13 *References*

So far, we have considered a variety of *pure* language features, including functional abstraction, basic types such as numbers and booleans, and structured types such as records and variants. These features form the backbone of most programming languages—including purely functional languages such as Haskell, "mostly functional" languages such as ML, imperative languages such as C, and object-oriented languages such as Java.

Most practical programming languages also include various *impure* features that cannot be described in the simple semantic framework we have used so far. In particular, besides just yielding results, evaluation of terms in these languages may assign to mutable variables (reference cells, arrays, mutable record fields, etc.), perform input and output to files, displays, or network connections, make non-local transfers of control via exceptions, jumps, or continuations, engage in inter-process synchronization and communication, and so on. In the literature on programming languages, such "side effects" of computation are more generally referred to as *computational effects*.

In this chapter, we'll see how one sort of computational effect—mutable references—can be added to the calculi we have studied. The main extension will be dealing explicitly with a *store* (or *heap*). This extension is straightforward to define; the most interesting part is the refinement we need to make to the statement of the type preservation theorem (13.5.3). We consider another kind of effect—exceptions and non-local transfer of control—in Chapter 14.

13.1 Introduction

Nearly every programming language[1] provides some form of *assignment* operation that changes the contents of a previously allocated piece of storage.

The system studied in this chapter is the simply typed lambda-calculus with `Unit` and references (Figure 13-1). The associated OCaml implementation is `fullref`.

1. Even "purely functional" languages such as Haskell, via extensions such as *monads*.

In some languages—notably ML and its relatives—the mechanisms for name-binding and those for assignment are kept separate. We can have a variable `x` whose value is the number `5`, or a variable `y` whose value is a *reference* (or *pointer*) to a mutable cell whose current contents is `5`, and the difference is visible to the programmer. We can add `x` to another number, but not assign to it. We can use `y` directly to assign a new value to the cell that it points to (by writing `y:=84`), but we cannot use it directly as an argument to `plus`. Instead, we must explicitly *dereference* it, writing `!y` to obtain its current contents. In most other languages—in particular, in all members of the C family, including Java—every variable name refers to a mutable cell, and the operation of dereferencing a variable to obtain its current contents is implicit.[2]

For purposes of formal study, it is useful to keep these mechanisms separate;[3] our development in this chapter will closely follow ML's model. Applying the lessons learned here to C-like languages is a straightforward matter of collapsing some distinctions and rendering certain operations such as dereferencing implicit instead of explicit.

Basics

The basic operations on references are *allocation, dereferencing,* and *assignment*. To allocate a reference, we use the `ref` operator, providing an initial value for the new cell.

```
r = ref 5;
```

```
▸ r : Ref Nat
```

The response from the typechecker indicates that the value of `r` is a reference to a cell that will always contain a number. To read the current value of this cell, we use the dereferencing operator `!`.

```
!r;
```

```
▸ 5 : Nat
```

To change the value stored in the cell, we use the assignment operator.

2. Strictly speaking, most variables of type `T` in C or Java should actually be thought of as pointers to cells holding values of type `Option(T)`, reflecting the fact that the contents of a variable can be either a proper value or the special value `null`.

3. There are also good arguments that this separation is desirable from the perspective of language design. Making the use of mutable cells an explicit choice rather than the default encourages a mostly functional programming style where references are used sparingly; this practice tends to make programs significantly easier to write, maintain, and reason about, especially in the presence of features like concurrency.

```
r := 7;
▸ unit : Unit
```

(The result the assignment is the trivial `unit` value; see §11.2.) If we dereference `r` again, we see the updated value.

```
!r;
▸ 7 : Nat
```

Side Effects and Sequencing

The fact that the result of an assignment expression is the trivial value `unit` fits nicely with the *sequencing* notation defined in §11.3, allowing us to write

```
(r:=succ(!r); !r);
▸ 8 : Nat
```

instead of the equivalent, but more cumbersome,

```
(λ_:Unit. !r) (r := succ(!r));
▸ 9 : Nat
```

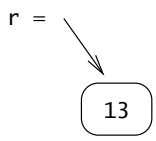
to evaluate two expressions in order and return the value of the second. Restricting the type of the first expression to `Unit` helps the typechecker to catch some silly errors by permitting us to throw away the first value only if it is really guaranteed to be trivial.

Notice that, if the second expression is also an assignment, then the type of the whole sequence will be `Unit`, so we can validly place it to the left of another `;` to build longer sequences of assignments:

```
(r:=succ(!r); r:=succ(!r); r:=succ(!r); r:=succ(!r); !r);
▸ 13 : Nat
```

References and Aliasing

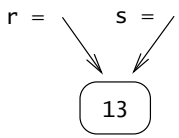
It is important to bear in mind the difference between the *reference* that is bound to `r` and the *cell* in the store that is pointed to by this reference.



