# Practical Foundations for Programming Languages

Robert Harper
Carnegie Mellon University

Spring Semester, 2007

[Draft of June 26, 2007 at 5:05pm.]

# Preface

This is a working draft of a book on the foundations of programming languages. The central organizing principle of the book is that programming language features may be seen as manifestations of an underlying type structure that governs its syntax and semantics. The emphasis, therefore, is on the concept of *type*, which codifies and organizes the computational universe in much the same way that the concept of *set* may be seen as an organizing principle for the mathematical universe. The purpose of this book is to explain this remark.

Comments and suggestions are most welcome, and should be sent to the author at `rwh@cs.cmu.edu`.

# Contents

# Part I

# Judgements and Rules

# Chapter 1

# Inductive Definitions

Inductive definitions are an indispensable tool in the study of programming languages. In this chapter we will develop the basic framework of inductive definitions, and give some examples of their use.

## 1.1 Objects and Judgements

We start with the notion of a *judgement*, or *assertion*, about one or more *objects* of study. We shall make use of many forms of judgement, including examples such as these:

| | |
|---|---|
| $n$ nat | $n$ is a natural number |
| $n = n_1 + n_2$ | $n$ is the sum of $n_1$ and $n_2$ |
| $a$ ast | $a$ is an abstract syntax tree |
| $\tau$ type | $\tau$ is a type |
| $e : \tau$ | expression $e$ has type $\tau$ |
| $e \Downarrow v$ | expression $e$ has value $v$ |

A judgement states that one or more objects have a property or stand in some relation to one another. The property or relation itself is called a *judgement form*, and the judgement that an object or objects have that property or stand in that relation is said to be an *instance* of that judgement form. A judgement form is also called a *predicate*, and the objects constituting an instance are its *subjects*.

We use the meta-variables $J$, $K$, and $L$ to stand for an unspecified judgement form, and write $a\ J$ to assert that $J$ holds of the object $a$. When it is not important to stress the subject of the judgement, we sometimes abuse notation and write $J$ stand for an unspecified instance of the judgement form $J$,

relying on context to disambiguate. When discussing particular judgement forms, we freely use prefix, infix, or mixfix notation, as illustrated by the above examples, in order to enhance readability.

We shall be deliberately vague about the universe of objects that may be involved in an inductive definition. As a rough-and-ready rule we permit any objects that can be constructed from finitely many other such objects by an effectively computable process. In particular, we shall assume that the universe of objects is closed under *tupling*, the formation of finite $n$-tuples of objects, written $(a_1, \ldots, a_n)$, where the $a_i$'s are objects. We shall also assume that the universe of objects contains infinitely many distinct *atoms*, also called *names* or *symbols*, and that it is closed under *tagging* of another object with a name. Tupling permits us to express judgements about multiple objects, and tagging permits us to discriminate among collections of objects.

## 1.2 Inference Rules

An inductive definition of judgement form, $J$, consists of a collection of *inference rules*

$$\frac{a_1\ J \quad \ldots \quad a_k\ J}{a\ J}\ , \tag{1.1}$$

where $a$ and each $a_1 \ldots, a_k$ are objects. The judgements above the horizontal line are called the *premises* of the rule, and the judgement below the line is called its *conclusion*. If a rule has no premises (that is, when $k$ is zero), the rule is called an *axiom*; otherwise it is called a *proper rule*.

An inference rule may be read as an implication stating that the premises are *sufficient* for the conclusion: to show $a\ J$, it is enough to show $a_1\ J$, $\ldots$, $a_k\ J$. When $k$ is zero, a rule states categorically that its conclusion holds, independently of any assumptions. Bear in mind that there may be, in general, many rules with the same conclusion, each specifying sufficient conditions for the conclusion. Consequently, if the conclusion of a rule holds, then it is not necessary that the premises hold, for it might have been derived by another rule.

We say that a judgement form, $J$, is closed under an inference rule of the form (1.1) iff $a_1\ J$, $\ldots$, $a_k\ J$ imply $a\ J$. The judgement form *inductively defined* by a set of rules of the form (1.1) is the *strongest*, or *most restrictive*, judgement form, $J$, closed under those rules. That is, the rules are taken as *necessary*, as well as sufficient, conditions for instances of that judgement to be derivable according to the rules: $a\ J$ holds *if, and only if,* there is some

rule of the form (1.1) such that $a_1\ J, \ldots, a_k\ J$ are all derivable according to the rules.

The following rules constitute an inductive definition of the judgement form $\mathsf{nat}$.

$$\frac{}{\mathtt{zero}\ \mathsf{nat}} \tag{1.2a}$$

$$\frac{a\ \mathsf{nat}}{\mathtt{succ}(a)\ \mathsf{nat}} \tag{1.2b}$$

According to this definition, the judgement $a\ \mathsf{nat}$ is holds exactly when either $a$ is $\mathtt{zero}$, or $a$ is $\mathtt{succ}(b)$ for some $b$ such that $b\ \mathsf{nat}$. In other words, $a\ \mathsf{nat}$ holds iff $a$ is a natural number.

Similarly, the following rules constitute an inductive definition of the judgement form $\mathsf{tree}$:

$$\frac{}{\mathtt{empty}\ \mathsf{tree}} \tag{1.3a}$$

$$\frac{a\ \mathsf{tree} \quad b\ \mathsf{tree}}{\mathtt{node}(a,b)\ \mathsf{tree}} \tag{1.3b}$$

According to these rules the judgement $a\ \mathsf{tree}$ holds exactly when $a$ is a binary tree, either the empty tree, $\mathtt{empty}$, or a node, $\mathtt{node}(a,b)$, with children $a$ and $b$ such that $a\ \mathsf{tree}$ and $b\ \mathsf{tree}$.

The judgement $a = b\ \mathsf{nat}$ expresses equality of $a\ \mathsf{nat}$ and $b\ \mathsf{nat}$. This judgement form is inductively defined by the following rules:

$$\frac{}{\mathtt{zero} = \mathtt{zero}\ \mathsf{nat}} \tag{1.4a}$$

$$\frac{a = b\ \mathsf{nat}}{\mathtt{succ}(a) = \mathtt{succ}(b)\ \mathsf{nat}} \tag{1.4b}$$

Similarly, the judgement $a = b\ \mathsf{tree}$ expresses equality of $a\ \mathsf{tree}$ and $b\ \mathsf{tree}$. This judgement form is inductively defined by these rules:

$$\frac{}{\mathtt{empty} = \mathtt{empty}\ \mathsf{tree}} \tag{1.5a}$$

$$\frac{a_1 = b_1\ \mathsf{tree} \quad a_2 = b_2\ \mathsf{tree}}{\mathtt{node}(a_1,a_2) = \mathtt{node}(b_1,b_2)\ \mathsf{tree}} \tag{1.5b}$$

These examples make informal use of the concept of a *rule scheme*, which specifies an infinite collection of rules by using *meta-variables*, here $a$ and $b$, to stand for arbitrary objects from the universe of discourse. An *instance* of a rule scheme is a rule obtained by replacing the meta-variables by an

object from the universe. We tacitly regard a rule scheme as standing for the collection of its instances, relying on context to determine which are the meta-variables of the scheme. (A more rigorous account of these informal conventions is given in Chapter 4.)

## 1.3 Derivations

To show that an instance of an inductively defined judgement form holds, it is enough to exhibit a *derivation* of it. A derivation of a judgement is a composition of rules, starting with axioms and ending with that judgement. A derivation has a natural tree structure arising from regarding the derivations of the premises of the rule as children of a node representing an instance of that rule. We usually depict such trees with the root (conclusion) at the bottom, and with the children of node representing a rule instance drawn as premises of that rule. Thus, if

$$\frac{a_1\ J \quad \ldots \quad a_k\ J}{a\ J} \tag{1.6}$$

is an inference rule and $\nabla_1, \ldots, \nabla_k$ are derivations of its premises, then

$$\frac{\nabla_1 \quad \ldots \quad \nabla_k}{a\ J} \tag{1.7}$$

is a derivation of its conclusion. In particular, if $k = 0$, then the node has no children, and is therefore a leaf of the derivation tree.

For example, here is a derivation of $\texttt{succ(succ(succ(zero)))}\ \texttt{nat}$ according to rules (1.2):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{zero nat}}}{\texttt{succ(zero) nat}}}{\texttt{succ(succ(zero)) nat}}}{\texttt{succ(succ(succ(zero))) nat}}\ . \tag{1.8}$$

Similarly, here is a derivation that $\texttt{node(node(empty,empty),empty)}\ \texttt{tree}$ according to rules (1.3):

$$\dfrac{\dfrac{\dfrac{}{\texttt{empty tree}} \quad \dfrac{}{\texttt{empty tree}}}{\texttt{node(empty,empty) tree}} \quad \dfrac{}{\texttt{empty tree}}}{\texttt{node(node(empty,empty),empty) tree}}\ . \tag{1.9}$$

To show that a judgement is derivable we need only find a derivation for it. There are two main methods for finding a derivation, called *forward chaining*, or *bottom-up construction*, and *backward chaining*, or *top-down construction*. Forward chaining starts with the axioms and works forward towards the desired judgement, whereas backward chaining starts with the desired judgement and works backwards towards the axioms.

More precisely, forward chaining search maintains a set of derivable judgements, and continually extends this set by adding to it the conclusion of any rule all of whose premises are in that set. Initially, the set is empty; the process terminates when the desired judgement occurs in the set. Assuming that all rules are considered at every stage, forward chaining will eventually find a derivation of any derivable judgement, but it is impossible (in general) to decide algorithmically when to stop extending the set and conclude that the desired judgement is not derivable. We may go on and on adding more judgements to the derivable set without ever achieving the intended goal. It is a matter of understanding the global properties of the rules to determine that a given judgement is not derivable.

Forward chaining is undirected in the sense that it does not take account of the end goal when deciding how to proceed at each step. In contrast, backward chaining is goal-directed. Backward chaining search maintains a set of current goals, judgements whose derivations are to be sought. Initially, this set consists solely of the judgement we wish to derive. At each stage, we remove a judgement from the goal set, and consider all rules whose conclusion is that judgement. For each such rule, we add to the goal set the premises of that rule. The process terminates when the goal set is empty, all goals having been achieved. As with forward chaining, backward chaining will eventually find a derivation of any derivable judgement, but there is no algorithmic method for determining in general whether the current goal is derivable. Thus we may futilely add more and more judgements to the goal set, never reaching a point at which all goals have been satisfied.

## 1.4 Rule Induction

If we know that an inductively defined judgement is derivable, then we know it can only be because there is some rule with that judgement as conclusion and such that each of its premises is derivable. Therefore, to show that $P(a)$ holds whenever $a\ J$, it is enough to show that for every rule of the form (1.1) in the definition of $J$, we must show that $P(a_1)$, ..., $P(a_k)$

imply $P(a)$. This is called the principle of *rule induction*, or *induction on derivations*, for the judgement $J$. Notice that the assumptions correspond to inductive hypotheses, and that the conclusion corresponds to the inductive step. When a rule has no assumptions, there are no assumptions, so the conclusion must be established outright—it is a base case of the induction.

The principle of rule induction determined by rule set (1.2) states that to show $P(a)$ whenever $a$ nat, it is enough to establish these two facts:

1. $P(\texttt{zero})$.
2. $P(\texttt{succ}(a))$, assuming $P(a)$.

This is just the familiar principle of *mathematical induction*, arising as a special case of the general principle of rule induction.

Similarly, the principle of rule induction associated with the rules (1.3) states that to show that $a$ tree implies $P(a)$, it is enough to show these two facts:

1. $P(\texttt{empty})$.
2. $P(\texttt{node}(a, b))$, assuming $P(a)$ and $P(b)$.

This is called the principle of *tree induction*, and is once again an instance of rule induction.

As a simple example of the use of rule induction, let us prove that if $a$ tree, then $a = a$ tree. We consider in turn the rules that may be used to derive $a$ tree:

**Rule** (1.3a) Applying Rule (1.5a) we obtain $\texttt{empty} = \texttt{empty}$ tree.

**Rule** (1.3b) Assume that $a = a$ tree and $b = b$ tree. It follows immediately from Rule (1.5b) that $\texttt{node}(a, b) = \texttt{node}(a, b)$ tree.

## 1.5 Iterated and Simultaneous Inductive Definitions

Inductive definitions are often *iterated*, meaning that one inductive definition builds on top of another. For example, the following rules, which define the judgement $a$ list stating that $a$ is a list of natural numbers.

$$\frac{}{\texttt{nil list}} \tag{1.10a}$$

$$\frac{a \text{ nat} \quad b \text{ list}}{\texttt{cons}(a, b) \text{ list}} \tag{1.10b}$$

The second rule refers to the judgement $a$ nat defined earlier.

It is also possible to give a *simultaneous inductive definition* of several judgements, $J_1, \ldots, J_n$, by a system of rules. The rules are considered to define the judgement forms all at once, so that each rule may refer to any of the judgements being defined. The general form of a rule is then

$$\frac{a_1\ J_{i_1} \quad \ldots \quad a_k\ J_{i_k}}{a\ J_i}\ , \tag{1.11}$$

where the premises involve a selection of the judgements being defined, and the conclusion is one of them.

As with singly inductive definitions, a collection of such rules inductively defines the strongest judgement forms $J_1, \ldots, J_n$ that are closed under these rules. This gives rise to a generalized form of rule induction, called *simultaneous rule induction*, in which we may show that $a_i\ P_i$ whenever $a_i\ J_i$ by showing that the properties $P_1, \ldots, P_n$ are simultaneously closed under each of the rules defining the judgement forms $J_1, \ldots, J_n$.

For example, consider the following rule set, which constitutes a simultaneous inductive definition of the judgement forms $a$ even, stating that $a$ is an even natural number, and $a$ odd, stating that $a$ is an odd natural number:

$$\frac{}{\texttt{zero}\ \text{even}} \tag{1.12a}$$

$$\frac{a\ \text{odd}}{\texttt{succ}(a)\ \text{even}} \tag{1.12b}$$

$$\frac{a\ \text{even}}{\texttt{succ}(a)\ \text{odd}} \tag{1.12c}$$

Simultaneous rule induction for this set of rules states that if we wish to show $P(a)$, whenever $a$ even, and $Q(a)$, whenever $a$ odd, it is enough to show

1. $P(\texttt{zero})$;
2. if $Q(a)$, then $P(\texttt{succ}(a))$;
3. if $P(a)$, then $Q(\texttt{succ}(a))$.

These proof obligations are derived by considering the rules defining the even and odd judgement forms.

## 1.6 Defining Functions by Rules

Another common use of inductive definitions is to define inductively its graph, which we then prove is a function. For example, one way to define the addition function on natural numbers is to define inductively the judgement $\mathsf{sum}(a, b, c)$, with the intended meaning that $c$ is the sum of $a$ and $b$, as follows:

$$\frac{b\ \mathsf{nat}}{\mathsf{sum}(\mathtt{zero}, b, b)} \tag{1.13a}$$

$$\frac{\mathsf{sum}(a, b, c)}{\mathsf{sum}(\mathtt{succ}(a), b, \mathtt{succ}(c))} \tag{1.13b}$$

We then must show that $c$ is uniquely determined as a function of $a$ and $b$. That is, we show that *for every* $a$ nat and $b$ nat, *there exists a unique* $c$ nat such that $\mathsf{sum}(a, b, c)$. This breaks down into two proof obligations:

1. (Existence) If $a$ nat and $b$ nat, then there exists $c$ nat such that $\mathsf{sum}(a, b, c)$.
2. (Uniqueness) If $a$ nat, $b$ nat, $c$ nat, $c'$ nat, $\mathsf{sum}(a, b, c)$, and $\mathsf{sum}(a, b, c')$, then $c = c'$ nat.

