# Part X

# Dynamic Typing

# Chapter 24

# Dynamic Typing

The formalization of type safety given in Chapter 9 states that a language is type safe iff it satisfies both *preservation* and *progress*. According to this account, "stuck" states — non-final states with no transition — must be rejected by the static type system as ill-typed. Although this requirement seems natural for relatively simple languages such as MinML, it is not immediately clear that our formalization of type safety scales to larger languages, nor is it entirely clear that the informal notion of safety is faithfully captured by the preservation and progress theorems.

One issue that we addressed in Chapter 9 was how to handle expressions such as `3 div 0`, which are well-typed, yet stuck, in apparent violation of the progress theorem. We discussed two possible ways to handle such a situation. One is to enrich the type system so that such an expression is ill-typed. However, this takes us considerably beyond the capabilities of current type systems for practical programming languages. The alternative is to ensure that such ill-defined states are not "stuck", but rather make a transition to a designated error state. To do so we introduced the notion of a checked error, which is explicitly detected and signalled during execution. Checked errors are constrasted with unchecked errors, which are ruled out by the static semantics.

In this chapter we will concern ourselves with question of why there should unchecked errors at all. Why aren't all errors, including type errors, checked at run-time? Then we can dispense with the static semantics entirely, and, in the process, execute more programs. Such a language is called *dynamically typed*, in contrast to MinML, which is *statically typed*.

One advantage of dynamic typing is that it supports a more flexible treatment of conditionals. For example, the expression

```
(if true then 7 else "7")+1
```

is statically ill-typed, yet it executes successfully without getting stuck or incurring a checked error. Why rule it out, simply because the type checker is unable to "prove" that the `else` branch cannot be taken? Instead we may shift the burden to the programmer, who is required to maintain invariants that ensure that no run-time type errors can occur, even though the program may contain conditionals such as this one.

Another advantage of dynamic typing is that it supports *heterogeneous data structures*, which may contain elements of many different types. For example, we may wish to form the "list"

```
[true, 1, 3.4, fn x=>x]
```

consisting of four values of distinct type. Languages such as ML preclude formation of such a list, insisting instead that all elements have the *same* type; these are called *homogenous* lists. The argument for heterogeneity is that there is nothing inherently "wrong" with such a list, particularly since its constructors are insensitive to the types of the components — they simply allocate a new node in the heap, and initialize it appropriately.

Note, however, that the additional flexibility afforded by dynamic typing comes at a cost. Since we cannot accurately predict the outcome of a conditional branch, nor the type of a value extracted from a heterogeneous data structure, we must program defensively to ensure that nothing bad happens, even in the case of a type error. This is achieved by turning type errors into checked errors, thereby ensuring progress and hence safety, even in the absence of a static type discipline. Thus dynamic typing catches type errors as late as possible in the development cycle, whereas static typing catches them as early as possible.

In this chapter we will investigate a dynamically typed variant of MinML in which type errors are treated as checked errors at execution time. Our analysis will reveal that, rather than being opposite viewpoints, dynamic typing is a *special case* of static typing! In this sense static typing is *more expressive* than dynamic typing, despite the superficial impression created by the examples given above. This viewpoint illustrates the *pay-as-you-go* principle of language design, which states that a program should only incur overhead for those language features that it actually uses. By viewing dynamic typing as a special case of static typing, we may avail ourselves of the benefits of dynamic typing whenever it is required, but avoid its costs whenever it is not.

## 24.1 Dynamic Typing

The fundamental idea of dynamic typing is to regard type clashes as *checked*, rather than *unchecked*, errors. Doing so puts type errors on a par with division by zero and other checked errors. This is achieved by augmenting the dynamic semantics with rules that explicitly check for stuck states. For example, the expression `true+7` is such an ill-typed, stuck state. By checking that the arguments of an addition are integers, we can ensure that progress may be made, namely by making a transition to `error`.

