formalize and prove meta-theoretical properties of an imperative core language IMP based on your experience with E. This homework also asks you to implement an extension of E in OCaml to gain experience translating formalization to implementation.

1 Language Design: IMP

In this section, we will formalize a variant of IMP from Chapter 2 of *FSPL* based on our experience from Homework Assignment 1. While it will be helpful to reference *FSPL* and *PFPL*, it is advised start from the principles you have learned thus far.

Consider the following syntax chart for IMP:

Typ	τ	::=	`num`	`num`	numbers
			`bool`	`bool`	booleans
Exp	e	::=	$\texttt{addr}[a]$	a	addresses (or "assignables")
			$\texttt{num}[n]$	n	numeral
			$\texttt{bool}[b]$	b	boolean
			$\texttt{plus}(e_1;e_2)$	$e_1 + e_2$	addition
			$\texttt{times}(e_1;e_2)$	$e_1 * e_2$	multiplication
			$\texttt{eq}(e_1;e_2)$	$e_1 == e_2$	equal
			$\texttt{le}(e_1;e_2)$	$e_1 <= e_2$	less-than-or-equal
			$\texttt{not}(e_1)$	$!e_1$	negation
			$\texttt{and}(e_1;e_2)$	$e_1 \mathbin{\&\&} e_2$	conjunction
			$\texttt{or}(e_1;e_2)$	$e_1 \mid\mid e_2$	disjunction
Cmd	c	::=	$\texttt{set}[a](e)$	$a := e$	assignment
			`skip`	`skip`	skip
			$\texttt{seq}(c_1;c_2)$	$c_1; c_2$	sequencing
			$\texttt{if}(e;c_1;c_2)$	$\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$	conditional
			$\texttt{while}(e;c_1)$	$\texttt{while}\ e\ \texttt{do}\ c_1$	looping
Addr	a				

Addresses a represent static memory store locations and are drawn from some unbounded set Addr. For simplicity, we fix all memory locations to only store numbers (as in FSPL). A store σ is thus a mapping from addresses to numbers, written as follows:

$$\sigma \quad ::= \quad \cdot \mid \sigma, a \hookrightarrow n$$

We rely on your prior experience with other programming languages for the semantics of the number, boolean, and command operations. Equality is polymorphic for both numbers and booleans, but all operators are monomorphic (e.g., $\texttt{and}(e_1; e_2)$ applies only to boolean arguments). With respect to order of evaluation, let us fix all operators to be left-to-right. Further, let us define $\texttt{and}(e_1; e_2)$ and $\texttt{or}(e_1; e_2)$ to be short circuiting.

Extra Credit. Complete this section where instead memory locations can store any values (i.e., numbers or booleans). Note that doing so will require extending the judgment forms with additional parameters.

We have chosen to stratify the syntax to separate commands c from expressions e to, at times, be able to focus on our discussion on either commands or expressions. By doing so, the semantics of IMP will require judgment forms both for expressions and commands.

1.1. Formalize the statics for IMP with two judgment forms $e : \tau$ and $c\,\mathsf{ok}$ that define well-typed IMP programs.

1.2. Formalize the dynamics for IMP by the following:

(a) Define values and final commands $e\,\mathsf{val}$ and $c\,\mathsf{final}$.

(b) Define a big-step operational semantics (i.e., evaluation semantics) with the judgment forms $\langle e, \sigma \rangle \Downarrow e'$ and $\langle c, \sigma \rangle \Downarrow \sigma'$. Notice that the expression and command judgment forms are not entirely parallel. The expression form returns a value-expression but not a store, while the command form returns a store but not a final-command.

(c) Define a small-step operational semantics with the judgment forms $\langle e, \sigma \rangle \longmapsto \langle e', \sigma' \rangle$ and $\langle c, \sigma \rangle \longmapsto \langle c', \sigma' \rangle$.

meta-theory

Statement theorem about PL (all expressions in the PL)

(d) i. State the canonical forms lemmas. No need to fully prove these, but it is important sketch enough of the proofs to convince yourself (and others) that you have the correct statements.

ii. State the progress and preservation lemmas for expressions. No need to fully prove these, but it is important to sketch enough of the proof to see that you have stated the appropriate canonical forms lemmas.

state "reasonable" assumptions about σ

iii. State and prove progress and preservation for commands.

