Principles of Programming Languages II
DRAFT

Scott Smith

November 4, 2019

Chapter 1

Environment-based Operational Semantics

1.1 F♭ Reviewed

Let us briefly review the grammar and substitution-based operational semantics from the PL I book [2]. Please see that book for details. We are going to use Xavier's syntax notation rather than the PL I capitalized-keywords business, which gets tiring to write after awhile. The syntax is the same, but `let rec` has been removed; it is not hard to encode with self-passing so is not adding a lot.

1.1.1 F♭ Syntax

x	::=		*variable names (strings)*
v	::=	x	*variable values*
		|`true`|`false`	*boolean values*
		|`0`|`1`|`-1`|`2`|`-2`|...	*integer values*
		|$\lambda x.e$	*function values*
e	::=	v	*value expressions*
		|`(`e`)`	*parenthesized expressions*
		|e `and` e|e `or` e|`not` e	*boolean expressions*
		|e `+` e|e `-` e|e `=` e|	*numerical expressions*
		|e e	*application expression*
		|`if` e `then` e `else` e	*conditional expressions*
		|`let` x `=` e `in` e	*let expression*

Note that in accordance with the above BNF, we will be using metavariables e, v, and x to represent expressions, values, and variables respectively.

Notions of bound and free variables and substitution are standard; see the PL I book.

1.1.2 Subsitution-based Operational Semantics

For reference, here are the **F**♭ operational semantics rules from the PL I book.

(value) $$\frac{}{v \Rightarrow v}$$

(not) $$\frac{e \Rightarrow v}{\texttt{not}\ \ e \Rightarrow \text{the negation of } v}$$

(and) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1\ \ \texttt{and}\ \ e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$$

(+) $$\frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1\ \ \texttt{+}\ \ e_2 \Rightarrow \text{the integer sum of } v_1 \text{ and } v_2}$$

(=) $$\frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1\ \ \texttt{=}\ \ e_2 \Rightarrow \texttt{true} \text{ if } v_1 \text{ and } v_2 \text{ are identical, else } \texttt{false}}$$

(`if true`*)* $$\frac{e_1 \Rightarrow \texttt{true}, \quad e_2 \Rightarrow v_2}{\texttt{if}\ \ e_1\ \ \texttt{then}\ \ e_2\ \ \texttt{else}\ \ e_3 \Rightarrow v_2}$$

(`if false`*)* $$\frac{e_1 \Rightarrow \texttt{false}, \quad e_3 \Rightarrow v_3}{\texttt{if}\ \ e_1\ \ \texttt{then}\ \ e_2\ \ \texttt{else}\ \ e_3 \Rightarrow v_3}$$

(application) $$\frac{e_1 \Rightarrow \lambda x.e, \quad e_2 \Rightarrow v_2, \quad e[v_2/x] \Rightarrow v}{e_1\ e_2 \Rightarrow v}$$

(let) $$\frac{e_1 \Rightarrow v_1 \quad e_2[v_1/x] \Rightarrow v_2}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow v_2}$$

1.2 Environment-based evaluator

Let us now get rid of substitutions, both because they are inefficient to implement in an interpreter and that we cannot make a program analysis out of an operational semantics with substitution.

The semantics will include an extra parameter ϵ, the *environment*, which map variables to values. But, for non-local variables in functions we will lose those definitions unless they are explicitly stashed. So, function values are not just $\lambda x.e$, they are *closures* $\langle \lambda x.e, \epsilon \rangle$ where ϵ is a mapping taking local variables to values.

It is more elegant to fully decouple the value grammar v from the expression grammar e at this point – closures are computation results so are only parts of values, and regular λ-expressions $\lambda x.e$ are still the non-value form of a function. Writing out the full grammars

(note we are overriding the definitions of e and v from the previous section here) we have

v	::=	$x \mid$ `true` $\mid$ `false` $\mid$ `0` $\mid$ `1` $\mid$ `-1` $\mid$ `2` $\mid$ `-2` $\mid \ldots$	*non-function values*
		$\mid \langle \lambda x.e, \epsilon \rangle$	*function closures*
ϵ	::=	$\{x \mapsto v, \ldots x \mapsto v\}$	*environments*
e	::=	$x \mid \lambda x.e \mid$ `true` $\mid$ `false` $\mid$ `0` $\mid$ `1` $\mid$ `-1` $\mid$ `2` $\mid$ `-2` $\mid \ldots$	
		$\mid e$ `and` $e \mid e$ `or` $e \mid$ `not` $e \mid e$ `+` $e \mid e$ `-` $e \mid e$ `=` e	
		$\mid e\, e \mid$ `if` e `then` e `else` $e \mid$ `let` x `=` e `in` e	

Environments ϵ are simple mappings from variables to values. It is a true mapping in that there is at most one mapping for any variable in any ϵ. To avoid ambiguity let us write out the basic operations etc on environments.