We give the proof of existence here, and leave the proof of uniqueness as an exercise. We proceed by induction on the rules defining $a$ nat. Let $P(a)$ be the proposition "if $b$ nat then there exists $c$ nat such that $\mathsf{sum}(a, b, c)$." We have two cases two consider:

**Rule** (1.2a) We are to show $P(\mathtt{zero})$. Assuming $b$ nat and taking $c$ to be $b$, we obtain $\mathsf{sum}(\mathtt{zero}, b, c)$ by Rule (1.13a).

**Rule** (1.2b) Assuming $P(a)$, we are to show $P(\mathtt{succ}(a))$. We proceed by an inner induction on the rules defining $b$ nat, which in this case amounts to a case analysis on the form of $b$.

- **Rule** (1.2a) We are to show $\mathsf{sum}(\mathtt{succ}(a), \mathtt{zero}, c)$ for some $c$. By the outer induction we know that $\mathsf{sum}(a, \mathtt{zero}, c')$ for some $c'$ nat, so, taking $c$ to be $\mathtt{succ}(c')$, we obtain $\mathsf{sum}(\mathtt{succ}(a), \mathtt{zero}, c)$ by Rule (1.13b).
- **Rule** (1.2b) Assuming $\mathsf{sum}(\mathtt{succ}(a), b, c')$ for some $c'$ nat, show that $\mathsf{sum}(\mathtt{succ}(a), \mathtt{succ}(b), c)$ for some $c$ nat. By the outer inductive hypothesis, there exists $c'$ nat such that $\mathsf{sum}(a, \mathtt{succ}(b), c')$, hence by Rule (1.13b), $\mathsf{sum}(\mathtt{succ}(a), \mathtt{succ}(b), \mathtt{succ}(c'))$.

As another example, the following rules define the height of a binary tree, expressed as the judgement $\mathsf{hgt}(a, b)$. The definition uses an auxiliary judgement, $\mathsf{max}(a, b, c)$, stating that $c$ nat is the larger of $a$ nat and $b$ nat.

$$\frac{}{\mathsf{hgt}(\mathtt{empty}, \mathtt{zero})} \tag{1.14a}$$

$$\frac{\mathsf{hgt}(a_1, b_1) \quad \mathsf{hgt}(a_2, b_2) \quad \mathsf{max}(b_1, b_2, b)}{\mathsf{hgt}(\mathtt{node}(a_1, a_2), \mathtt{succ}(b))} \tag{1.14b}$$

It is easy to prove by tree induction that this judgement has mode $(\forall, \exists)$ with inputs and outputs being binary trees.

The intended mode of a judgement is often indicated by the notation we use to express it. For example, when giving an inductive definition of a function, we often use equations to indicate the intended input and output relationships. For example, we may re-state the inductive definition of addition (1.13) using equations.

$$\frac{a \ \mathsf{nat}}{a + \mathtt{zero} = a \ \mathsf{nat}} \tag{1.15a}$$

$$\frac{a + b = c \ \mathsf{nat}}{a + \mathtt{succ}(b) = \mathtt{succ}(c) \ \mathsf{nat}} \tag{1.15b}$$

When using this notation we tacitly incur the obligation to prove that the mode of the judgement is such that the object on the right-hand side of the equations is determined as a function of those on the left. Having done so, we abuse the notation by using the relation as function, writing just $a + b$ for the unique $c$ such that $a + b = c$ nat.

## 1.7 Exercises

1. Give an inductive definition of the judgement "$\nabla$ is a derivation of $J$" for an arbitrary inductively defined judgement form $J$.

2. Give an inductive definition of the forward-chaining and backward-chaining search strategies.

# Chapter 2

# Hypothetical Judgements

A *categorical* judgement is an unconditional assertion about some object of the universe. The inductively defined judgements given in Chapter 1 are all categorical. In contrast, a *hypothetical judgement* is made on the basis of one or more *hypotheses*, or *assumptions*, that entail a *conclusion*. We will consider two forms of hypothetical judgement, the *derivability* judgement and the *admissibility* judgement, which are both defined relative to some fixed set of rules. These two forms of hypothetical judgement share a common set of *structural properties* that characterize the concept of reasoning under hypotheses.

## 2.1 Derivability

For a given set of rules defining a collection of categorical judgements, we define the *derivability* judgement, written $J \vdash K$, where $J$ and $K$ are categorical judgements, to mean that we may derive the judgement $K$ from the extension of our rule set with $J$ as a new axiom (*i.e.*, a rule without premises having $J$ as conclusion). The assertion $J$ is called the *hypothesis*, and $K$ the *conclusion*, of the hypothetical judgement.

The hypothetical judgement is naturally extended to permit $K$ to be hypothetical to obtain the *iterated* form

$$J_1 \vdash J_2 \vdash \ldots J_n \vdash K, \tag{2.1}$$

which we abbreviate to

$$J_1, \ldots, J_n \vdash K. \tag{2.2}$$

We often use $\Gamma$ to stand for a finite sequence of assertions, writing $\Gamma \vdash K$ to mean that $K$ is derivable from the judgements $\Gamma$.

There is a close correspondence between inference rules and derivability judgements. Each inference rule defining a judgement form gives rise to a valid derivability judgement. For if

$$\frac{J_1 \quad \ldots \quad J_n}{J} \tag{2.3}$$

is a primitive rule, then the judgement $J_1, \ldots, J_n \vdash J$ is valid, since adding the hypotheses as axioms enables us to apply the displayed rule to derive the conclusion of that rule. Conversely, if $J_1, \ldots, J_n \vdash J$ is valid, then there is a derivation of $J$ obtained by composing rules starting with the hypotheses $J_i$ as axioms. Equivalently, we say that the inference rule (2.3) is *derivable* iff $J_1, \ldots, J_n \vdash J$. The derivation of $J$ is essentially a compound inference rule with the $J_i$'s as premises and $J$ as conclusion.

For example, the derivability judgement

$$a \ \texttt{nat} \vdash \texttt{succ(succ(}a\texttt{))} \ \texttt{nat} \tag{2.4}$$

is valid according to Rules (1.2). For if we regard the premise $a$ nat as a new axiom, then we may derive succ(succ($a$)) nat from it according to those rules, as follows:

$$\dfrac{\dfrac{\overline{a \ \texttt{nat}}}{\texttt{succ(}a\texttt{)} \ \texttt{nat}}}{\texttt{succ(succ(}a\texttt{))} \ \texttt{nat}} \tag{2.5}$$

This derivation consists of a composition of Rules (1.2), starting with $a$ nat as an axiom and ending with succ(succ($a$)) nat as conclusion. In other words, the rule

$$\frac{a \ \texttt{nat}}{\texttt{succ(succ(}a\texttt{))} \ \texttt{nat}} \tag{2.6}$$

is derivable.

It is interesting to observe that the derivability of this rule is entirely independent of the choice of the object $a$. In particular, we may choose $a$ to be some rubbish object, say junk, and observe that

$$\texttt{junk} \ \texttt{nat} \vdash \texttt{succ(succ(junk))} \ \texttt{nat} \tag{2.7}$$

is valid. For if we treat junk nat as a new axiom, then surely we can derive succ(succ(junk)) nat by using the rules defining the natural numbers, even though we cannot derive junk nat from these rules.

Because evidence for a derivability judgement consists of a derivation from axioms, certain *structural properties* follow, independently of the rule set under consideration.

**Reflexivity** Every judgement is a consequence of itself: $\Gamma, J \vdash J$. The conclusion is justified because it is regarded as an axiom.

**Weakening** If $\Gamma \vdash J$, then $\Gamma, K \vdash J$. The derivation of $J$ makes use of the rules and the premises $\Gamma$, and is not affected by the (unexercised) option to use $K$ as an axiom.

**Exchange** If $\Gamma_1, J_1, J_2, \Gamma_2 \vdash J$, then $\Gamma_1, J_2, J_1, \Gamma_2 \vdash J$. The relative ordering of the axioms is immaterial.

**Contraction** If $\Gamma, J, J \vdash K$, then $\Gamma, J \vdash K$. Since we can use any assumption any number of times, stating it more than once is the same as stating it once.

**Transitivity** If $\Gamma, K \vdash J$ and $\Gamma \vdash K$, then $\Gamma \vdash J$. If we replace an axiom by a derivation of it, the result remains a derivation of its conclusion.

The weakening and exchange properties together imply that a finite sequence of hypotheses $\Gamma$ may just as well be regarded as a finite set, since set membership is not affected by duplication of elements or by the order in which elemets are specified. In most cases we treat the hypotheses of an iterated hypothetical judgement as a finite set, which amounts to the tacit use of the exchange and contraction properties of the hypothetical judgement.

Derivability is a relatively strong condition that is stable under extension of the set of rules defining a judgement. That is, if a rule is derivable from one set of rules, it remains derivable from any extension of that set of rules. The existence of a derivation depends only on what rules are available, and not on which rules are absent. Another characterization of derivability is explored in Exercise 1 on page 21.

## 2.2 Admissibility

The *admissibility* judgement, written $J \models K$, is a weaker form of hypothetical judgement whose meaning is that $K$ is derivable from the given set of rules whenever $J$ is derivable from the same set of rules. Equivalently, the admissibility judgement is a simple conditional assertion stating that *if* $J$ is derivable from the rules, *then* so is $K$. As with derivability, we may iterate the admissibility judgement, writing $J_1, \ldots, J_n \models K$ to mean that *if* $J_1$ is derivable *and* ... *and* $J_n$ is derivable, *then* $K$ is derivable. Equivalently, we

say that the rule

$$\frac{J_1 \quad \ldots \quad J_n}{J} \tag{2.8}$$

is *admissible* iff $J_1, \ldots, J_n \models J$.

For example, for an abritrary object $a$, the admissibility judgement

$$\texttt{succ}(a)\ \texttt{nat} \models a\ \texttt{nat} \tag{2.9}$$

is valid with respect to Rules (1.2). This may be proved by rule induction, for if $\texttt{succ}(a)\ \texttt{nat}$, then this can only be by virtue of Rule (1.2b). But then the desired conclusion must hold, since it is the premise of the inference. Equivalently, we may say that the rule

$$\frac{\texttt{succ}(a)\ \texttt{nat}}{a\ \texttt{nat}} \tag{2.10}$$

is admissible.

Admissibility is, in general, strictly weaker than derivability: if $J_1, \ldots, J_n \vdash J$ is valid, then so is $J_1, \ldots, J_n \models J$, but the converse need not be the case. To see why the implication left to right holds, assume that $J_1, \ldots, J_n \vdash J$. To show $J_1, \ldots, J_n \models J$, assume further that each $J_i$ is derivable from the original rules, which is to say that $\vdash J_1$, $\ldots$, $\vdash J_n$ are all valid derivability judgements with no hypotheses. But then by weakening and transitivity it follows that $\vdash K$, which means that $K$ is derivable in the original set of rules. On the other hand, we have already seen that $\texttt{succ}(a)\ \texttt{nat} \models a\ \texttt{nat}$, but

$$\texttt{succ}(a)\ \texttt{nat} \not\vdash a\ \texttt{nat}. \tag{2.11}$$

That is, there is no way to compose rules starting with $\texttt{succ}(a)\ \texttt{nat}$ and end up with $a\ \texttt{nat}$. To see this, take $a = \texttt{junk}$ and observe that, even with $\texttt{succ(junk)}\ \texttt{nat}$ as a new axiom, there is no way to derive $\texttt{junk}\ \texttt{nat}$.

The admissibility judgement enjoys the same structural properties as derivability.

**Reflexivity** If $J$ is derivable from the original rules, then $J$ is derivable from the original rules: $J \models J$.

**Weakening** If $J$ is derivable from the original rules assuming that each of the judgements in $\Gamma$ are derivable from these rules, then $J$ must also be derivable assuming that $\Gamma$ and also $K$ are derivable from the original rules: if $\Gamma \models J$, then $\Gamma, K \models J$.

**Exchange** The order of assumptions in an iterated implication does not matter.

**Contraction** Assuming the something twice is the same as assuming it once.

**Transitivity** If $\Gamma, K \models J$ and $\Gamma \models K$, then $\Gamma \models J$. If the assumption $K$ is used, then we may instead appeal to the assumed derivability of $K$.

As with derivability we often make tacit use of exchange and contraction by stating the iterated form using finite sets, rather than sequences, of assumptions.

In contrast to derivability, admissibility is not stable under expansion of the rule set. For example, suppose we expanded Rules (1.2) with the following (fanciful) rule:

$$\frac{}{\texttt{succ(junk) nat}} \tag{2.12}$$

But relative to this expanded rule set, `succ(`$a$`) nat` $\not\models$ $a$ `nat`, even though it was valid with respect to the original. For if the premise were derived using the additional rule, there would be no derivation of `junk nat`, so the conclusion fails. In other words, admissibility is sensitive to which rules are *absent* from, as well as to which rules are *present* in, an inductive definition. At bottom a proof of admissibility amounts to an exhaustive analysis of the possible ways of deriving the premises, showing in each case that the conclusion is derivable.

Another way to compare derivability to admissibility is to note that whereas an admissibility judgement may be *vacuously* true (because the hypothesis is not derivable), a derivability judgement *never* holds vacuously (because it adds the hypothesis to the rules as a new axiom). Thus, relative to Rules (1.2), the admissibility judgement `junk nat` $\models$ `succ(junk) nat` is vacuously valid, because the hypothesis is not derivable according to those rules (as may be seen by a simple rule induction). The corresponding derivability judgement `junk nat` $\vdash$ `succ(junk) nat` is also valid, but not vacuously so! Rather, the conclusion holds because we can apply Rule (1.2b) to the hypothesis to obtain the conclusion.

## 2.3 Derivability Judgements in Rules

Inference rules, as defined in Chapter 1, are limited to categorical judgements. It is often useful to extend the concept of a rule to permit derivability judgements as a premise or the conclusion of a rule. To illustrate this,

let us consider the formalization of some elementary principles of logic using inference rules. We will consider the categorical judgment form $\phi$ true, where $\phi$ is a proposition involving the familiar connectives such as conjunction and implication.[1]

Suppose that $\phi$ has the form of a conjunction, $\phi_1 \wedge \phi_2$, of $\phi_1$ and $\phi_2$. In this case to show that $\phi$ true, it is enough to show $\phi_1$ true and $\phi_2$ true. Conversely, if we know $\phi$ true, then we know $\phi_1$ true and $\phi_2$ true. Thus we have the following two rules for conjunction:

$$\frac{\phi_1 \text{ true} \quad \phi_2 \text{ true}}{\phi_1 \wedge \phi_2 \text{ true}} \tag{2.13a}$$

$$\frac{\phi_1 \wedge \phi_2 \text{ true}}{\phi_1 \text{ true}} \tag{2.13b}$$

$$\frac{\phi_1 \wedge \phi_2 \text{ true}}{\phi_2 \text{ true}} \tag{2.13c}$$

These rules fit into the framework of Chapter 1, and present no serious challenges.