If we make a copy of r, for example by binding its value to another variable s,

```
  s = r;
▸ s : Ref Nat
```

what gets copied is only the reference (the *arrow* in the diagram), not the cell:



We can verify this by assigning a new value into s

```
  s := 82;
▸ unit : Unit
```

and reading it out via r:

```
  !r;
▸ 82 : Nat
```

The references r and s are said to be *aliases* for the same cell.

13.1.1 EXERCISE [⋆]: Draw a similar diagram showing the effects of evaluating the expressions a = {ref 0, ref 0} and b = (λx:Ref Nat. {x,x}) (ref 0). □

Shared State

The possibility of aliasing can make programs with references quite tricky to reason about. For example, the expression (r:=1; r:=!s), which assigns 1 to r and then immediately overwrites it with s's current value, has exactly the same effect as the single assignment r:=!s, *unless* we write it in a context where r and s are aliases for the same cell.

Of course, aliasing is also a large part of what makes references useful. In particular, it allows us to set up "implicit communication channels"—shared state—between different parts of a program. For example, suppose we define a reference cell and two functions that manipulate its contents:

```
  c = ref 0;
▸ c : Ref Nat
```

```
  incc = λx:Unit. (c := succ (!c); !c);
▸ incc : Unit → Nat
  decc = λx:Unit. (c := pred (!c); !c);
▸ decc : Unit → Nat
```

Calling `incc`

```
  incc unit;
▸ 1 : Nat
```

results in changes to `c` that can be observed by calling `decc`:

```
  decc unit;
▸ 0 : Nat
```

If we package `incc` and `decc` together into a record

```
  o = {i = incc, d = decc};
▸ o : {i:Unit→Nat, d:Unit→Nat}
```

then we can pass this whole structure around as a unit and use its components to perform incrementing and decrementing operations on the shared piece of state in `c`. In effect, we have constructed a simple kind of *object*. This idea is developed in detail in Chapter 18.

References to Compound Types

A reference cell need not contain just a number: the primitives above allow us to create references to values of any type, including functions. For example, we can use references to functions to give a (not very efficient) implementation of arrays of numbers, as follows. Write `NatArray` for the type `Ref (Nat→Nat)`.

```
  NatArray = Ref (Nat→Nat);
```

To build a new array, we allocate a reference cell and fill it with a function that, when given an index, always returns `0`.

```
  newarray = λ_:Unit. ref (λn:Nat.0);
▸ newarray : Unit → NatArray
```

To look up an element of an array, we simply apply the function to the desired index.

```
  lookup = λa:NatArray. λn:Nat. (!a) n;
▸ lookup : NatArray → Nat → Nat
```

The interesting part of the encoding is the `update` function. It takes an array, an index, and a new value to be stored at that index, and does its job by creating (and storing in the reference) a new function that, when it is asked for the value at this very index, returns the new value that was given to `update`, and on all other indices passes the lookup to the function that was previously stored in the reference.

```
  update = λa:NatArray. λm:Nat. λv:Nat.
             let oldf = !a in
             a := (λn:Nat. if equal m n then v else oldf n);
▸ update : NatArray → Nat → Nat → Unit
```

13.1.2 EXERCISE [⋆⋆]: If we defined `update` more compactly like this

```
  update = λa:NatArray. λm:Nat. λv:Nat.
             a := (λn:Nat. if equal m n then v else (!a) n);
```

would it behave the same? □

References to values containing other references can also be very useful, allowing us to define data structures such as mutable lists and trees. (Such structures generally also involve *recursive types,* which we introduce in Chapter 20.)

Garbage Collection

A last issue that we should mention before we move on formalizing references is storage *de*allocation. We have not provided any primitives for freeing reference cells when they are no longer needed. Instead, like many modern languages (including ML and Java) we rely on the run-time system to perform *garbage collection,* collecting and reusing cells that can no longer be reached by the program. This is *not* just a question of taste in language design: it is extremely difficult to achieve type safety in the presence of an explicit deallocation operation. The reason for this is the familiar *dangling reference* problem: we allocate a cell holding a number, save a reference to it in some data structure, use it for a while, then deallocate it and allocate a new cell holding a boolean, possibly reusing the same storage. Now we can have two names for the same storage cell—one with type `Ref Nat` and the other with type `Ref Bool`.