The idea is easily illustrated by example. Consider the rules for function application in MinML given in Chapter 8, which we repeat here for convenience:

$$\frac{v \text{ value} \quad v_1 \text{ value} \quad (v = \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\, \texttt{is}\, e\, \texttt{end})}{\texttt{apply(}v, v_1\texttt{)} \mapsto \{v, v_1/f, x\}e}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{apply(}e_1, e_2\texttt{)} \mapsto \texttt{apply(}e_1', e_2\texttt{)}}$$

$$\frac{v_1 \text{ value} \quad e_2 \mapsto e_2'}{\texttt{apply(}v_1, e_2\texttt{)} \mapsto \texttt{apply(}v_1, e_2'\texttt{)}}$$

In addition to these rules, which govern the well-typed case, we add the following rules governing the ill-typed case:

$$\frac{v \text{ value} \quad v_1 \text{ value} \quad (v \neq \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\, \texttt{is}\, e\, \texttt{end})}{\texttt{apply(}v, v_1\texttt{)} \mapsto \texttt{error}}$$

$$\frac{}{\texttt{apply(error,}e_2\texttt{)} \mapsto \texttt{error}}$$

$$\frac{v_1 \text{ value}}{\texttt{apply(}v_1\texttt{,error)} \mapsto \texttt{error}}$$

The first rule states that a run-time error arises from any attempt to apply a non-function to an argument. The other two define the propagation of such errors through other expressions — once an error occurs, it propagates throughout the entire program.

By entirely analogous means we may augment the rest of the semantics of MinML with rules to check for type errors at run time. Once we have done so, it is safe to eliminate the static semantics in its entirety.[1] Having done

[1] We may then simplify the language by omitting type declarations on variables and functions, since these are no longer of any use.

so, every expression is well-formed, and hence preservation holds vacuously. More importantly, the progress theorem also holds because we have augmented the dynamic semantics with transitions from every ill-typed expression to `error`, ensuring that there are no "stuck" states. Thus, the dynamically typed variant of MinML is safe in same sense as the statically typed variant. The meaning of safety does not change, only the means by which it is achieved.

## 24.2 Implementing Dynamic Typing

Since both the statically- and the dynamically typed variants of MinML are safe, it is natural to ask which is better. The main difference is in how early errors are detected — at compile time for static languages, at run time for dynamic languages. Is it better to catch errors early, but rule out some useful programs, or catch them late, but admit more programs? Rather than attempt to settle this question, we will sidestep it by showing that the apparent dichotomy between static and dynamic typing is illusory by showing that dynamic typing is a *mode of use* of static typing. From this point of view static and dynamic typing are matters of design for a particular program (which to use in a *given* situation), rather than a doctrinal debate about the design of a programming language (which to use in *all* situations).

To see how this is possible, let us consider what is involved in implementing a dynamically typed language. The dynamically typed variant of MinML sketched above includes rules for run-time type checking. For example, the dynamic semantics includes a rule that explicitly checks for an attempt to apply a non-function to an argument. How might such a check be implemented? The chief problem is that the natural representations of data values on a computer do not support such tests. For example, a function might be represented as a word representing a pointer to a region of memory containing a sequence of machine language instructions. An integer might be represented as a word interpreted as a two's complement integer. But given a word, you cannot tell, in general, whether it is an integer or a code pointer.

To support run-time type checking, we must adulterate our data representations to ensure that it is possible to implement the required checks. We must be able to tell by looking at the value whether it is an integer, a boolean, or a function. Having done so, we must be able to recover the underlying value (integer, boolean, or function) for direct calculation. When-

ever a value of a type is created, it must be marked with appropriate information to identify the sort of value it represents.

There are many schemes for doing this, but at a high level they all amount to attaching a *tag* to a "raw" value that identifies the value as an integer, boolean, or function. Dynamic typing then amounts to checking and stripping tags from data during computation, transitioning to `error` whenever data values are tagged inappropriately. From this point of view, we see that dynamic typing should *not* be described as "run-time type checking", because we are not checking *types* at run-time, but rather *tags*. The difference can be seen in the application rule given above: we check only that the first argument of an application is some function, not whether it is well-typed in the sense of the MinML static semantics.