(e) **"It's Just Semantics."**

i. Give the alternative non–short-circuiting big-step and small-step semantics for the $\texttt{and}(e_1; e_2)$ expression. That is, give rules for judgment forms $\langle e, \sigma \rangle \Downarrow e'$ and $\langle e, \sigma \rangle \longmapsto \langle e', \sigma' \rangle$ that replace the rules given above the $\texttt{and}(e_1; e_2)$ expression.

ii. Does short-circuiting versus non-short-circuiting affect the derivability in the big-step semantics? In other words, considering the set of rules defining the big-step semantics with short-circuiting $\texttt{and}(e_1; e_2)$ versus the set of rules defining the big-step semantics with non-short-circuiting $\texttt{and}(e_1; e_2)$, is there a judgment that is

$\langle e, \sigma \rangle \Downarrow e'$

$\langle e, \sigma \rangle \mapsto \langle e', \sigma' \rangle$ w/ and w/o short-circuiting derivable in one system but not the other? If yes, provide such a judgment and brief explanation. If not, give a brief explanation why there is no difference in derivability.

iii. How about for the small-step semantics? Please be clear and concise.

(f) **Manual Program Verification.** Let us write even(n) for the predicate on numbers that is true for even numbers. Prove the following statement: if $\langle \texttt{while}\ e\ \texttt{do}\ a := a+2, \sigma\rangle \Downarrow \sigma'$ such that even($\sigma(a)$), then even($\sigma(a)$). You may rely on your background knowledge about the even predicate.

2 Language Implementation: ETPS

In this section, we will implement in OCaml the language **ETPS**, that is, the language that combines language **E** (numbers and strings), language **T** (primitive recursion over natural numbers), language **P** (nullary-binary products), and language **S** (nullary-binary sums). We have already "implemented" **ETPS** in the meta-language of grammars and judgments, so when we say "implement in OCaml," we consider a formulaic translation using one meta-language (grammars and judgments) to another (OCaml).

When implementing a language, it is most effective to work by "growing the language" with test-driven development along the way. That is, start with the datatypes defining a small sublanguage and implement all functions (e.g., type checking, substitution, evaluation, reduction) and then incrementally update the datatypes and functions with additional language features one-at-a-time. Observe that this suggested approach is in contrast to proceeding one-phase-at-a-time: first defining the syntax, then implementing the type checker, then implementing substitution, then implementing evaluation, etc.

2.1. Implement language **E** along with thorough unit tests.

2.2. Implement language **ET** along with thorough unit tests.

2.3. Implement language **ETP** along with thorough unit tests.

2.4. Implement language **ETPS** along with thorough unit tests.

Explain your testing strategy and justify that your test cases attempt to cover your code as thoroughly as possible (e.g., they attempt to cover different execution paths of your implementation with each test). Write this explanation as comments alongside your test code.

Translating a Language to OCaml. When we say "implement Language X in OCaml," we mean precisely the following components.

- Define the syntactic forms as OCaml datatypes (i.e., each meta-variable becomes a datatype).

For terminals, we need to decide on a representation (e.g., variables `var` as OCaml `string`s).

e	**type** `exp`
τ	**type** `typ`
x	**type** `var = string`
n	**type** `num = int`
s	**type** `str = string`
Γ	**type** `typctx`

For testing and debugging, it is helpful to have functions that mediate between abstract syntax (i.e., OCaml values of the above types) and concrete syntax (e.g., a string of ASCII characters). Going from concrete to abstract syntax is called *parsing*, and going from abstract to concrete is called *pretty-printing*. We will need one for each datatype, for example,

```
parse_exp : string -> exp
pp_exp    : exp -> string
```

For this assignment, implementing parsing is optional (and not recommended until completing everything else).

- Implement OCaml functions for each function and judgment form.

$[e'/x]e$	**val** `subst : exp -> var -> exp -> exp`
$e\,\mathsf{val}$	**val** `is_val : exp -> bool`
$\Gamma \vdash e : \tau$	**val** `exp_typ : typctx -> exp -> typ option`
$e \Downarrow e'$	**val** `eval : exp -> exp`
$e \longmapsto e'$	**val** `step : exp -> exp`

For substitution `subst`, make sure that you implement *shadowing* correctly.