13.1.3 EXERCISE [⋆⋆]: Show how this can lead to a violation of type safety. □

13.2 Typing

The typing rules for `ref`, `:=`, and `!` follow straightforwardly from the behaviors we have given them.

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{ref}\ t_1 : \texttt{Ref}\ T_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Ref}\ T_1}{\Gamma \vdash \texttt{!}t_1 : T_1} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Ref}\ T_1 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\texttt{:=}t_2 : \texttt{Unit}} \qquad \text{(T-Assign)}$$

13.3 Evaluation

A more subtle aspect of the treatment of references appears when we consider how to formalize their operational behavior. One way to see why is to ask, "What should be the *values* of type `Ref T`?" The crucial observation that we need to take into account is that evaluating a `ref` operator should *do* something—namely, allocate some storage—and the result of the operation should be a reference to this storage.

What, then, is a reference?

The run-time store in most programming language implementations is essentially just a big array of bytes. The run-time system keeps track of which parts of this array are currently in use; when we need to allocate a new reference cell, we allocate a large enough segment from the free region of the store (4 bytes for integer cells, 8 bytes for cells storing `Floats`, etc.), mark it as being used, and return the index (typically, a 32- or 64-bit integer) of the start of the newly allocated region. These indices are references.

For present purposes, there is no need to be quite so concrete. We can think of the store as an array of *values,* rather than an array of bytes, abstracting away from the different sizes of the run-time representations of different values. Furthermore, we can abstract away from the fact that references (i.e., indexes into this array) are numbers. We take references to be elements of some uninterpreted set $\mathcal{L}$ of *store locations,* and take the store to be simply a partial function from locations l to values. We use the metavariable μ to range over stores. A reference, then, is a location—an abstract index into the store. We'll use the word *location* instead of *reference* or *pointer* from now on to emphasize this abstract quality.[4]

4. Treating locations abstractly in this way will prevent us from modeling the *pointer arith-*

Next, we need to extend our operational semantics to take stores into account. Since the result of evaluating an expression will in general depend on the contents of the store in which it is evaluated, the evaluation rules should take not just a term but also a store as argument. Furthermore, since the evaluation of a term may cause side effects on the store that may affect the evaluation of other terms in the future, the evaluation rules need to return a new store. Thus, the shape of the single-step evaluation relation changes from $\mathtt{t} \longrightarrow \mathtt{t}'$ to $\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'$, where μ and μ' are the starting and ending states of the store. In effect, we have enriched our notion of *abstract machines,* so that a machine state is not just a program counter (represented as a term), but a program counter plus the current contents of the store.

To carry through this change, we first need to augment all of our existing evaluation rules with stores:

$$(\lambda \mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \mid \mu \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \mid \mu \qquad \text{(E-AppAbs)}$$

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{t}_1\ \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2 \mid \mu'} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_2' \mid \mu'}{\mathtt{v}_1\ \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{v}_1\ \mathtt{t}_2' \mid \mu'} \qquad \text{(E-App2)}$$

Note that the first rule here returns the store μ unchanged: function application, in itself, has no side effects. The other two rules simply propagate side effects from premise to conclusion.

Next, we make a small addition to the *syntax* of our terms. The result of evaluating a `ref` expression will be a fresh location, so we need to include locations in the set of things that can be results of evaluation—i.e., in the set of values:

v ::=		*values:*
	λx:T.t	*abstraction value*
	unit	*unit value*
	l	*store location*

Since all values are also terms, this means that the set of terms should include locations.

metic found in low-level languages such as C. This limitation is intentional. While pointer arithmetic is occasionally very useful (especially for implementing low-level components of run-time systems, such as garbage collectors), it cannot be tracked by most type systems: knowing that location n in the store contains a `Float` doesn't tell us anything useful about the type of location $n + 4$. In C, pointer arithmetic is a notorious source of type safety violations.

t ::=		*terms:*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	unit	*constant* unit
	ref t	*reference creation*
	!t	*dereference*
	t:=t	*assignment*
	l	*store location*

Of course, making this extension to the syntax of terms does not mean that we intend *programmers* to write terms involving explicit, concrete locations: such terms will arise only as intermediate results of evaluation. In effect, the term language in this chapter should be thought of as formalizing an *intermediate language,* some of whose features are not made available to programmers directly.