To clarify these points, we will make explicit the manipulation of tags required to support dynamic typing. To begin with, we revise the grammar of MinML to make a distinction between *tagged* and *untagged* values, as follows:

| | | | |
|---|---|---|---|
| *Expressions* | $e$ | $::=$ | $x \mid v \mid o(e_1, \ldots, e_n) \mid \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \mid \texttt{apply}(e_1, e_2)$ |
| *TaggedValues* | $v$ | $::=$ | $\texttt{Int}(n) \mid \texttt{Bool}(\texttt{true}) \mid \texttt{Bool}(\texttt{false}) \mid \texttt{Fun}(\texttt{fun}\, x\, (y{:}\tau_1){:}\tau_2\, \texttt{is}\, e\, \texttt{end})$ |
| *UntaggedValues* | $u$ | $::=$ | $\texttt{true} \mid \texttt{false} \mid n \mid \texttt{fun}\, x\, (y{:}\tau_1){:}\tau_2\, \texttt{is}\, e\, \texttt{end}$ |

Note that *only* tagged values arise as expressions; untagged values are used strictly for "internal" purposes in the dynamic semantics. Moreover, we do not admit general tagged expressions such as $\texttt{Int}(e)$, but only explicitly-tagged values.

Second, we introduce *tag checking* rules that determine whether or not a tagged value has a given tag, and, if so, extracts its underlying untagged value. In the case of functions these are given as rules for deriving judgements of the form $v$ is_fun $u$, which checks that $v$ has the form $\texttt{Fun}(u)$, and extracts $u$ from it if so, and for judgements of the form $v$ isnt_fun, that checks that $v$ does not have the form $\texttt{Fun}(u)$ for any untagged value $u$.

$$\frac{}{\texttt{Fun}(u)\ \text{is\_fun}\ u}$$

$$\frac{}{\texttt{Int}(\_)\ \text{isnt\_fun}} \qquad \frac{}{\texttt{Bool}(\_)\ \text{isnt\_fun}}$$

Similar judgements and rules are used to identify integers and booleans, and to extract their underlying untagged values.

Finally, the dynamic semantics is re-formulated to make use of these judgement forms. For example, the rules for application are as follows:

$$\frac{v_1 \text{ value} \quad v \text{ is\_fun } \texttt{fun}\ f\ \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\ \texttt{is}\ e\ \texttt{end}}{\texttt{apply(}v, v_1\texttt{)} \mapsto \{v, v_1/f, x\}e}$$

$$\frac{v \text{ value} \quad v \text{ isnt\_fun}}{\texttt{apply(}v, v_1\texttt{)} \mapsto \texttt{error}}$$

Similar rules govern the arithmetic primitives and the conditional expression. For example, here are the rules for addition:

$$\frac{v_1 \text{ value} \quad v_2 \text{ value} \quad v_1 \text{ is\_int } n_1 \quad v_2 \text{ is\_int } n_2 \quad (n = n_1 + n_2)}{+(v_1, v_2) \mapsto \texttt{Int(}n\texttt{)}}$$

Note that we must explicitly check that the arguments are tagged as integers, and that we must apply the integer tag to the result of the addition.

$$\frac{v_1 \text{ value} \quad v_2 \text{ value} \quad v_1 \text{ isnt\_int}}{+(v_1, v_2) \mapsto \texttt{error}}$$

$$\frac{v_1 \text{ value} \quad v_2 \text{ value} \quad v_1 \text{ is\_int } n_1 \quad v_2 \text{ isnt\_int}}{+(v_1, v_2) \mapsto \texttt{error}}$$

These rules explicitly check for non-integer arguments to addition.

## 24.3 Dynamic Typing as Static Typing

Once tag checking is made explicit, it is easier to see its hidden costs in both time and space — time to check tags, to apply them, and to extract the underlying untagged values, and space for the tags themselves. This is a significant overhead. Moreover, this overhead is imposed *whether or not* the original program is statically type correct. That is, even if we can prove that no run-time type error can occur, the dynamic semantics nevertheless dutifully performs tagging and untagging, just as if there were no type system at all.

This violates a basic principle of language design, called the *pay-as-you-go* principle. This principle states that a language should impose the cost of a feature only to the extent that it is actually used in a program. With dynamic typing we pay for the cost of tag checking, even if the program is statically well-typed! For example, if all of the lists in a program are

homogeneous, we should not have to pay the overhead of supporting heterogeneous lists. The choice should be in the hands of the programmer, not the language designer.