Definition 1.1. *Basic definitions on environments include*

- $x \in domain(\epsilon)$ *iff* ϵ *is of the form* $\{\ldots x \mapsto v, \ldots\}$, *i.e. it has some mapping of* x.
- $\epsilon(x) = v$ *where* $\epsilon = \{\ldots, x \mapsto v, \ldots\}$. *Looking up a variable not in the domain is undefined.*
- $(\epsilon_1 \cup \epsilon_2)$ *is the least* ϵ' *such that* $\epsilon'(x) = \epsilon_2(x)$ *if* $x \in domain(\epsilon_2)$, *and otherwise* $\epsilon'(x) = \epsilon_1(x)$.
- *We say* e/ϵ *is* closed *if all free variables in* e *are in* $domain(\epsilon)$.
- *We extend environment lookup notation so that* $\epsilon(e) = e[v_1/x_1, ..., v_n/x_n]$ *for* $\epsilon = \{x_1 \mapsto v_1, \ldots x_n \mapsto v_n\}$.
- *We will let* $\epsilon_\emptyset = \{\}$ *be shorthand for the empty environment.*

We can now define the operational semantics rules.

(variable) $$\frac{\epsilon(x) = v}{\epsilon \vdash x \Rightarrow v}$$

(closure) $$\frac{}{\epsilon \vdash \lambda x.e \Rightarrow \langle \lambda x.e, \epsilon \rangle}$$

(value) $$\frac{v \text{ not a variable or a closure}}{\epsilon \vdash v \Rightarrow v}$$

(not) $$\frac{\epsilon \vdash e \Rightarrow v}{\epsilon \vdash \texttt{not}\ \ e \Rightarrow \text{the negation of } v}$$

(and) $$\frac{\epsilon \vdash e_1 \Rightarrow v_1 \quad \epsilon \vdash e_2 \Rightarrow v_2}{\epsilon \vdash e_1\ \ \texttt{and}\ \ e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$$

(+) $$\frac{\epsilon \vdash e_1 \Rightarrow v_1, \quad \epsilon \vdash e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{\epsilon \vdash e_1\ \ \texttt{+}\ \ e_2 \Rightarrow \text{the integer sum of } v_1 \text{ and } v_2}$$

(=) $$\frac{\epsilon \vdash e_1 \Rightarrow v_1, \quad \epsilon \vdash e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{\epsilon \vdash e_1\ \ \texttt{=}\ \ e_2 \Rightarrow \texttt{true} \text{ if } v_1 \text{ and } v_2 \text{ are identical, else } \texttt{false}}$$

(`if true`) $$\frac{\epsilon \vdash e_1 \Rightarrow \texttt{true}, \quad \epsilon \vdash e_2 \Rightarrow v_2}{\epsilon \vdash \texttt{if}\ \ e_1\ \ \texttt{then}\ \ e_2\ \ \texttt{else}\ \ e_3 \Rightarrow v_2}$$

(`if false`) $$\frac{\epsilon \vdash e_1 \Rightarrow \texttt{false}, \quad \epsilon \vdash e_3 \Rightarrow v_3}{\epsilon \vdash \texttt{if}\ \ e_1\ \ \texttt{then}\ \ e_2\ \ \texttt{else}\ \ e_3 \Rightarrow v_3}$$

(application) $$\frac{\epsilon \vdash e_1 \Rightarrow \langle \lambda x.e, \epsilon' \rangle \quad \epsilon \vdash e_2 \Rightarrow v_2, \quad \epsilon' \cup \{x \mapsto v_2\} \vdash e \Rightarrow v}{\epsilon \vdash e_1\ e_2 \Rightarrow v}$$

(let) $$\frac{\epsilon \vdash e_1 \Rightarrow v_1 \quad \epsilon \cup \{x \mapsto v_1\} \vdash e_2 \Rightarrow v_2}{\epsilon \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Rightarrow v_2}$$

We can now define the (big step) relation.

Definition 1.2. *$\epsilon \vdash e \Rightarrow v$ for e/ϵ closed is the least relation closed under the above rules.*

Lemma 1.1. *The substitution and environment evaluators are equivalent:*

1. *For all expressions e and for non-function values v, $e \Rightarrow v$ in the substitution-based system iff $\epsilon_\emptyset \vdash e \Rightarrow v$ in the environment-based system.*
2. *For all expressions e, $e \Rightarrow \lambda x.e'$ in the substitution-based system iff $\epsilon_\emptyset \vdash e \Rightarrow \langle \lambda x.e'', \epsilon' \rangle$ in the environment-based system, where $\epsilon'(\lambda x.e'') = \lambda x.e'$.*

There is a bit of a mismatch here on the result because function values differ between the two systems, the former performing substitutions and the latter not. The second clause puts them back on equal footing by substituting the final environment in.

1.3 A Store-of-Environments evaluator

We are now going to define a variation on the above where rather than putting environments ϵ directly in closures, we put a "reference" r to the environment in the closure and then we additionally have a store, σ, which like a heap is a mapping from references r to environments. We want to make sure we make a fresh reference for each new environment, otherwise we get overlaps. Let us now define this revised evaluator.

v	::=	$\ldots$	*non-function values as before*
		$\mid \langle \lambda x.e, r \rangle$	*function closures*
ϵ	::=	$\{x \mapsto v, \ldots x \mapsto v\}$	*environments as before*
e	::=	$\ldots$	*expressions as before*
r	::=		*references, abstract now*
σ	::=	$\{r \mapsto \epsilon, \ldots r \mapsto \epsilon\}$	*store*

Stores σ are maps like environments and we use the same notation for extension and lookup etc. Their domain of references r are for now abstract nonces, similar to the cells c of FbS. If you like things concrete think of them as natural numbers. When we make a new closure we want to make a fresh $r \mapsto \epsilon$ to add to the store.