In terms of this expanded syntax, we can state evaluation rules for the new constructs that manipulate locations and the store. First, to evaluate a dereferencing expression $!t_1$, we must first reduce t_1 until it becomes a value:

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{!t_1 \mid \mu \longrightarrow !t_1' \mid \mu'} \qquad \text{(E-Deref)}$$

Once t_1 has finished reducing, we should have an expression of the form $!l$, where l is some location. A term that attempts to dereference any other sort of value, such as a function or unit, is erroneous. The evaluation rules simply get stuck in this case. The type safety properties in §13.5 assure us that well-typed terms will never misbehave in this way.

$$\frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu} \qquad \text{(E-DerefLoc)}$$

Next, to evaluate an assignment expression $t_1 := t_2$, we must first evaluate t_1 until it becomes a value (i.e., a location),

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1 := t_2 \mid \mu \longrightarrow t_1' := t_2 \mid \mu'} \qquad \text{(E-Assign1)}$$

and then evaluate t_2 until it becomes a value (of any sort):

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1 := t_2 \mid \mu \longrightarrow v_1 := t_2' \mid \mu'} \qquad \text{(E-Assign2)}$$

Once we have finished with t_1 and t_2, we have an expression of the form $l := v_2$, which we execute by updating the store to make location l contain v_2:

$$l := v_2 \mid \mu \longrightarrow \text{unit} \mid [l \mapsto v_2]\mu \qquad \text{(E-Assign)}$$

(The notation $[l \mapsto \mathtt{v}_2]\mu$ here means "the store that maps l to $\mathtt{v}_2$ and maps all other locations to the same thing as μ." Note that the term resulting from this evaluation step is just `unit`; the interesting result is the updated store.)

Finally, to evaluate an expression of the form $\mathtt{ref}\ \mathtt{t}_1$, we first evaluate $\mathtt{t}_1$ until it becomes a value:

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{ref}\ \mathtt{t}_1 \mid \mu \longrightarrow \mathtt{ref}\ \mathtt{t}_1' \mid \mu'} \qquad \text{(E-Ref)}$$

Then, to evaluate the `ref` itself, we choose a fresh location l (i.e., a location that is not already part of the domain of μ) and yield a new store that extends μ with the new binding $l \mapsto \mathtt{v}_1$.

$$\frac{l \notin \mathit{dom}(\mu)}{\mathtt{ref}\ \mathtt{v}_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto \mathtt{v}_1)} \qquad \text{(E-RefV)}$$

The term resulting from this step is the name l of the newly allocated location.

Note that these evaluation rules do not perform any kind of garbage collection: we simply allow the store to keep growing without bound as evaluation proceeds. This does not affect the correctness of the results of evaluation (after all, the definition of "garbage" is precisely parts of the store that are no longer reachable and so cannot play any further role in evaluation), but it means that a naive implementation of our evaluator will sometimes run out of memory where a more sophisticated evaluator would be able to continue by reusing locations whose contents have become garbage.

13.3.1 Exercise [⋆⋆⋆]: How might our evaluation rules be refined to model garbage collection? What theorem would we then need to prove, to argue that this refinement is correct? □

13.4 Store Typings

Having extended our syntax and evaluation rules to accommodate references, our last job is to write down typing rules for the new constructs—and, of course, to check that they are sound. Naturally, the key question is, "What is the type of a location?"

When we evaluate a term containing concrete locations, the type of the result depends on the contents of the store that we start with. For example, if we evaluate the term $!l_2$ in the store $(l_1 \mapsto \mathtt{unit},\ l_2 \mapsto \mathtt{unit})$, the result is `unit`; if we evaluate the same term in the store $(l_1 \mapsto \mathtt{unit},\ l_2 \mapsto \lambda\mathtt{x{:}Unit.x})$, the result is $\lambda\mathtt{x{:}Unit.x}$. With respect to the former store, the location $\mathtt{l}_2$ has type `Unit`, and with respect to the latter it has type `Unit→Unit`. This observation leads us immediately to a first attempt at a typing rule for locations:

$$\frac{\Gamma \vdash \mu(l) : \mathtt{T}_1}{\Gamma \vdash l : \mathtt{Ref}\ \mathtt{T}_1}$$

That is, to find the type of a location l, we look up the current contents of l in the store and calculate the type $\mathtt{T}_1$ of the contents. The type of the location is then $\mathtt{Ref}\ \mathtt{T}_1$.