It turns out that we can eat our cake and have it too! The key is a simple, but powerful, observation: dynamic typing is but a mode of use of static typing, provided that our static type system includes a type of *tagged data*! Dynamic typing emerges as a particular style of programming with tagged data.

The point is most easily illustrated using ML. The type of tagged data values for MinML may be introduced as follows:

```
(* The type of tagged values. *)
datatype tagged =
  Int of int |
  Bool of bool |
  Fun of tagged -> tagged
```

Values of type `tagged` are marked with a value constructor indicating their outermost form. Tags may be manipulated using pattern matching.

Second, we introduce operations on tagged data values, such as addition or function call, that explicitly check for run-time type errors.

```
exception TypeError

fun checked_add (m:tagged, n:tagged):tagged =
    case (m,n) of
      (Int a, Int b) => Int (a+b)
    | (_, _) => raise TypeError

fun checked_apply (f:tagged, a:tagged):tagged =
    case f of
      Fun g => g a
    | _ => raise TypeError
```

Observe that these functions correspond precisely to the instrumented dynamic semantics given above.

Using these operations, we can then build heterogeneous lists as values of type `tagged list`.

```
val het_list : tagged list =
    [Int 1, Bool true, Fun (fn x => x)]
val f : tagged = hd(tl(tl het_list))
val x : tagged = checked_apply (f, Int 5)
```

The tags on the elements serve to identify what sort of element it is: an integer, a boolean, or a function.

It is enlightening to consider a dynamically typed version of the factorial function:

```
fun dyn_fact (n : tagged) =
    let fun loop (n, a) =
        case n
          of Int m =>
             (case m
                of 0 => a
                 | m => loop (Int (m-1),
                              checked_mult (m, a)))
           | _ => raise RuntimeTypeError
      in loop (n, Int 1)
    end
```

Notice that tags must be manipulated within the loop, even though we can prove (by static typing) that they are not necessary! Ideally, we would like to hoist these checks out of the loop:

```
fun opt_dyn_fact (n : tagged) =
    let fun loop (0, a) = a
          | loop (n, a) = loop (n-1, n*a)
     in case n
          of Int m => Int (loop (m, 1))
           | _ => raise RuntimeTypeError
    end
```

It is *very hard* for a compiler to do this hoisting reliably. But if you consider dynamic typing to be a special case of static typing, as we do here, there is no obstacle to doing this optimization yourself, as we have illustrated here.

# Chapter 25

# Featherweight Java

We will consider a tiny subset of the Java language, called *Featherweight Java*, or FJ, that models subtyping and inheritance in Java. We will then discuss design alternatives in the context of FJ. For example, in FJ, as in Java, the subtype relation is tightly coupled to the subclass relation. Is this necessary? Is it desirable? We will also use FJ as a framework for discussing other aspects of Java, including interfaces, privacy, and arrays.

## 25.1 Abstract Syntax

The abstract syntax of FJ is given by the following grammar:

| | | | |
|---|---|---|---|
| *Classes* | $C$ | $::=$ | $\texttt{class}\, c\, \texttt{extends}\, c\, \{\underline{c}\, \underline{f}\texttt{;}\ k\, \underline{d}\}$ |
| *Constructors* | $k$ | $::=$ | $c\texttt{(}\underline{c}\, \underline{x}\texttt{)}\, \{\texttt{super(}\underline{x}\texttt{);}\ \texttt{this.}\underline{f}\texttt{=}\underline{x}\texttt{;}\}$ |
| *Methods* | $d$ | $::=$ | $c\, m\texttt{(}\underline{c}\, \underline{x}\texttt{)}\, \{\texttt{return}\, e\texttt{;}\}$ |
| *Types* | $\tau$ | $::=$ | $c$ |
| *Expressions* | $e$ | $::=$ | $x \mid e\texttt{.}f \mid e\texttt{.}m\texttt{(}\underline{e}\texttt{)} \mid \texttt{new}\, c\texttt{(}\underline{e}\texttt{)} \mid \texttt{(}c\texttt{)}\, e$ |

The variable $f$ ranges over a set of *field names*, $c$ over a set of *class names*, $m$ over a set of *method names*, and $x$ over a set of *variable names*. We assume that these sets are countably infinite and pairwise disjoint. We assume that there is a distinguished class name, `Object`, standing for the root of the class hierarchy. It's role will become clear below. We assume that there is a distinguished variable `this` that cannot otherwise be declared in a program.

