Part XII

State

Chapter 34

Fluid Binding

Recall from Chapter 13 that under the dynamic scope discipline evaluation is defined for expressions with free variables whose bindings are determined by capture-incurring substitution. Evaluation aborts if the binding of a variable is required in a context in which no binding for it exists. Otherwise, it uses whatever bindings for its free variables happen to be active at the point at which it is evaluated. In essence the bindings of variables are determined as late as possible during execution—just in time for evaluation to proceed. However, we found that as a language design dynamic scoping is deficient in (at least) two respects:

- Bound variables may not always be renamed in an expression without changing its meaning.
- Since the scopes of variables are resolved dynamically, it is difficult to ensure type safety.

These difficulties can be overcome by distinguishing two different concepts, namely *static binding* of variables, which is defined by substitution, and *dynamic*, or *fluid*, *binding* of *symbols*, which is defined by storing and retrieving bindings from a table during execution.

34.1 Statics

The language fragment $\mathcal{L}\{\texttt{fluid}\}$ is defined by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Expr	e	$::=$	$\texttt{put}[a](e_1;e_2)$	$\texttt{put}\,a\,\texttt{is}\,e_1\,\texttt{in}\,e_2$
		$\mid$	$\texttt{get}[a]$	$\texttt{get}\,a$

The variable a ranges over some fixed set of *symbols*. The expression `put` a `is` e_1 `in` e_2, called a *fluid let*, binds the symbol a to the value e_1 for the duration of the evaluation of e_2, at which point the binding of a reverts to what it was prior to the execution. It is important to note that `bind` does not bind a in either argument; rather, the symbol, a, is simply a parameter of the operation. The expression `get` a evaluates to the value of the current binding of a, if it has one, and is stuck otherwise.

The static semantics of $\mathcal{L}\{\texttt{fluid}\}$ is defined by judgements of the form

$$\mathcal{A}\,\mathcal{X} \mid \Sigma\,\Gamma \vdash e : \tau,$$

where $\mathcal{A}$ is a finite set of symbols, $\mathcal{X}$ is a finite set of variables disjoint from $\mathcal{A}$, Σ is a finite set of hypotheses of the form $a : \tau$, where $a \in \mathcal{A}$, and Γ is a finite set of hypotheses of the form $x : \tau$, where $x \in \mathcal{X}$. We insist that no symbol or variable be declared more than once in Σ or Γ, respectively. We usually drop explicit mention of the sets of $\mathcal{A}$ and $\mathcal{X}$, since they can be recoved from Σ and Γ.

The following rules comprise the static semantics of $\mathcal{L}\{\texttt{fluid}\}$:

$$\frac{\Sigma \vdash a : \tau}{\Sigma\,\Gamma \vdash \texttt{get}\,[a] : \tau} \tag{34.1a}$$

$$\frac{\Sigma \vdash a : \tau_1 \quad \Sigma\,\Gamma \vdash e_1 : \tau_1 \quad \Sigma\,\Gamma \vdash e_2 : \tau_2}{\Sigma\,\Gamma \vdash \texttt{put}\,[a]\,(e_1; e_2) : \tau_2} \tag{34.1b}$$

Rule (34.1b) treats the symbol a as given by Σ, rather than introducing a "new" symbol, as would be the case for a statically scoped `let` construct: neither Σ nor Γ are extended in Rule (34.1b).

34.2 Dynamics

The dynamics of $\mathcal{L}\{\texttt{fluid}\}$ is defined using a technique called *shallow binding* in which we maintain an association of values to symbols that changes in a stack-like manner during execution. We define a family of transition judgements of the form $e \mapsto_\theta e'$, where θ is a finite mapping assigning closed values to symbols.[1] If $a \in \mathit{dom}(\theta)$, then we may write $\theta = \theta' \otimes \langle a : e \rangle$. If $a \notin \mathit{dom}(\theta)$, then we shall, as a notational convenience, regard θ has having the form $\theta' \otimes \langle a : \bullet \rangle$ in which a is considered bound to the "black hole",

[1] By a "closed value" we mean, as usual, one that has no free variables. However, it may contain symbols whose bindings are not yet determined.

$\bullet$. We will write $\langle a : \_ \rangle$ to stand ambiguously for either $\langle a : \bullet \rangle$ or $\langle a : e \rangle$ for some expression e.

The dynamic semantics of $\mathcal{L}\{\texttt{fluid}\}$ is given by the following rules:

$$\frac{e \ \mathsf{val}}{\texttt{get}\,[a] \longmapsto_{\theta \otimes \langle a : e \rangle} e} \tag{34.2a}$$

$$\frac{e_1 \longmapsto_\theta e_1'}{\texttt{put}\,[a]\,(e_1; e_2) \longmapsto_\theta \texttt{put}\,[a]\,(e_1'; e_2)} \tag{34.2b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \longmapsto_{\theta \otimes \langle a : e_1 \rangle} e_2'}{\texttt{put}\,[a]\,(e_1; e_2) \longmapsto_{\theta \otimes \langle a : \_ \rangle} \texttt{put}\,[a]\,(e_1; e_2')} \tag{34.2c}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{val}}{\texttt{put}\,[a]\,(e_1; e_2) \longmapsto_\theta e_2} \tag{34.2d}$$

Rule (34.2a) specifies that $\texttt{get}\,[a]$ evaluates to the current binding of a, if any. Rule (34.2b) specifies that the binding for the symbol a is to be evaluated before the binding is created. Rule (34.2c) evaluates e_2 in an environment in which the symbol a is bound to the value e_1, regardless of whether or not a is already bound in the environment. Rule (34.2d) eliminates the fluid binding for a once evaluation of the extent of the binding has completed.

The dynamic semantics specifies that there is no transition of the form $\texttt{get}\,[a] \longmapsto_{\theta \otimes \langle a : \bullet \rangle} e$ for any e. Since the static semantics does not rule out such states, we define the judgement $e \ \mathsf{unbound}_\theta$ by the following rules:[2]

$$\frac{}{\texttt{get}\,[a] \ \mathsf{unbound}_{\theta \otimes \langle a : \bullet \rangle}} \tag{34.3a}$$

$$\frac{e_1 \ \mathsf{unbound}_\theta}{\texttt{put}\,[a]\,(e_1; e_2) \ \mathsf{unbound}_\theta} \tag{34.3b}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{unbound}_{\theta \otimes \langle a : e_1 \rangle}}{\texttt{put}\,[a]\,(e_1; e_2) \ \mathsf{unbound}_\theta} \tag{34.3c}$$

This judgement is used to state the progress theorem for $\mathcal{L}\{\texttt{fluid}\}$ in the next section.

[2] In the context of other language constructs, stuck states would have to be propagated through the evaluated arguments of a compound expression as described in Chapter 11.

34.3 Type Safety

Define the auxiliary judgement $\theta : \Sigma$ by the following rules:

$$\frac{}{\varnothing : \varnothing} \tag{34.4a}$$

$$\frac{\Sigma \vdash e : \tau \quad \theta : \Sigma}{\theta \otimes \langle a : e \rangle : \Sigma, a : \tau} \tag{34.4b}$$

$$\frac{\theta : \Sigma}{\theta \otimes \langle a : \bullet \rangle : \Sigma, a : \tau} \tag{34.4c}$$

These rules specify that if a symbol is bound to a value, then that value must be of the type associated to the symbol by Σ. No demand is made in the case that the symbol is unbound (equivalently, bound to a "black hole").

