Chapter 1

References

In the (typed or untyped) λ-calculus, or in "pure" functional languages, a variable is immutable in that once bound to a value as the result of a substitution, its contents never change. While it may appear to be too restrictive or even strange from the perspective of imperative programming, immutability of variables allows us to consider λ-abstractions as equivalent to mathematical functions whose meaning does not change, and thus makes functional programming more elegant and readable than other programming paradigms. For example, the following λ-abstraction denotes a mathematical function taking a boolean value x and returning a logical conjunction of x and y where the value of y is determined at the time of creating a closure out of it:

$$\lambda x\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false}$$

Then the meaning of the λ-abstraction (or the closure created out of it, to be precise) does not change throughout the evaluation; hence we only have to look at the λ-abstraction itself to learn what it means.

This chapter extends the simply typed λ-calculus with *mutable references*, or *references* for short, in the presence of which λ-abstractions no longer denote mathematical functions. We will introduce three new constructs for manipulating references; references may be thought of as another name for pointers familiar from imperative programming, and all these constructs find their counterparts in imperative languages:

- $\mathsf{ref}\ e$ creates or *allocates* a reference pointing to the value to which e evaluates.
- $!e$ obtains a reference by evaluating e, and then *dereferences* it, *i.e.*, retrieves the contents of it.
- $e := e'$ obtains a reference and a value by evaluating e and e', and then updates the contents of the reference with the value. That is, it *assigns* a new value to a reference.

It is easy to see that if a λ-abstraction is "contaminated" with references, it no longer denotes a mathematical function. For example, the meaning of the following λ-abstraction depends on the contents of a reference in y, and we cannot decide its meaning once and for all:

$$\lambda x\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ !y\ \mathsf{else}\ \mathsf{false}$$

In other words, each time we invoke the above λ-abstraction (or the closure created out of it, to be precise), we have to look up an environment (called a *store* or a *heap*) to obtain the value of $!y$. It is certainly either true or false, but we can never decide its meaning by looking only at the λ-abstraction itself. Hence it does not denote a mathematical function.

In general, we refer to those constructs that destroy the connection between λ-abstractions and mathematical functions, or the "purity" of the λ-calculus, as *computational effects*. References are the most common form of computational effects; other kinds of computational effects include continuations, exceptions, and input/output. A functional language with such features as references is often called an "impure" functional language.[1]

[1] SML is an impure functional language. Haskell is a pure functional language, although it comes with a few impure language constructs.

Below we extend the type system and the operational semantics of the simply typed λ-calculus to incorporate the three constructs for references. The development will be incremental in that if we remove the new constructs, the "impure" definition reverts to the "pure" definition of the simply typed λ-calculus. An important implication is that immutability of variables is *not* affected by the new constructs — it is the contents of a reference that change; the contents of a variable never change!

1.1 Abstract syntax and type system

We augment the simply typed λ-calculus with three constructs for references; for the sake of simplicity, we do not consider base types P:

$$\begin{array}{rrcl} \textsf{type} & A & ::= & P \mid A \rightarrow A \mid \textsf{unit} \mid \textsf{ref}\ A \\ \textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\,e \mid e\ e \mid () \mid \textsf{ref}\ e \mid\ !e \mid e := e \\ \textsf{value} & v & ::= & \lambda x\!:\!A.\,e \mid () \end{array}$$

A *reference type* ref A is the type for references pointing to values of type A, or equivalently, the type for references whose contents are values of type A. All the constructs for references use reference types, and behave as follows:

- ref e evaluates e to obtain a value v and then allocates a reference initialized with v; hence, if e has type A, then ref e has type ref A. It uses the same keyword ref as in reference types to maintain consistency with the syntax of SML.
- $!e$ evaluates e to obtain a reference and then retrieves its contents; hence, if e has type ref A, then $!e$ has type A.
- $e := e'$ evaluates e to obtain a reference and then updates its contents with a value obtained by evaluating e'. Since the result of updating the contents of a reference is computationally meaningless, $e := e'$ is assigned a unit type unit.

Thus we obtain the following typing rules for the constructs for references:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \textsf{ref}\ e : \textsf{ref}\ A}\ \textsf{Ref} \qquad \frac{\Gamma \vdash e : \textsf{ref}\ A}{\Gamma \vdash\ !e : A}\ \textsf{Deref} \qquad \frac{\Gamma \vdash e : \textsf{ref}\ A \quad \Gamma \vdash e' : A}{\Gamma \vdash e := e' : \textsf{unit}}\ \textsf{Assign}$$

For proving type safety, we need to elaborate the typing judgment $\Gamma \vdash e : A$ (see Section 1.3), but for type-checking a given expression, the above typing rules suffice.