As a notational convenience we use "underbarring" to stand for sequences of phrases. For example, $\underline{d}$ stands for a sequence of $d$'s, whose

individual elements we designate $d_1, \ldots, d_k$, where $k$ is the length of the sequence. We write $\underline{c}\,\underline{f}$ for the sequence $c_1\,f_1, \ldots, c_k\,f_k$, where $k$ is the length of the sequences $\underline{c}$ and $\underline{f}$. Similar conventions govern the other uses of sequence notation.

The class expression

$$\texttt{class}\,c\,\texttt{extends}\,c'\,\{\underline{c}\,\underline{f}\texttt{;}\,k\,\underline{d}\}$$

declares the class $c$ to be a subclass of the class $c'$. The subclass has additional fields $\underline{c}\,\underline{f}$, single constructor $k$, and method suite $\underline{d}$. The methods of the subclass may override those of the superclass, or may be new methods specific to the subclass.

The constructor expression

$$c\texttt{(}\underline{c'}\,\underline{x'}\texttt{,}\,\underline{c}\,\underline{x}\texttt{)}\,\{\texttt{super(}\underline{x'}\texttt{);}\,\texttt{this.}\underline{f}\texttt{=}\underline{x}\texttt{;}\}$$

declares the constructor for class $c$ with arguments $\underline{c'}\,\underline{x'}, \underline{c}\,\underline{x}$, corresponding to the fields of the superclass followed by those of the subclass. The variables $\underline{x'}$ and $\underline{x}$ are bound in the body of the constructor. The body of the constructor indicates the initialization of the superclass with the arguments $\underline{x'}$ and of the subclass with arguments $\underline{x}$.

The method expression

$$c\,m\texttt{(}\underline{c}\,\underline{x}\texttt{)}\,\{\texttt{return}\,e\texttt{;}\}$$

declares a method $m$ yielding a value of class $c$, with arguments $\underline{x}$ of class $\underline{c}$ and body returning the value of the expression $e$. The variables $\underline{x}$ and `this` are bound in $e$.

The set of types is, for the time being, limited to the set of class names. That is, the only types are those declared by a class. In Java there are more types than just these, including the primitive types `integer` and `boolean` and the array types.

The set of expressions is the minimal "interesting" set sufficient to illustrate subtyping and inheritance. The expression $e\texttt{.}f$ selects the contents of field $f$ from instance $e$. The expression $e\texttt{.}m\texttt{(}\underline{e}\texttt{)}$ invokes the method $m$ of instance $e$ with arguments $\underline{e}$. The expression $\texttt{new}\,c\texttt{(}\underline{e}\texttt{)}$ creates a new instance of class $c$, passing arguments $\underline{e}$ to the constructor for $c$. The expression $\texttt{(}c\texttt{)}\,e$ casts the value of $e$ to class $c$.

The methods of a class may invoke one another by sending messages to `this`, standing for the instance itself. We may think of `this` as a bound variable of the instance, but we will arrange things so that renaming of `this` is never necessary to avoid conflicts.

```
class Pt extends Object {
  int x;
  int y;
  Pt (int x, int y) {
     super(); this.x = x; this.y = y;
  }
  int getx () { return this.x; }
  int gety () { return this.y; }
}
class CPt extends Pt {
  color c;
  CPt (int x, int y, color c) {
    super(x,y);
    this.c = c;
  }
  color getc () { return this.c; }
}
```

Figure 25.1: A Sample FJ Program

A *class table* $T$ is a finite function assigning classes to class names. The classes declared in the class table are bound within the table so that all classes may refer to one another via the class table.

A *program* is a pair $(T, e)$ consisting of a class table $T$ and an expression $e$. We generally suppress explicit mention of the class table, and consider programs to be expressions.

A small example of FJ code is given in Figure 25.1. In this example we assume given a class `Object` of all objects and make use of types `int` and `color` that are not, formally, part of FJ.