Theorem 34.1 (Preservation)**.** *If $e \mapsto_\theta e'$, where $\theta : \Sigma$ and $\Sigma \vdash e : \tau$, then $\Sigma \vdash e' : \tau$.*

Proof. By rule induction on Rules (34.2). Rule (34.2a) is handled by the definition of $\theta : \Sigma$. Rule (34.2b) follows immediately by induction. Rule (34.2d) is handled by inversion of Rules (34.1). Finally, Rule (34.2c) is handled by inversion of Rules (34.1) and induction. □

Theorem 34.2 (Progress)**.** *If $\Sigma \vdash e : \tau$ and $\theta : \Sigma$, then either e val, or e $\mathsf{unbound}_\theta$, or there exists e' such that $e \mapsto_\theta e'$.*

Proof. By induction on Rules (34.1). For Rule (34.1a), we have $\Sigma \vdash a : \tau$ from the premise of the rule, and hence, since $\theta : \Sigma$, we have either $\theta(a) = \bullet$ (unbound) or $\theta(a) = e$ for some e such that $\Sigma \vdash e : \tau$. In the former case we have e $\mathsf{unbound}_\theta$, and in the latter we have $\mathtt{get}[a] \mapsto_\theta e$. For Rule (34.1b), we have by induction that either e_1 val or e_1 $\mathsf{unbound}_\theta$, or $e_1 \mapsto_\theta e_1'$. In the latter two cases we may apply Rule (34.2b) or Rule (34.3b), respectively. If e_1 val, we apply induction to obtain that either e_2 val, in which case Rule (34.2d) applies; e_2 $\mathsf{unbound}_\theta$, in which case Rule (34.3b) applies; or $e_2 \mapsto_\theta e_2'$, in which case Rule (34.2c) applies. □

34.4 Symbol Generation

The language $\mathcal{L}\{\texttt{fluid new}\}$ enriches $\mathcal{L}\{\texttt{fluid}\}$ with the ability to generate new symbols during execution. The syntax of this extension is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	$::=$	$\texttt{sym}(\sigma;\tau)$	$\langle\sigma\rangle\tau$
Expr	e	$::=$	$\texttt{new}[\sigma](a.e)$	$\nu(a{:}\sigma.e)$
		$\mid$	$\texttt{gen}(e)$	$\texttt{gen}(e)$

The type $\texttt{sym}(\sigma;\tau)$ consists of expressions of type τ that require a symbol of type σ for their execution. Such an expression is introduced by the symbol abstraction construct, $\texttt{new}[\sigma](a.e)$. It is eliminated by $\texttt{gen}(e)$, which generates a "new" symbol at execution time and supplies it to e prior to executing it.

The static semantics of $\mathcal{L}\{\texttt{fluid new}\}$ is given by the following rules:

$$\frac{\Sigma, a:\sigma\,\Gamma \vdash e:\tau}{\Sigma\,\Gamma \vdash \texttt{new}[\sigma](a.e) : \texttt{sym}(\sigma;\tau)} \tag{34.5a}$$

$$\frac{\Sigma\,\Gamma \vdash e : \texttt{sym}(\sigma;\tau)}{\Sigma\,\Gamma \vdash \texttt{gen}(e) : \tau} \tag{34.5b}$$

Rule (34.5a) extends Σ with a new symbol whose uniqueness is guaranteed by the convention on bound variables. If a already occurs as the subject of some typing assumption in Σ, then we must rename a in $\texttt{new}[\sigma](a.e)$ prior to applying the rule. Rule (34.5b) generates a fresh symbol to be used in place of the symbol introduced by e in accordance with Rule (34.5a).

The dynamic semantics of $\mathcal{L}\{\texttt{fluid new}\}$ is given by the following rules:

$$\frac{}{\texttt{new}[\sigma](a.e)\ \texttt{val}} \tag{34.6a}$$

$$\frac{e \mapsto_\theta e'}{\texttt{gen}(e) \mapsto_\theta \texttt{gen}(e')} \tag{34.6b}$$

$$\frac{a\ \textit{fresh}}{\texttt{gen}(\texttt{new}[\sigma](a.e)) \mapsto_\theta e} \tag{34.6c}$$

The freshness condition on Rule (34.6c) may always be met by α-varying the symbol name in the expression $\texttt{new}[\sigma](a.e)$ prior to applying the rule to ensure that the symbol a is not otherwise active.

While the intention of the freshness condition seems clear, it is not precisely defined by Rules (34.6). To make it precise, we may define a transition system between states of the form $e \mathbin{@} \nu$, where ν is a finite set of symbols and e is an expression involving at most the symbols in ν. The set ν is to be thought of as the set of *active* symbols, so that a *fresh* symbol is one that lies outside of this set. The transition judgement, $e \mathbin{@} \nu \mapsto_\theta e' \mathbin{@} \nu'$, is defined for states $e \mathbin{@} \nu$ such that $dom(\theta) \subseteq \nu$. This ensures that the mapping θ governs only active symbols. An initial state has the form $e \mathbin{@} \emptyset$, which requires that e type-check with respect to the empty set of assumptions about the types of symbols. A final state has the form $e \mathbin{@} \nu$, where e val.

The rules defining state transition are as follows:

$$\frac{a \in \nu}{\mathtt{get}\,[a] \mathbin{@} \nu \mapsto_{\theta \otimes \langle a : e \rangle} e \mathbin{@} \nu} \tag{34.7a}$$

$$\frac{e_1 \mathbin{@} \nu \mapsto_\theta e_1' \mathbin{@} \nu'}{\mathtt{put}\,[a]\,(e_1; e_2) \mathbin{@} \nu \mapsto_\theta \mathtt{put}\,[a]\,(e_1'; e_2) \mathbin{@} \nu'} \tag{34.7b}$$

$$\frac{e_1 \text{ val} \quad e_2 \mathbin{@} \nu \mapsto_{\theta \otimes \langle a : e_1 \rangle} e_2' \mathbin{@} \nu'}{\mathtt{put}\,[a]\,(e_1; e_2) \mathbin{@} \nu \mapsto_{\theta \otimes \langle a : \_ \rangle} \mathtt{put}\,[a]\,(e_1; e_2') \mathbin{@} \nu'} \tag{34.7c}$$

$$\frac{e_1 \text{ val} \quad e_2 \text{ val}}{\mathtt{put}\,[a]\,(e_1; e_2) \mathbin{@} \nu \mapsto_\theta e_2 \mathbin{@} \nu} \tag{34.7d}$$

$$\frac{e \mathbin{@} \nu \mapsto_\theta e' \mathbin{@} \nu'}{\mathtt{gen}(e) \mathbin{@} \nu \mapsto_\theta \mathtt{gen}(e') \mathbin{@} \nu'} \tag{34.7e}$$

$$\frac{a \notin \nu}{\mathtt{gen}(\mathtt{new}\,[\sigma]\,(a.e)) \mathbin{@} \nu \mapsto_\theta e \mathbin{@} \nu \cup \{ a \}} \tag{34.7f}$$

Rule (34.7f) makes explicit that the symbol a is to be chosen (by α-equivalence) outside of the set ν of active symbols. Observe that the set of active symbols grows monotonically with transition: if $e \mathbin{@} \nu \mapsto_\theta e' \mathbin{@} \nu'$, then $\nu' \supseteq \nu$.