Now suppose that $\phi$ has the form of an implication, $\phi_1 \supset \phi_2$. To show that $\phi$ true, we temporarily assume that $\phi_1$ true and deduce from this that $\phi_2$ true. Conversely, if we know that $\phi_1 \supset \phi_2$ true, then we know that it is possible to deduce $\phi_2$ true from $\phi_1$ true. This leads to the following two rules:

$$\frac{\phi_1 \text{ true} \vdash \phi_2 \text{ true}}{\phi_1 \supset \phi_2 \text{ true}} \tag{2.14a}$$

$$\frac{\phi_1 \supset \phi_2 \text{ true}}{\phi_1 \text{ true} \vdash \phi_2 \text{ true}} \tag{2.14b}$$

These rules illustrate the use of derivability in both the premises and the conclusion of a rule.

Allowing a derivability in the conclusion of a rule presents no serious complications, since we may regard a rule of the form

$$\frac{J_1 \quad \ldots \quad J_n}{K_1, \ldots, K_m \vdash K} \tag{2.15}$$

as alternate notation for the rule

$$\frac{J_1 \quad \ldots \quad J_n \quad K_1 \quad \ldots \quad K_m}{K}, \tag{2.16}$$

[1] For a more thorough account of the rules of logic and how they relate to programming languages, please see Chapters 29 and 30.

which expresses the same thing, namely that we may derive $K$ from derivations of $J_1, \ldots, J_n, K_1, \ldots, K_m$. Thus, Rule (2.14b) may be written instead in the form

$$\frac{\phi_1 \supset \phi_2 \text{ true} \quad \phi_1 \text{ true}}{\phi_2 \text{ true}} \tag{2.17}$$

Rule (2.14a) makes use of a derivability judgement in its premise. How are we to make sense of this? Reading the premise as a derivability judgement, the rule states that *if* we are able to derive $\phi_2$ true in the extension of the "current" rule set with the axiom $\phi_1$ true, *then* we may derive $\phi_1 \supset \phi_2$ true in the "current" rule set. The "current" rule set contains the original rules, augmented by any additional axioms that have been added during the deduction. For example, if $\phi_2$ is itself an implication, then a derivation of its truth would further extend the rule set with the antecedent of the implication in order to deduce the consequent.

This captures the right intuition, but there is a technical snag lurking in the notation. The semantics of the derivability judgement given in Section 2.1 on page 13 is defined for a *previously given* inductive definition, whereas we wish to consider rules such as Rule (2.14a) as *part of the definition itself*. To account for this we must generalize the concept of a rule.

## 2.4 Generalized Rules

To justify the use of the derivability judgements in the premise of a rule, we will reduce the general case, which admits this possibility, to the special case considered in Chapter 1, which does not. Suppose we have a collection of *generalized rules* of the form

$$\frac{\Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma_n \vdash J_n}{J}\ . \tag{2.18}$$

The hypotheses $\Gamma_i$ are the *local hypotheses* of the $i$th premise of the inference. The meaning of such a rule is that $J$ is derivable iff each of the $J_i$'s are derivable from the extension of those rules with the local hypotheses $\Gamma_i$ as axioms.

To capture the idea of the "current" rule set, a generalized rule of the form (2.18) is understood to apply in any extension of the rules with a set of hypotheses, called the *global*, or *ambient*, hypotheses of the inference. This may be made explicit by writing Rule (2.18) in the following form:

$$\frac{\Gamma\,\Gamma_1 \vdash J_1 \quad \ldots \quad \Gamma\,\Gamma_n \vdash J_n}{\Gamma \vdash J}\ , \tag{2.19}$$

The global hypotheses, $\Gamma$, may be chosen arbitrarily to reflect any extensions to the rule set that may be in effect at the point of application of the rule. The conclusion is stated relative to global hypotheses, $\Gamma$, which are further augmented by the local hypotheses in the derivation of each of the premises of the rule.

By making the global and local hypotheses explicit we enable an alternate reading of a collection of generalized rules as a simultaneous inductive definition of an infinite family of judgements indexed by finite hypothesis sets. That is, we regard the judgement $\Gamma \vdash J$ as the "$\Gamma$th instance" of an infinite family of judgements indexed by hypothesis sets. From this point of view the generalized rule (2.19) is not generalized at all, but is rather just an ordinary inference rule defining a $\Gamma$-indexed family of categorical judgements.

For this to capture the intended interpretation of a generalized rule, we must ensure that the members of the family "fit together" properly so that $\Gamma \vdash J$ behaves like a hypothetical judgement. That is, we must ensure that the family obey the structural properties of a hypothetical judgement stated in Section 2.1 on page 13. This may be achieved by implicitly including the following *structural rules* along with any set of generalized rules.

$$\frac{}{\Gamma, J \vdash J} \tag{2.20a}$$

$$\frac{\Gamma \vdash J}{\Gamma, K \vdash J} \tag{2.20b}$$

$$\frac{\Gamma_1, J_2, J_2, \Gamma_2 \vdash J}{\Gamma_1, J_1, J_2, \Gamma_2 \vdash J} \tag{2.20c}$$

$$\frac{\Gamma, J, J \vdash K}{\Gamma, J \vdash K} \tag{2.20d}$$

$$\frac{\Gamma \vdash K \quad \Gamma, K \vdash J}{\Gamma \vdash J} \tag{2.20e}$$

In practice we avoid the need to include Rules (2.20c) and (2.20d) by treating $\Gamma$ as a finite *set* of hypotheses. Moreover, by stating the rules in explicit form (2.19), we also ensure that Rule (2.20b) is admissible. This leaves only the structural rules (2.20a), which we include as part of a generalized inductive definition, and (2.20e), which we usually prove to be admissible.

The principle of rule induction extends to generalized inductive definitions as well. To state the induction principle it is necessary to make explicit

the subjects of the judgements in a generalized rule:

$$\frac{\Gamma\,\Gamma_1 \vdash a_1\ J_1 \quad \ldots \quad \Gamma\,\Gamma_n \vdash a_n\ J_n}{\Gamma \vdash a\ J}\ . \tag{2.21}$$

Let $P_\Gamma$ be a family of predicates indexed by hypothesis sets $\Gamma$. To show that $P_\Gamma(a)$ whenever $\Gamma \vdash a\ J$, it is enough to satisfy these requirements:

1. To satisfy the (implied) structural rule of reflexivity, we require that $P_{\Gamma,a\ J}(a)$. That is, if $a\ J$ is a hypothesis in $\Gamma$, then we must have $P_\Gamma(a)$.

2. To satisfy a rule of the form (2.21), it is enough to show that $P_\Gamma(a)$ whenever $P_{\Gamma\,\Gamma_1}(a_1), \ldots, P_{\Gamma\,\Gamma_n}(a_n)$.

These requirements presume that the structural rules, other than reflexivity, are admissible. If not, we must also show that the family, $P$, is closed under each of the inadmissible structural rules.

## 2.5 Exercises

1. Define $\Gamma' \vdash \Gamma$ to mean that $\Gamma' \vdash J_i$ for each $J_i$ in $\Gamma$. Show that $\Gamma \vdash J$ iff whenever $\Gamma' \vdash \Gamma$, it follows that $\Gamma' \vdash J$. For the implication right-to-left, take $\Gamma' = \Gamma$. For the implication left-to-right, repeatedly appeal to transitivity to obtain the desired conclusion.

2. Show that it is possible to make sense of admissibility judgements in the conclusions (only!) of inference rules by an analysis reminiscent of that used to justify derivability judgements in the conclusion. Hint: make use of the interpretation of generalized rules as a simultaneous inductive definition of a family of judgement forms described in Section 2.3 on page 17.

# Chapter 3

# Syntactic Objects

We will make use of two sorts of objects for representing syntax, *strings* and *abstract syntax trees*. Strings provide a convenient linear representation useful primarily for human interaction, but are not suitable for manipulation and analysis. Abstract syntax trees expose the hierarchical structure of syntax, and are much more suitable than strings for analysis and mechanization.

## 3.1 Strings

An *alphabet* is a (finite or infinite) collection of *symbols*. An alphabet is specified by a set of judgements of the form $c_1\ \mathsf{sym}, \ldots, c_n\ \mathsf{sym}$, which we collectively designate by $\Sigma$. The judgement $s\ \mathsf{str}$ is inductively defined by the following rules:

$$\frac{}{\varepsilon\ \mathsf{str}} \tag{3.1a}$$

$$\frac{c\ \mathsf{sym} \quad s\ \mathsf{str}}{c \cdot s\ \mathsf{str}} \tag{3.1b}$$

Thus a string is essentially a list of characters, with the null string being the empty list.

This definition makes use of the judgement $c\ \mathsf{sym}$, which specifies the alphabet over which the strings are defined. To specify the alphabet, we consider hypothetical judgements of the form

$$c_1\ \mathsf{sym}, \ldots, c_n\ \mathsf{sym} \vdash s\ \mathsf{str},$$

which we abbreviate by writing $\Sigma \vdash s\ \mathsf{str}$. We shall have occasion to work with strings over various alphabets, usually leaving the precise specifica-

tion of the symbols implicit. We use the judgement $c$ char to state that $c$ is a character of some unspecified standard alphabet, such as the ASCII or Unicode character set, and write $s$ $\mathsf{str}_{\mathsf{char}}$ to state that $s$ is a string of symbols $c$ such that $c$ char.

When specialized to Rules (3.1), the principle of rule induction states that to show $P(s)$ holds whenever $s$ str, it is enough to show

1. $P(\varepsilon)$, and
2. if $P(s)$ and $c$ sym, then $P(c \cdot s)$.

This is sometimes called the principle of *string induction*. It is essentially equivalent to induction over the length of a string, except that there is no need to define the length of a string in order to use it.

The following rules constitute an inductive definition of the judgement $s_1 \hat{} s_2 = s$ str, stating that $s$ is the result of concatenating the strings $s_1$ and $s_2$.

$$\frac{}{\varepsilon \hat{} s = s \text{ str}} \tag{3.2a}$$

$$\frac{s_1 \hat{} s_2 = s \text{ str}}{(c \cdot s_1) \hat{} s_2 = c \cdot s \text{ str}} \tag{3.2b}$$

It is easy to prove by string induction on the first argument that this judgement has mode $(\forall, \forall, \exists!)$. Thus, it determines a total function of its first two arguments.

Strings are usually written as juxtapositions of characters, writing just *abcd* for the four-letter string $a \cdot (b \cdot (c \cdot (d \cdot \varepsilon)))$. Concatentation is also written as juxtaposition, and individual characters are often identified with the corresponding unit-length string. This means that *abcd* can be thought of in many ways, for example as the concatenations *ab cd*, *a bcd*, or *abc d*, or even $\varepsilon$ *abcd* or *abcd* $\varepsilon$, as may be convenient in a given situation.

## 3.2 Names

Names arise frequently in the study of programming languages. They are used as variables, as labels of fields in data structures, as targets of branches, and so forth. The "spelling" of a name is of no intrinsic significance, but serves only to distinguish one name from another. Consequently, we shall treat names as atoms, abstracting away any structure other than their identity. We assume given a judgement $x$ name that holds for infinitely

many objects $x$, and a judgement $x \# y$, where $x$ name and $y$ name, stating that $x$ and $y$ are distinct names. We further assume that names are distinguishable from all other forms of syntactic object, so that there is no ambiguity in their use.

The judgement $[x \leftrightarrow y]z = z'$ name, which swaps names $x$ with $y$ and leaves all other names fixed, is inductively defined by the following rules.

$$\frac{}{[x \leftrightarrow y]x = y \text{ name}} \tag{3.3a}$$

$$\frac{}{[x \leftrightarrow y]y = x \text{ name}} \tag{3.3b}$$

$$\frac{x \# z \text{ name} \quad y \# z \text{ name}}{[x \leftrightarrow y]z = z \text{ name}} \tag{3.3c}$$

This judgement has mode $(\forall, \forall, \forall, \exists!)$, which means that the fourth argument is determined as a function of the first three.

## 3.3 Abtract Syntax Trees

An *abstract syntax tree*, or *ast*, is an ordered tree whose nodes are labelled an *operator*. Each operator is assigned an *arity*, which determines the number of children of any node labelled with that operator. The assignment of arities to operators is specified by an *operator signature*, $\Omega$, which is a finite set of judgements of the form $\mathsf{ar}(o_1) = n_1, \ldots, \mathsf{ar}(o_k) = n_k$, where $n$ nat, $o_i$ name for each $1 \leq i \leq k$, and $o_i \# o_j$ whenever $i \neq j$.

For a given operator signature, $\Omega$, the judgement $a$ ast is inductively defined by the following rule:

$$\frac{\mathsf{ar}(o) = n \qquad a_1 \text{ ast} \quad \ldots \quad a_n \text{ ast}}{o(a_1, \ldots, a_n) \text{ ast}} \tag{3.4a}$$

When $\Omega \vdash \mathsf{ar}(o) = \mathtt{zero}$, this rule has no premises, and hence a node labelled with $o$ is a leaf of the abstract syntax tree.

### 3.3.1 Variables and Substitution

We shall frequently consider ast's involving occurrences of one or more *variables* standing for a fixed, but unspecified, ast. Variables are represented by names, and are given meaning by substitution. To avoid confusion, we

assume that variable names are disjoint from operator names. The hypothetical judgment

$$x_1 \text{ ast}, \ldots, x_n \text{ ast} \vdash a \text{ ast}, \tag{3.5}$$

where $x_1, \ldots, x_n$ are pairwise distinct names, states that $a$ is an ast possibly involving the variables $x_1, \ldots, x_n$. A set of hypotheses of this form is sometimes called a *variable context*; the meta-variable $\mathcal{X}$ stands for an arbitrary (possibly empty) variable context. We write $x \# \mathcal{X}$ to indicate that $x \# x_i$ for each $x_i$ such that $\mathcal{X} \vdash x_i$ ast.

Substitution is defined by the judgement $[a/x]b = c$, which states that $c$ is the result of substituting the ast $a$ for all occurrences of the name $x$ in $b$. This judgement is inductively defined by the following rules:

$$\frac{}{[a/x]x = a} \tag{3.6a}$$

$$\frac{x \# y}{[a/x]y = y} \tag{3.6b}$$

$$\frac{[a/x]b_1 = c_1 \quad \ldots \quad [a/x]b_n = c_n}{[a/x]o(b_1, \ldots, b_n) = o(c_1, \ldots, c_n)} \tag{3.6c}$$

### 3.3.2 Structural Induction

*Structural induction* is the principle of rule induction applied to Rules (3.4) for any fixed operator signature $\Omega$. Specifically, to show $P(a)$ whenever $a$ ast, it is enough to show that, for each operator $o$ such that $\Omega \vdash \mathsf{ar}(o) = n$, if $P(a_1)$, ..., $P(a_n)$, then $P(o(a_1, \ldots, a_n))$. The base cases of the induction correspond to the operators $o$ for which $\Omega \vdash \mathsf{ar}(o) = \mathtt{zero}$.

Structural induction extends naturally to abstract syntax trees with variables by parameterizing with respect to the variable context, $\mathcal{X}$. Specifically, to show that $P_{\mathcal{X}}(a)$ whenever $\mathcal{X} \vdash a$ ast, it is enough to show the following:

1. $P_{\mathcal{X}}(x)$ whenever $\mathcal{X} \vdash x$ ast.
2. whenever $\Omega \vdash \mathsf{ar}(o) = n$, if $P_{\mathcal{X}}(a_1), \ldots, P_{\mathcal{X}}(a_n)$, then $P_{\mathcal{X}}(o(a_1 \ldots, a_n))$.