The revised relation is of the form $\epsilon \vdash \langle e, \sigma \rangle \Rightarrow \langle v, \sigma' \rangle$ - the store is threaded through the computation in a manner similar to FbS. We can now define the operational semantics rules as a variation on the previous system. Most of the added "baggage" is because we need to explicitly thread the store here.

$$\textit{(var)} \quad \frac{\epsilon(x) = v}{\epsilon \vdash \langle x, \sigma\rangle \Rightarrow \langle v, \sigma\rangle}$$

$$\textit{(closure)} \quad \frac{\sigma' = \sigma \cup \{r \mapsto \epsilon\} \quad r \text{ is fresh}}{\epsilon \vdash \langle \lambda x.e, \sigma\rangle \Rightarrow \langle\langle \lambda x.e, r\rangle, \sigma'\rangle}$$

$$\textit{(value)} \quad \frac{v \text{ not a variable or a closure}}{\epsilon \vdash \langle v, \sigma\rangle \Rightarrow \langle v, \sigma\rangle}$$

$$\textit{(not)} \quad \frac{\epsilon \vdash \langle e, \sigma\rangle \Rightarrow \langle v, \sigma'\rangle \quad v' \text{ is the negation of } v}{\epsilon \vdash \langle \sigma, \texttt{not } e\rangle \Rightarrow \langle v', \sigma'\rangle}$$

$$\textit{(and)} \quad \frac{\epsilon \vdash \langle e_1, \sigma\rangle \Rightarrow \langle v_1, \sigma'\rangle \quad \epsilon \vdash \langle e_2, \sigma'\rangle \Rightarrow \langle v_2, \sigma''\rangle \quad v \text{ is } v_1 \wedge v_2}{\epsilon \vdash \langle e_1 \texttt{ and } e_2, \sigma\rangle \Rightarrow \langle v, \sigma''\rangle}$$

$$\textit{(+)} \quad \frac{\epsilon \vdash \langle e_1, \sigma\rangle \Rightarrow \langle v_1, \sigma'\rangle \quad \epsilon \vdash \langle e_2, \sigma'\rangle \Rightarrow \langle v_2, \sigma''\rangle \text{ where } v_1, v_2, \sigma'' \in \mathbb{Z} \text{ and } v_1 + v_2 = v}{\epsilon \vdash \langle e_1 \texttt{ + } e_2, \sigma\rangle \Rightarrow \langle v, \sigma''\rangle}$$

$$\textit{(=)} \quad \frac{\epsilon \vdash \langle e_1, \sigma\rangle \Rightarrow \langle v_1, \sigma'\rangle \quad \epsilon \vdash \langle e_2, \sigma'\rangle \Rightarrow \langle v_2, \sigma''\rangle \text{ where } v_1, v_2, \sigma'' \in \mathbb{Z}, v = (v_1 = v_2)}{\epsilon \vdash \langle e_1 \texttt{ = } e_2, \sigma\rangle \Rightarrow \langle v, \sigma''\rangle}$$

$$\textit{(}\texttt{if true}\textit{)} \quad \frac{\epsilon \vdash \langle e_1, \sigma\rangle \Rightarrow \langle \texttt{true}, \sigma'\rangle \quad \epsilon \vdash \langle e_2, \sigma'\rangle \Rightarrow \langle v_2, \sigma''\rangle}{\epsilon \vdash \langle \sigma, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3\rangle \Rightarrow \langle v_2, \sigma''\rangle}$$

$$\textit{(}\texttt{if false}\textit{)} \quad \frac{\epsilon \vdash \langle e_1, \sigma\rangle \Rightarrow \langle \texttt{false}, \sigma'\rangle \quad \epsilon \vdash \langle \sigma', e_3\rangle \Rightarrow \langle v_3, \sigma''\rangle}{\epsilon \vdash \langle \sigma, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3\rangle \Rightarrow \langle v_3, \sigma''\rangle}$$

$$\textit{(app)} \quad \frac{\begin{array}{l}\epsilon \vdash \langle e_1, \sigma\rangle \Rightarrow \langle\langle \lambda x.e, r\rangle, \sigma'\rangle \quad \epsilon \vdash \langle e_2, \sigma'\rangle \Rightarrow \langle v_2, \sigma''\rangle \\ \sigma''(r) = \epsilon' \quad \epsilon' \cup \{x \mapsto v_2\} \vdash \langle e, \sigma''\rangle \Rightarrow \langle v, \sigma'''\rangle\end{array}}{\epsilon \vdash \langle e_1\ e_2, \sigma\rangle \Rightarrow \langle v, \sigma'''\rangle}$$

$$\textit{(let)} \quad \frac{\epsilon \vdash \langle e_1, \sigma\rangle \Rightarrow \langle v_1, \sigma'\rangle \quad \epsilon \cup \{x \mapsto v_1\} \vdash \langle e_2, \sigma'\rangle \Rightarrow \langle v_2, \sigma'''\rangle}{\epsilon \vdash \langle \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, \sigma\rangle \Rightarrow \langle v_2, \sigma'''\rangle}$$

The closure rule is the main change, notice how we only store a reference r to the environment in the closure, and we put the environment itself in the store. Since each time we make a closure we pick a fresh r there will be no overlap on different environments. Note that "fresh" is a hack, it is a side effect and we are doing math here. But it is a well-known and accepted hack. To be more accurate though, we should be passing along the most recent reference name used, and pick a strictly "bigger" one to guarantee it is fresh, then keep passing along this latest one. The Leroy encoding of state passing style did this.