Having begun in this way, we need to go a little further to reach a consistent state. In effect, by making the type of a term depend on the store, we have changed the typing relation from a three-place relation (between contexts, terms, and types) to a four-place relation (between contexts, *stores,* terms, and types). Since the store is, intuitively, part of the context in which we calculate the type of a term, let's write this four-place relation with the store to the left of the turnstile: $\Gamma \mid \mu \vdash \mathtt{t} : \mathtt{T}$. Our rule for typing references now has the form

$$\frac{\Gamma \mid \mu \vdash \mu(l) : \mathtt{T}_1}{\Gamma \mid \mu \vdash l : \mathtt{Ref}\ \mathtt{T}_1}$$

and all the rest of the typing rules in the system are extended similarly with stores. The other rules do not need to do anything interesting with their stores—just pass them from premise to conclusion.

However, there are two problems with this rule. First, typechecking is rather inefficient, since calculating the type of a location l involves calculating the type of the current contents $\mathtt{v}$ of l. If l appears many times in a term $\mathtt{t}$, we will re-calculate the type of $\mathtt{v}$ many times in the course of constructing a typing derivation for $\mathtt{t}$. Worse, if $\mathtt{v}$ itself contains locations, then we will have to recalculate *their* types each time they appear. For example, if the store contains

$$\begin{array}{l}(l_1 \mapsto \lambda\mathtt{x{:}Nat.\ 999},\\ \ l_2 \mapsto \lambda\mathtt{x{:}Nat.\ (!}l_1\mathtt{)\ x},\\ \ l_3 \mapsto \lambda\mathtt{x{:}Nat.\ (!}l_2\mathtt{)\ x},\\ \ l_4 \mapsto \lambda\mathtt{x{:}Nat.\ (!}l_3\mathtt{)\ x},\\ \ l_5 \mapsto \lambda\mathtt{x{:}Nat.\ (!}l_4\mathtt{)\ x}),\end{array}$$

then calculating the type of l_5 involves calculating those of l_4, l_3, l_2, and l_1.

Second, the proposed typing rule for locations may not allow us to derive anything at all, if the store contains a *cycle*. For example, there is no finite typing derivation for the location l_2 with respect to the store

$$\begin{array}{l}(l_1 \mapsto \lambda\mathtt{x{:}Nat.\ (!}l_2\mathtt{)\ x},\\ \ l_2 \mapsto \lambda\mathtt{x{:}Nat.\ (!}l_1\mathtt{)\ x}),\end{array}$$

since calculating a type for l_2 requires finding the type of l_1, which in turn involves l_1, etc. Cyclic reference structures do arise in practice (e.g., they can

be used for building doubly linked lists), and we would like our type system to be able to deal with them.

13.4.1 EXERCISE [⋆]: Can you find a term whose evaluation will create this particular cyclic store? □

Both of these problems arise from the fact that our proposed typing rule for locations requires us to recalculate the type of a location every time we mention it in a term. But this, intuitively, should not be necessary. After all, when a location is first created, we know the type of the initial value that we are storing into it. Moreover, although we may later store other values into this location, those other values will always have the same type as the initial one. In other words, we always have in mind a single, definite type for every location in the store, which is fixed when the location is allocated. These intended types can be collected together as a *store typing*—a finite function mapping locations to types. We'll use the metavariable Σ to range over such functions.

Suppose we are *given* a store typing Σ describing the store μ in which some term $\mathtt{t}$ will be evaluated. Then we can use Σ to calculate the type of the result of $\mathtt{t}$ without ever looking directly at μ. For example, if Σ is $(l_1 \mapsto \mathtt{Unit}, l_2 \mapsto \mathtt{Unit}\to\mathtt{Unit})$, then we may immediately infer that $!l_2$ has type $\mathtt{Unit}\to\mathtt{Unit}$. More generally, the typing rule for locations can be reformulated in terms of store typings like this:

$$\frac{\Sigma(l) = \mathtt{T}_1}{\Gamma \mid \Sigma \vdash l : \mathtt{Ref}\ \mathtt{T}_1} \qquad \text{(T-Loc)}$$

Typing is again a four-place relation, but it is parameterized on a store *typing* rather than a concrete store. The rest of the typing rules are analogously augmented with store typings.

Of course, these typing rules will accurately predict the results of evaluation only if the concrete store used during evaluation actually conforms to the store typing that we assume for purposes of typechecking. This proviso exactly parallels the situation with free variables in all the calculi we have seen up to this point: the substitution lemma (9.3.8) promises us that, if $\Gamma \vdash \mathtt{t} : \mathtt{T}$, then we can replace the free variables in $\mathtt{t}$ with values of the types listed in Γ to obtain a closed term of type $\mathtt{T}$, which, by the type preservation theorem (9.3.9) will evaluate to a final result of type $\mathtt{T}$ if it yields any result at all. We will see in §13.5 how to formalize an analogous intuition for stores and store typings.