The subscript $\mathcal{X}$ on the property $P$ serves only to indicate that we are considering the property only for ast's over the variables declared in $\mathcal{X}$.

As an example, we may prove by structural induction that the substitution judgement has mode $(\forall, \forall, \forall, \exists!)$ over abstract syntax trees and names.

**Theorem 3.1.** *If* $\mathcal{X}, x$ ast $\vdash a$ ast, *where* $x \# \mathcal{X}$, *and* $\mathcal{X} \vdash b$ ast, *then there exists a unique* $c$ *such that* $\mathcal{X} \vdash c$ ast *and* $[a/x]b = c$.

*Proof.* We show that if $\mathcal{X}, x$ ast $\vdash b$ ast, then whenever $\mathcal{X} \vdash a$ ast, there exists a unique $c$ such that $\mathcal{X} \vdash c$ ast and $[a/x]b = c$.

We have two cases to consider, the first of which decomposes into two sub-cases.

1. Names $y$ such that $\mathcal{X}, x$ ast $\vdash y$ ast:
   (a) If $y = x$, then by Rule (3.6a) we have $[a/x]x = a$, and no other rule applies.
   (b) If $y \# x$, then by Rule (3.6b) we have $[a/x]y = y$, and no other rule applies.
2. Operators $o$ such that $\Omega \vdash \mathsf{ar}(o) = n$: suppose that for each $1 \leq i \leq n$ there exists a unique $c_i$ such that $[a/x]a_i = c_i$. Then by Rule (3.6c) we may take $c = o(c_1, \ldots, c_n)$, and no other rule applies.

□

Since $c$ is determined by $a$, $x$, and $b$, we often write $[a/x]b$ for the unique $c$ such that $[a/x]b = c$.

## 3.4 Abstract Binding Trees

Abstract syntax trees express the hierarchical structure of syntax, providing the foundation for the very useful principle of proof by structural induction. Another commonality among languages is the concept of *binding* a name within a *scope*, the range of significance of the binding. *Abstract binding trees*, or *abt's*, extend abstract syntax trees with an additional construct, called an *abstractor*, for binding one or more names in another abt.

To support operators that bind variables in certain argument positions, we generalize the arities introduced in Section 3.3 on page 25. Instead of being simply a number, the *arity* of an operator is now a finite sequence of natural numbers, one per argument position of the operator, specifying the *valence* of that argument. The valence of an argument is, in turn, a natural number specifying the number of names bound in that argument. This generalizes the system of arities in Section 3.3 on page 25 in that the arity $(0, 0, \ldots, 0)$ of length $k$ is the arity of a $k$-argument operator binding no variables in any argument position. A *signature* is a finite set of judgements of

the form $\mathsf{ar}(o) = (n_1, \ldots, n_k)$, where $o$ name, specifying the arity of operator $o$. We assume that no operator name is assigned more than one arity in a given signature.

The definition of abstract binding trees is given by a set of generalized rules involving hypothetical judgements of the form $\mathcal{X} \vdash a\ \mathsf{abt}^n$, where $n$ nat and $\mathcal{X}$ is a variable context. This judgement states that $a$ is an abt of valence $n$ possibly involving the variables declared in $\mathcal{X}$. The variable context, $\mathcal{X}$, has the form $x_1\ \mathsf{abt}^0, \ldots, x_n\ \mathsf{abt}^0$ in which $x_i$ name for each $1 \leq i \leq n$ and $x_i \# x_j$ whenever $i \neq j$.

$$\frac{}{\mathcal{X}, x\ \mathsf{abt}^0 \vdash x\ \mathsf{abt}^0} \tag{3.7a}$$

$$\frac{\mathsf{ar}(o) = (n_1, \ldots, n_k) \qquad \mathcal{X} \vdash a_1\ \mathsf{abt}^{n_1} \quad \ldots \quad \mathcal{X} \vdash a_k\ \mathsf{abt}^{n_k}}{\mathcal{X} \vdash o(a_1, \ldots, a_k)\ \mathsf{abt}^0} \tag{3.7b}$$

$$\frac{x \# \mathcal{X} \quad \mathcal{X}, x\ \mathsf{abt}^0 \vdash a\ \mathsf{abt}^n}{\mathcal{X} \vdash x.a\ \mathsf{abt}^{n+1}} \tag{3.7c}$$

If an abt has valence $n$, then it has the form $x_1.x_2.\ldots.x_n.a$, where $a$ is not an abstractor and no variable is declared more than once. We usually abbreviate this to just $x_1, \ldots, x_n.a$, omitting the prefix entirely when $n$ is zero. We sometimes write $a$ abt to mean $a\ \mathsf{abt}^0$.

**Lemma 3.2.** *There is at most one $n$ such that $\mathcal{X} \vdash a\ \mathsf{abt}^n$, and if $\mathcal{X} \vdash a\ \mathsf{abt}^n$, then there is a least $\mathcal{X}_0 \subseteq \mathcal{X}$ such that $\mathcal{X}_0 \vdash a\ \mathsf{abt}^n$.*

### 3.4.1 Structural Induction

The principle of structural induction introduced in Section 3.3 on page 25 extends to abstract binding trees. Let $P^n_{\mathcal{X}}$ be a family of predicates indexed by a variable context, $\mathcal{X}$, and a valence, $n$. To show that $P^n_{\mathcal{X}}(a)$ whenever $\mathcal{X} \vdash a\ \mathsf{abt}^n$, it is enough to show the following facts:

1. $P^0_{\mathcal{X}, x\ \mathsf{abt}^0}(x)$.
2. For each operator, $o$, of arity $(m_1, \ldots, m_k)$, if $P^{m_1}_{\mathcal{X}}(a_1)$ and $\ldots$ and $P^{m_k}_{\mathcal{X}}(a_k)$, then $P^0_{\mathcal{X}}(o(a_1, \ldots, a_k))$.
3. For some and any $x$ name such that $x \# \mathcal{X}$, if $P^n_{\mathcal{X}, x\ \mathsf{abt}^0}(a)$, then $P^{n+1}_{\mathcal{X}}(x.a)$.

The last clause requires that the bound variable, $x$, be chosen "fresh" in the sense of being apart from the active variables, $\mathcal{X}$, and, moreover, that the proof be independent of the exact choice of $x$, provided that it satisfies this freshness condition.

### 3.4.2 Apartness and Name Swapping

The relation of a variable, $x$ *lying apart* from an abt, $a$, is fundamental. Informally, this judgement means that the abt $a$ does not involve the variable $x$ except possibly as a bound variable. The apartness judgement, $x \# a\ \mathsf{abt}^n$, is inductively defined by the following rules.

$$\frac{x \# y\ \mathsf{name}}{x \# y\ \mathsf{abt}^0} \tag{3.8a}$$

$$\frac{x \# a_1\ \mathsf{abt}^{n_1} \quad \ldots \quad x \# a_k\ \mathsf{abt}^{n_k}}{x \# o(a_1, \ldots, a_k)\ \mathsf{abt}^0} \tag{3.8b}$$

$$\frac{}{x \# x\,.\,a\ \mathsf{abt}^{n+1}} \tag{3.8c}$$

$$\frac{x \# y\ \mathsf{name} \quad x \# a\ \mathsf{abt}^n}{x \# y\,.\,a\ \mathsf{abt}^{n+1}} \tag{3.8d}$$

We say that a variable, $x$, *lies within*, or *is free in*, an abt, $a$, written $x \in a\ \mathsf{abt}$, iff it is *not* the case that $x \# a\ \mathsf{abt}$. We leave as an exercise to give an inductive definition of this judgement.

The result, $a'$, of *swapping* one variable, $x$, for another, $y$, within an abt, $a$, written $[x \leftrightarrow y]a = a'\ \mathsf{abt}$ is also inductively defined.

$$\frac{[x \leftrightarrow y]z = z'\ \mathsf{name}}{[x \leftrightarrow y]z = z'\ \mathsf{abt}^0} \tag{3.9a}$$

$$\frac{[x \leftrightarrow y]a_1 = a_1'\ \mathsf{abt}^{n_1} \quad \ldots \quad [x \leftrightarrow y]a_k = a_k'\ \mathsf{abt}^{n_k}}{[x \leftrightarrow y]o(a_1, \ldots, a_k) = o(a_1', \ldots, a_k')\ \mathsf{abt}^0} \tag{3.9b}$$

$$\frac{[x \leftrightarrow y]z = z'\ \mathsf{name} \quad [x \leftrightarrow y]a = a'\ \mathsf{abt}^n}{[x \leftrightarrow y]z\,.\,a = z'\,.\,a'\ \mathsf{abt}^{n+1}} \tag{3.9c}$$

It is easy to check that $a'$ is determined uniquely by $x$, $y$, and $a$, and hence defines a function of the latter three arguments.

### 3.4.3 Renaming of Bound Variables

A chief characteristic of a binding operator is that the choice of bound variables does not matter. This is captured by treating as equivalent any two abt's that differ only in the choice of bound variables, but are otherwise identical. This relation is called, for historical reasons, $\alpha$-equivalence.

$$\frac{}{\mathcal{X}, x\ \mathsf{abt}^0 \vdash x =_\alpha x\ \mathsf{abt}^0} \tag{3.10a}$$

$$\frac{\mathcal{X} \vdash a_1 =_\alpha b_1\ \mathsf{abt}^{n_1} \quad \ldots \quad \mathcal{X} \vdash a_k =_\alpha b_k\ \mathsf{abt}^{n_k}}{\mathcal{X} \vdash o(a_1, \ldots, a_k) =_\alpha o(b_1, \ldots, b_k)\ \mathsf{abt}^0} \tag{3.10b}$$

$$\frac{x \# \mathcal{X} \quad \mathcal{X}, x\ \mathsf{abt}^0 \vdash a =_\alpha b\ \mathsf{abt}^n}{\mathcal{X} \vdash x\,.\,a =_\alpha x\,.\,b\ \mathsf{abt}^{n+1}} \tag{3.10c}$$

$$\frac{x \# y\ \mathsf{name} \quad y \# \mathcal{X} \quad \mathcal{X}, y\ \mathsf{abt}^0 \vdash [x \leftrightarrow y]a =_\alpha b\ \mathsf{abt}^n}{\mathcal{X} \vdash x\,.\,a =_\alpha y\,.\,b\ \mathsf{abt}^{n+1}} \tag{3.10d}$$

We write $\mathcal{X} \vdash a =_\alpha b$ for $\mathcal{X} \vdash a =_\alpha b\ \mathsf{abt}^n$ for some (unique, if it exists) $n$ nat. Further, we write just $a =_\alpha b$ to mean $\mathcal{X} \vdash a =_\alpha b$, where $\mathcal{X}$ is the least variable context covering $a$ and $b$.

There are many characterizations of $\alpha$-equivalence.

**Theorem 3.3.** *The following rules of $\alpha$-equivalence are derivable relative to the set of rules* (3.10)*.*

$$\frac{x \# y\ \mathsf{name} \quad y \# \mathcal{X}}{\mathcal{X} \vdash x\,.\,a =_\alpha y\,.\,[x \leftrightarrow y]a\ \mathsf{abt}^{n+1}} \tag{3.11a}$$

$$\frac{x \# z\ \mathsf{name} \quad y \# z\ \mathsf{name} \quad z \# \mathcal{X} \quad \mathcal{X}, z\ \mathsf{abt}^0 \vdash [x \leftrightarrow z]a =_\alpha [y \leftrightarrow z]b\ \mathsf{abt}^n}{\mathcal{X} \vdash x\,.\,a =_\alpha y\,.\,b\ \mathsf{abt}^{n+1}} \tag{3.11b}$$

**Theorem 3.4.** *$\alpha$-equivalence is reflexive, symmetric, and transitive.*

### 3.4.4 Substitution

*Substitution* is the process of replacing all free occurrences of a variable, $x$, in an abt, $b$, by another, $a$, which is written $[a/x]b = c\ \mathsf{abt}^n$.

$$\frac{}{\mathcal{X} \vdash [a/x]x = a\ \mathsf{abt}^0} \tag{3.12a}$$

$$\frac{x \# y \ \mathsf{name}}{\mathcal{X} \vdash [a/x]y = y \ \mathsf{abt}^0} \tag{3.12b}$$

$$\frac{\mathcal{X} \vdash [a/x]b_1 = c_1 \ \mathsf{abt}^{n_1} \quad \ldots \quad \mathcal{X} \vdash [a/x]b_k = c_k \ \mathsf{abt}^{n_k}}{\mathcal{X} \vdash [a/x]o(b_1, \ldots, b_k) = o(c_1, \ldots, c_k) \ \mathsf{abt}^0} \tag{3.12c}$$

$$\frac{x \# y \ \mathsf{name} \quad y \# \mathcal{X} \quad \mathcal{X}, y \ \mathsf{abt}^0 \vdash [a/x]b = c \ \mathsf{abt}^n}{\mathcal{X} \vdash [a/x]y.b = y.c \ \mathsf{abt}^{n+1}} \tag{3.12d}$$

$$\frac{\mathcal{X} \vdash b =_\alpha b' \ \mathsf{abt}^n \quad \mathcal{X} \vdash [a/x]b' = c \ \mathsf{abt}^n}{\mathcal{X} \vdash [a/x]b = c \ \mathsf{abt}^n} \tag{3.12e}$$

Rule (3.12e), which states that substitution respects $\alpha$-equivalence, ensures that the apartness conditions on Rule (3.12d) may be imposed without loss of generality.

**Theorem 3.5.** *1. If $\mathcal{X} \vdash a \ \mathsf{abt}^0$ and $\mathcal{X}, x \ \mathsf{abt}^0 \vdash b \ \mathsf{abt}^n$, then there exists $\mathcal{X} \vdash c \ \mathsf{abt}^n$ such that $\mathcal{X} \vdash [a/x]b = c \ \mathsf{abt}^n$.*

*2. If $\mathcal{X} \vdash a \ \mathsf{abt}^0$, $\mathcal{X} \vdash [a/x]b = c \ \mathsf{abt}^n$ and $\mathcal{X} \vdash [a/x]b = c' \ \mathsf{abt}^n$, then $\mathcal{X} \vdash c =_\alpha c' \ \mathsf{abt}^n$.*

As a notational convenience we often drop the variable context and valence from the substitution judgement, writing just $[a/x]b = c$. Furthermore, in view of Theorem 3.5, we write $[a/x]b$ for the unique (up to $\alpha$-equivalence) $c$ such that $[a/x]b = c$. This makes sense provided that abt's are always identified up to $\alpha$-equivalence.

## 3.5 Exercises

1. Give an inductive definition of the two-place judgement $|s| = n \ \mathsf{str}$, where $s \ \mathsf{str}$ and $n \ \mathsf{nat}$, stating that a string $s$ has length $n$, namely the number of symbols occurring within it. Use the principle of string induction to show that this judgement has mode $(\forall, \exists!)$, and hence defines a function.

2. Give an inductive definition of equality of strings, and show that string concatenation is associative. Specifically, define the judgement $s_1 = s_2 \ \mathsf{str}$, and show that if $s_1 \hat{} s_2 = s_{12} \ \mathsf{str}$, $s_{12} \hat{} s_3 = s_{123} \ \mathsf{str}$, $s_1 \hat{} s_{23} = s'_{123} \ \mathsf{str}$, and $s_2 \hat{} s_2 = s_{23} \ \mathsf{str}$, then $s_{123} = s'_{123} \ \mathsf{str}$.