## 25.2 Static Semantics

The static semantics of FJ is defined by a collection of judgments of the following forms:

| | |
|---|---|
| $\tau <: \tau'$ | *subtyping* |
| $\Gamma \vdash e : \tau$ | *expression typing* |
| $d$ ok in $c$ | *well-formed method* |
| $C$ ok | *well-formed class* |
| $T$ ok | *well-formed class table* |
| $\mathsf{fields}(c) = \underline{c}\ \underline{f}$ | *field lookup* |
| $\mathsf{type}(m, c) = \underline{c} \to c$ | *method type* |

The rules defining the static semantics follow.
Every variable must be declared:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \tag{25.1}$$

The types of fields are defined in the class table.

$$\frac{\Gamma \vdash e_0 : c_0 \quad \mathsf{fields}(c_0) = \underline{c}\ \underline{f}}{\Gamma \vdash e_0.f_i : c_i} \tag{25.2}$$

The argument and result types of methods are defined in the class table.

$$\frac{\Gamma \vdash e_0 : c_0 \quad \Gamma \vdash \underline{e} : \underline{c} \quad \mathsf{type}(m, c_0) = \underline{c'} \to c \quad \underline{c} <: \underline{c'}}{\Gamma \vdash e_0.m(\underline{e}) : c} \tag{25.3}$$

Instantiation must provide values for all instance variables as arguments to the constructor.

$$\frac{\Gamma \vdash \underline{e} : \underline{c} \quad \underline{c} <: \underline{c'} \quad \mathsf{fields}(c) = \underline{c'}\ \underline{f}}{\Gamma \vdash \mathsf{new}\, c(\underline{e}) : c} \tag{25.4}$$

All casts are statically valid, but must be checked at run-time.

$$\frac{\Gamma \vdash e_0 : d}{\Gamma \vdash (c)\, e_0 : c} \tag{25.5}$$

The subtyping relation is read directly from the class table. Subtyping is the smallest reflexive, transitive relation containing the subclass relation:

$$\frac{}{\tau \texttt{ <: } \tau} \tag{25.6}$$

$$\frac{\tau \texttt{ <: } \tau' \quad \tau' \texttt{ <: } \tau''}{\tau \texttt{ <: } \tau''} \tag{25.7}$$

$$\frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c' \,\{\ldots\texttt{;}\ \ldots\}}{c \texttt{ <: } c'} \tag{25.8}$$

A well-formed class has zero or more fields, a constructor that initializes the superclass and the subclass fields, and zero or more methods. To account for method override, the typing rules for each method are relative to the class in which it is defined.

$$\frac{\begin{array}{c} k = c(\underline{c'}\ \underline{x'}, \underline{c}\ \underline{x})\,\{\texttt{super(}\underline{x'}\texttt{);}\ \texttt{this.}\underline{f}\texttt{=}\underline{x}\texttt{;}\} \\ \mathsf{fields}(c') = \underline{c'}\ \underline{f'} \quad \underline{c''}\ \mathsf{ok\ in}\ c \end{array}}{\texttt{class}\, c\, \texttt{extends}\, c' \,\{\underline{c}\ \underline{f}\texttt{;}\ k\ \underline{c''}\}\ \mathsf{ok}} \tag{25.9}$$

Method overriding takes account of the type of the method in the superclass. The subclass method must have the same argument types and result type as in the superclass.

$$\frac{\begin{array}{c} T(c) = \texttt{class}\, c\, \texttt{extends}\, c' \,\{\ldots\texttt{;}\ \ldots\} \\ \mathsf{type}(m, c') = \underline{c} \to c_0 \quad \underline{x}\texttt{:}\underline{c}, \texttt{this}\texttt{:}c \vdash e_0 : c_0 \end{array}}{c_0\, m(\underline{c}\ \underline{x})\,\{\texttt{return}\, e_0\texttt{;}\}\ \mathsf{ok\ in}\ c} \tag{25.10}$$

A method table is well-formed iff all of its classes are well-formed:

$$\frac{\forall c \in \mathrm{dom}(T)\ T(c)\ \mathsf{ok}}{T\ \mathsf{ok}} \tag{25.11}$$

Note that well-formedness of a class is relative to the method table!