We can now define the (big step) relation.

Definition 1.3. *$\epsilon \vdash \langle e, \sigma\rangle \Rightarrow \langle v, \sigma'\rangle$ for e/ϵ closed is the least relation closed under the above rules. An initial computation starts with an empty environment and store: $\epsilon_\emptyset \vdash \langle e, \sigma_\emptyset\rangle \Rightarrow \langle v, \sigma\rangle$.*

We can again easily prove equivalence; this time let us just state equivalence on integer or boolean results, equivalence on functions is analogous to the previous Lemma.

Lemma 1.2. *The environment and store evaluators are equivalent: For all expressions e and for non-function values v, $\epsilon_\emptyset \vdash e \Rightarrow v$ in the environment-based system iff $\epsilon_\emptyset \vdash \langle e, \sigma_\emptyset \rangle \Rightarrow \langle v, \sigma \rangle$ in the store-based system.*

1.4 Call site stacks as references

We want to make one final modification to this operational semantics before we will be ready to "hobble" it to make a program analysis.

Above the references r were arbitrary entities, now we will make them something more concrete that we can then make finite so our program analysis will terminate. In particular we make them stacks (lists in particular) of call sites, which we call *contexts*, C. We also need to distinguish different call sites and so tweak the language grammar ever so slightly to add a tag $e\ ^t\ e$ to each call site where t is distinct for each different call site in the program.

Here are the changed grammar entities.

v	::=	$\ldots$	*non-function values as before*
		$\mid \langle \lambda x.e, C \rangle$	*function closures*
e	::=	$\cdots \mid e\ ^t\ e$	*expressions nearly as before*
t	::=		*call site tags, think numbers*
C	::=	$[t; \ldots; t]$	*references, now call site lists*
σ	::=	$\{C \mapsto \epsilon, \ldots C \mapsto \epsilon\}$	*store*

The rules are nearly identical, but we need to keep track of the current call site list C since it will be the reference used in the store. Here are the closure and app rules, for the others just tack a C on front as they are unchanged otherwise.

(closure)
$$\frac{\sigma' = \sigma \cup \{C \mapsto \epsilon\}}{\epsilon, C \vdash \langle \lambda x.e, \sigma \rangle \Rightarrow \langle \langle \lambda x.e, C \rangle, \sigma' \rangle}$$

(app)
$$\frac{\begin{array}{l}\epsilon, C \vdash \langle e_1, \sigma \rangle \Rightarrow \langle \langle \lambda x.e, C' \rangle, \sigma' \rangle \quad \epsilon, C \vdash \langle e_2, \sigma' \rangle \Rightarrow \langle v_2, \sigma'' \rangle \\ \sigma''(C') = \epsilon' \quad \epsilon' \cup \{x \mapsto v_2\}, (t :: C) \vdash \langle e, \sigma'' \rangle \Rightarrow \langle v, \sigma''' \rangle\end{array}}{\epsilon, C \vdash \langle e_1\ ^t\ e_2, \sigma \rangle \Rightarrow \langle v, \sigma''' \rangle}$$

The $\cup$ operation in the closure rule needs to be clarified – in the context of let bindings the environment may be extended as more let expressions are entered, but the context C is not changing. The solution is to take $\cup$ to always replace if the key C is already present, as that environment will always be an extension. When we get to making an analysis out of this it is more subtle, the mapping becomes a multi-mapping due to non-determinism.

Definition 1.4. *$\epsilon, C \vdash \langle e, \sigma \rangle \Rightarrow \langle v, \sigma' \rangle$ for e/ϵ closed is the least relation closed under the above rules. An initial computation starts with an empty environment, call stack, and store: $\epsilon_\emptyset, [] \vdash \langle e, \sigma_\emptyset \rangle \Rightarrow \langle v, \sigma \rangle$.*

Lemma 1.3. *The store-reference and store-call-stack evaluators are equivalent: For all expressions e and for non-function values $\epsilon_\emptyset \vdash \langle e, \sigma_\emptyset \rangle \Rightarrow \langle v, \sigma \rangle$ in the store-reference system iff $\epsilon_\emptyset, [] \vdash \langle e, \sigma_\emptyset \rangle \Rightarrow \langle v, \sigma \rangle$ in the store-call-stack system.*

To establish this result we need to know that the stacks C are "as good as fresh references", in that we will not accidentally give two different environments the same address in the store. If our language had e.g. `while`-loops this would in fact not be the case! Going around the while loop we could repeatedly make the same stack. Fortunately our language does not have while loops, and when we encode them with recursion they will be putting a new frame on the call stack each time around the loop. A proof was written by Zach and appears in [1].

Chapter 2

Program Analysis

In this chapter we will "hobble" the (complete) evaluator $\epsilon, C \vdash \langle e, \sigma \rangle \Rightarrow \langle v, \sigma' \rangle$ defined in the previous chapter so it can be implemented as a total function. Since a hobbled evaluator always terminates, it can be used by a compiler to optimize programs and/or to help find potential run-time errors. Well, the previous sentence is not completely accurate, it must *feasibly* terminate – a doubly-exponential algorithm would not be very good! This is in fact a serious issue as many program analyses are exponential in the worst case.