To prove safety we define a state $e \mathbin{@} \nu$ to be well-formed iff there exists a symbol typing Σ such that (a) $a \in \nu$ implies $\Sigma = \Sigma', a : \tau$ for some type τ, and (b) $\Sigma \vdash e : \sigma$ for some type σ. It is then straightforward to formulate and prove type safety, following along the lines of Section 34.3 on page 312, but treating ν as the set of active symbols. The main difference compared to the static case is that the proof of preservation for Rule (34.7f) relies on the invariance of typing under renaming of parameters, as described in Chapter 3.

34.5 Subtleties of Fluid Binding

Fluid binding in the context of a first-order language is easy to understand. If the expression $\texttt{put}\,a\,\texttt{is}\,e_1\,\texttt{in}\,e_2$ has a type such as `nat`, then its execution consists of the evaluation of e_2 to a number in the presence of a binding of a to the value of expression e_1. When execution is completed, the binding of a is dropped (reverted to its state in the surrounding context), and the value is returned. Since this value is a number, it cannot contain any reference to a, and so no issue of its binding arises.

But what if the type of $\texttt{put}\,a\,\texttt{is}\,e_1\,\texttt{in}\,e_2$ is a function type, so that the returned value is a λ-abstraction? In that case the body of the λ may contain references to the symbol a whose binding is dropped upon return. This raises an important question about the interaction between fluid binding and higher-order functions. For example, consider the expression

$$\texttt{put}\,a\,\texttt{is}\,\overline{17}\,\texttt{in}\,\lambda(x\texttt{:nat}.\,x + \texttt{get}\,a), \tag{34.8}$$

which has type `nat`, given that a is a symbol of the same type. Let us assume, for the sake of discussion, that a is unbound at the point at which this expression is evaluated. Doing so binds a to the number $\overline{17}$, and returns the function $\lambda(x\texttt{:nat}.\,x + \texttt{get}\,a)$. This function contains the symbol a, but is returned to a context in which the symbol a is not bound. This means that, for example, application of the expression (34.8) to an argument will incur an error because the symbol a is not bound.

Contrast this with the similar expression

$$\texttt{let}\,y\,\texttt{be}\,\overline{17}\,\texttt{in}\,\lambda(x\texttt{:nat}.\,x + y), \tag{34.9}$$

in which we have replaced the fluid-bound symbol, a, by a statically bound variable, y. This expression evaluates to $\lambda(x\texttt{:nat}.\,x + \overline{17})$, which adds 17 to its argument when applied. There is never any possibility of an unbound identifier arising at execution time, precisely because the identification of scope and extent ensures that the association between a variable and its binding is never violated.

It is not possible to say that either of these two behaviors is "right" or "wrong," but experience has shown that providing only one or the other of these behaviors is a mistake. Static binding is an important mechanism for encapsulation of behavior in a program; without static binding, one cannot ensure that the meaning of a variable is unchanged by the context in which it is used. The main use of fluid binding is to avoid having to pass "extra" parameters to a function in order to specialize its behavior. Instead we rely

on fluid binding to establish the binding of a symbol for the duration of execution of the function, avoiding the need to re-specify it at each call site.

For example, let e stand for the value of expression (34.8), a λ-abstraction whose body is dependent on the binding of the symbol a. This imposes the requirement that the programmer provide a binding for a whenever e is applied to an argument. For example, the expression

$$\texttt{put}\, a\, \texttt{is}\, \overline{7}\, \texttt{in}\, (e(\overline{9}))$$

evaluates to $\overline{15}$, and the expression

$$\texttt{put}\, a\, \texttt{is}\, \overline{8}\, \texttt{in}\, (e(\overline{9}))$$

evaluates to $\overline{17}$. Writing just $e(\overline{9})$, without a surrounding binding for a, results in a run-time error attempting to retrieve the binding of the unbound symbol a.

The alternative to fluid binding is to add an additional parameter to e for the binding of the symbol a, so that one would write

$$e'(\overline{7})(\overline{9})$$

and

$$e'(\overline{8})(\overline{9}),$$

respectively, where e' is the λ-abstraction

$$\lambda(a{:}\texttt{nat}.\,\lambda(x{:}\texttt{nat}.\,x + a)).$$

Using additional arguments can be slightly inconvenient, though, when several call sites have the same binding for a. Using fluid binding we may write

$$\texttt{put}\, a\, \texttt{is}\, \overline{7}\, \texttt{in}\, \langle e(\overline{8}), e(\overline{9})\rangle,$$

whereas using an additional argument we must write

$$\langle e'(\overline{7})(\overline{8}), e'(\overline{7})(\overline{9})\rangle.$$

However, this sort of redundancy can be mitigated by simply factoring out the common part, writing

$$\texttt{let}\, f\, \texttt{be}\, e'(\overline{7})\, \texttt{in}\, \langle f(\overline{8}), f(\overline{9})\rangle.$$

One might argue, then, that it is all a matter of taste. However, a significant drawback of using fluid binding is that the requirement to provide

a binding for a is not apparent in the type of e, whereas the type of e' reflects the demand for an additional argument. One may argue that the type system *should* record the dependency of a computation on a specified set of fluid-bound symbols. For example, the expression e might be given a type of the form $\texttt{nat} \to_a \texttt{nat}$, reflecting the demand that a binding for a be provided at the call site. A type system of this sort is developed in Chapter 40.

34.6 Exercises

1. Complete the formalization of $\mathcal{L}\{\texttt{fluid}\,\texttt{new}\}$ and prove type safety for it.

2. Formalize *deep binding* using the stack machine of Chapter 27.

Chapter 35

Mutable Storage

Data structures constructed from sums, products, and recursive types are all *immutable* in that their structure does not change over time as a result of computation. For example, evaluation of an expression such as $\langle 2+3, 4*5 \rangle$ results in the ordered pair $\langle 5, 20 \rangle$, which cannot subsequently be altered by further computation. Creation of a value (of any type) is "forever" in that no subsequent computation can change it. Such data structures are said to be *persistent* in that their value persists throughout the rest of the computation. In particular if we project the components of a pair, and construct another pair from it, the original and the newly constructed pair continue to exist side-by-side. This behavior is particularly significant when working with recursive types, such as lists and trees, because the operations performed on them are *non-destructive*. Inserting an element into a persistent binary search tree does not modify the original tree; rather it constructs another tree with the new element inserted, leaving the original intact and available for further computation.