Finally, note that, for purposes of typechecking the terms that programmers actually write, we do not need to do anything tricky to guess what store typing we should use. As we remarked above, concrete location constants

arise only in terms that are the intermediate results of evaluation; they are not in the language that programmers write. Thus, we can simply typecheck the programmer's terms with respect to the *empty* store typing. As evaluation proceeds and new locations are created, we will always be able to see how to extend the store typing by looking at the type of the initial values being placed in newly allocated cells; this intuition is formalized in the statement of the type preservation theorem below (13.5.3).

Now that we have dealt with locations, the typing rules for the other new syntactic forms are quite straightforward. When we create a reference to a value of type $\mathtt{T}_1$, the reference itself has type $\mathtt{Ref}\ \mathtt{T}_1$.

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \mid \Sigma \vdash \mathtt{ref}\ \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1} \qquad \text{(T-Ref)}$$

Notice that we do not need to extend the store typing here, since the *name* of the new location will not be determined until run time, while Σ records only the association between already-allocated storage cells and their types.

Conversely, if $\mathtt{t}_1$ evaluates to a location of type $\mathtt{Ref}\ \mathtt{T}_{11}$, then dereferencing $\mathtt{t}_1$ is guaranteed to yield a value of type $\mathtt{T}_{11}$.

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash\ !\mathtt{t}_1 : \mathtt{T}_{11}} \qquad \text{(T-Deref)}$$

Finally, if $\mathtt{t}_1$ denotes a cell of type $\mathtt{Ref}\ \mathtt{T}_{11}$, then we can store $\mathtt{t}_2$ into this cell as long as the type of $\mathtt{t}_2$ is also $\mathtt{T}_{11}$:

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_{11} \qquad \Gamma \mid \Sigma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{t}_1 := \mathtt{t}_2 : \mathtt{Unit}} \qquad \text{(T-Assign)}$$

Figure 13-1 summarizes the typing rules (and the syntax and evaluation rules, for easy reference) for the simply typed lambda-calculus with references.

13.5 Safety

Our final job in this chapter is to check that standard type safety properties continue to hold for the calculus with references. The progress theorem ("well-typed terms are not stuck") can be stated and proved almost as before (cf. 13.5.7); we just need to add a few straightforward cases to the proof, dealing with the new constructs. The preservation theorem is a bit more interesting, so let's look at it first.

Since we have extended both the evaluation relation (with initial and final stores) and the typing relation (with a store typing), we need to change the statement of preservation to include these parameters. Clearly, though, we

→ Unit Ref — *Extends $\lambda_{\to}$ with Unit (9-1 and 11-2)*

Syntax

t	::=		*terms:*
		x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
		unit	*constant* unit
		ref t	*reference creation*
		!t	*dereference*
		t:=t	*assignment*
		l	*store location*
v	::=		*values:*
		λx:T.t	*abstraction value*
		unit	*constant unit*
		l	*store location*
T	::=		*types:*
		T→T	*type of functions*
		Unit	*unit type*
		Ref T	*type of reference cells*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		Γ, x:T	*term variable binding*
μ	::=		*stores:*
		$\varnothing$	*empty store*
		$\mu, l = v$	*location binding*
Σ	::=		*store typings:*
		$\varnothing$	*empty store typing*
		Σ, l:T	*location typing*

Evaluation $\boxed{t \mid \mu \longrightarrow t' \mid \mu'}$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1\ t_2 \mid \mu \longrightarrow t_1'\ t_2 \mid \mu'} \quad \text{(E-App1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1\ t_2 \mid \mu \longrightarrow v_1\ t_2' \mid \mu'} \quad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \mid \mu \longrightarrow [x \mapsto v_2]t_{12} \mid \mu \quad \text{(E-AppAbs)}$$

$$\frac{l \notin dom(\mu)}{\text{ref } v_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto v_1)} \quad \text{(E-RefV)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{\text{ref } t_1 \mid \mu \longrightarrow \text{ref } t_1' \mid \mu'} \quad \text{(E-Ref)}$$

$$\frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu} \quad \text{(E-DerefLoc)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{!t_1 \mid \mu \longrightarrow !t_1' \mid \mu'} \quad \text{(E-Deref)}$$

$$l{:=}v_2 \mid \mu \longrightarrow \text{unit} \mid [l \mapsto v_2]\mu \quad \text{(E-Assign)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1{:=}t_2 \mid \mu \longrightarrow t_1'{:=}t_2 \mid \mu'} \quad \text{(E-Assign1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1{:=}t_2 \mid \mu \longrightarrow v_1{:=}t_2' \mid \mu'} \quad \text{(E-Assign2)}$$