Before we give the reduction rules for the new constructs, let us consider a couple of examples exploiting references. Both examples use syntactic sugar let $x = e$ in e' for $(\lambda x\!:\!A.\,e')\ e$ for some type A. That is, let $x = e$ in e' first evaluates e and store the result in x; then it evaluates e'. We may think of the rule Let below as the typing rule for let $x = e$ in e':

$$\frac{\Gamma \vdash e : A \quad \Gamma, x : A \vdash e' : B}{\Gamma \vdash \textsf{let}\ x = e\ \textsf{in}\ e' : B}\ \textsf{Let}$$

Both examples also assume common constructs for types bool and int (*e.g.*, if e then e_1 else e_2 as a conditional construct, $+$ for addition, $-$ for subtraction, $=$ for equality, and so on). We use a wildcard pattern _ for a variable not used in the body of a λ-abstraction:

The first example exploits references to simulate arrays of integers. We choose a functional representation of arrays by defining type iarray for arrays of integers as follows:

$$\textsf{iarray} = \textsf{ref}\ (\textsf{int} \rightarrow \textsf{int})$$

That is, we represent an array of integers as a function taking an index (of type int) and returning a corresponding element of the array. We need the following constructs for arrays:

- $\mathsf{new} : \mathsf{unit} \to \mathsf{iarray}$ for creating a new array.
  new $()$ returns a new array of indefinite size; all elements are initialized as 0.
- $\mathsf{access} : \mathsf{iarray} \to \mathsf{int} \to \mathsf{int}$ for accessing an array.
  $\mathsf{access}\ a\ i$ returns the i-th element of array a.
- $\mathsf{update} : \mathsf{iarray} \to \mathsf{int} \to \mathsf{int} \to \mathsf{unit}$ for updating an array.
  $\mathsf{update}\ a\ i\ n$ updates the i-the element of array a with integer n.

Exercise 1.1. Implement new, access, and update.

We implement new and access according to the definition of type iarray:

$$\begin{array}{rcl} \mathsf{new} & = & \lambda\_ : \mathsf{unit}.\ \mathsf{ref}\ \lambda i : \mathsf{int}.\ 0 \\ \mathsf{access} & = & \lambda a : \mathsf{iarray}.\ \lambda i : \mathsf{int}.\ (!a)\ i \end{array}$$

To implement update, we have to first retrieve a function of type $\mathsf{int} \to \mathsf{int}$ from a given array and then build a new function of type $\mathsf{int} \to \mathsf{int}$:

$$\begin{array}{rcl} \mathsf{update} & = & \lambda a : \mathsf{iarray}.\ \lambda i : \mathsf{int}.\ \lambda n : \mathsf{int}. \\ & & \mathsf{let}\ old = !a\ \mathsf{in} \\ & & a := \lambda j : \mathsf{int}.\ \mathsf{if}\ i = j\ \mathsf{then}\ n\ \mathsf{else}\ old\ j \end{array}$$

The following implementation of update has a correct type, but is wrong: a in the body does not point to the old array that exists *before* the update, but ends up pointing to the same array that is created *after* the update:

$$\begin{array}{rcl} \mathsf{update} & = & \lambda a : \mathsf{iarray}.\ \lambda i : \mathsf{int}.\ \lambda n : \mathsf{int}. \\ & & a := \lambda j : \mathsf{int}.\ \mathsf{if}\ i = j\ \mathsf{then}\ n\ \mathsf{else}\ (!a)\ j \end{array}$$

The wrong implementation of update illustrates that a reference a can be assigned a value that dereferences the same reference a. We can exploit such "self-references" to implement recursive functions without using the fixed point construct. The second example implements the following recursive function (written in the syntax from Section **??**) which takes an integer n and returns the sum of integers from 0 to n:

$$\mathsf{fun}\ f\ n : \mathsf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 0\ \mathsf{else}\ n + f\ (n-1)$$

To implement the above recursive function, we first allocate a reference f initialized with a dummy function of type $\mathsf{int} \to \mathsf{int}$. Then we assign the reference f a function which dereferences the same reference f when its argument is not equal to 0, thereby effecting a recursive call:

$$\begin{array}{l} \mathsf{let}\ f = \mathsf{ref}\ \lambda n : \mathsf{int}.\ 0\ \mathsf{in} \\ \mathsf{let}\ \_ = f := \lambda n : \mathsf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 0\ \mathsf{else}\ n + (!f)\ (n-1)\ \mathsf{in} \\ !f \end{array}$$

