lec15: Mutable state

Jana Dunfield

April 5, 2022

1 Introduction

Our language so far has been *purely functional*: it has no features that allow

- output (other than a final result v), such as writing to a console, output file, or network;
- input (other than input given as part of the source program), such as reading from a file, from a user device such as a keyboard, or from the network;
- *mutable state*, in which (some) program variables can be updated (re-assigned).

Except for *extremely* purely functional languages that do not provide these features in any form[1], we would like our semantics to describe these language features. Even if we only use purely functional languages, real machine architectures are not purely functional, and we would like semantics that can describe the connection between a purely functional source language and the machine code produced by a compiler.

Our existing judgment forms $e \mapsto e'$ and $e \Downarrow v$ are not suitable to describe these features. (It is *possible*, but not easy; it requires fairly drastic changes to the language, and the resulting semantics is especially difficult to understand.)

In these notes, we extend our operational semantics to describe one non-purely-functional language feature: *mutable state*.

2 State

> "State is the coldest of all cold monsters." —Friedrich Nietzsche

The form of mutable state we'll consider is more or less the same as that found in Standard ML. Specifically:

- We will continue to not support writing to a variable as such: in $(\texttt{Lam}\ x\ e_{\text{body}})$, whatever was passed for the argument x (in a `Call` expression) is the value of x throughout e_{body}; there is no general "assignment statement".
- However, our language will support *locations* in a *store*, which *can* be updated.

The resulting language is, roughly, "functional by default, imperative by request". If you want to maintain a counter that is updated, without passing the counter around as an extra argument or extra result (in our language, this could be done using a `Pair`), you can do that—but you must "request" that by explicitly allocating a location.

[1] Haskell, for example, provides input and output through monads, which hide the non-functional behaviour behind an interface that is purely functional.

2.1 Features

We will add four new expression forms: three that allocate, read, and write references; and one to describe locations themselves.

Locations	Loc	::=	$L_1 \mid L_2 \mid L_3 \mid \ldots$	
Expressions	e	::=	$\ldots$	
			$\mid$ (Alloc e)	allocate a reference (or location) initialized to e
			$\mid$ (Get e)	read the contents of a location
			$\mid$ (Set e e)	update the contents of a location
			$\mid$ Loc	location
Values	v	::=	$\ldots$	
			$\mid$ Loc	
Stores	Σ	::=	emp	empty store
			$\mid \Sigma, \text{Loc} \triangleright v$	store Σ, with additional location Loc containing v

Think of L_1, L_2, ... as addresses in memory (not necessarily allocated in that order).
We will change the judgment form $e \mapsto e'$ to a new form:

$$\Sigma; e \mapsto \Sigma'; e'$$

This judgment is read: *starting in store Σ, expression e steps to store Σ' and expression e'.*

2.2 Stepping rule

Instead of step-context deriving $\mathcal{C}[e] \mapsto \mathcal{C}[e']$, we will have a rule that derives $\Sigma; \mathcal{C}[e] \mapsto \Sigma'; \mathcal{C}[e']$.

$\boxed{\Sigma; e \mapsto \Sigma'; e'}$ starting in store Σ, expression e steps to store Σ' and expression e'

$$\frac{\Sigma; e \mapsto_R \Sigma'; e'}{\Sigma; \mathcal{C}[e] \mapsto \Sigma'; \mathcal{C}[e']} \text{ Step-context}$$

(I capitalized the S in Step-context so that we would not have two rules with exactly the same name, one for the original $e \mapsto e'$ judgment form, one for the new form $\Sigma; e \mapsto \Sigma'; e'$.)

2.3 Reduction rules

Our existing reduction rules (red-beta, etc.) should not affect the store, so they can all be copied over, only adding Σ on both sides of $\mapsto_R$.

Our three new reduction rules, one for each of our three new expression forms, all make use of Σ.

$\boxed{\Sigma; e \mapsto_R \Sigma'; e'}$ starting in store Σ, expression e reduces to store Σ' and expression e'

$$\frac{}{\Sigma; (\texttt{Call } (\texttt{Lam x } e_{body})\ v) \mapsto_R \Sigma; [v/x]e_{body}} \text{ Red-call}$$

$$\frac{\text{Loc} \notin \text{locs}(\Sigma)}{\Sigma; (\texttt{Alloc } v) \mapsto_R \Sigma, \text{Loc} \rhd v; \text{Loc}} \text{ Red-ref}$$

$$\frac{(\text{Loc} \rhd v) \in \Sigma}{\Sigma; (\texttt{Get Loc}) \mapsto_R \Sigma; v} \text{ Red-get}$$

$$\frac{}{\Sigma_1, \text{Loc} \rhd v_{old}, \Sigma_2; (\texttt{Set Loc } v_{new}) \mapsto_R \Sigma_1, \text{Loc} \rhd v_{new}, \Sigma_2; ()} \text{ Red-set}$$