Notice that, as a design decision, we handle errors differently for statics (`exp_typ`) versus dynamics (`eval` and `step`). By error, we mean the inability to derive the judgment. For typing (`exp_typ` Γ e), we return a `typ option` where `Some(`τ`)` indicates we have a derivation for $\Gamma \vdash e : \tau$ and `None` means that we are not able to derive $\Gamma \vdash e : \tau$ for any τ. For `eval` and `step`, we instead raise an exception if the input expression does not allow deriving the judgment of interest. For instance, the single-step function (`step` e) should return an e' such that $e \longmapsto e'$ or otherwise raise an `Invalid_argument` exception.

- Implement a multiple-steps function

```
steps_pap : exp -> exp
```

that iterates the single-step function until a value. We will test progress and preservation at each step. Since you *meta-theory* proofs have shown progress and preservation, calls to step should never raise an exception (unless you have a bug in translation).

To be precise, let us define a judgment form $e \hookrightarrow^{:} e'$ for `steps_pap`:

$$\frac{e : \tau \qquad e\,\mathsf{val}}{e \hookrightarrow^{:} e} \qquad\qquad \frac{e : \tau \qquad e \longmapsto e' \qquad e' \hookrightarrow^{:} e''}{e \hookrightarrow^{:} e''}$$

As a side effect, use the pretty-printing functions above to print the expression and the type at each step.

3 Final Project Preparation: Pre-Proposal

3.1. Start thinking about the final project. Choose a partner. Review the course website for more detailed information about the class project.

To receive full points on this homework, you must do the following:

- Answer the pre-proposal questions.
  1. Who are the members of your team?
  2. What basic problem will your project try to solve?

  If you already have a research project, consider how to incorporate it! Write your pre-proposal in about 250 words. Be clear about your interests and what background reading you have done thus far.
- Scan the titles of papers at (at least) five top PL conferences. Name these conferences (including years), and include the conference URL with this information. For each conference, name the paper title and abstract that seems most interesting to you from that conference's proceedings that year. Include a citation along with a URL for each paper. Using BibTeX and DBLP is recommended.

If $\mathcal{D}: \langle \text{while } e \text{ do } a := a+2, \sigma\rangle \Downarrow \sigma'$ s.t. $\text{even}(\sigma(a))$

then $\text{even}(\sigma'(a))$.

Proof By induction on the derivation $\mathcal{D}$ of $\langle \text{while } e \text{ do } a := a+2, \sigma\rangle \Downarrow \sigma'$

Case

$$\mathcal{D}: \frac{\langle e, \sigma\rangle \Downarrow \text{false}}{\langle \text{while } e \text{ do } \underbrace{a := a+2}_{c}, \sigma\rangle \Downarrow \sigma} \qquad \left[\frac{\langle e, \sigma\rangle \Downarrow \text{false}}{\langle \text{while } e \text{ do } c, \sigma\rangle \Downarrow \sigma}\right]$$

So $\sigma' = \sigma$ and $\text{even}(\sigma(a))$ by assumption

So $\text{even}(\sigma'(a))$ ✓

Case

$$\mathcal{D}: \frac{\mathcal{D}_1: \langle e, \sigma\rangle \Downarrow \text{true} \quad \mathcal{D}_2: \langle a := a+2, \sigma\rangle \Downarrow \sigma'' \quad \mathcal{D}_3: \langle \text{while } e \text{ do } a := a+2, \sigma''\rangle \Downarrow \sigma'}{\langle \text{while } e \text{ do } a := a+2, \sigma\rangle \Downarrow \sigma'}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \text{true} \quad \langle c_1, \sigma\rangle \Downarrow \sigma'' \quad \langle \text{while } e \text{ do } c_1, \sigma''\rangle \Downarrow \sigma'}{\langle \text{while } e \text{ do } c_1, \sigma\rangle \Downarrow \sigma'}$$

$$\boxed{\langle c, \sigma\rangle \Downarrow \sigma'}$$

Need to show even($\sigma'(a)$)

To apply the i.h. even($\sigma''(a)$)

To get even($\sigma''(a)$) need to examine $\mathcal{D}_2$

Cases of $\mathcal{D}_2$

Case $\mathcal{D}_2$:

$$\dfrac{\dfrac{\dfrac{}{\langle a, \sigma \rangle \Downarrow \sigma(a)} \quad \dfrac{}{\langle 2, \sigma \rangle \Downarrow 2}}{\langle a+2, \sigma \rangle \Downarrow \sigma(a)+2}}{\langle a := a+2, \sigma \rangle \Downarrow \underbrace{\sigma, a \mapsto \sigma(a)+2}_{\sigma''}}$$