continued…

Figure 13-1: References

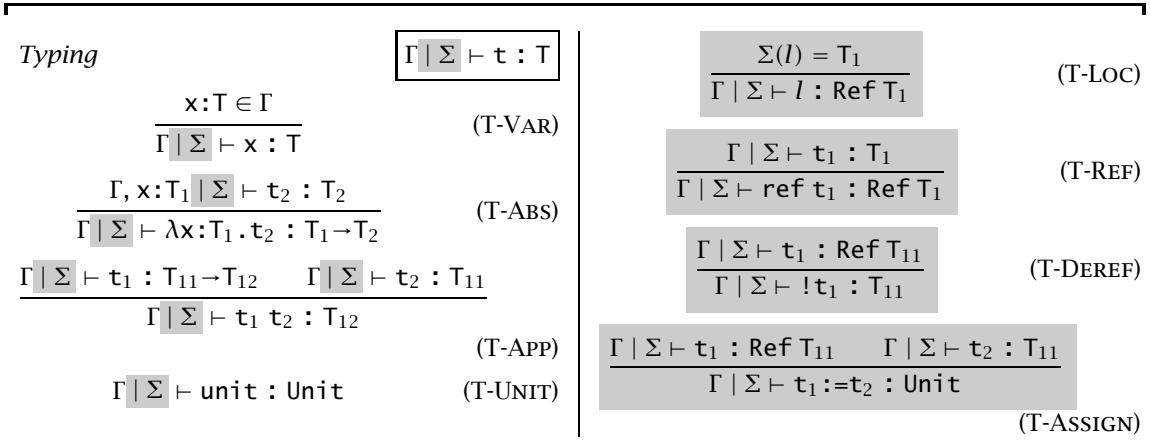


Figure 13-1: References (continued)

cannot just add stores and store typings without saying anything about how they are related.

If $\Gamma \mid \Sigma \vdash t : T$ and $t \mid \mu \longrightarrow t' \mid \mu'$, then $\Gamma \mid \Sigma \vdash t' : T$. (*Wrong!*)

If we typecheck with respect to some set of assumptions about the types of the values in the store and then evaluate with respect to a store that violates these assumptions, the result will be disaster. The following requirement expresses the constraint we need.

13.5.1 DEFINITION: A store μ is said to be *well typed* with respect to a typing context Γ and a store typing Σ, written $\Gamma \mid \Sigma \vdash \mu$, if $dom(\mu) = dom(\Sigma)$ and $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in dom(\mu)$. □

Intuitively, a store μ is consistent with a store typing Σ if every value in the store has the type predicted by the store typing.

13.5.2 EXERCISE [⋆⋆]: Can you find a context Γ, a store μ, and two different store typings Σ_1 and Σ_2 such that both $\Gamma \mid \Sigma_1 \vdash \mu$ and $\Gamma \mid \Sigma_2 \vdash \mu$? □

We can now state something closer to the desired preservation property:

If

$\Gamma \mid \Sigma \vdash t : T$
$t \mid \mu \longrightarrow t' \mid \mu'$
$\Gamma \mid \Sigma \vdash \mu$

then $\Gamma \mid \Sigma \vdash \mathtt{t}' : \mathtt{T}$. (*Less wrong.*)

This statement is fine for all of the evaluation rules except the allocation rule E-RefV. The problem is that this rule yields a store with a larger domain than the initial store, which falsifies the conclusion of the above statement: if μ' includes a binding for a fresh location l, then l cannot be in the domain of Σ, and it will not be the case that $\mathtt{t}'$ (which definitely mentions l) is typable under Σ.

Evidently, since the store can increase in size during evaluation, we need to allow the store typing to grow as well. This leads us to the final (correct) statement of the type preservation property:

13.5.3 Theorem [Preservation]: If

$$\Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$$
$$\Gamma \mid \Sigma \vdash \mu$$
$$\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'$$

then, for some $\Sigma' \supseteq \Sigma$,

$$\Gamma \mid \Sigma' \vdash \mathtt{t}' : \mathtt{T}$$
$$\Gamma \mid \Sigma' \vdash \mu'. \qquad \square$$

Note that the preservation theorem merely asserts that there is *some* store typing $\Sigma' \supseteq \Sigma$ (i.e., agreeing with Σ on the values of all the old locations) such that the new term $\mathtt{t}'$ is well typed with respect to Σ'; it does not tell us exactly what Σ' is. It is intuitively clear, of course, that Σ' is either Σ or else it is exactly $(\mu, l \mapsto \mathtt{T}_1)$, where l is a newly allocated location (the new element of the domain of μ') and $\mathtt{T}_1$ is the type of the initial value bound to l in the extended store $(\mu, l \mapsto \mathtt{v}_1)$, but stating this explicitly would complicate the statement of the theorem without actually making it any more useful: the weaker version above is already in the right form (because its conclusion implies its hypothesis) to "turn the crank" repeatedly and conclude that every *sequence* of evaluation steps preserves well-typedness. Combining this with the progress property, we obtain the usual guarantee that "well-typed programs never go wrong."