Rule Red-ref uses an auxiliary definition, of a (mathematical) function locs:

$$\begin{aligned} \text{locs}(\text{emp}) &= \{\} \\ \text{locs}(\Sigma, \text{Loc} \rhd v) &= \text{locs}(\Sigma) \cup \{\text{Loc}\} \end{aligned}$$

For example, the locations in the store $\text{emp}, L_1 \rhd 1, L_2 \rhd (\texttt{Lam x x}), L_3 \rhd ()$ are given by

$$\begin{aligned} \text{locs}(\text{emp}, L_1 \rhd 1, L_2 \rhd (\texttt{Lam x x}), L_3 \rhd ()) &= \text{locs}(\text{emp}, L_1 \rhd 1, L_2 \rhd (\texttt{Lam x x})) \cup \{L_3\} \\ &= \text{locs}(\text{emp}, L_1 \rhd 1) \cup \{L_2\} \cup \{L_3\} \\ &= \text{locs}(\text{emp}) \cup \{L_1\} \cup \{L_2\} \cup \{L_3\} \\ &= \{\} \cup \{L_1\} \cup \{L_2\} \cup \{L_3\} = \{L_1, L_2, L_3\} \end{aligned}$$

The choice of `()` in Red-set is quite arbitrary (it is the choice made by my favourite language, Standard ML); other vaguely reasonable options include returning v_{new} (similar to C's semantics for "chained" assignment), returning v_{old}, or returning Loc.

In addition to the meaningful, but arbitrary, choice of what expression to return, there are many ways to express "change Lv_{old} to Lv_{new}", some of which we discussed in class. The way I have chosen here is different from all of those: Red-set "decomposes" the store into a left-hand part Σ_1 and a right-hand part Σ_2, with Lv_{old} in the middle.

Exercise 1. Rewrite Red-get to use the same technique as the Red-set above.

2.4 Evaluation contexts

In class, as we worked out the reduction rules, we added to the evaluation contexts.

For each new expression form, we need a production for each of that form's subexpressions, allowing us to reduce non-values to values. Thus, `Alloc` has one production for its one subexpression, etc.

$$\begin{array}{rrl} \text{Evaluation contexts} & ::= & \ldots \\ & & \mid (\texttt{Alloc}\ \mathcal{C}) \\ & & \mid (\texttt{Get}\ \mathcal{C}) \\ & & \mid (\texttt{Set}\ \mathcal{C}\ e) \\ & & \mid (\texttt{Set}\ v\ \mathcal{C}) \end{array}$$

3 Typing

We add a new form of type, Ref T, which describes references to locations (if you like, pointers to locations) containing values of type T. For example, Ref int is a reference to an integer, and Ref (bool → bool) is a reference to a function over booleans.

$$\begin{array}{rrl} \text{Types} & S, T & ::= \ldots \\ & & \mid \mathsf{Ref}\ T \end{array}$$

The expression $(\texttt{Alloc}\ e)$ is the introduction form for Ref T. The expressions $(\texttt{Get}\ e)$ and $(\texttt{Set}\ e_1\ e_2)$ are elimination forms for Ref T. (The fact that Ref T has two elimination forms with no clear symmetry—an immediate symmetry of the kind seen in $(\texttt{Proj}_1\ e)$ and $(\texttt{Proj}_2\ e)$—seems out of line with natural deduction, but mutable state seems essentially out of line with natural deduction, so not much can be done.)

We might then expect to write the following typing rules. (These are actually wrong!)

$$\frac{\Gamma \vdash e : S}{\Gamma \vdash (\texttt{Alloc}\ e) : \mathsf{Ref}\ S}\ \text{type-alloc} \qquad \frac{\Gamma \vdash e : \mathsf{Ref}\ S}{\Gamma \vdash (\texttt{Get}\ e) : S}\ \text{type-get} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Ref}\ S \qquad \Gamma \vdash e_2 : S}{\Gamma \vdash (\texttt{Set}\ e_1\ e_2) : \mathsf{unit}}\ \text{type-set}$$

Rule type-alloc says that if e has type S, then $(\texttt{Alloc}\ e)$ has type Ref S, since it creates a reference to a location containing something of type S.

Rule type-get says that if e is a reference to a location containing something of type S, then $(\texttt{Get}\ e)$ has type S, because it extracts the contents of the location.

Rule type-set says that if e_1 is a reference to a location containing something of type S and e_2 has type S, then $(\texttt{Set}\ e_1\ e_2)$ has type unit.

These seem fine as far they go, but if we check our grammar of expressions (Section 2.1), we added four new expression forms. So we need (at least) four new typing rules. We're missing a rule for locations, Loc, and now we have a problem: we can't write such a rule.

A location has a type only if it's been allocated, and we don't have anything to tell us which locations have been allocated (or to tell us the type of the contents). To do this, we need to change the judgment form from

$$\Gamma \vdash e : S$$

to

$$\Lambda; \Gamma \vdash e : S$$

where Λ is a *store typing* that lists which locations exist (have been allocated), along with their types.