This behavior is in sharp contrast to conventional textbook treatments of data structures such as lists and trees, which are almost invariably defined by *destructive* operations that modify, or *mutate*, the data structure "in place". Inserting an element into a binary tree changes the tree itself to include the new element; the original tree is lost in the process, and all references to it reflect the change. Such data structures are said to be *ephemeral*, in that changes to them destroy the original. In some cases ephemeral data structures are essential to the task at hand; in other cases a persistent representation would do just as well, or even better. For example, a data structure modeling a shared database accessed by many users simultaneously is naturally ephemeral in that the changes made by one user are to be im-

mediately propagated to the computations made by another. On the other hand, data structures used internally to a body of code, such as a search tree, need no such capability and are often profitably represented in persistent form.

A good programming language should naturally support both persistent and ephemeral data structures. This is neatly achieved by distinguishing between a *value* of a type, τ, and a *mutable cell* holding a value of type τ that may change over time. The type τ `ref` is the type of references to (names for) mutable cells that contain a value of type τ. Using reference types we may represent a variety of persistent and mutable data structures. For example, a value of type $\texttt{nat} \times \texttt{(nat ref)}$ is a pair consisting of a natural number and a mutable cell containing a natural number. The pair itself cannot be changed, but the contents of its second component can be altered during execution.. On the other hand, a value of type $\texttt{nat ref} \times \texttt{nat ref}$ is a pair both of whose components are mutable cells whose contents may change. This is different from a value of type $\texttt{(nat} \times \texttt{nat)ref}$, which is a cell whose contents is a pair of natural numbers that may change from time to time.

35.1 Statics

The language $\mathcal{L}\{\texttt{ref}\}$ of mutable cells is described by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{ref}(\tau)$	$\tau\,\texttt{ref}$
Expr	e	::=	$\texttt{loc}[l]$	l
		|	$\texttt{ref}(e)$	$\texttt{ref}(e)$
		|	$\texttt{get}(e)$	$\texttt{!}\,e$
		|	$\texttt{set}(e_1;e_2)$	$e_1\,\texttt{<-}\,e_2$

Mutable cells are handled *by reference*; a mutable cell is represented by a *location*, which is the *name*, or *abstract address*, of the cell. The meta-variable l ranges over locations, an infinite set of symbols reserved as names for reference cells. Locations in expressions arise during execution, but are not available to the programmer as forms of expression. The expression $\texttt{ref}(e)$ allocates a "new" reference cell with initial contents of type τ being the value of the expression e. The expression $\texttt{get}(e)$ retrieves the contents of the cell given by the value of e, and $\texttt{set}(e_1;e_2)$ sets the contents of the cell given by the value of e_1 to the value of e_2.

In practice we consider $\mathcal{L}\{\texttt{ref}\}$ as an extension to another language, such as $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, with mutable data. However, for the purposes of this chapter we study $\mathcal{L}\{\texttt{ref}\}$ in isolation from other language features. The beauty of type systems is that we may do so; the presence of a type of mutable references does not disrupt the behavior of the other constructs of the programming language in which they are embedded.

The static semantics of $\mathcal{L}\{\texttt{ref}\}$ consists of a set of rules for deriving typing judgements of the form $e : \tau$ that are indexed by two sets of parameters, one for locations and one for variables, and hypothetical in two forms of hypotheses specifying the type of the contents of a location and the type of the binding of a variable. The fully explicit form of the typing judgement for $\mathcal{L}\{\texttt{ref}\}$ is

$$\mathcal{L}\,\mathcal{X} \mid \Lambda\,\Gamma \vdash e : \tau,$$

where $\mathcal{L}$ is a finite set of locations, $\mathcal{X}$ is a finite set of variables, Λ is a finite set of assumptions of the form $l : \tau$, one for each $l \in \mathcal{L}$, and Γ is a finite set of assumptions of the form $x : \tau$, one for each $x \in \mathcal{X}$. As usual, we usually omit explicit mention of the parameters, writing just $\Lambda\,\Gamma \vdash e : \tau$.

The static semantics of $\mathcal{L}\{\texttt{ref}\}$ is specified by the following rules:

$$\frac{}{\Lambda, l : \tau\,\Gamma \vdash \texttt{loc}[l] : \texttt{ref}(\tau)} \tag{35.1a}$$

$$\frac{\Lambda\,\Gamma \vdash e : \tau}{\Lambda\,\Gamma \vdash \texttt{ref}(e) : \texttt{ref}(\tau)} \tag{35.1b}$$

$$\frac{\Lambda\,\Gamma \vdash e : \texttt{ref}(\tau)}{\Lambda\,\Gamma \vdash \texttt{get}(e) : \tau} \tag{35.1c}$$

$$\frac{\Lambda\,\Gamma \vdash e_1 : \texttt{ref}(\tau_2) \quad \Lambda\,\Gamma \vdash e_2 : \tau_2}{\Lambda\,\Gamma \vdash \texttt{set}(e_1; e_2) : \tau_2} \tag{35.1d}$$

The type of the expression $\texttt{loc}[l]$ is a reference type, whereas the type assigned to l in the hypothesis is the type of its contents. The return type of $\texttt{set}(e_1; e_2)$ is τ_2, the type of the assigned value, which is returned by the assignment.

35.2 Dynamics

A *memory* is a finite function mapping each of a finite set of locations to closed value (one with no free variables). We write $\emptyset$ for the empty memory, $\langle l : e \rangle$ for the memory μ with domain $\{ l \}$ such that $\mu(l) = e$, and

$\mu_1 \otimes \mu_2$, where $dom(\mu_1) \cap dom(\mu_2) = \emptyset$, for the smallest memory μ such that $\mu(l) = \mu_1(l)$ if $l \in dom(\mu_1)$ and $\mu(l) = \mu_2(l)$ if $l \in dom(\mu_2)$. Whenever we write $\mu_1 \otimes \mu_2$ it is tacitly assumed that μ_1 and μ_2 are disjoint.

The dynamic semantics of $\mathcal{L}\{\texttt{ref}\}$ consists of a transition systems between states of the form $e \mathbin{@} \mu$, where μ is a memory and e is an expression with no free variables. We will maintain the invariant that, in a state $e \mathbin{@} \mu$, the locations occurring in e, and in the contents of any cell in μ, lie within the domain of μ. An initial state has the form $e \mathbin{@} \emptyset$ in which the memory is empty and there are no locations occurring in e. A final state is one of the form $e \mathbin{@} \mu$, where e is a value.