3. Give an inductive definition of *simultaneous substitution* of a sequence of $n$ ast's for a sequence of $n$ distinct variables within an ast, written $[a_1, \ldots, a_n / x_1, \ldots, x_n]b = c$. Show that $c$ is uniquely determined, and hence we may write $[a_1, \ldots, a_n / x_1, \ldots, x_n]b$ for the unique such $c$.

4. Suppose that `let` is an operator of arity $(0,1)$ and that `plus` is an operator of arity $(0,0)$. Determine whether or not each of the following $\alpha$-equivalences are valid.

$$\texttt{let}(x, x.x) =_\alpha \texttt{let}(x, y.y) \tag{3.13a}$$
$$\texttt{let}(y, x.x) =_\alpha \texttt{let}(y, y.y) \tag{3.13b}$$
$$\texttt{let}(x, x.x) =_\alpha \texttt{let}(y, y.y) \tag{3.13c}$$
$$\texttt{let}(x, x.\texttt{plus}(x, y)) =_\alpha \texttt{let}(x, z.\texttt{plus}(z, y)) \tag{3.13d}$$
$$\texttt{let}(x, x.\texttt{plus}(x, y)) =_\alpha \texttt{let}(x, y.\texttt{plus}(y, y)) \tag{3.13e}$$

# Chapter 4

# General Judgements

The concept of a *variable* is central to all of mathematics, including the study of programming languages. Informally, a variable stands for a fixed, but unspecified, object drawn from the universe of discourse. In elementary algebra variables range over real numbers, and when we write formulas such as $x^2 + 2x + 1$, we implicitly understand $x$ to stand for an unspecified real number. In the study of programming languages we make use of variables that range over a wide variety of objects, but the principle remains the same: a variable stands for a fixed, but unspecified, element of the universe. Put in other terms, an expression involving a variable stands for *any* of its possible instances obtained by replacing the variable with an object from the universe.

To capture this more formally, we introduce the concept of the *general*, or *schematic*, judgement form. This judgement makes sense whenever we are considering judgements over a universe of objects for which we have a notion of a *variable* and the notion of *substitution* of such an object for that variable in another such object. The class of abstract syntax trees defined in Chapter 3 is one example, as is the broader class of abstract binding trees. The latter class admits the additional concept of $\alpha$-equivalence, which we shall require is respected by all of our judgement forms.

## 4.1 Generality

The *general* judgement may be defined over any class of syntactic objects involving variables for which there is an associated substitution function. For example, the class of abstract syntax trees defined in Chapter 3 admits a general judgement, as will further enhancements to that class to be intro-

duced later in the book. For the time being we will write $a$ syn to indicate that $a$ is an object of a syntactic class supporting variable substitution.

A *general* judgement over has the form $x$ syn $\mid J$, where $x$ name and $J$ is a categorical judgement possibly involving the name $x$. This judgement asserts that $[a/x]J$ holds whenever $a$ syn. The general judgement is naturally extended to the *iterated* form

$$x_1 \text{ syn} \mid \cdots \mid x_n \text{ syn} \mid J, \tag{4.1}$$

where $x_i$ name for each $1 \leq i \leq n$, and $x_i \# x_j$ whenever $i \neq j$. The iterated generality judgement is usually abbreviated to

$$x_1 \text{ syn}, \ldots, x_n \text{ syn} \mid J. \tag{4.2}$$

We let $\mathcal{X}$ range over finite sequences of hypotheses of the form $x_i$ syn subject to the constraints just mentioned. Thus, if $\mathcal{X}$ is $x_1$ syn, $\ldots$, $x_n$ syn, then the general judgement $\mathcal{X} \vdash J$ asserts that $[a_1, \ldots, a_n/x_1, \ldots, x_n]J$ holds for every $a_i$ syn for each $1 \leq i \leq n$.

To show that such a general judgement is derivable, it suffices to exhibit a Evidence for the derivability of the general judgement consists of a *derivation scheme*, $\nabla$, involving $x_1, \ldots, x_n$ such that

$$[a_1, \ldots, a_n/x_1, \ldots, x_n]\nabla$$

is a derivation of

$$[a_1, \ldots, a_n/x_1, \ldots, x_n]J$$

for every choice of instantiating objects $a_1, \ldots, a_n$. Thus, a derivation scheme is a uniform derivation of the conclusion that is insensitive to the choice of objects $x_i$.

The general judgement is often combined with the (derivability) hypothetical judgement to obtain the *hypothetico-general* judgement

$$x_1 \text{ syn}, \ldots, x_m \text{ syn} \mid J_1, \ldots, J_n \vdash J, \tag{4.3}$$

where the variables $x_1, \ldots, x_n$ govern the hypotheses and the conclusion of the hypothetical judgement. Expanding the notation, we see that this judgement asserts that for every choice of objects $a_i$ syn for each $1 \leq i \leq m$, if the rule set is extended with the axioms $[a_1, \ldots, a_m/x_1, \ldots, x_m]J_i$ for each $1 \leq i \leq n$, then $[a_1, \ldots, a_m/x_1, \ldots, x_m]J$ is also derivable.

For example, the following hypothetico-general judgement is derivable relative to Rules (1.2):

$$x \text{ syn} \mid x \text{ nat} \vdash \texttt{succ(succ(}x\texttt{))} \text{ nat}.$$

The derivability of this hypothetico-general judgement means that every instance

$$a\ \mathsf{nat} \vdash \mathtt{succ(succ(}a\mathtt{))}\ \mathsf{nat}$$

of the derivability judgement is valid according to Rules (1.2), which is indeed the case. Evidence for this consists of the derivation scheme, $\nabla$,

$$\dfrac{\dfrac{\overline{x\ \mathsf{nat}}}{\mathtt{succ(}x\mathtt{)}\ \mathsf{nat}}}{\mathtt{succ(succ(}x\mathtt{))}\ \mathsf{nat}}\ , \tag{4.4}$$

which involves $x$ as a parameter. Each instance of $\nabla$ obtained by substituting an object $a$ for $x$ is a derivation of the corresponding instance of the hypothetical judgement.

## 4.2 Structural Properties

The general judgement enjoys structural properties that are reminiscent of those enjoyed by the hypothetical judgement:

**Proliferation** If $\mathcal{X} \mid J$, then $\mathcal{X}, x\ \mathsf{syn} \mid J$.

**Swapping** If $\mathcal{X}_1, x_1\ \mathsf{syn}, x_2\ \mathsf{syn}, \mathcal{X}_2 \mid J$, then $\mathcal{X}_1, x_2\ \mathsf{syn}, x_1\ \mathsf{syn}, \mathcal{X}_2 \mid J$.

**Duplication** If $\mathcal{X}, x\ \mathsf{syn}, x\ \mathsf{syn} \mid J$, then $\mathcal{X}, x\ \mathsf{syn} \mid J$.

**Substitution** If $\mathcal{X}, x\ \mathsf{syn} \mid J$, then $\mathcal{X} \mid [a/x]J$, provided that $\mathcal{X} \mid a\ \mathsf{syn}$.

Proliferation of variables corresponds to weakening, and substitution corresponds to transitivity. By considering $\mathcal{X}$ to be a finite *set* of hypotheses, we render implicit the principles of swapping and duplication, much as we render implicit exchange and contraction of hypotheses.

Combining the structural properties of the hypothetical and general judgements, we obtain the following set of *structural rules* governing the hypothetico-general judgement.

$$\overline{\mathcal{X} \mid \Gamma, J \vdash J} \tag{4.5a}$$

$$\dfrac{\mathcal{X} \mid \Gamma \vdash K}{\mathcal{X} \mid \Gamma, J \vdash K} \tag{4.5b}$$

$$\dfrac{\mathcal{X} \mid \Gamma \vdash K \quad \mathcal{X} \mid \Gamma, K \vdash J}{\mathcal{X} \mid \Gamma \vdash J} \tag{4.5c}$$

$$\frac{\mathcal{X} \mid \Gamma \vdash J \quad x \# \mathcal{X}}{\mathcal{X}, x \text{ syn} \mid \Gamma \vdash J} \tag{4.5d}$$

$$\frac{\mathcal{X}, x \text{ syn} \mid \Gamma \vdash J \quad \mathcal{X} \vdash a \text{ syn}}{\mathcal{X} \mid [a/x]\Gamma \vdash [a/x]J} \tag{4.5e}$$

In most cases we drop explicit mention of the variables in a hypothetico-general judgement. For example, we may write $x$ nat $\vdash$ succ($x$) nat to mean $x$ syn $\mid$ $x$ nat $\vdash$ succ($x$) nat, relying on the choice of meta-variable names to determine the intended variable declarations.

## 4.3 Generalized Rules

Inductive definitions may be further generalized to permit hypothetico-general judgements in the premises and conclusions of rules. A generalized rule in this sense has the form

$$\frac{\mathcal{X}_1 \mid \Gamma_1 \vdash J_1 \quad \ldots \quad \mathcal{X}_k \mid \Gamma_k \vdash J_k}{J} \ . \tag{4.6}$$

Each $\mathcal{X}_i$ declares the *local variables*, and each $\Gamma_i$ specifies the *local hypotheses*, of the inference.

A generalized rule applies relative to an arbitrary set of *global variables*, $\mathcal{X}$, and *global hypotheses*, $\Gamma$. This can be made explicit by stating a generalized rule in the following form:

$$\frac{\mathcal{X}\,\mathcal{X}_1 \mid \Gamma\,\Gamma_1 \vdash J_1 \quad \ldots \quad \mathcal{X}\,\mathcal{X}_k \mid \Gamma\,\Gamma_k \vdash J_k}{\mathcal{X} \mid \Gamma \vdash J} \ . \tag{4.7}$$

The pair $\mathcal{X} \mid \Gamma$ is called the *global context* of the inference, and each pair $\mathcal{X}_i \mid \Gamma_i$ is called the *local context* of the $i$th premise of the rule. Thus a generalized rule states that $J$ is derivable in the global context iff each of the premises $J_i$ is derivable in the extension of the global context by the local context of that premise.

To avoid confusion, in Rule (4.7) we tacitly require that each local variables lie apart from the global variables; that is, we demand that $\mathcal{X}_i \# \mathcal{X}$ for each $1 \leq i \leq n$. In most cases the local variables are derived from the bound variables in the subject of a judgement, and hence are of interest only when working over the universe of abt's. In such cases the local apartness restrictions may always be met, provided that we insist that the judgement form respect $\alpha$-equivalence. This may be imposed by including

the following rule in any generalized inductive definition over the class of abt's:

$$\frac{\mathcal{X} \mid \Gamma \vdash a' \; J \quad \mathcal{X} \vdash a =_\alpha a'}{\mathcal{X} \mid \Gamma \vdash a \; J} \tag{4.8}$$

As in Chapter 2, we regard a generalized inductive definition as an ordinary inductive definition of an *infinite family* of categorical judgements indexed by the context $\mathcal{X} \mid \Gamma$. The premises of the rule may then be seen as referring to various instances of this family, and the conclusion to one such instance. To ensure that this family behaves like hypothetico-general judgements, we must impose the combined structural rules given in Section 4.2 on page 35. In principle each of these rules is implicitly included in any generalized inductive definition, but in practice we arrange that most of these rules are admissible, leaving only reflexivity to be handled explicitly. Weakening and proliferation are admissible because we have ensured that a generalized rule is applicable in any context. Swapping and duplication, and exchange and contraction, are admissible because we treat the variables and hypotheses as sets, rather than sequences. This leaves transitivity and substitution to be proved admissible on a case-by-case basis.

The principle of rule induction associated with a generalized inductive definition follows the same pattern as that given in Chapter 2. First, let us re-state the generalized rule (4.7) in a form that makes the subjects of the judgements explicit:

$$\frac{\mathcal{X}\,\mathcal{X}_1 \mid \Gamma\,\Gamma_1 \vdash a_1 \; J_1 \quad \ldots \quad \mathcal{X}\,\mathcal{X}_k \mid \Gamma\,\Gamma_k \vdash a_k \; J_k}{\mathcal{X} \mid \Gamma \vdash a \; J} \; . \tag{4.9}$$

Let $P_{\mathcal{X}|\Gamma}$ be a family of predicates indexed by contexts $\mathcal{X} \mid \Gamma$. To show that $P_{\mathcal{X}|\Gamma}(a)$ holds whenever $\mathcal{X} \mid \Gamma \vdash a \; J$, it is enough to establish the following conditions:

1. Reflexivity: $P_{\mathcal{X}|\Gamma, a \; J}(a)$.
2. Closure under rules: for each rule of the form (4.9), if $P_{\mathcal{X}\,\mathcal{X}_1|\Gamma\,\Gamma_1}(a_1)$, and $\ldots$ and $P_{\mathcal{X}\,\mathcal{X}_n|\Gamma\,\Gamma_n}(a_n)$, then $P_{\mathcal{X}|\Gamma}(a)$.
3. Renaming: if $\mathcal{X} \vdash a =_\alpha a'$, then $P_{\mathcal{X}|\Gamma}(a)$ iff $P_{\mathcal{X}|\Gamma}(a')$

If any of the other structural rules are inadmissible, then they must be considered explicitly for the induction.

## 4.4 Exercises

# Chapter 5

# Transition Systems

*Transition systems* are used to describe the execution behavior of programs by defining an abstract computing device with a set, $S$, of *states* that are related by a *transition judgement*, $\mapsto$. The transition judgement describes how the state of the machine evolves during execution.

## 5.1 Transition Systems

An *(ordinary) transition system* is specified by the following judgements:

1. $s$ state, asserting that $s$ is a *state* of the transition system.
2. $s$ final, where $s$ state, asserting that $s$ is a *final* state.
3. $s$ initial, where, $s$ state, asserting that $s$ is an *initial* state.
4. $s \mapsto s'$, where $s$ state and $s'$ state, asserting that state $s$ may transition to state $s'$.

We require that if $s$ final, then for no $s'$ do we have $s \mapsto s'$. In general, a state $s$ for which there is no $s' \in S$ such that $s \mapsto s'$ is said to be *stuck*. All final states are stuck, but not all stuck states need be final!

A *transition sequence* is a sequence of states $s_0, \ldots, s_n$ such that $s_0$ initial, and $s_i \mapsto s_{i+1}$ for every $0 \leq i < n$. A transition sequence is *maximal* iff $s_n \not\mapsto$; it is *complete* iff it is maximal and, in addition, $s_n$ final. Thus every complete transition sequence is maximal, but maximal sequences are not necessarily complete. A transition system is *deterministic* iff for every state $s$ there exists at most one state $s'$ such that $s \mapsto s'$, otherwise it is *non-deterministic*.

A *labelled transition system* over a set of labels, $I$, is a generalization of a transition system in which the single transition judgement, $s \mapsto s'$ is replaced by an $I$-indexed family of transition judgements, $s \overset{i}{\longmapsto} s'$, where $s$ and $s'$ are states of the system. In typical situations the family of transition relations is given by a simultaneous inductive definition in which each rule may make reference to any member of the family.