A program is well-formed iff its method table is well-formed and the expression is well-formed:

$$\frac{T\ \mathsf{ok} \quad \emptyset \vdash e : \tau}{(T, e)\ \mathsf{ok}} \tag{25.12}$$

The auxiliary lookup judgments determine the types of fields and methods of an object. The types of the fields of an object are determined by the following rules:

$$\frac{}{\mathsf{fields}(\texttt{Object}) = \bullet} \tag{25.13}$$

$$\frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c'\, \{\underline{c}\, \underline{f}\texttt{;}\ \ldots\} \quad \mathsf{fields}(c') = \underline{c'}\ \underline{f'}}{\mathsf{fields}(c) = \underline{c'}\ \underline{f'}, \underline{c}\ \underline{f}} \tag{25.14}$$

The type of a method is determined by the following rules:

$$\frac{\begin{array}{c} T(c) = \texttt{class}\, c\, \texttt{extends}\, c'\, \{\ldots\texttt{;}\ \ldots\ \underline{c''}\} \\ c_i'' = c_i\, m\texttt{(}\underline{c_i}\ \underline{x}\texttt{)}\, \{\texttt{return}\, e\texttt{;}\} \end{array}}{\mathsf{type}(m_i, c) = \underline{c_i} \to c_i} \tag{25.15}$$

$$\frac{\begin{array}{c} T(c) = \texttt{class}\, c\, \texttt{extends}\, c'\, \{\ldots\texttt{;}\ \ldots\ \underline{c''}\} \\ m \notin \underline{c''} \quad \mathsf{type}(m, c') = \underline{c_i} \to c_i \end{array}}{\mathsf{type}(m, c) = \underline{c_i} \to c_i} \tag{25.16}$$

## 25.3 Dynamic Semantics

The dynamic semantics of FJ may be specified using SOS rules similar to those for MinML. The transition relation is indexed by a class table $T$, which governs the semantics of casting and dynamic dispatch (which see below). In the rules below we omit explicit mention of the class table for the sake of brevity.

An instance of a class has the form $\texttt{new}\, c\texttt{(}\underline{e}\texttt{)}$, where each $e_i$ is a value.

$$\frac{\underline{e}\ \mathsf{value}}{\texttt{new}\, c\texttt{(}\underline{e}\texttt{)}\ \mathsf{value}} \tag{25.17}$$

Since we arrange that there be a one-to-one correspondence between instance variables and constructor arguments, an instance expression of this form carries all of the information required to determine the values of the fields of the instance. This makes clear that an instance is essentially just a labelled collection of fields. Each instance is labelled with its class, which is used to guide method dispatch.

Field selection retrieves the value of the named field from either the subclass or its superclass, as appropriate.

$$\frac{\mathsf{fields}(c) = \underline{c'}\ \underline{f'}, \underline{c}\ \underline{f} \quad \underline{e'}\ \mathsf{value} \quad \underline{e}\ \mathsf{value}}{\mathtt{new}\, c(\underline{e'}, \underline{e})\,.\, f'_i \mapsto e'_i} \tag{25.18}$$

$$\frac{\mathsf{fields}(c) = \underline{c'}\ \underline{f'}, \underline{c}\ \underline{f} \quad \underline{e'}\ \mathsf{value} \quad \underline{e}\ \mathsf{value}}{\mathtt{new}\, c(\underline{e'}, \underline{e})\,.\, f_i \mapsto e_i} \tag{25.19}$$

Message send replaces `this` by the instance itself, and replaces the method parameters by their values.

$$\frac{\mathsf{body}(m, c) = \underline{x} \to e_0 \quad \underline{e}\ \mathsf{value} \quad \underline{e'}\ \mathsf{value}}{\mathtt{new}\, c(\underline{e})\,.\, m(\underline{e'}) \mapsto \{\underline{e'}/\underline{x}\}\{\mathtt{new}\, c(\underline{e})/\mathtt{this}\}e_0} \tag{25.20}$$

Casting checks that the instance is of a sub-class of the target class, and yields the instance.