Our development here is based on "Abstracting Abstract Machines" (AAM for short), a paper by Might and Van Horn [4]. We could just re-use their development, but their analysis only works over CPS-converted programs which can make some things more difficult to see, and I have simplified their approach, getting rid of one parameter in the evaluation relation.

2.1 The Basic Analysis

We start from the store-call-stack evaluator at the end of the previous chapter. All we need to do is to make the store and value space finite; to make the store finite we will just make the references finite, by arbitrarily pruning the stack.

Here is the full grammar. Note how we have finitary integers here, only keeping the sign, and also include unknown-int and unknown-bool values. Unknown-int is needed when we for example add a positive and a negative number – it is unknown what its sign will be. Other than these changes to atomic data the language grammar is the same.

The major change is for the store: when we hobble the analysis we may end up re-using a key C in the store, and we don't want to overwrite the old value as it could still be needed in the future – to overwrite the old value would make the analyses *un-sound*, it should always return a "superset" of what the actual program does and that property would fail.

So, the solution is that instead of overwriting we just include them all, making it a multi-map. We will notate this as a map where the key lookup in general returns a *set* of environments $\{\epsilon_1, \ldots, \epsilon_n\}$.

v	::=	$x \mid \texttt{true} \mid \texttt{false} \mid \texttt{(-)} \mid \texttt{(0)} \mid \texttt{(+)}$	*non-function values*
		$\mid \langle \lambda x.e, \epsilon \rangle$	*function closures*
ϵ	::=	$\{x \mapsto v, \ldots x \mapsto v\}$	*environments*
E	::=	$\{\epsilon, \ldots, \epsilon\}$	*environment sets*
e	::=	$x \mid \lambda x.e \mid \texttt{true} \mid \texttt{false} \mid \{\ldots \texttt{-1}, 0, 1, \ldots\}$	
		$\mid e \texttt{ and } e \mid e \texttt{ or } e \mid \texttt{not } e \mid e \texttt{ + } e \mid e \texttt{ - } e \mid e \texttt{ = } e$	
		$\mid e \,^t\, e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \mid \texttt{let } x \texttt{ = } e \texttt{ in } e$	
t	::=		*call site tags, think numbers*
C	::=	$[t; \ldots; t]$	*call site lists aka contexts*
σ	::=	$\{C \mapsto E, \ldots C \mapsto E\}$	*store*

Since stores are now multi-maps we need to re-define store extension and lookup.

Definition 2.1. *Basic definitions on stores now include*

- $C \in domain(\sigma)$ *iff* σ *is of the form* $\{\ldots C \mapsto E, \ldots\}$, *i.e. it has some mapping of* C.
- $\sigma(C) = E$ *where* $\sigma = \{\ldots, C \mapsto E, \ldots\}$. *Looking up a context not in the domain is undefined. To assert a particular environment is in a store map result we then assert* $\epsilon \in \sigma(C)$.
- $(\sigma_1 \cup \sigma_2)$ *is the least* σ' *such that* $\sigma'(C) = \sigma_1(C) \cup \sigma_2(C)$. *Observe that we are unioning up the respective multi-map results here; this is also how we may create multi-maps to begin with if the stores formerly had only singleton sets of environments mapped.*
- *We must be careful on how sets of environments are unioned –* $E_1 \cup E_2$ *will union the* ϵ *except in the case where one environment strictly subsumes another; in that case the smaller environment will be elided. This subtle issue is only needed to address let-binding since there is no context change in the let rule and environments may interfere in the store if we are not careful.*
- *We will let* $\sigma_\emptyset = \{\}$ *be shorthand for the empty store.*

The rules are mostly unchanged from the previous evaluation system. For functions and function call only the store operations in closure and app and the context append in app change. The integer rules are different because we have abstracted their values to only positive, negative, and zero in order to make them finite. The hatted operators below are over this finite value space; for example, $\texttt{(+)} \in \texttt{(+)} \hat{+} \texttt{(-)}$ - adding a positive and a negative

can produce a positive (it also can produce a negative or a 0…).

$$\textit{(var)} \quad \frac{\epsilon(x) = v}{\epsilon, C \vdash \langle x, \sigma\rangle \widehat{\Rightarrow} \langle v, \sigma\rangle}$$

$$\textit{(closure)} \quad \frac{\sigma' = \sigma \cup \{C \mapsto \epsilon\}}{\epsilon, C \vdash \langle \lambda x.e, \sigma\rangle \widehat{\Rightarrow} \langle \langle \lambda x.e, C\rangle, \sigma'\rangle}$$

$$\textit{(number)} \quad \frac{e \in \mathbb{Z} \quad v \text{ is the sign of } e}{\epsilon, C \vdash \langle e, \sigma\rangle \widehat{\Rightarrow} \langle v, \sigma\rangle}$$

$$\textit{(boolean)} \quad \frac{v \text{ is a boolean}}{\epsilon, C \vdash \langle v, \sigma\rangle \widehat{\Rightarrow} \langle v, \sigma\rangle}$$

$$\textit{(not)} \quad \frac{\epsilon, C \vdash \langle e, \sigma\rangle \widehat{\Rightarrow} \langle v, \sigma'\rangle \quad v' \text{ is the negation of } v}{\epsilon, C \vdash \langle \sigma, \texttt{not}\ \ e\rangle \widehat{\Rightarrow} \langle v', \sigma'\rangle}$$