It is often necessary to consider families of transition relations in which there is a distinguished unlabelled transition, $s \mapsto s'$, in addition to the indexed transitions. It is sometimes convenient to regard this distinguished transition as labelled by a special, anonymous label not otherwise in $I$. For historical reasons this distinguished label is often designated by $\tau$ or $\epsilon$, but we will simply use an unadorned arrow. The unlabelled form is often called a *silent* transition, in contrast to the labelled forms, which announce their presence with a label.

## 5.2 Iterated Transition

Let $s \mapsto s'$ be a transition judgement, whether drawn from an indexed set of such judgements or not.

The *reflexive, transitive closure* of this judgement, written $s \mapsto^* s'$, is inductively defined by the following rules.

$$\frac{}{s \mapsto^* s} \tag{5.1a}$$

$$\frac{s \mapsto s' \quad s' \mapsto^* s''}{s \mapsto^* s''} \tag{5.1b}$$

The principle of rule induction for these rules states that to show that $P(s, s')$ holds whenever $s \mapsto^* s'$, it is enough to show these two properties of $P$:

1. $P(s, s)$.
2. if $s \mapsto s'$ and $P(s', s'')$, then $P(s, s'')$.

The first requirement is to show that $P$ is reflexive. The second is to show that $P$ is *closed under head expansion*, or *converse evaluation*. Using this principle, it is easy to prove that $\mapsto^*$ is reflexive and transitive.

The *n-times iterated* transition judgement, $s \mapsto^n s'$, where $n \geq 0$, is inductively defined by the following rules.

$$\frac{}{s \mapsto^0 s} \tag{5.2a}$$

$$\frac{s \mapsto s' \quad s' \mapsto^n s''}{s \mapsto^{n+1} s''} \tag{5.2b}$$

**Theorem 5.1.** *For all states $s$ and $s'$, $s \mapsto^* s'$ iff $s \mapsto^k s'$ for some $k \geq 0$.*

Finally, we write $\downarrow s$ to indicate that there exists some $s'$ final such that $s \mapsto^* s'$.

## 5.3 Simulation and Bisimulation

A *strong simulation* between two transition systems $\mapsto_1$ and $\mapsto_2$ is given by a binary relation, $s_1 \, S \, s_2$, between their respective states such that if $s_1 \, S \, s_2$, then $s_1 \mapsto_1 s_1'$ implies $s_2 \mapsto_2 s_2'$ for some state $s_2'$ such that $s_1' \, S \, s_2'$. Two states, $s_1$ and $s_2$, are *strongly similar* iff there is a strong simulation, $S$, such that $s_1 \, S \, s_2$. Two transition systems are strongly similar iff each initial state of the first is strongly similar to an initial state of the second. Finally, two states are *strongly bisimilar* iff there is a single relation $S$ such that both $S$ and its converse are simulations. Note that this is stronger than merely requiring that there be simulations in both directions!

A strong simulation between two labelled transition systems over the same set, $I$, of labels consists of a relation $S$ between states such that for each $i \in I$ the relation $S$ is a strong simulation between $\overset{i}{\longmapsto}_1$ and $\overset{i}{\longmapsto}_2$. That is, if $s_1 \, S \, s_2$, then $s_1 \overset{i}{\longmapsto}_1 s_1'$ implies that $s_2 \overset{i}{\longmapsto}_2 s_2'$ for some $s_2'$ such that $s_1' \, S \, s_2'$. In other words the simulation must preserve labels, and not just transition.

The requirements for strong simulation are rather stringent: every step in the first system must be mimicked by a similar step in the second, up to the simulation relation in question. This means, in particular, that a sequence of steps in the first system can only be simulated by a sequence of steps of the same length in the second—there is no possibility of performing "extra" work to achieve the simulation.

A *weak simulation* between transition systems is a binary relation between states such that if $s_1 \, S \, s_2$, then $s_1 \mapsto_1 s_1'$ implies that $s_2 \mapsto_2^* s_2'$ for some $s_2'$ such that $s_1' \, S \, s_2'$. That is, every step in the first may be matched by zero or more steps in the second. A *weak bisimulation* is such that both it and its converse are weak simulations. We say that states $s_1$ and $s_2$ are *weakly (bi)similar* iff there is a weak (bi)simulation $S$ such that $s_1 \, S \, s_2$.

The corresponding notion of weak simulation for labelled transitions involves the silent transition. The idea is that to weakly simulate the labelled transition $s_1 \stackrel{i}{\longmapsto}_1 s_1'$, we do not wish to permit multiple *labelled* transitions between related states, but rather to permit any number of *unlabelled* transitions to accompany the labelled transition. A relation between states is a *weak simulation* iff it satisfies both of the following conditions whenever $s_1 \, S \, s_2$:

1. If $s_1 \longmapsto_1 s_1'$, then $s_2 \longmapsto_2^* s_2'$ for some $s_2'$ such that $s_1' \, S \, s_2'$.
2. If $s_1 \stackrel{i}{\longmapsto}_1 s_1'$, then $s_2 \longmapsto_2^* \stackrel{i}{\longmapsto}_2 \longmapsto_2^* s_2'$ for some $s_2'$ such that $s_1' \, S \, s_2'$.

That is, every silent transition must be mimicked by zero or more silent transitions, and every labelled transition must be mimicked by a corresponding labelled transition, preceded and followed by any number of silent transitions. As before, a *weak bisimulation* is a relation between states such that both it and its converse are weak simulations. Finally, two states are *weakly (bi)similar* iff there is a weak (bi)simulation between them.

## 5.4 Exercises

1. Prove that $S$ is a weak simulation for the ordinary transition system $\longmapsto$ iff $S$ is a strong simulation for $\longmapsto^*$.
2. A similar result about labelled transition systems.

# Part II

# Levels of Syntax

# Chapter 6

# Concrete Syntax

The *concrete syntax* of a language is a means of representing expressions as strings that may be written on a page or entered using a keyboard. The concrete syntax usually is designed to enhance readability and to eliminate ambiguity. While there are good methods for eliminating ambiguity, improving readability is, to a large extent, a matter of taste.

In this chapter we introduce the main methods for specifying concrete syntax, using as an example an illustrative expression language, called $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, that supports elementary arithmetic on the natural numbers and simple computations on strings. In addition, $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ includes a construct for binding the value an expression to a variable within a specified scope.

## 6.1 Lexical Structure

The first phase of syntactic processing is to convert from a character-based representation to a symbol-based representation of the input. This is called *lexical analysis*, or *lexing*. The main idea is to aggregate characters into symbols that serve as tokens for subsequent phases of analysis. For example, the numeral `467` is written as a sequence of three consecutive characters, one for each digit, but is regarded as a single token, namely the number 467. Similarly, an identifier such as `temp` comprises four letters, but is treated as a single symbol representing the entire word. Moreover, many character-based representations include empty "white space" (spaces, tabs, newlines, and, perhaps, comments) that are discarded by the lexical analyzer.[1]

[1] In some languages white space *is* significant, in which case it must be converted to symbolic form for subsequent processing.

The character representation of symbols is, in most cases, conveniently described using *regular expressions*. The lexical structure of $\mathcal{L}\{\texttt{num str}\}$ is specified as follows:

| | | | |
|---|---|---|---|
| Item | itm | ::= | kwd \| id \| num \| str \| spl |
| Keyword | kwd | ::= | $\texttt{l}\cdot\texttt{e}\cdot\texttt{t}\cdot\varepsilon \mid \texttt{b}\cdot\texttt{e}\cdot\texttt{g}\cdot\varepsilon \mid \texttt{i}\cdot\texttt{n}\cdot\varepsilon$ |
| Identifier | id | ::= | ltr (ltr \| dig)$^*$ |
| Numeral | num | ::= | dig dig$^*$ |
| Literal | str | ::= | qum (num \| dig)$^*$qum |
| Special | spl | ::= | + \| * \| @ \| ( \| ) \| \| |
| Letter | ltr | ::= | a \| b \| ... |
| Digit | dig | ::= | 0 \| 1 \| ... |
| Quote | qum | ::= | " |

A lexical item is either a keyword, an identifier, a numeral, a string literal, or a special symbol. There are three keywords, specified as sequences of characters, for emphasis. Identifiers start with a letter and may involve letters or digits. Numerals are non-empty sequences of digits. String literals are sequences of letters or digits surrounded by quotes. The special symbols, letters, digits, and quote marks are as enumerated. (Observe that in these latter classes we are tacitly identifying a character with the unit length string consisting of that character.)

The job of the lexical analyzer is to translate a character string into a token string using the above definitions as a guide. The input string is scanned, ignoring white space, and translating lexical items into tokens, which are specified by the following rules:

$$\frac{s\ \mathsf{str}}{\texttt{ID}[s]\ \mathsf{tok}} \tag{6.1a}$$

$$\frac{n\ \mathsf{nat}}{\texttt{NUM}[n]\ \mathsf{tok}} \tag{6.1b}$$

$$\frac{s\ \mathsf{str}}{\texttt{STR}[s]\ \mathsf{tok}} \tag{6.1c}$$

$$\frac{}{\texttt{LET}\ \mathsf{tok}} \tag{6.1d}$$

$$\frac{}{\texttt{BE}\ \mathsf{tok}} \tag{6.1e}$$

$$\frac{}{\texttt{IN}\ \mathsf{tok}} \tag{6.1f}$$

$$\frac{}{\texttt{ADD}\ \mathsf{tok}} \tag{6.1g}$$

$$\frac{}{\texttt{MUL}\ \mathsf{tok}} \tag{6.1h}$$

$$\frac{}{\texttt{CAT}\ \mathsf{tok}} \tag{6.1i}$$

$$\frac{}{\texttt{VB}\ \mathsf{tok}} \tag{6.1j}$$

$$\frac{}{\texttt{LP}\ \mathsf{tok}} \tag{6.1k}$$

$$\frac{}{\texttt{RP}\ \mathsf{tok}} \tag{6.1l}$$

Lexical analysis is inductively defined by the following judgement forms:

| | |
|---|---|
| $s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}$ | Scan an item |
| $s\ \mathsf{kwd} \longleftrightarrow t\ \mathsf{tok}$ | Scan a keyword |
| $s\ \mathsf{id} \longleftrightarrow t\ \mathsf{tok}$ | Scan an identifier |
| $s\ \mathsf{num} \longleftrightarrow t\ \mathsf{tok}$ | Scan a number |
| $s\ \mathsf{spl} \longleftrightarrow t\ \mathsf{tok}$ | Scan a symbol |

The definition of these forms makes use of some auxiliary judgements corresponding to the primitive concepts used in the definition of the lexical structure of the language.

$$\frac{s\ \mathsf{kwd} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{6.2a}$$

$$\frac{s\ \mathsf{id} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{6.2b}$$

$$\frac{s\ \mathsf{num} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{6.2c}$$

$$\frac{s\ \mathsf{str} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{6.2d}$$

$$\frac{s\ \mathsf{spl} \longleftrightarrow t\ \mathsf{tok}}{s\ \mathsf{itm} \longleftrightarrow t\ \mathsf{tok}} \tag{6.2e}$$

$$\frac{s = \mathtt{l} \cdot \mathtt{e} \cdot \mathtt{t} \cdot \varepsilon \ \mathsf{str}}{s \ \mathsf{kwd} \longleftrightarrow \mathtt{LET} \ \mathsf{tok}} \tag{6.2f}$$

$$\frac{s = \mathtt{b} \cdot \mathtt{e} \cdot \varepsilon \ \mathsf{str}}{s \ \mathsf{kwd} \longleftrightarrow \mathtt{LET} \ \mathsf{tok}} \tag{6.2g}$$

$$\frac{s = \mathtt{i} \cdot \mathtt{n} \cdot \varepsilon \ \mathsf{str}}{s \ \mathsf{kwd} \longleftrightarrow \mathtt{LET} \ \mathsf{tok}} \tag{6.2h}$$

$$\frac{s = s_1 \hat{} s_2 \ \mathsf{str} \quad s_1 \ \mathsf{ltr} \quad s_2 \ \mathsf{lord}}{s \ \mathsf{id} \longleftrightarrow \mathtt{ID}[s] \ \mathsf{tok}} \tag{6.2i}$$

$$\frac{s = s_1 \hat{} s_2 \ \mathsf{str} \quad s_1 \ \mathsf{dig} \quad s_2 \ \mathsf{dgs} \quad s \ \mathsf{num} \longleftrightarrow n \ \mathsf{nat}}{s \ \mathsf{num} \longleftrightarrow \mathtt{NUM}[n] \ \mathsf{tok}} \tag{6.2j}$$

$$\frac{s = s_1 \hat{} s_2 \hat{} s_3 \ \mathsf{str} \quad s_1 \ \mathsf{qum} \quad s_2 \ \mathsf{lord} \quad s_2 \ \mathsf{qum}}{s \ \mathsf{str} \longleftrightarrow \mathtt{STR}[s_2] \ \mathsf{tok}} \tag{6.2k}$$

$$\frac{s = \mathtt{+} \cdot \varepsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{ADD} \ \mathsf{tok}} \tag{6.2l}$$

$$\frac{s = \mathtt{*} \cdot \varepsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{MUL} \ \mathsf{tok}} \tag{6.2m}$$

$$\frac{s = \mathtt{@} \cdot \varepsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{CAT} \ \mathsf{tok}} \tag{6.2n}$$

$$\frac{s = \mathtt{(} \cdot \varepsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{LP} \ \mathsf{tok}} \tag{6.2o}$$

$$\frac{s = \mathtt{)} \cdot \varepsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{RP} \ \mathsf{tok}} \tag{6.2p}$$

$$\frac{s = \mathtt{|} \cdot \varepsilon \ \mathsf{str}}{s \ \mathsf{spl} \longleftrightarrow \mathtt{VB} \ \mathsf{tok}} \tag{6.2q}$$

We leave it to the reader to provide definitions of the remaining auxiliary judgement forms.

Translation from a character string to a token string is defined by the judgement $s \ \mathsf{fil} \longleftrightarrow t \ \mathsf{tokstr}$, which is inductively defined by the following rule:

$$\frac{s = s_1 \hat{} s_2 \hat{} s_3 \ \mathsf{str} \quad s_1 \ \mathsf{whs} \quad s_2 \ \mathsf{itm} \longleftrightarrow t \ \mathsf{tok} \quad s_3 \ \mathsf{fil} \longleftrightarrow ts \ \mathsf{tokstr}}{s \ \mathsf{fil} \longleftrightarrow t \cdot ts \ \mathsf{tokstr}} \tag{6.3a}$$

In words, the translation of a character string to a token string consists of repeatedly skipping white space and translating a lexical item into a token, which is then emitted to the output.

## 6.2 Context-Free Grammars

The standard method for defining concrete syntax is by giving a *context-free grammar* for the language. A grammar consists of three components:

1. The *tokens*, or *terminals*, over which the grammar is defined.
2. The *syntactic classes*, or *non-terminals*, which are disjoint from the terminals.
3. The *rules*, or *productions*, which have the form $A ::= \alpha$, where $A$ is a non-terminal and $\alpha$ is a string of terminals and non-terminals.

Each syntactic class is a collection of token strings. The rules determine which strings belong to which syntactic classes.

When defining a grammar, we often abbreviate a set of productions,

$$\begin{aligned} A &::= \alpha_1 \\ &\vdots \\ A &::= \alpha_n, \end{aligned}$$

each with the same left-hand side, by the *compound* production

$$A ::= \alpha_1 \mid \ldots \mid \alpha_n,$$

which specifies a set of alternatives for the syntactic class $A$.