The rules defining values and the transition judgement of the dynamic semantics of $\mathcal{L}\{\texttt{ref}\}$ are given as follows:

$$\frac{}{\texttt{loc}[l]\ \mathsf{val}} \tag{35.2a}$$

$$\frac{e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'}{\texttt{ref}(e) \mathbin{@} \mu \mapsto \texttt{ref}(e') \mathbin{@} \mu'} \tag{35.2b}$$

$$\frac{e\ \mathsf{val}}{\texttt{ref}(e) \mathbin{@} \mu \mapsto \texttt{loc}[l] \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{35.2c}$$

$$\frac{e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'}{\texttt{get}(e) \mathbin{@} \mu \mapsto \texttt{get}(e') \mathbin{@} \mu'} \tag{35.2d}$$

$$\frac{e\ \mathsf{val}}{\texttt{get}(\texttt{loc}[l]) \mathbin{@} \mu \otimes \langle l : e \rangle \mapsto e \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{35.2e}$$

$$\frac{e_1 \mathbin{@} \mu \mapsto e_1' \mathbin{@} \mu'}{\texttt{set}(e_1; e_2) \mathbin{@} \mu \mapsto \texttt{set}(e_1'; e_2) \mathbin{@} \mu'} \tag{35.2f}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mathbin{@} \mu \mapsto e_2' \mathbin{@} \mu'}{\texttt{set}(e_1; e_2) \mathbin{@} \mu \mapsto \texttt{set}(e_1; e_2') \mathbin{@} \mu'} \tag{35.2g}$$

$$\frac{e\ \mathsf{val}}{\texttt{set}(\texttt{loc}[l]; e) \mathbin{@} \mu \otimes \langle l : e' \rangle \mapsto e \mathbin{@} \mu \otimes \langle l : e \rangle} \tag{35.2h}$$

In Rule (35.2c) it is tacitly assumed that l is chosen so as not to occur in the domain of μ, corresponding to the intuition that l is a "new" location in memory.

35.3 Safety

As usual, type safety is the conjunction of preservation and progress for well-formed machine states. Informally, the state $e \mathbin{@} \mu$ is well-formed if (a) μ is well-formed relative to itself, and (b) e is well-formed relative to μ. The latter condition means that $e : \tau$ for some type τ, assuming that the locations have the types given to them by μ. The former means that the contents of each location, l, in μ has the type given to it relative to the types given to *all* the other locations by μ, including the location l itself.

The judgement $e \mathbin{@} \mu \ \mathsf{ok}$ is defined by the following rule:

$$\frac{\Lambda \vdash e : \tau \quad \Lambda \vdash \mu : \Lambda}{e \mathbin{@} \mu \ \mathsf{ok}} \tag{35.3}$$

The hypotheses Λ are the types of the locations in the domain of μ. Since any location may appear in the expression e, it must be checked relative to the assumptions Λ. This is defined formally by the following rules:

$$\frac{}{\Lambda \vdash \emptyset : \emptyset} \tag{35.4a}$$

$$\frac{\Lambda \vdash e : \tau \quad \Lambda \vdash \mu' : \Lambda'}{\Lambda \vdash \mu' \otimes \langle l : e \rangle : \Lambda', l : \tau} \tag{35.4b}$$

To account for circular dependencies, the contents of each location in memory is type-checked relative to the typing assumptions for all locations in memory.

Theorem 35.1 (Preservation)**.** *If $e \mathbin{@} \mu$ ok and $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$, then $e' \mathbin{@} \mu'$ ok.*

Proof. The proof is by rule induction on Rules (35.2). For the sake of the induction we prove the following stronger result: if $\Lambda \vdash e : \tau$, $\Lambda \vdash \mu : \Lambda$, and $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$, then there exists $\Lambda' \supseteq \Lambda$ such that $\Lambda' \vdash e' : \tau$ and $\Lambda' \vdash \mu' : \Lambda'$.

Consider Rule (35.2c). We have $\mathtt{ref}(e) \mathbin{@} \mu \mapsto \mathtt{loc}[l] \mathbin{@} \mu'$, where e val and $\mu' = \mu \otimes \langle l : e \rangle$. By inversion of typing $\Lambda \vdash e : \sigma$ and $\tau = \mathtt{ref}(\sigma)$. Taking $\Lambda' = \Lambda, l : \sigma$, observe that $\Lambda' \supseteq \Lambda$, and $\Lambda' \vdash \mathtt{loc}[l] : \mathtt{ref}(\tau)$. Finally, we have $\Lambda' \vdash \mu' : \Lambda'$, since $\Lambda' \vdash \mu : \Lambda$ and $\Lambda' \vdash e : \sigma$ by assumption and weakening.

The other cases follow a similar pattern. □

Theorem 35.2 (Progress). *If $e \mathbin{@} \mu$ ok then either $e \mathbin{@} \mu$ final or there exists $e' \mathbin{@} \mu'$ such that $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$.*

Proof. By rule induction on Rules (35.1). Consider, for example, Rule (35.1c). We have $\Lambda \vdash \mathtt{get}(e) : \tau$ because $\Lambda \vdash e : \tau$. By induction either e val or there exists μ' and e' such that $e \mathbin{@} \mu \mapsto e' \mathbin{@} \mu'$. In the latter case we have $\mathtt{get}(e) \mathbin{@} \mu \mapsto \mathtt{get}(e') \mathbin{@} \mu'$ by Rule (35.2d). In the former it follows from an analysis of Rules (35.1) that $e = \mathtt{loc}[l]$ for some location l such that $\Lambda = \Lambda', l : \tau$. Since $\Lambda \vdash \mu : \Lambda$, it follows that $\mu = \mu' \otimes \langle l : e' \rangle$ for some μ' and e' such that $\Lambda \vdash e' : \tau$. But then by Rule (35.2e) we have $\mathtt{get}(e) \mathbin{@} \mu \mapsto e' \mathbin{@} \mu$.

The remaining cases follow a similar pattern. □

35.4 References and Recursion

The definition of the judgement $e \mathbin{@} \mu$ ok is reminiscent of the typing rule for general recursion given in Chapter 15. Much like general recursion, the condition on μ in Rule (35.3) specifies that we are to assume that which we are going to prove, namely that the types of the locations are as given by Λ. This is necessary to account for circular dependencies that arise when the contents of one location contains (directly or indirectly through other mutable cells) the location itself.

It is reasonable to wonder whether it is possible to construct a cyclic memory by evaluating a closed expression in the empty memory, or must be start with a cycle in order to encounter one during execution? The answer is not so obvious. For example, certain forms of circularity are excluded for typing reasons. For example, there cannot arise in a computation a location l containing the pair $\langle \overline{0}, l \rangle$, for otherwise the l would have to have a type of the form $\tau\,\mathtt{ref}$, where τ is of the form $\tau_1 \times \tau$, which is impossible.

Based on this we may begin to suspect that no circularities can arise during execution, but this turns out not to be the case. In fact we can exploit such circularities to implement general recursion using a technique called *back-patching*. This shows that the self-reference in the typing of memories is inherent, since it must be at least as strong as the typing rule for general recursion. Back-patching is illustrated by the following example:

```
let r be ref(λ n:nat.n) in
let f be λ n:nat.ifz(n, 1, n'.n * get(r)(n')) in
let _ be set(r,f) in f
```

This expression returns a function of type $\texttt{nat} \to \texttt{nat}$ that is obtained by (a) allocating a reference cell initialized arbitrarily with a function of this type, (b) defining a λ-abstraction in which each "recursive call" consists of retrieving and applying the function stored in that cell, (c) assigning this function to the cell, and (d) returning that function.