1.2 Operational semantics

The operational semantics for references needs a reduction judgment that departs from the previous reduction judgment $e \mapsto e'$ for the simply typed λ-calculus. To see why, consider how to reduce $\mathsf{ref}\ v$ for a value v. The operational semantics needs to allocate a reference initialized with v, but allocates it where? In fact, the abstract syntax given in the previous section does not even include expressions for results of allocating references. That is, what is the syntax for the result of reducing $\mathsf{ref}\ v$? Thus we are led to extend both the abstract syntax to include values for references and the reduction judgment to record the contents of all references allocated during an evaluation.

We use a *store* ψ to record the contents of all references. A store is a mapping from *locations* to values, where a location l is a value for a reference, or simply another name for a reference. (We do not call l a "pointer" to emphasize that arithmetic operations may not be applied on l.) As we will see in the reduction rules below, a fresh location l is created only by reducing $\mathsf{ref}\ v$, in which case the store is extended so as to map l to v. We define a store as an unordered collection of bindings of the form $l \mapsto v$:

$$
\begin{array}{rrcll}
\textsf{expression} & e & ::= & \cdots \mid l & \text{location} \\
\textsf{value} & v & ::= & \cdots \mid l & \text{location} \\
\textsf{store} & \psi & ::= & \cdot \mid \psi, l \mapsto v &
\end{array}
$$

We write $dom(\psi)$ for the domain of ψ, *i.e.*, the set of locations mapped to certain values under ψ. Formally we define $dom(\psi)$ as follows:

$$
\begin{array}{rcl}
dom(\cdot) & = & \varnothing \\
dom(\psi, l \mapsto v) & = & dom(\psi) \cup \{l\}
\end{array}
$$

We write $[l \mapsto v]\psi$ for the store obtained by updating the contents of l in ψ with v. Note that in order for $[l \mapsto v]\psi$ to be defined, l must be in $dom(\psi)$:

$$[l \mapsto v](\psi', l \mapsto v') \quad = \quad \psi', l \mapsto v$$

We write $\psi(l)$ for the value to which l is mapped under ψ; in order for $\psi(l)$ to be defined, l must be in $dom(\psi)$:

$$(\psi', l \mapsto v)(l) \quad = \quad v$$

Since the reduction of an expression may need to access or update a store, we use the following reduction judgment which carries a store along with an expression being reduced:

$$e \mid \psi \mapsto e' \mid \psi' \qquad \Leftrightarrow \qquad \textit{e with store } \psi \textit{ reduces to } e' \textit{ with store } \psi'$$

In the judgment $e \mid \psi \mapsto e' \mid \psi'$, we definitely have $e \neq e'$, but ψ and ψ' may be the same if the reduction of e does not make a change to ψ. The reduction rules are given as follows:

$$\frac{e_1 \mid \psi \mapsto e_1' \mid \psi'}{e_1\ e_2 \mid \psi \mapsto e_1'\ e_2 \mid \psi'}\ \mathit{Lam} \qquad \frac{e_2 \mid \psi \mapsto e_2' \mid \psi'}{(\lambda x\!:\!A.\, e)\ e_2 \mid \psi \mapsto (\lambda x\!:\!A.\, e)\ e_2' \mid \psi'}\ \mathit{Arg}$$

$$\frac{}{(\lambda x\!:\!A.\, e)\ v \mid \psi \mapsto [v/x]e \mid \psi}\ \mathit{App}$$

$$\frac{e \mid \psi \mapsto e' \mid \psi'}{\mathsf{ref}\ e \mid \psi \mapsto \mathsf{ref}\ e' \mid \psi'}\ \mathit{Ref} \qquad \frac{l \notin dom(\psi)}{\mathsf{ref}\ v \mid \psi \mapsto l \mid \psi, l \mapsto v}\ \mathit{Ref}'$$

$$\frac{e \mid \psi \mapsto e' \mid \psi'}{!e \mid \psi \mapsto !e' \mid \psi'}\ \mathit{Deref} \qquad \frac{\psi(l) = v}{!l \mid \psi \mapsto v \mid \psi}\ \mathit{Deref}'$$

$$\frac{e \mid \psi \mapsto e'' \mid \psi'}{e := e' \mid \psi \mapsto e'' := e' \mid \psi'}\ \mathit{Assign} \qquad \frac{e \mid \psi \mapsto e' \mid \psi'}{l := e \mid \psi \mapsto l := e' \mid \psi'}\ \mathit{Assign}'$$

$$\frac{}{l := v \mid \psi \mapsto () \mid [l \mapsto v]\psi}\ \mathit{Assign}''$$

Note that locations are runtime values: they are not part of the syntax for the source language; they are created only at runtime by reducing $\mathsf{ref}\ e$.