A context-free grammar determines a simultaneous inductive definition of its syntactic classes. Specifically, we regard each non-terminal, $A$, as a judgement form, $s\ A$, over strings of terminals. To each production of the form

$$A ::= s_1\, A_1\, s_2\, \ldots\, s_n\, A_n\, s_{n+1} \tag{6.4}$$

we associate an inference rule

$$\frac{s'_1\ A_1 \quad \ldots \quad s'_n\ A_n}{s_1\, s'_1\, s_2\, \ldots\, s_n\, s'_n\, s_{n+1}\ A}\ . \tag{6.5}$$

The collection of all such rules constitutes an inductive definition of the syntactic classes of the grammar.

Recalling that juxtaposition of strings is short-hand for their concatenation, we may re-write the preceding rule as follows:

$$\frac{s'_1\ A_1 \quad \ldots \quad s'_n\ A_n \quad s = s_1 \hat{} s'_1 \hat{} s_2 \hat{} \ldots s_n \hat{} s'_n \hat{} s_{n+1}}{s\ A}\ . \tag{6.6}$$

This formulation makes clear that $s\ A$ holds whenever $s$ can be partitioned as described so that $s'_i\ A$ for each $1 \leq i \leq n$. Since string concatenation is not invertible, the decomposition is not unique, and so there may be many different ways in which the rule applies.

## 6.3 Grammatical Structure

The concrete syntax of $\mathcal{L}\{\texttt{num str}\}$ may be specified by a context-free grammar over the tokens defined in Section 6.1 on page 45. The grammar has only one syntactic class, exp, which is defined by the following compound production:

| | | | |
|---|---|---|---|
| Expression | exp | ::= | num \| str \| id \| LP exp RP \| exp ADD exp \| exp MUL exp \| exp CAT exp \| VB exp VB \| LET id BE exp IN exp |
| Number | num | ::= | NUM[$n$] ($n$ nat) |
| String | str | ::= | STR[$s$] ($s$ str) |
| Identifier | id | ::= | ID[$s$] ($s$ str) |

This grammar makes use of some standard notational conventions to improve readability: we identify a token with the corresponding unit-length string, and we use juxtaposition to denote string concatenation.

Applying the interpretation of a grammar as an inductive definition, we obtain the following rules:

$$\frac{n\ \mathsf{num}}{\texttt{NUM[}n\texttt{]}\ \mathsf{exp}} \tag{6.7a}$$

$$\frac{s\ \mathsf{str}}{\texttt{STR[}s\texttt{]}\ \mathsf{exp}} \tag{6.7b}$$

$$\frac{s\ \mathsf{str}}{\texttt{ID[}s\texttt{]}\ \mathsf{exp}} \tag{6.7c}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\ \texttt{ADD}\ s_2\ \mathsf{exp}} \tag{6.7d}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\ \texttt{MUL}\ s_2\ \mathsf{exp}} \tag{6.7e}$$

$$\frac{s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s_1\ \texttt{CAT}\ s_2\ \mathsf{exp}} \tag{6.7f}$$

$$\frac{s\ \mathsf{exp}}{\mathtt{VB}\,s\,\mathtt{VB}\ \mathsf{exp}} \tag{6.7g}$$

$$\frac{s\ \mathsf{exp}}{\mathtt{LP}\,s\,\mathtt{RP}\ \mathsf{exp}} \tag{6.7h}$$

$$\frac{s_1\ \mathsf{id} \quad s_2\ \mathsf{exp} \quad s_3\ \mathsf{exp}}{\mathtt{LET}\,s_1\,\mathtt{BE}\,s_2\,\mathtt{IN}\,s_3\ \mathsf{exp}} \tag{6.7i}$$

$$\frac{s\ \mathsf{str}}{\mathtt{ID}[s]\ \mathsf{id}} \tag{6.7j}$$

$$\frac{n\ \mathsf{nat}}{\mathtt{NUM}[n]\ \mathsf{num}} \tag{6.7k}$$

To emphasize the role of string concatentation, we may rewrite Rule (6.7e) as follows:

$$\frac{s = s_1\,\mathtt{MUL}\,s_2\ \mathsf{str} \quad s_1\ \mathsf{exp} \quad s_2\ \mathsf{exp}}{s\ \mathsf{exp}} \tag{6.8}$$

That is, $s$ exp is derivable if $s$ is the concatentation of $s_1$, the multiplication sign, and $s_2$, where $s_1$ exp and $s_2$ exp.

## 6.4 Ambiguity

Apart from subjective matters of readability, a principal goal of concrete syntax design is to eliminate ambiguity. The grammar of arithmetic expressions given above is *ambiguous* in the sense that some token strings may be thought of as arising in several different ways. More precisely, there are token strings $s$ for which there is more than one derivation ending with $s$ exp according to Rules (6.7).

For example, consider the character string `1+2*3`, which, after lexical analysis, is translated to the token string

$$\mathtt{NUM[1]\ ADD\,NUM[2]\ MUL\,NUM[3]}.$$

Since string concatenation is associative, this token string can be thought of as arising in several ways, including

$$\mathtt{NUM[1]\ ADD}\,{}_\wedge\mathtt{NUM[2]\ MUL\,NUM[3]}$$

and

$$\mathtt{NUM[1]\ ADD\,NUM[2]}\,{}_\wedge\,\mathtt{MUL\,NUM[3]},$$

where the caret indicates the concatenation point.

One consequence of this observation is that the same token string may be seen to be grammatical according to the rules given in Section 6.3 on page 50 in two different ways. According to the first reading, the expression is principally an addition, with the first argument being a number, and the second being a multiplication of two numbers. According to the second reading, the expression is principally a multiplication, with the first argument being the addition of two numbers, and the second being a number.

Ambiguity is a *purely syntactic* property of grammars; it has nothing to do with the "meaning" of a string. For example, the token string

```
NUM[1] ADD NUM[2] ADD NUM[3],
```

also admits two readings. It is immaterial that both readings have the same meaning under the usual interpretation of arithmetic expressions. Moreover, nothing prevents us from interpreting the token `ADD` to mean "division," in which case the two readings would hardly coincide! Nothing in the syntax itself precludes this interpretation, so we do not regard it as relevant to whether the grammar is ambiguous.

To eliminate ambiguity the grammar of $\mathcal{L}\{\texttt{num str}\}$ given in Section 6.3 on page 50 must be re-structured to ensure that every grammatical string has at most one derivation according to the rules of the grammar. The main method for achieving this is to introduce precedence and associativity conventions that ensure there is only one reading of any token string. Parenthesization may be used to override these conventions, so there is no fundamental loss of expressive power in doing so.

Precedence relationships are introduced by *layering* the grammar by introducing new syntactic classes.

| | | | |
|---|---|---|---|
| Factor | fct | ::= | num \| str \| id \| `LP` exp `RP` |
| Term | trm | ::= | fct \| fct `MUL` trm \| `VB` fct `VB` |
| Expression | exp | ::= | trm \| trm `ADD` exp \| trm `CAT` exp |
| Program | prg | ::= | exp \| `LET` id `BE` trm `IN` prg |

The effect of this grammar is to ensure that `let` has the lowest precedence, addition and concatenation intermediate precedence, and multiplication and length the highest precedence. Moreover, all forms are right-associative. Different choices are possible, according to taste; this grammar illustrates one way to resolve ambiguity.

## 6.5 Informal Conventions

Throughout this book we will encounter many programming languages. In order to write down examples, it is essential to give each one a concrete syntax, and hence we must specify a lexical and grammatical structure for each. To avoid getting bogged down in such syntactic details, we will employ a *short-form* grammatical specification that leaves implicit the lexical structure, and ignores problems of ambiguity that must be solved in practice, but which can be safely glossed over in an informal development. In addition, to avoid introducing names for the various syntactic categories, the grammar is formulated using *typical element* notation, rather than standard grammar notation.

These conventions are best illustrated by example. Using the short-form presentation, the concrete syntax of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ may be defined as follows:

$$\text{Expr}\quad e \quad ::= n \mid \texttt{"}s\texttt{"} \mid s \mid e_1\texttt{+}e_2 \mid e_1\texttt{*}e_2 \mid e_1\texttt{\^{}}e_2 \mid \texttt{|}e\texttt{|} \mid \texttt{let}\,x\,\texttt{be}\,e_1\,\texttt{in}\,e_2$$

When using short-form grammar notation we leave it to the reader to determine the intended lexical structure, to restructure the grammar to avoid ambiguity, and to introduce grammatical classes for each of the typical elements. Moreover, we usually tacitly include parenthesization to improve readability and to resolve ambiguity in examples.

## 6.6 Exercises

# Chapter 7

# Abstract Syntax

The concrete syntax of a language is concerned with the linear representation of the phrases of a language as strings of symbols—the form in which we write them on paper, type them into a computer, and read them from a page. The main goal of concrete syntax design is to enhance the readability and writability of the language, based on subjective criteria such as similarity to other languages, ease of editing using standard tools, and so forth.

But languages are also the subjects of study, as well as the instruments of expression. As such the concrete syntax of a language is just a nuisance. When analyzing a language mathematically we are only interested in the deep structure of its phrases, not their surface representation. The *abstract syntax* of a language exposes the hierarchical and binding structure of the language, and suppresses the linear notation used to write it on the page.

*Parsing* is the process of translation from concrete to abstract syntax. It consists of analyzing the linear representation of a phrase in terms of the grammar of the language and transforming it into an abstract syntax tree or an abstract binding tree that reveals the deep structure of the phrase.

## 7.1 Abstract Syntax Trees

The abstract syntax tree representation of $\mathcal{L}\{\texttt{num str}\}$ is specified by the following assignment of arities to operators.

$$\begin{aligned}
\mathsf{ar}(\texttt{num[}n\texttt{]}) &= 0 \quad (n\ \mathsf{nat}) \\
\mathsf{ar}(\texttt{str[}s\texttt{]}) &= 0 \quad (s\ \mathsf{str}) \\
\mathsf{ar}(\texttt{id[}s\texttt{]}) &= 0 \quad (s\ \mathsf{str}) \\
\mathsf{ar}(\texttt{plus}) &= 2 \\
\mathsf{ar}(\texttt{times}) &= 2 \\
\mathsf{ar}(\texttt{cat}) &= 2 \\
\mathsf{ar}(\texttt{len}) &= 1 \\
\mathsf{ar}(\texttt{let}) &= 3
\end{aligned}$$

Observe that identifiers are regarded as operators of arity 0, and that the `let` construct is regarded as an operator with three arguments, the first of which is expected to be an identifier (but this is not enforced).

Specializing the rules for abstract syntax trees to this signature, we obtain the following inductive definition of the abstract syntax of arithmetic expressions:

$$\frac{n\ \mathsf{nat}}{\texttt{num[}n\texttt{]}\ \mathsf{ast}} \tag{7.1a}$$

$$\frac{s\ \mathsf{str}}{\texttt{str[}s\texttt{]}\ \mathsf{ast}} \tag{7.1b}$$

$$\frac{s\ \mathsf{str}}{\texttt{id[}s\texttt{]}\ \mathsf{ast}} \tag{7.1c}$$

$$\frac{a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{plus(}a_1\texttt{,}a_2\texttt{)}\ \mathsf{ast}} \tag{7.1d}$$

$$\frac{a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{times(}a_1\texttt{,}a_2\texttt{)}\ \mathsf{ast}} \tag{7.1e}$$

$$\frac{a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{cat(}a_1\texttt{,}a_2\texttt{)}\ \mathsf{ast}} \tag{7.1f}$$

$$\frac{a\ \mathsf{ast}}{\texttt{len(}a\texttt{)}\ \mathsf{ast}} \tag{7.1g}$$

$$\frac{s\ \mathsf{str} \quad a_1\ \mathsf{ast} \quad a_2\ \mathsf{ast}}{\texttt{let(id[}s\texttt{],}a_1\texttt{,}a_2\texttt{)}\ \mathsf{ast}} \tag{7.1h}$$

Strictly speaking, the last rule is a specialization of the rule induced by the arity assignment for `let` in which we demand that the first argument be an identifier.

## 7.2 Parsing Into Abstract Syntax Trees

The process of translation from concrete to abstract syntax is called *parsing*. We will define parsing as a judgement between the concrete and abstract syntax of a language. This judgement will have the mode $(\forall, \exists^{\leq 1})$, which states that the parser is a partial function of its input, being undefined for ungrammatical token strings, but otherwise uniquely determining the abstract syntax tree representation of each well-formed input.

The parsing judgements for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ follow the unambiguous grammar given in Chapter 6:

| | |
|---|---|
| $s$ prg $\longleftrightarrow$ $a$ ast | Parse as a program |
| $s$ exp $\longleftrightarrow$ $a$ ast | Parse as an expression |
| $s$ trm $\longleftrightarrow$ $a$ ast | Parse as a term |
| $s$ fct $\longleftrightarrow$ $a$ ast | Parse as a factor |

These judgements are inductively defined simultaneously by the following rules:

$$\frac{n\ \mathsf{nat}}{\texttt{NUM[}n\texttt{]}\ \mathsf{fct} \longleftrightarrow \texttt{num[}n\texttt{]}\ \mathsf{ast}} \tag{7.2a}$$

$$\frac{s\ \mathsf{str}}{\texttt{STR[}s\texttt{]}\ \mathsf{fct} \longleftrightarrow \texttt{str[}s\texttt{]}\ \mathsf{ast}} \tag{7.2b}$$

$$\frac{s\ \mathsf{str}}{\texttt{ID[}s\texttt{]}\ \mathsf{fct} \longleftrightarrow \texttt{id[}s\texttt{]}\ \mathsf{ast}} \tag{7.2c}$$

$$\frac{s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}}{\texttt{LP}\ s\ \texttt{RP}\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}} \tag{7.2d}$$

$$\frac{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{trm} \longleftrightarrow a\ \mathsf{ast}} \tag{7.2e}$$

$$\frac{s_1\ \mathsf{fct} \longleftrightarrow a_1\ \mathsf{ast} \quad s_2\ \mathsf{trm} \longleftrightarrow a_2\ \mathsf{ast}}{s_1\ \texttt{MUL}\ s_2\ \mathsf{trm} \longleftrightarrow \texttt{times}(a_1, a_2)\ \mathsf{ast}} \tag{7.2f}$$

$$\frac{s\ \mathsf{fct} \longleftrightarrow a\ \mathsf{ast}}{\texttt{VB}\ s\ \texttt{VB}\ \mathsf{trm} \longleftrightarrow \texttt{len}(a)\ \mathsf{ast}} \tag{7.2g}$$

$$\frac{s\ \mathsf{trm} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}} \tag{7.2h}$$

$$\frac{s_1\ \mathsf{trm} \longleftrightarrow a_1\ \mathsf{ast} \quad s_2\ \mathsf{exp} \longleftrightarrow a_2\ \mathsf{ast}}{s_1\ \mathtt{ADD}\ s_2\ \mathsf{exp} \longleftrightarrow \mathtt{plus(}a_1\mathtt{,}a_2\mathtt{)}\ \mathsf{ast}} \tag{7.2i}$$

$$\frac{s_1\ \mathsf{trm} \longleftrightarrow a_1\ \mathsf{ast} \quad s_2\ \mathsf{exp} \longleftrightarrow a_2\ \mathsf{ast}}{s_1\ \mathtt{CAT}\ s_2\ \mathsf{exp} \longleftrightarrow \mathtt{cat(}a_1\mathtt{,}a_2\mathtt{)}\ \mathsf{ast}} \tag{7.2j}$$

$$\frac{s\ \mathsf{exp} \longleftrightarrow a\ \mathsf{ast}}{s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{ast}} \tag{7.2k}$$

$$\frac{s_1\ \mathsf{fct} \longleftrightarrow \mathtt{id[}s\mathtt{]}\ \mathsf{ast} \quad s_2\ \mathsf{trm} \longleftrightarrow a_2\ \mathsf{ast} \quad s_3\ \mathsf{prg} \longleftrightarrow a_3\ \mathsf{ast}}{\mathtt{LET}\ s_1\ \mathtt{BE}\ s_2\ \mathtt{IN}\ s_3\ \mathsf{prg} \longleftrightarrow \mathtt{let(id[}s\mathtt{]}\mathtt{,}a_2\mathtt{,}a_3\mathtt{)}\ \mathsf{ast}} \tag{7.2l}$$

A successful parse implies that the token string must have been derived according to the rules of the unambiguous grammar and that the result is a well-formed abstract syntax tree.