$$\textit{(and)} \quad \frac{\epsilon, C \vdash \langle e_1, \sigma\rangle \widehat{\Rightarrow} \langle v_1, \sigma'\rangle \quad \epsilon, C \vdash \langle e_2, \sigma'\rangle \widehat{\Rightarrow} \langle v_2, \sigma''\rangle \quad v \text{ is } v_1 \wedge v_2}{\epsilon, C \vdash \langle e_1\ \ \texttt{and}\ \ e_2, \sigma\rangle \widehat{\Rightarrow} \langle v, \sigma''\rangle}$$

$$\textit{(+)} \quad \frac{\epsilon, C \vdash \langle e_1, \sigma\rangle \widehat{\Rightarrow} \langle v_1, \sigma'\rangle \quad \epsilon, C \vdash \langle e_2, \sigma'\rangle \widehat{\Rightarrow} \langle v_2\rangle \text{ where } v_1, v_2 \in \hat{\mathbb{Z}} \text{ and } v \in v_1 \hat{+} v_2}{\epsilon, C \vdash \langle e_1\ \ \texttt{+}\ \ e_2, \sigma\rangle \widehat{\Rightarrow} \langle v, \sigma''\rangle}$$

$$\textit{(=)} \quad \frac{\epsilon, C \vdash \langle e_1, \sigma\rangle \widehat{\Rightarrow} \langle v_1, \sigma'\rangle \quad \epsilon, C \vdash \langle e_2, \sigma'\rangle \widehat{\Rightarrow} \langle v_2, \sigma''\rangle \text{ where } v_1, v_2 \in \hat{\mathbb{Z}}, v \in (v_1 \hat{=} v_2)}{\epsilon, C \vdash \langle e_1\ \ \texttt{=}\ \ e_2, \sigma\rangle \widehat{\Rightarrow} \langle v, \sigma''\rangle}$$

$$\textit{(if true)} \quad \frac{\epsilon, C \vdash \langle e_1, \sigma\rangle \widehat{\Rightarrow} \langle \texttt{true}, \sigma'\rangle \quad \epsilon, C \vdash \langle e_2, \sigma'\rangle \widehat{\Rightarrow} \langle v_2, \sigma''\rangle}{\epsilon, C \vdash \langle \texttt{if}\ \ e_1\ \ \texttt{then}\ \ e_2\ \ \texttt{else}\ \ e_3, \sigma\rangle \widehat{\Rightarrow} \langle v_2, \sigma''\rangle}$$

$$\textit{(if false)} \quad \frac{\epsilon, C \vdash \langle e_1, \sigma\rangle \widehat{\Rightarrow} \langle \texttt{false}, \sigma'\rangle \quad \epsilon, C \vdash \langle \sigma', e_3\rangle \widehat{\Rightarrow} \langle v_3, \sigma''\rangle}{\epsilon, C \vdash \langle \texttt{if}\ \ e_1\ \ \texttt{then}\ \ e_2\ \ \texttt{else}\ \ e_3, \sigma\rangle \widehat{\Rightarrow} \langle v_3, \sigma''\rangle}$$

$$\textit{(app)} \quad \frac{\begin{array}{l}\epsilon, C \vdash \langle e_1, \sigma\rangle \widehat{\Rightarrow} \langle \langle \lambda x.e, C'\rangle, \sigma'\rangle \quad \epsilon, C \vdash \langle e_2, \sigma'\rangle \widehat{\Rightarrow} \langle v_2, \sigma''\rangle \\ \epsilon' \in \sigma''(C') \quad \epsilon' \cup \{x \mapsto v_2\}, (t ::_k C) \vdash \langle e, \sigma''\rangle \widehat{\Rightarrow} \langle v, \sigma'''\rangle\end{array}}{\epsilon, C \vdash \langle e_1^{\,t}\, e_2, \sigma\rangle \widehat{\Rightarrow} \langle v, \sigma'''\rangle}$$

$$\textit{(let)} \quad \frac{\epsilon, C \vdash \langle e_1, \sigma\rangle \widehat{\Rightarrow} \langle v_1, \sigma'\rangle \quad \epsilon \cup \{x \mapsto v_1\}, C \vdash \langle e_2, \sigma'\rangle \widehat{\Rightarrow} \langle v_2, \sigma''\rangle}{\epsilon, C \vdash \langle \texttt{let}\ x = e_1\ \texttt{in}\ e_2, \sigma\rangle \widehat{\Rightarrow} \langle v_2, \sigma''\rangle}$$

Notice that list cons, ::, has been replaced with k-cons, $::_k$. This operation conses an element to the front and prunes any elements past length k off of the end of the list. This is key to making function calls finitary, eventually they will start re-using environments since no new keys C will be manufactured. The particular number k is a parameter to the analysis, this style of analysis is termed kCFA [3]. k is usually something small like 0, 1, or 2 as it in practice gets very slow for larger k.