1.3 Type safety

This section proves type safety of the simply typed λ-calculus extended with references. First of all, we have to extend the type system for locations which are valid (runtime) expressions but have not been given a typing rule. The development of the typing rule for locations gives rise to a new form of typing judgment, since deciding the type of a location requires information on a store, but the previous typing judgment $\Gamma \vdash e : A$ does not include information on a store. Then a proof of type safety rewrites the typing rules Ref, Deref, Assign in terms of the new typing judgment, although these typing rules suffice for the purpose of typechecking an expression with no location in it.

Let us begin with the following (blatantly wrong) typing rule for locations:

$$\frac{}{\Gamma \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc}_1$$

The rule Loc_1 does not make sense: we wish to assign l a reference type $\mathsf{ref}\ A$ only if it is mapped to a value of type A under a certain store, but the rule does not even inspect the value to which l is mapped. The following typing rule uses a new form of typing judgment involving a store ψ, and assigns l a reference type $\mathsf{ref}\ A$ only if it is mapped to a value of type A under ψ:

$$\frac{\psi(l) = v \quad \Gamma \vdash v : A}{\Gamma \mid \psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc}_2$$

The rule Loc_2 is definitely an improvement over the rule Loc_1, but it is still inadequate: it uses an ordinary typing judgment $\Gamma \vdash v : A$ on the assumption that value v does not contain locations, but in general, any value in a store may contain locations. For example, it is perfectly fine to store a pair of locations in a store. Thus we are led to typecheck v in the premise of the rule Loc_2 using the same typing judgment as in the conclusion:

$$\frac{\psi(l) = v \quad \Gamma \mid \psi \vdash v : A}{\Gamma \mid \psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc}_3$$

Unfortunately the rule Loc_3 has a problem, too: if a location l is mapped to a value containing l under ψ, the derivation of $\Gamma \mid \psi \vdash l : \mathsf{ref}\ A$ never terminates because of the infinite chain of typing judgments all of which address the same location l:

$$\frac{\psi(l) = \cdots\ l\ \cdots \qquad \overline{\begin{array}{c}\dfrac{\vdots}{\Gamma \mid \psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc}_3 \\ \vdots\end{array}}\\ \Gamma \mid \psi \vdash \cdots\ l\ \cdots : A}{\Gamma \mid \psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc}_3$$

Thus, for example, it is impossible to prove type safety for the expression in Section 1.1 for implementing the recursive function adding integers from 0 to n.

We fix the rule Loc_3 by analyzing a store only once, rather than each time a location is encountered during typechecking. To this end, we introduce a *store typing context* which records types of all values in a store:

$$\text{store typing context} \qquad \Psi \quad ::= \quad \cdot \mid \Psi, l \mapsto A$$

The idea is that if a store maps location l_i to value v_i of type A_i, it is given a store typing context mapping l_i to A_i (for $i = 1, \cdots, n$). Given a store typing context Ψ corresponding to a store ψ, then, we use Ψ, instead of ψ, for typechecking expressions:

$$\Gamma \mid \Psi \vdash e : A \quad \Leftrightarrow \quad \textit{expression } e \textit{ has type } A \textit{ under typing context } \Gamma \textit{ and store typing context } \Psi$$

We write $dom(\Psi)$ for the domain of Ψ:

$$
\begin{array}{rcl}
dom(\cdot) & = & \varnothing \\
dom(\Psi, l \mapsto A) & = & dom(\Psi) \cup \{l\}
\end{array}
$$

We write $\Psi(l)$ for the type to which l is mapped under Ψ; in order for $\Psi(l)$ to be defined, l must be in $dom(\Psi)$:

$$
(\Psi', l \mapsto A)(l) \quad = \quad A
$$

Now the typing rule for locations looks up a store typing context included in the typing judgment:

$$
\frac{\Psi(l) = A}{\Gamma \mid \Psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc}
$$