3.1 Store typing

The grammar for store typings is extremely similar to the grammar for typing contexts Γ:

$$\begin{array}{rrcl} \text{Store typings} & \Lambda & ::= & \emptyset \\ & & | & \Lambda, \text{Loc} : S \end{array}$$

For example, the store typing $\emptyset, L_1 : \text{int}, L_2 : \text{bool}$ describes a store with two locations L_1 (containing an integer) and L_2 (containing a boolean).

3.2 The new typing rules

We can now write a typing rule for locations that follows the rule type-assum, except that instead of taking the type S directly from the store typing Λ, we add `Ref` to it:

$$\frac{(\text{Loc} : S) \in \Lambda}{\Lambda; \Gamma \vdash \text{Loc} : \text{Ref } S} \text{ type-loc}$$

Our proposed rules for the features specific to references only change slightly: whenever we had Γ, we need to have $\Lambda; \Gamma$.

$$\frac{\Lambda; \Gamma \vdash e : S}{\Lambda; \Gamma \vdash (\texttt{Alloc } e) : \text{Ref } S} \text{ type-alloc} \qquad \frac{\Lambda; \Gamma \vdash e : \text{Ref } S}{\Lambda; \Gamma \vdash (\texttt{Get } e) : S} \text{ type-get}$$

$$\frac{\Lambda; \Gamma \vdash e_1 : \text{Ref } S \qquad \Lambda; \Gamma \vdash e_2 : S}{\Lambda; \Gamma \vdash (\texttt{Set } e_1\ e_2) : \text{unit}} \text{ type-set}$$

3.3 Updating the existing typing rules

The other rules have to be changed similarly. For example, the rule for `Lam`, which is

$$\frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash (\texttt{Lam } x\ e) : (S \to T)} \to\text{Intro}$$

becomes

$$\frac{\Lambda; \Gamma, x : S \vdash e : T}{\Lambda; \Gamma \vdash (\texttt{Lam } x\ e) : (S \to T)} \to\text{Intro}$$

3.4 Updating the statements of preservation and progress

If we mechanically changed the statement of preservation by adding Λs and Σs, we might get

Conjecture 1 (Type Preservation)**.**
If $\Lambda; \emptyset \vdash e : S$
and $\Sigma; e \mapsto \Sigma'; e'$
then $\Lambda; \emptyset \vdash e' : S$.

The typing context is still empty, but we need to allow a non-empty store typing, because Σ might contain locations.

This statement is not right, because it doesn't say relate Σ and Λ. Specifically:

- The store typing Λ must have a type for every location in Σ. For example, if Σ contains $L_3 \triangleright \cdots$ then Λ must have a type for L_3.
- If Λ says that a location, say L_3, contains an integer—that is, $(L_3 : \mathsf{int}) \in \Lambda$—then the value associated with L_3 in Σ must have type int. If $(L_3 : \mathsf{int}) \in \Lambda$ but $(L_3 \triangleright \mathtt{False}) \in \Sigma$, then Σ should *not* have the store typing Λ.
- The store typing Λ should not have more locations than the store: we don't want to say "the empty store has the store typing $\emptyset, L_3 : \mathsf{int}$".

The judgment form for store typing will be $\vdash \Sigma : \Lambda$, read "store Σ has store typing Λ".

The (hopefully) correct statement is, then, as follows. It says that if initial store Σ has store typing Λ, and e steps to e' with an updated store Σ', then there is a (possibly empty) extension Λ' to the store context such that e' has type S under (Λ, Λ')—that is, Λ' added to the end of Λ.

Conjecture 2 (Type Preservation)**.**
If $\Lambda; \emptyset \vdash e : S$
and $\vdash \Sigma : \Lambda$
and $\Sigma; e \mapsto \Sigma'; e'$
then there exists Λ' *such that* $\Lambda, \Lambda'; \emptyset \vdash e' : S$.

3.5 Rules for store typing

I'm doing this slightly differently from the lecture: I'll use two judgment forms, $\vdash \Sigma : \Lambda$ (as used in the statement of type preservation) and $\Lambda \vdash \Sigma : \Lambda_0$, which "recursively" traverses through Σ and Λ_0 maintaining Λ unchanged.

The $\vdash \Sigma : \Lambda$ form has only one rule, with one premise; this rule "delegates" the real work to the second judgment form.

$\boxed{\vdash \Sigma : \Lambda}$ Store Σ has store typing Λ

$$\frac{\Lambda \vdash \Sigma : \Lambda}{\vdash \Sigma : \Lambda}$$

$\boxed{\Lambda \vdash \Sigma : \Lambda_0}$ Under store typing Λ, partial store Σ has partial store typing Λ_0

$$\frac{}{\Lambda \vdash \mathsf{emp} : \emptyset} \qquad \frac{\Lambda \vdash \Sigma : \Lambda_0 \qquad \Lambda \vdash \mathrm{Loc} : S}{\Lambda (\vdash (\Sigma, \mathrm{Loc} \triangleright v) : (\Lambda_0, \mathrm{Loc} : S)}$$

References