Definition 2.2. *$\epsilon, C \vdash \langle e, \sigma\rangle \widehat{\Rightarrow} \langle v, \sigma'\rangle$ for e/ϵ closed is the least relation closed under the above rules. An initial computation starts with an empty environment, call stack, and store: $\epsilon_\emptyset, C_\emptyset \vdash \langle e, \sigma_\emptyset\rangle \widehat{\Rightarrow} \langle v, \sigma\rangle$.*

Lemma 2.1 (Soundness)**.** *The analysis evaluator above is a conservative approximation of the actual evaluation relation: For all expressions e and for non-function values v, $\epsilon_\emptyset, C_\emptyset \vdash \langle e, \sigma_\emptyset \rangle \Rightarrow \langle v, \sigma \rangle$ in the store-reference system implies $\epsilon_\emptyset, C_\emptyset \vdash \langle e, \sigma_\emptyset \rangle \widehat{\Rightarrow} \langle v', \sigma \rangle$ in the analysis, where $v \in v'$ (the "in" here means e.g.* $5 \in (+)$*).*

Lemma 2.2 (Nondeterminism)**.** *The $\epsilon_\emptyset, C_\emptyset \vdash \langle e, \sigma_\emptyset \rangle \widehat{\Rightarrow} \langle v, \sigma \rangle$ relation is not a function, there may be more than one value corresponding to initial expression e.*

Lemma 2.3 (Decidability)**.** *The $\epsilon_\emptyset, C_\emptyset \vdash \langle e, \sigma_\emptyset \rangle \widehat{\Rightarrow} \langle v, \sigma \rangle$ relation is decidable: given an e it is possible to compute the complete result value set $\{v_1, \ldots, v_n\}$, the largest set such that for each v_i, $\epsilon_\emptyset, C_\emptyset \vdash \langle e, \sigma_\emptyset \rangle \widehat{\Rightarrow} \langle v_i, \sigma \rangle$.*

Acknowledgements

Thanks to Leandro Facchinetti for serving as a sounding board for this presentation of program analysis, and for suggesting several improvements to the representations.

Chapter 3

Symbolic Execution

Here is an initial draft of the symbolic execution system.

x			*variables*
i^n		$\subseteq x$'s, $n \in \mathbb{N}^+$	*input variables*
v	::=	$s \mid \lambda x.e$	*values*
s	::=	$x^C \mid$ `true` $\mid$ `false` $\mid \cdots \mid \{\ldots$`-1,0,1,`$\ldots\}$	
		$\mid s$ `and` $s \mid s$ `or` $s \mid$ `not` $s \mid s$ `+` $s \mid s$ `-` $s \mid s$ `=` s	*symbolic values*
ϕ	::=	s with top node one of `and/or/not/=` $\mid x^C \mapsto v$	*formulae*
Φ	::=	$\phi \wedge \cdots \wedge \phi$	*global formula*
e	::=	$x \mid \lambda x.e \mid$ `true` $\mid$ `false` $\mid \{\ldots$`-1,0,1,`$\ldots\}$	
		$\mid e$ `and` $e \mid e$ `or` $e \mid$ `not` $e \mid e$ `+` $e \mid e$ `-` $e \mid e$ `=` e	
		$\mid e\ ^t e \mid$ `if` e `then` e `else` $e \mid$ `let` x `=` e `in` e	
t	::=		*call site tags, think numbers*
C	::=	$[t;\ldots;t]$	*call site lists aka contexts*

We require that all expressions e additionally are "alphatized" – they cannot bind the same variable in multiple places. This is not a practical restriction as any otherwise identical variables can be renamed. We will reserve special program variable `input` to denote an input effect, it should not be bound and programs are considered closed even if `input` is free. The syntax for input will be `let x = input in ...`. This fits into the grammar above and will require no modifications to the FbDK which is why we are not introducing new syntax for input – it will let us re-use the same **F♭** parsers, pretty printers, etc.

Global constraints on inputs are accumulated in a master formula Φ. It can equivalently be viewed as a big conjunction of ϕ or as a set of ϕ, we will pun across the two forms. We will also be putting variable bindings in Φ by adding $x^C \mapsto v$ to Φ; this is just an equivalence as far as the logical interpretation of Φ goes, so think "=" when you see these $\mapsto$; we use a different symbol to make things more readable (also recall the = only works on numbers). Notation "$\Phi(x^C) = v$" is shorthand for $\Phi = \cdots \wedge x^C \mapsto v \wedge \ldots$. It is an invariant that there will never be two different such mappings for any variable x^C in Φ, so this operation is always in fact a function.

Variables i^n for $n \in \mathbb{N}^+$ are special variables in Φ which are used to name successive inputs to the program. The first input statement will return a result equivalent to i^1, the second i^2, etc. We define a function Φ++ which given a Φ returns the least $n > 0$ such that i^n does not occur in Φ.

The formula Φ must always be *satisfiable*: there must be some assignment of numbers/-booleans to the variables in Φ such that all of the conjuncts ϕ_j of Φ hold. We write $\textsc{Sat}(\Phi)$ as a predicate asserting this fact. Function $\textsc{Sats}(\Phi)$ returns a set of all possible such mappings of variables to values.