$$\frac{c <: c' \quad \underline{e}\ \mathsf{value}}{(c')\ \mathtt{new}\, c(\underline{e}) \mapsto \mathtt{new}\, c(\underline{e})} \tag{25.21}$$

These rules determine the order of evaluation:

$$\frac{e_0 \mapsto e'_0}{e_0\,.\, f \mapsto e'_0\,.\, f} \tag{25.22}$$

$$\frac{e_0 \mapsto e'_0}{e_0\,.\, m(\underline{e}) \mapsto e'_0\,.\, m(\underline{e})} \tag{25.23}$$

$$\frac{e_0\ \mathsf{value} \quad \underline{e} \mapsto \underline{e'}}{e_0\,.\, m(\underline{e}) \mapsto e_0\,.\, m(\underline{e'})} \tag{25.24}$$

$$\frac{\underline{e} \mapsto \underline{e'}}{\mathtt{new}\, c(\underline{e}) \mapsto \mathtt{new}\, c(\underline{e'})} \tag{25.25}$$

$$\frac{e_0 \mapsto e'_0}{(c)\ e_0 \mapsto (c)\ e'_0} \tag{25.26}$$

Dynamic dispatch makes use of the following auxiliary relation to find the correct method body.

$$\frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c'\, \{\ldots\texttt{;}\ \ldots\ \underline{d}\} \quad d_i = c_i\, m\texttt{(}\underline{c_i}\ \underline{x}\texttt{)}\, \{\texttt{return}\, e\texttt{;}\}}{\mathsf{body}(m_i, c) = \underline{x} \to e} \tag{25.27}$$

$$\frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c'\, \{\ldots\texttt{;}\ \ldots\ \underline{d}\} \quad m \notin \underline{d} \quad \mathsf{type}(m, c') = \underline{x} \to e}{\mathsf{body}(m, c) = \underline{x} \to e} \tag{25.28}$$

Finally, we require rules for evaluating sequences of expressions from left to right, and correspondingly defining when a sequence is a value (*i.e.*, consists only of values).

$$\frac{e_1\ \mathsf{value} \quad \ldots \quad e_{i-1}\ \mathsf{value} \quad e_i \mapsto e_i'}{e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n \mapsto e_1, \ldots, e_{i-1}, e_i', e_{i+1}, \ldots, e_n} \tag{25.29}$$

$$\frac{e_1\ \mathsf{value} \quad \ldots \quad e_n\ \mathsf{value}}{\underline{e}\ \mathsf{value}} \tag{25.30}$$

This completes the dynamic semantics of FJ.

## 25.4 Type Safety

The safety of FJ is stated in the usual manner by the Preservation and Progress Theorems.

Since the dynamic semantics of casts preserves the "true" type of an instance, the type of an expression may become "smaller" in the subtype ordering during execution.

**Theorem 80 (Preservation)**
*Assume that $T$ is a well-formed class table. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau'$ for some $\tau'$ such that $\tau' <: \tau$.*

The statement of Progress must take account of the possibility that a cast may fail at execution time. Note, however, that field selection or message send can never fail — the required field or method will always be present.

**Theorem 81 (Progress)**
*Assume that $T$ is a well-formed class table. If $e : \tau$ then either*

1. $v$ value*, or*
2. $e$ *contains an instruction of the form* `(`$c$`)` `new` $d$`(`$e_0$`)` *with* $e_0$ value *and* $d \not<: c$*, or*
3. *there exists* $e'$ *such that* $e \mapsto e'$*.*

It follows that if no casts occur in the source program, then the second case cannot arise. This can be sharpened somewhat to admit source-level casts for which it is known statically that the type of casted expression is a subtype of the target of the cast. However, we cannot predict, in general, statically whether a given cast will succeed or fail dynamically.

**Lemma 82 (Canonical Forms)**
*If $e : c$ and $e$* value*, then $e$ has the form* `new` $d$`(`$e_0$`)` *with* $e_0$ value *and* $d$ `<:` $c$*.*

## 25.5 Acknowledgement

This chapter is based on "Featherweight Java: A Minimal Core Calculus for Java and GJ" by Atsushi Igarashi, Benjamin Pierce, and Philip Wadler.