35.5 Subtyping

Reference types interact poorly with subtyping. To see why, let us apply the principle of subsumption to derive a sound subtyping rule for references. Suppose that r has type $\sigma\,\texttt{ref}$. There are two elimination forms that may be applied to r:

1. Retrieve its contents as a value of type σ.
2. Replace its contents with a value of type σ.

If $\sigma <: \tau$, then retrieving the contents of r yields a value of type τ, by subsumption. This suggests that reference types be considered covariant:

$$\frac{\sigma <: \tau}{\sigma\,\texttt{ref} <: \tau\,\texttt{ref}} \quad \textbf{??}$$

On the other hand, if $\tau <: \sigma$, then we may store a value of type τ into r. This suggests that reference types be considered contravariant:

$$\frac{\tau <: \sigma}{\sigma\,\texttt{ref} <: \tau\,\texttt{ref}} \quad \textbf{??}$$

Combining these two observations, we see that reference types are *invariant*:

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\sigma\,\texttt{ref} <: \tau\,\texttt{ref}} \tag{35.5}$$

In practice the only way to satisfy the premises of the rule is for σ and τ to be identical.

Similar restrictions govern mutable array types, whose components are mutable cells that can be assigned and retrieved just as for reference types. A naïve interpretation of array types would suggest that arrays be covariant, since a sequence of values of type σ is also a sequence of values of type τ, provided that $\sigma <: \tau$. But this overlooks the possibility of assigning to the elements of the array, which is inconsistent with covariance. The result is that mutable array types must be regarded as *invariant* to ensure type safety (*pace* well-known languages that stipulate otherwise).

35.6 Exercises

Chapter 36

Dynamic Classification

Sum types may be used to classify data values by labelling them with a class identifier that determines the type of the associated data item. For example, a sum type of the form $\sum \langle i_0 : \tau_0, \ldots, i_{n-1} : \tau_{n-1} \rangle$ consists of n distinct classes of data, with the ith class labelling a value of type τ_i. A value of this type is introduced by the expression `in[`i`](`e_i`)`, where $0 \leq i < n$ and $e_i : \tau_i$, and is eliminated by an n-ary case analysis binding the variable x_i to the value of type τ_i labelled with class i.

Sum types are useful in situations where the type of a data item can only be determined at execution time, for example when processing input from an external data source. For example, a data stream from a sensor might consist of several different types of data according to the form of a stimulus. To ensure safe processing the items in the stream are labeled with a class that determines the type of the underlying datum. The items are processed by performing a case analysis on the class, and passing the underlying datum to a handler for items of that class.

A difficulty with using sums for this purpose, however, is that the developer must specify in advance the classes of data that are to be considered. That is, sums support *static classification* of data based on a fixed collection of classes. While this works well in the vast majority of cases, there are situations where static classification is inadequate, and *dynamic classification* is required. For example, we may wish to classify data in order to keep it secret from an intermediary in a computation. By creating a fresh class at execution time, two parties engaging in a communication can arrange that they, and only they, are able to compute with a given datum; all others must merely handle it passively without examining its structure or value.

One example of this sort of interaction arises when programming with exceptions, as described in Chapter 28. One may consider the value associated with an exception to be a secret that is shared between the program component that raises the exception and the program component that handles it. No other intervening handler may intercept the exception value; only the designated handler is permitted to process it. This behavior may be readily modelled using dynamic classification. Exception values are dynamically classified, with the class of the value known only to the raiser and to the intended handler, and to no others.

One may wonder why dynamic, as opposed to static, classification is appropriate for exception values. To do otherwise—that is, to use static classification—would require a global commitment to the possible forms of exception value that may be used in a program. This creates problems for modularity, since any such global commitment must be made for the whole program, rather than for each of its components separately. Dynamic classification ensures that when any two components are integrated, the classes they introduce are disjoint from one another, avoiding integration problems while permitting separate development.

In this chapter we study two separable concepts, *dynamic classification*, and *dynamic classes*. *Dynamic classification* permits the classification of data values using dynamically generated symbols (as described in Section 34.4 on page 313 of Chapter 34). A value is classified by tagging it with a symbol that determines the type of its associated value. A classified value is inspected by pattern matching against a finite set of known classes, dispatching according to whether the class of the value is among them, with a default behavior if it is not. *Dynamic classes* treat class labels as a form of dynamic data. This allows a class to be communicated between components at run-time without embedding it into another data structure. Dynamic classes, in combination with product and existential types (Chapter 24), are sufficient to implement dynamic classification.

36.1 Dynamic Classification

The language $\mathcal{L}\{\texttt{classified}\}$ uses dynamically generated symbols as class identifiers.[1] We rely on the implicit identification of α-equivalent expressions to ensure that a dynamically generated symbol is distinct from any of

[1] Dynamic symbol generation was introduced in Chapter 34. However, the use of symbols for dynamic classification does not imply that there is any form of fluid binding involved!

the finitely many symbols that have been previously generated.

The syntax of $\mathcal{L}\{\texttt{classified}\}$ is given by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	`clsfd`	`clsfd`
Expr	e	::=	`in[`a`](`e`)`	`in[`a`](`e`)`
		$\mid$	`ccase(`e`;`e_0`;`r_1`,...,`r_n`)`	`ccase` e $\{r_1 \mid \ldots \mid r_n\}$ `ow` e_0
Rule	r	::=	`in?[`a`](`x`.`e`)`	`in[`a`](`x`)` $\Rightarrow$ e

The expression `in[`a`](`e`)` classifies the value of the expression e by labelling it with the symbol a. The expression `ccase` e $\{r_1 \mid \ldots \mid r_n\}$ `ow` e_0 analyzes the class of e according to the rule $r_1, \ldots, r_n$. Each rule has the form `in[`a_i`](`x_i`)` $\Rightarrow$ e_i, consisting of a symbol, a_i, representing a candidate class of the analyzed value; a variable, x_i, representing the associated data value for a value of that class; and an expression, e_i, to be evaluated in the case that the analyzed expression is labelled with class a_i. If the class of the analyzed value does not match any of the rules, the default expression, e_0, is evaluated instead. A default case is required, since no static type system can, in general, circumscribe the set of possible classes of a classified value, and hence pattern matches on classified values cannot be guaranteed to be exhaustive.

The static semantics of $\mathcal{L}\{\texttt{classified}\}$ consists of a judgement of the form $\Sigma\,\Gamma \vdash e : \tau$, where Σ specifies the types of the symbols (as described in Chapter 34), and Γ specifies the types of the variables. The definition makes use of an auxiliary judgement of the form $\Sigma\,\Gamma \vdash r : \tau$, specifying that a rule, r, matches a classified value of the form `in[`a`](`e`)` and yields a value of type τ. These judgements are inductively defined by the following rules:

$$\frac{\Sigma \vdash a : \tau \quad \Sigma\,\Gamma \vdash e : \tau}{\Sigma\,\Gamma \vdash \texttt{in[}a\texttt{](}e\texttt{)} : \texttt{clsfd}} \tag{36.1a}$$

$$\frac{\Sigma\,\Gamma \vdash e : \texttt{clsfd} \quad \Sigma\,\Gamma \vdash e_0 : \tau \quad \Sigma\,\Gamma \vdash r_1 : \tau \quad \ldots \quad \Sigma\,\Gamma \vdash r_n : \tau}{\Sigma\,\Gamma \vdash \texttt{ccase(}e\texttt{;}e_0\texttt{;}r_1\texttt{,}\ldots\texttt{,}r_n\texttt{)} : \tau} \tag{36.1b}$$

$$\frac{\Sigma \vdash a : \sigma \quad \Sigma\,\Gamma, x : \sigma \vdash e : \tau}{\Sigma\,\Gamma \vdash \texttt{in?[}a\texttt{](}x\texttt{.}e\texttt{)} : \tau} \tag{36.1c}$$