(var) $$\frac{\Phi(x) = v}{C \vdash \langle x, \Phi\rangle \Rightarrow \langle v, \Phi\rangle}$$

(value) $$\frac{}{C \vdash \langle v, \Phi\rangle \Rightarrow \langle v, \Phi\rangle}$$

(not) $$\frac{C \vdash \langle e, \Phi\rangle \Rightarrow \langle \phi, \Phi'\rangle}{C \vdash \langle \Phi, \texttt{not } e\rangle \Rightarrow \langle \texttt{not } \phi, \Phi'\rangle}$$

(and) $$\frac{C \vdash \langle e_1, \Phi\rangle \Rightarrow \langle \phi_1, \Phi'\rangle \quad C \vdash \langle e_2, \Phi'\rangle \Rightarrow \langle \phi_2, \Phi''\rangle}{C \vdash \langle e_1 \texttt{ and } e_2, \Phi\rangle \Rightarrow \langle \phi_1 \texttt{ and } \phi_2, \Phi''\rangle}$$

(+) $$\frac{C \vdash \langle e_1, \Phi\rangle \Rightarrow \langle s_1, \Phi'\rangle \quad C \vdash \langle e_2, \Phi'\rangle \Rightarrow \langle s_2, \Phi''\rangle}{C \vdash \langle e_1 \texttt{ + } e_2, \Phi\rangle \Rightarrow \langle s_1 \texttt{ + } s_2, \Phi''\rangle}$$

(=) $$\frac{C \vdash \langle e_1, \Phi\rangle \Rightarrow \langle s_1, \Phi'\rangle \quad C \vdash \langle e_2, \Phi'\rangle \Rightarrow \langle s_2, \Phi''\rangle}{C \vdash \langle e_1 \texttt{ = } e_2, \Phi\rangle \Rightarrow \langle s_1 \texttt{ = } s_2, \Phi''\rangle}$$

(`if true`*)* $$\frac{C \vdash \langle e_1, \Phi\rangle \Rightarrow \langle \phi, \Phi'\rangle \quad \Phi'' = \Phi' \wedge \phi \quad C \vdash \langle e_2, \Phi''\rangle \Rightarrow \langle v_2, \Phi'''\rangle}{C \vdash \langle \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, \Phi\rangle \Rightarrow \langle v_2, \Phi'''\rangle}$$

(`if false`*)* $$\frac{C \vdash \langle e_1, \Phi\rangle \Rightarrow \langle \phi, \Phi'\rangle \quad \Phi'' = \Phi' \wedge (\texttt{not } \phi) \quad C \vdash \langle e_2, \Phi''\rangle \Rightarrow \langle v_2, \Phi'''\rangle}{C \vdash \langle \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, \Phi\rangle \Rightarrow \langle v_2, \Phi'''\rangle}$$

(app) $$\frac{\begin{array}{l} C \vdash \langle e_1, \Phi\rangle \Rightarrow \langle \lambda x.e, \Phi'\rangle \quad C \vdash \langle e_2, \Phi'\rangle \Rightarrow \langle v_2, \Phi''\rangle \\ t :: C \vdash \langle e[x^{t::C}/x], (\Phi'' \wedge x^{t::C} \mapsto v_2)\rangle \Rightarrow \langle v, \Phi'''\rangle \end{array}}{C \vdash \langle e_1{}^t\, e_2, \Phi\rangle \Rightarrow \langle v, \Phi'''\rangle}$$

(let) $$\frac{C \vdash \langle e_1, \Phi\rangle \Rightarrow \langle v_1, \Phi'\rangle \quad C \vdash \langle e_2[x^C/x], (\Phi' \wedge x^C \mapsto v_1)\rangle \Rightarrow \langle v_2, \Phi''\rangle \quad e_1 \neq \texttt{input}}{C \vdash \langle \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, \Phi\rangle \Rightarrow \langle v_2, \Phi''\rangle}$$

(input) $$\frac{C \vdash \langle e_2[x^C/x], (\Phi \wedge x^C = i^{\Phi++})\rangle \Rightarrow \langle v_2, \Phi'\rangle}{C \vdash \langle \texttt{let } x \texttt{ = input in } e_2, \Phi\rangle \Rightarrow \langle v_2, \Phi'\rangle}$$

Definition 3.1 (Symbolic evaluation)**.** *$C \vdash \langle e, \Phi\rangle \Rightarrow \langle v, \Phi'\rangle$ for e closed is the least relation closed under the above rules for which* $\textsc{Sat}(\Phi')$ *holds. An initial symbolic computation starts with an empty context and formula: $C_\emptyset \vdash \langle e, \Phi_\emptyset\rangle \Rightarrow \langle v, \Phi\rangle$.*

In an implementation of an evaluator, at any point where Φ becomes unsatisfiable that execution path is no longer possible and it can be aborted, but the above relation will only check satisfiability at the end of a computation.

Lemma 3.1 (Soundness)**.** *For programs with no input, this symbolic evaluator works just like a regular evaluator: For all* `input`*-free expressions e and for non-function values v, $e \Rightarrow v$*

in the standard substitution-based **F**$\mathbb{D}$ *operational semantics iff* $C_\emptyset \vdash \langle e, \Phi_\emptyset \rangle \Rightarrow \langle v, \Phi \rangle$ *in the symbolic evaluator.*

Lemma 3.2 (Nondeterminism)**.** *The* $C_\emptyset \vdash \langle e, \Phi_\emptyset \rangle \Rightarrow \langle v, \Phi \rangle$ *relation is not a function, there may be more than one value corresponding to initial expression* e*.*

For example the program `let x = input in if x = 0 then true else false` could evaluate to either `true` or `false`.

Bibliography

[1] Leandro Facchinetti, Zachary Palmer, and Scott Smith. Higher-order demand-driven program analysis. *TOPLAS*, 41, July 2019.

[2] Mike Grant, Zachary Palmer, and Scott Smith. *Principles of Programming Languages*. http://pl.cs.jhu.edu/pl/book, 2019.

[3] Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages*. PhD thesis, Pittsburgh, PA, USA, 1991. UMI Order No. GAX91-26964.

[4] David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP*, 2010.