To prove preservation, we need a few technical lemmas. The first is an easy extension of the standard substitution lemma (9.3.8).

13.5.4 Lemma [Substitution]: If $\Gamma, \mathtt{x}:\mathtt{S} \mid \Sigma \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \mid \Sigma \vdash \mathtt{s} : \mathtt{S}$, then $\Gamma \mid \Sigma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} : \mathtt{T}$. $\square$

Proof: Just like Lemma 9.3.8. $\square$

The next states that replacing the contents of a cell in the store with a new value of appropriate type does not change the overall type of the store.

13.5.5 LEMMA: If

$$\Gamma \mid \Sigma \vdash \mu$$
$$\Sigma(l) = \mathsf{T}$$
$$\Gamma \mid \Sigma \vdash \mathtt{v} : \mathsf{T}$$

then $\Gamma \mid \Sigma \vdash [l \mapsto \mathtt{v}]\mu$. □

Proof: Immediate from the definition of $\Gamma \mid \Sigma \vdash \mu$. □

Finally, we need a kind of weakening lemma for stores, stating that, if a store is extended with a new location, the extended store still allows us to assign types to all the same terms as the original.

13.5.6 LEMMA: If $\Gamma \mid \Sigma \vdash \mathtt{t} : \mathsf{T}$ and $\Sigma' \supseteq \Sigma$, then $\Gamma \mid \Sigma' \vdash \mathtt{t} : \mathsf{T}$. □

Proof: Easy induction. □

Now we can prove the main preservation theorem.

Proof of 13.5.3: Straightforward induction on evaluation derivations, using the lemmas above and the inversion property of the typing rules (a straightforward extension of 9.3.1). □

The statement of the progress theorem (9.3.5) must also be extended to take stores and store typings into account:

13.5.7 THEOREM [PROGRESS]: Suppose $\mathtt{t}$ is a closed, well-typed term (that is, $\varnothing \mid \Sigma \vdash \mathtt{t} : \mathsf{T}$ for some T and Σ). Then either $\mathtt{t}$ is a value or else, for any store μ such that $\varnothing \mid \Sigma \vdash \mu$, there is some term $\mathtt{t}'$ and store μ' with $\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'$. □

Proof: Straightforward induction on typing derivations, following the pattern of 9.3.5. (The canonical forms lemma, 9.3.4, needs two additional cases stating that all values of type `Ref T` are locations and similarly for `Unit`.) □

13.5.8 EXERCISE [RECOMMENDED, ⋆⋆⋆]: Is the evaluation relation in this chapter normalizing on well-typed terms? If so, prove it. If not, write a well-typed factorial function in the present calculus (extended with numbers and booleans). □

13.6 Notes

The presentation in this chapter is adapted from a treatment by Harper (1994, 1996). An account in a similar style is given by Wright and Felleisen (1994).

The combination of references (or other computational effects) with ML-style polymorphic type inference raises some quite subtle problems (cf. §22.7) and has received a good deal of attention in the research literature. See Tofte (1990), Hoang et al. (1993), Jouvelot and Gifford (1991), Talpin and Jouvelot (1992), Leroy and Weis (1991), Wright (1992), Harper (1994, 1996), and the references cited there.

Static prediction of possible aliasing is a long-standing problem both in compiler implementation (where it is called *alias analysis*) and in programming language theory. An influential early attempt by Reynolds (1978, 1989) coined the term *syntactic control of interference*. These ideas have recently seen a burst of new activity—see O'Hearn et al. (1995) and Smith et al. (2000). More general reasoning techniques for aliasing are discussed in Reynolds (1981) and Ishtiaq and O'Hearn (2001) and other references cited there.

A comprehensive discussion of garbage collection can be found in Jones and Lins (1996). A more semantic treatment is given by Morrisett et al. (1995).

Find out the cause of this effect,
Or rather say, the cause of this defect,
For this effect defective comes by cause. —*Hamlet II, ii, 101*

The finger pointing at the moon is not the moon. —*Buddhist saying*