All other typing rules are obtained by replacing $\Gamma \vdash e : A$ by $\Gamma \mid \Psi \vdash e : A$ in the previous typing rules:

$$
\frac{x : A \in \Gamma}{\Gamma \mid \Psi \vdash x : A}\ \mathsf{Var} \qquad \frac{\Gamma, x : A \mid \Psi \vdash e : B}{\Gamma \mid \Psi \vdash \lambda x{:}A.\, e : A \rightarrow B}\ {\rightarrow}\mathsf{I}
$$

$$
\frac{\Gamma \mid \Psi \vdash e : A \rightarrow B \quad \Gamma \mid \Psi \vdash e' : A}{\Gamma \mid \Psi \vdash e\, e' : B}\ {\rightarrow}\mathsf{E} \qquad \frac{}{\Gamma \mid \Psi \vdash () : \mathsf{unit}}\ \mathsf{Unit}
$$

$$
\frac{\Gamma \mid \Psi \vdash e : A}{\Gamma \mid \Psi \vdash \mathsf{ref}\ e : \mathsf{ref}\ A}\ \mathsf{Ref} \qquad \frac{\Gamma \mid \Psi \vdash e : \mathsf{ref}\ A}{\Gamma \mid \Psi \vdash !e : A}\ \mathsf{Deref}
$$

$$
\frac{\Gamma \mid \Psi \vdash e : \mathsf{ref}\ A \quad \Gamma \mid \Psi \vdash e' : A}{\Gamma \mid \Psi \vdash e := e' : \mathsf{unit}}\ \mathsf{Assign}
$$

The remaining question is: how do we decide a store typing context Ψ corresponding to a store ψ? We use a new judgment

$$
\Gamma \vdash \psi :: \Psi
$$

to mean that Ψ corresponds to ψ under typing context Γ, or simply, ψ is *well-typed* with Ψ under typing context Γ. Then the goal is to give an inference rule for the judgment $\Gamma \vdash \psi :: \Psi$.

Suppose that ψ maps l_i to v_i for $i = 1, \cdots, n$. Loosely speaking, Ψ maps l_i to A_i if v_i has type A_i. Then how do we verify that v_i has type A_i? Since we are in the process of deciding Ψ (which is unknown yet), $\Gamma \mid \cdot \vdash v_i : A_i$ may appear to be the right judgment. The judgment is, however, inadequate because v_i itself may contain locations pointing to other values in the same store ψ. Therefore we have to typecheck *all locations simultaneously using the same typing context currently being decided*:

$$
\frac{dom(\Psi) = dom(\psi) \quad \Gamma \mid \Psi \vdash \psi(l) : \Psi(l) \ \text{ for every } l \in dom(\psi)}{\Gamma \vdash \psi :: \Psi}\ \mathsf{Store}
$$

Type safety is is stated as follows:

Theorem 1.2 (Progress). *Suppose that expression e satisfies $\cdot \mid \Psi \vdash e : A$ for some store typing context Ψ and type A. Then either:*
(1) e is a value, or
(2) for any store ψ such that $\cdot \vdash \psi :: \Psi$, there exist some expression e' and store ψ' such that $e \mid \psi \mapsto e' \mid \psi'$.

Theorem 1.3 (Type preservation). *Suppose*
$\Gamma \mid \Psi \vdash e : A$,
$\Gamma \vdash \psi :: \Psi$,
$e \mid \psi \mapsto e' \mid \psi'$.
Then there exists a store typing context Ψ' such that
$\Gamma \mid \Psi' \vdash e' : A$,
$\Psi \subset \Psi'$,
$\Gamma \vdash \psi' :: \Psi'$.

In Theorem 1.3, ψ' may extend ψ by the rule Ref', in which case Ψ' also extends Ψ, *i.e.*, $\Psi' \supset \Psi$ and $\Psi' \neq \Psi$. Note also that ψ' is not always a superset of ψ (*i.e.*, $\psi' \not\supset \psi$) because the rule $Assign''$ updates ψ without extending it. Even in this case, however, $\Psi' \supset \Psi$ still holds because Ψ' and Ψ are the same.

We use type safety to show that a well-typed expression cannot go wrong. Suppose that we are reducing a closed well-typed expression e with a well-typed store ψ. That is, $\cdot \vdash \psi :: \Psi$ and $\cdot \mid \Psi \vdash e : A$ hold for some store typing context Ψ and type A. If e is already a value, the reduction has been finished. Otherwise Theorem 1.2 guarantees that there exist some expression e' and store ψ' such that $e \mid \psi \mapsto e' \mid \psi'$. By Theorem 1.3, then, there exists a store typing context Ψ' such that $\cdot \mid \Psi' \vdash e' : A$ and $\cdot \vdash \psi' :: \Psi'$. That is, e' is a close well-typed expression and ψ' is a well-typed store (with Ψ'). Therefore the reduction of e with ψ cannot go wrong!