The dynamic semantics of these operations is an entirely straightforward extension of the semantics of dynamic symbol generation given in Section 34.4 on page 313.

$$\frac{e\ \mathsf{val}}{\texttt{in[}a\texttt{](}e\texttt{)}\ \mathsf{val}} \tag{36.2a}$$

$$\frac{e \mathbin{@} \nu \mapsto e_0 \mathbin{@} \nu'}{\mathtt{in}[a]\,(e) \mathbin{@} \nu \mapsto \mathtt{in}[a]\,(e_0) \mathbin{@} \nu'} \tag{36.2b}$$

$$\frac{e \mathbin{@} \nu \mapsto e' \mathbin{@} \nu'}{\mathtt{ccase}(e; e_0; r_1, \ldots, r_n) \mathbin{@} \nu \mapsto \mathtt{ccase}(e'; e_0; r_1, \ldots, r_n) \mathbin{@} \nu'} \tag{36.2c}$$

$$\frac{\mathtt{in}[a]\,(e)\ \mathsf{val}}{\mathtt{ccase}(\mathtt{in}[a]\,(e); e_0; \epsilon) \mathbin{@} \nu \mapsto e_0 \mathbin{@} \nu} \tag{36.2d}$$

$$\frac{\mathtt{in}[a_1]\,(e_1)\ \mathsf{val}}{\begin{array}{c}\mathtt{ccase}(\mathtt{in}[a_1]\,(e_1); e_0; \mathtt{in?}[a_1]\,(x_1.e_1'), \ldots, \mathtt{in?}[a_n]\,(x_n.e_n')) \mathbin{@} \nu \\ \mapsto \\ [e_1/x_1]e_1' \mathbin{@} \nu\end{array}} \tag{36.2e}$$

$$\frac{\mathtt{in}[a]\,(e)\ \mathsf{val} \quad a \neq a_1 \quad n > 0}{\begin{array}{c}\mathtt{ccase}(\mathtt{in}[a_1]\,(e_1); e_0; \mathtt{in?}[a_1]\,(x_1.e_1'), \ldots, \mathtt{in?}[a_n]\,(x_n.e_n')) \mathbin{@} \nu \\ \mapsto \\ \mathtt{ccase}(\mathtt{in}[a]\,(e); e_0; \mathtt{in?}[a_2]\,(x_2.e_2'), \ldots, \mathtt{in?}[a_n]\,(x_n.e_n')) \mathbin{@} \nu\end{array}} \tag{36.2f}$$

Rule (36.2d) specifies that the default case is evaluated when all rules have been exhausted (that is, the sequence of rules is empty). Rules (36.2e) and (36.2f) specify that each rule is considered in turn, matching the class of the analyzed expression to the class of each of the successive rules of the case analysis.

The statement and proof of type safety for $\mathcal{L}\{\mathtt{classified}\}$ proceeds along the lines of the safey proofs given in Chapters 17, 18, and 34.

Theorem 36.1 (Preservation)**.** *Suppose that $e \mathbin{@} \nu \mapsto e' \mathbin{@} \nu'$, where $\Sigma \vdash e : \tau$ and $\Sigma \vdash a : \tau_a$ whenever $a \in \nu$. Then $\nu' \supseteq \nu$, and there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \vdash e' : \tau$ and $\Sigma' \vdash a' : \tau_{a'}$ for each $a' \in \nu'$.*

Lemma 36.2 (Canonical Forms)**.** *Suppose that $\Sigma \vdash e : \mathtt{clsfd}$ and e val. Then $e = \mathtt{in}[a]\,(e')$ for some a such that $\Sigma \vdash a : \tau$ and some e' such that e' val and $\Sigma \vdash e' : \tau$.*

Theorem 36.3 (Progress)**.** *Suppose that $\Sigma \vdash e : \tau$, and that if $a \in \nu$, then $\Sigma \vdash a : \tau_a$ for some type τ_a. Then either e val, or $e \mathbin{@} \nu \mapsto e' \mathbin{@} \nu'$ for some ν' and e'.*

36.2 Dynamic Classes

Dynamic classification may be used in combination with higher-order functions to provide controlled access to data among the components of a program as described in the introduction to this chapter. Given a dynamically generated (and hence globally unique) symbol, a, of type τ, one may define two functions of type $\tau \to \texttt{clsfd}$ and $\texttt{clsfd} \to \tau$ that, respectively, classify a value of type τ with class a and declassify a value classified by a, failing otherwise. Either or both of these functions may be passed out of the scope of the binder that introduced the symbol a, ensuring that knowledge of its identity is confined to these two operations. Any component with access to the classification operation may create a value with class a, and only those components with access to the declassification operation may recover the classified value.

A more direct way to enforce privacy is to treat classes themselves as values of type $\tau\,\texttt{class}$, where τ is the type of data labelled by that class. The language $\mathcal{L}\{\texttt{class}\}$ consists of the dynamic symbol generation mechanism described in Chapter 34 together with the primitive operations for the type $\tau\,\texttt{class}$. The syntax of $\mathcal{L}\{\texttt{class}\}$ is specified by the following grammar:

Category	*Item*		*Abstract*	*Concrete*
Type	τ	::=	$\texttt{class}(\tau)$	$\tau\,\texttt{class}$
Expr	e	::=	$\texttt{cls}[a]$	$\texttt{cls}[a]$
		$\mid$	$\texttt{ccase}[t.\sigma](e; e_0; r_1, \ldots, r_n)$	$\texttt{ccase}\,e\,\{r_1 \mid \ldots \mid r_n\}\,\texttt{ow}\,e_0$
Rule	r	$\mid$	$\texttt{cls?}[a](e)$	$\texttt{cls}[a] \Rightarrow e$

The type $\tau\,\texttt{class}$ represents the type of classes with associated values of type τ. A value of this type has the form $\texttt{cls}[a]$, where a is a symbol labelling a class of values. The expression $\texttt{ccase}\,e\,\{r_1 \mid \ldots \mid r_n\}\,\texttt{ow}\,e_0$ is analogous to the class case construct of $\mathcal{L}\{\texttt{classified}\}$, except that there is no data associated with each class. The abstractor, $t.\sigma$, in the syntax of the class case construct plays a critical role in the static semantics of $\mathcal{L}\{\texttt{class}\}$.

The static semantics of $\mathcal{L}\{\texttt{class}\}$ must take care to propagate type identity information gained during pattern matching. For suppose that e is an expression of type $\texttt{class}(\tau)$, and that we analyze its class using a series of rules of the form $\texttt{cls?}[a_i](e_i)$, where each symbol a_i has the corresponding type τ_i. The type of e ensures that its value is of the form $\texttt{cls}[a]$ for some class symbol a of type τ. The typing rule for the case analysis must allow for the possibility that a is one of the a_i's, in which case we must propagate the fact that τ_i is, in fact, τ. This is achieved by assigning the case analysis the type $[\tau/t]\sigma$, and insisting that for each $1 \leq i \leq n$, we have that

$e_i : [\tau_i/t]\sigma$. In the case that a is a_i, then we are, in effect, treating an expression of type $[\tau_i/t]\sigma$ as an expression of type $[\tau/t]\sigma$, which is justified by the equality of τ_i and τ.

The static semantics of $\mathcal{L}\{\texttt{class}\}$ consists of expression typing judgements of the form $\Sigma\,\Gamma \vdash e : \tau$, and rule typing judgements of the form $\Sigma\,\Gamma \vdash r : \texttt{class}(\tau) > \tau'$. These judgements are inductively defined by the following rules:

$$\frac{\Sigma \vdash a : \tau}{\Sigma\,\Gamma \vdash \texttt{cls}[a] : \texttt{class}(\tau)} \tag{36.3a}$$

$$\frac{\begin{array}{c}\Sigma\,\Gamma \vdash e : \texttt{class}(\tau) \qquad \Sigma\,\Gamma \vdash e_0 : [\tau/t]\sigma \\ \Sigma\,\Gamma \vdash r_1 : \texttt{class}(\tau_1) > [\tau_1/t]\sigma \quad \ldots \quad \Sigma\,\Gamma \vdash r_n : \texttt{class}(\tau_n) > [\tau_n/t]\sigma\end{array}}{\Sigma\,\Gamma \vdash \texttt{ccase}[t.\sigma](e; e_0; r_1, \ldots, r_n) : [\tau/t]\sigma} \tag{36.3b}$$

$$\frac{\Sigma \vdash a : \tau \quad \Sigma\,\Gamma \vdash e : \tau'}{\Sigma\,\Gamma \vdash \texttt{cls?}[a](e) : \texttt{class}(\tau) > \tau'} \tag{36.3c}$$

Rule (36.3a) specifies that the class $\texttt{cls}[a]$ has type $\texttt{class}(\tau)$, where τ is the type associated to the class a by Σ. Rule (36.3b) specifies the type of a case analysis on a class of type $\texttt{class}(\tau)$ to be $[\tau/t]\sigma$, where each rule yields a value of type $[\tau_i/t]\sigma$, and the default case is of type $[\tau/t]\tau$.

The dynamic semantics of $\mathcal{L}\{\texttt{class}\}$ is similar to that of $\mathcal{L}\{\texttt{classified}\}$. States have the form $e \,@\, \nu$, where ν is a finite set of symbols. Final states are those for which e val; all states are initial states. The rules defining the judgement $e \,@\, \nu \mapsto e' \,@\, \nu'$ are easily derived from Rules (36.2), and are omitted here for the sake of brevity.

Theorem 36.4 (Preservation)**.** *Suppose that $e \,@\, \nu \mapsto e' \,@\, \nu'$, where $\Sigma \vdash e : \tau$ and $\Sigma \vdash a : \tau_a$ whenever $a \in \nu$. Then $\nu' \supseteq \nu$, and there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \vdash e' : \tau$ and $\Sigma' \vdash a' : \tau_{a'}$ for each $a' \in \nu'$.*

Proof. By rule induction on the dynamic semantics of $\mathcal{L}\{\texttt{class}\}$. The most interesting case arises when $e = \texttt{cls}[a]$ and $a = a_i$ for some rule $\texttt{cls}[a_i] \Rightarrow e_i$. By inversion of typing we know that $e_i : [\tau_i/t]\sigma$. We are to show that $e_i : [\tau/t]\sigma$. This follows directly from the observation that if $a = a_i$, then by unicity of typing, $\tau_i = \tau$. □

Lemma 36.5 (Canonical Forms)**.** *Suppose that $\Sigma \vdash e : \tau$ `class` and e val. Then $e =$ `cls[a]` for some a such that $\Sigma \vdash a : \tau$.*

Proof. By rule induction on Rules (36.3), taking account of the definition of values. □

Theorem 36.6 (Progress). *Suppose that* $\Sigma \vdash e : \tau$, *and that if* $a \in \nu$, *then* $\Sigma \vdash a : \tau_a$ *for some type* τ_a. *Then either* e `val`, *or* e `@`$\nu \mapsto e'$ `@`ν' *for some* ν' *and* e'.

Proof. By rule induction on Rules (36.3). For a case analysis of the form `ccase` $e\,\{r_1 \mid \ldots \mid r_n\}$ `ow` e_0, where e `val`, we have by Lemma 36.5 on the facing page that $e =$ `cls[`a`]` for some a. Either $a = a_i$ for some rule `cls[`a_i`]` $\Rightarrow e_i$ in $r_1, \ldots, r_n$, in which case we progress to e_i, or else we progress to e_0. □

36.3 From Classes to Classification

Dynamic classification is definable in a language with dynamic class types, existential types, and product types. Specifically, the type `clsfd` may be considered to stand for the existential type

$$\exists(t.t\,\texttt{class} \times t).$$

According to this identification, the classified value `in[`a`]` `(`e`)` is the package

$$\texttt{pack}\,\tau\,\texttt{with}\,\langle \texttt{cls[}a\texttt{]}, e\rangle\,\texttt{as}\,\exists(t.t\,\texttt{class} \times t),$$

where a is a symbol of type τ. Case analysis is performed by opening the package and dispatching on its encapsulated class component. To be specific, suppose that the class case expression `ccase` $e\,\{r_1 \mid \ldots \mid r_n\}$ `ow` e' has type ρ, where r_i is the rule `in[`a_i`]` `(`x_i`)` $\Rightarrow e_i$, with $x_i : \tau_i \vdash e_i : \rho$. This expression is defined to be

$$\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,\langle x, y\rangle\texttt{:}t\,\texttt{class} \times t\,\texttt{in}\,(e_{body}(y)),$$

where e_{body} is an expression to be defined shortly. Case analysis proceeds by opening the package, e, representing the classified value, and decomposing it into a class, x, and an underlying value, y. The body of the `open` analyzes the class x, yielding a function of type $t \to \rho$, where t is the abstract type introduced by the `open`. This function is applied to y, the value that is labelled by x in the package.

The core of the case analysis, namely the expression e_{body}, analyzes the encapsulated class, x, of the package. The case analysis is parameterized

by the type abstractor $u\,.\,u \to \rho$, where u is not free in ρ. The overall type of the case is $[t/u]u \to \rho = t \to \rho$, which ensures that the application to y to the classified value is well-typed. Each branch of the case analysis has type $\tau_i \to \rho$, as required by Rule (36.3b). In sum, the expression e_{body} is defined to be the expression

$$\texttt{ccase}\, x\, \{r'_1 \mid \ldots \mid r'_n\}\, \texttt{ow}\, \lambda(\_ : t.\, e_0),$$

where for each $1 \leq i \leq n$, the rule r'_i is defined to be

$$\texttt{cls}[a_i] \Rightarrow \lambda(x_i : \tau_i.\, e_i).$$

One may check that the static and dynamic semantics of $\mathcal{L}\{\texttt{classified}\}$ are derivable according to these definitions.

36.4 Exercises

1. Derive the Standard ML exception mechanism from the machinery developed here.