**Theorem 7.1.** *1. If $s$* fct $\longleftrightarrow a$ ast*, then $s$* fct *and $a$* ast*.*

*2. If $s$* trm $\longleftrightarrow a$ ast*, then $s$* trm *and $a$* ast*.*

*3. If $s$* exp $\longleftrightarrow a$ ast*, then $s$* exp *and $a$* ast*.*

*4. If $s$* prg $\longleftrightarrow a$ ast*, then $s$* prg *and $a$* ast*.*

Moreover, if a string is generated according to the rules of the grammar, then it has a parse as an ast.

**Theorem 7.2.** *1. If $s$* fct*, then there is a unique $a$ such that $s$* fct $\longleftrightarrow a$ ast*.*

*2. If $s$* trm*, then there is a unique $a$ such that $s$* trm $\longleftrightarrow a$ ast*.*

*3. If $s$* exp*, then there is a unique $a$ such that $s$* exp $\longleftrightarrow a$ ast*.*

*4. If $s$* prg*, then there is a unique $a$ such that $s$* prg $\longleftrightarrow a$ ast*.*

## 7.3 Parsing Into Abstract Binding Trees

The representation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ using abstract syntax trees exposes the hierarchical structure of the language, but does not manage the binding and scope of variables in a `let` expression. In this section we revise the parser given in Section 7.1 on page 56 to translate from token strings (as before) to abstract binding trees to make explicit the binding and scope of identifiers in a program.

The abstract binding tree representation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is specified by the following assignment of (generalized) arities to operators:

$$\begin{aligned}
\mathsf{ar}(\texttt{num}[n]) &= () \\
\mathsf{ar}(\texttt{str}[s]) &= () \\
\mathsf{ar}(\texttt{plus}) &= (0,0) \\
\mathsf{ar}(\texttt{times}) &= (0,0) \\
\mathsf{ar}(\texttt{cat}) &= (0,0) \\
\mathsf{ar}(\texttt{len}) &= (0) \\
\mathsf{ar}(\texttt{let}) &= (0,1)
\end{aligned}$$

The arity of the operator `let` specifies that it takes two arguments, the second of which is an abstractor of valence 1, meaning that it binds one variable in the second argument position. Observe that identifiers are no longer declared as operators; instead, identifiers are translated by the parser into variables.

The revised parsing judgement, $s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{abt}$, between strings $s$ and abt's $a$, is defined by a collection of rules similar to those given in Section 7.2 on page 57. These rules take the form of a generalized inductive definition (see Chapter 2) in which the premises and conclusions of the rules involve hypothetical judgments of the form

$$\texttt{ID}[s_1]\ \mathsf{fct} \longleftrightarrow x_1\ \mathsf{abt}, \ldots, \texttt{ID}[s_n]\ \mathsf{fct} \longleftrightarrow x_n\ \mathsf{abt} \vdash s\ \mathsf{prg} \longleftrightarrow a\ \mathsf{abt},$$

where the $x_i$'s are pairwise distinct variable names. The hypotheses of the judgement dictate how identifiers are to be parsed as variables, for it follows from the reflexivity of the hypothetical judgement that

$$\Gamma, \texttt{ID}[s]\ \mathsf{fct} \longleftrightarrow x\ \mathsf{abt} \vdash \texttt{ID}[s]\ \mathsf{fct} \longleftrightarrow x\ \mathsf{abt}.$$

To maintain the association between identifiers and variables we must ensure that when parsing a `let` expression we update the hypotheses to

record the association between the bound identifier and a corresponding variable:

$$\frac{\Gamma \vdash s_1 \ \mathsf{exp} \longleftrightarrow a_1 \ \mathsf{abt} \quad \Gamma, \mathtt{ID[}s\mathtt{]} \ \mathsf{fct} \longleftrightarrow x \ \mathsf{abt} \vdash s_2 \ \mathsf{prg} \longleftrightarrow a_2 \ \mathsf{abt}}{\Gamma \vdash \mathtt{LET\ ID[}s\mathtt{]\ BE}\, s_1\, \mathtt{IN}\, s_2 \ \mathsf{prg} \longleftrightarrow \mathtt{let}(a_1, x.a_2) \ \mathsf{abt}} \tag{7.3a}$$

We demand that the concrete syntax of a `let` expression specify an identifier in the leading position, and update the hypotheses governing the parsing of this identifier when parsing the body of the `let`.

This elegant formulation of parsing using hypothetical judgements and generalized rules suffers from a shortcoming that must be rectified. What happens if an inner `let` expression binds the *same* identifier as an outer `let` expression? This will lead to the introduction of two hypotheses, say $\mathtt{ID[}s\mathtt{]}\ \mathsf{fct} \longleftrightarrow x_1\ \mathsf{abt}$ and $\mathtt{ID[}s\mathtt{]}\ \mathsf{fct} \longleftrightarrow x_2\ \mathsf{abt}$, for the same identifier, destroying the unicity property of the parsing judgement! Each occurrence of $\mathtt{ID[}s\mathtt{]}$ may be parsed either as the variable $x_1$ or as the variable $x_2$, with no means to control which is chosen at any point.

To rectify this we must resort to less elegant methods. Rather than use hypotheses, we instead maintain an explicit *symbol table* to record the association between identifiers and variables. We must define explicitly the procedures for creating and extending symbol tables, and for looking up an identifier in the symbol table to determine its associated variable. This gives us the freedom to implement the usual shadowing policy for re-used identifiers, according to which the most recent binding governs the association of identifiers and variables.

The main change to the parsing judgement is that the hypothetical judgement

$$\Gamma \vdash s \ \mathsf{prg} \longleftrightarrow a \ \mathsf{abt}$$

is reduced to the categorical judgement

$$s \ \mathsf{prg} \longleftrightarrow a \ \mathsf{abt}\ [\sigma],$$

where $\sigma$ is a symbol table. (Analogous changes must be made to the other parsing judgements.) The symbol table is now an argument to the judgement form, rather than an implicit mechanism for performing inference under hypotheses.

The rule for parsing `let` expressions is then formulated as follows:

$$\frac{s_1 \ \mathsf{exp} \longleftrightarrow a_1 \ \mathsf{abt}\ [\sigma] \quad \sigma' = \sigma[\mathtt{ID[}s\mathtt{]} \mapsto x] \quad s_2 \ \mathsf{prg} \longleftrightarrow a_2 \ \mathsf{abt}\ [\sigma']}{\mathtt{let\ ID[}s\mathtt{]\ be}\, s_1\, \mathtt{in}\, s_2 \ \mathsf{prg} \longleftrightarrow \mathtt{let}(a_1, x.a_2) \ \mathsf{abt}\ [\sigma]} \tag{7.4}$$

This rule is quite similar to the hypothetical form, the difference being that we must manage the symbol table explicitly. In particular, we must include a rule for parsing identifiers, rather than relying on the reflexivity of the hypothetical judgement to do it for us.

$$\frac{\sigma(\texttt{ID[}s\texttt{]}) = x}{\texttt{ID[}s\texttt{]}\ \mathsf{id} \longleftrightarrow x\ \mathsf{name}\ [\sigma]} \tag{7.5}$$

The premise of this rule states that $\sigma$ maps the identifier $\texttt{ID[}s\texttt{]}$ to the variable $x$.

Symbol tables may be defined to be finite sequences of ordered pairs of the form $(\texttt{ID[}s\texttt{]}, x)$, where $\texttt{ID[}s\texttt{]}$ is an identifier and $x$ is a variable name. Using this representation it is straightforward to define the following judgement forms:

| | |
|---|---|
| $\sigma\ \mathsf{symtab}$ | well-formed symbol table |
| $\sigma' = \sigma[\texttt{ID[}s\texttt{]} \mapsto x]$ | add new association |
| $\sigma(\texttt{ID[}s\texttt{]}) = x$ | lookup identifier |

We leave the precise definitions of these judgements as an exercise for the reader.

## 7.4 Informal Conventions

The abstract syntax of a language is often divided into several syntactic categories, rather than consolidated into a single category of abstract syntax. For example, in many languages it is natural to distinguish expressions from types, with types permitted to occur in expressions, but expressions not permitted to occur in types. Formally, this can be achieved by distinguishing two different signatures of operators, say $\Omega_{\mathsf{type}}$ and $\Omega_{\mathsf{expr}}$, and consider types to be abt's over the former signature, and expressions to be abt's over the latter.

In practice we avoid such formal specifications implicit by abusing the short-form grammar notation introduced in Chapter 6 to specify the abstract binding structure of a language. This is best illustrated by example. The abstract syntax of $\mathcal{L}\{\texttt{num str}\}$ may be specified as follows:

$$\begin{array}{lrcl} \text{Type} & \tau & ::= & \texttt{nat} \mid \texttt{str} \\ \text{Expr} & e & ::= & x \mid \texttt{num[}n\texttt{]} \mid \texttt{str[}s\texttt{]} \mid \texttt{plus(}e_1\texttt{,}e_2\texttt{)} \mid \texttt{times(}e_1\texttt{,}e_2\texttt{)} \mid \\ & & & \texttt{cat(}e_1\texttt{,}e_2\texttt{)} \mid \texttt{len(}e\texttt{)} \mid \texttt{let(}e_1\texttt{,}x\texttt{.}e_2\texttt{)} \end{array}$$

This specification is to be read as defining two judgements, $\tau$ type stating that $\tau$ is a well-formed type, and $e$ exp, stating that $e$ is a well-formed expression. The grammar for types specifies two operators of arity $()$, namely `nat` and `str`, and the grammar for expressions specifies two operators of arity $()$, three of arity $(0,0)$, one of arity $(0)$, and one of arity $(0,1)$. The arities are implied by the given form of the typical element in the grammar. For example, the typical element $\texttt{let}(e_1, x.e_2)$ specifies that `let` is an expression operator of arity $(0,1)$. This is because it has two arguments, the second of which is an abstractor of valence one.

Usually it is useful to introduce a more readable form of concrete syntax for each form of abstract syntax. To avoid the complexities of a full specification of the concrete syntax of a language, we instead give a chart showing the correspondence between the abstract and concrete syntax. For example, a concrete syntax for $\mathcal{L}\{\texttt{num str}\}$ may be (incompletely) specified by the following chart:

| Abstract | Concrete |
|---:|---:|
| $\texttt{num}[n]$ | $n$ |
| $\texttt{str}[s]$ | $\texttt{"}s\texttt{"}$ |
| $\texttt{plus}(e_1, e_2)$ | $e_1\texttt{+}e_2$ |
| $\texttt{times}(e_1, e_2)$ | $e_1\texttt{*}e_2$ |
| $\texttt{cat}(e_1, e_2)$ | $e_1\texttt{\^{}}e_2$ |
| $\texttt{len}(e)$ | $\texttt{|}e\texttt{|}$ |
| $\texttt{let}(e_1, x.e_2)$ | $\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2$ |

We occasionally make a further distinction between two forms of concrete syntax, the *implementation syntax*, which is used in a computer implementation, and the *blackboard syntax*, which is used in handwritten contexts.

## 7.5 Exercises

1. Give an unambiguous grammar for numbers, and show how to parse it.

2. Show that the parser may be "run backwards" to obtain an *unparser*, or *pretty printer*. This means that you are to show that the parser has mode $(\exists, \forall)$.

3. Implement symbol tables.

# Part III

# Static and Dynamic Semantics

# Chapter 8

# Static Semantics

The *static semantics* of a language consists of a collection of rules for imposing constraints on the formation of programs, called a *type system*. Phrases of the language are classified by *types*, which govern how they may be used in combination with other phrases. Roughly speaking, the type of a phrase predicts the form of its value, and a phrase is said to be *well-typed* if it is constructed consistently with these predictions. For example, the sum of two expressions of numeric type is itself of numeric type, which expresses the evident fact that the sum of type numbers is itself a number. On the other hand, the sum of an expression of string type with any other expression is *ill-typed*, expressing that addition is undefined on strings.

It is rather straightforward to formulate a type system for simple calculator-like languages which do not involve variable binding. The task becomes more interesting once variables are introduced, for then we must employ hypothetico-general judgements to account for their types. The static semantics of such languages takes the form of a generalized inductive definition, for which we must show that the structural rules are admissible (as described in Chapter 4).

## 8.1 Static Semantics of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$

The static semantics of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ consists of a collection of rules for deriving judgements of the form $e : \tau$, where $e$ $\mathsf{exp}$ and $\tau$ $\mathsf{type}$. The static semantics is specified by a generalized inductive definition of the family of judgements

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

where $\mathcal{X}$ is a variable context consisting of declarations of the form $x$ exp, where $x$ is a variable name, and where $\Gamma$ is a *typing context*, a finite set of hypotheses of the form $x : \tau$, one for each $x$ declared in $\mathcal{X}$.

$$\frac{}{\mathcal{X}, x\ \texttt{exp} \mid \Gamma, x : \tau \vdash x : \tau} \tag{8.1a}$$

$$\frac{}{\mathcal{X} \mid \Gamma \vdash \texttt{str}[s] : \texttt{str}} \tag{8.1b}$$

$$\frac{}{\mathcal{X} \mid \Gamma \vdash \texttt{num}[n] : \texttt{num}} \tag{8.1c}$$

$$\frac{\mathcal{X} \mid \Gamma \vdash e_1 : \texttt{num} \quad \mathcal{X} \mid \Gamma \vdash e_2 : \texttt{num}}{\mathcal{X} \mid \Gamma \vdash \texttt{plus}(e_1, e_2) : \texttt{num}} \tag{8.1d}$$

$$\frac{\mathcal{X} \mid \Gamma \vdash e_1 : \texttt{str} \quad \mathcal{X} \mid \Gamma \vdash e_2 : \texttt{str}}{\mathcal{X} \mid \Gamma \vdash \texttt{cat}(e_1, e_2) : \texttt{str}} \tag{8.1e}$$

$$\frac{\mathcal{X} \mid \Gamma \vdash e_1 : \tau_1 \quad \mathcal{X}, x\ \texttt{exp} \mid \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\mathcal{X} \mid \Gamma \vdash \texttt{let}(e_1, x.e_2) : \tau_2} \tag{8.1f}$$

In practice we usually elide the variable context $\mathcal{X}$ since it can be deduced from the typing context. In that case the foregoing rules would be written as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{8.2a}$$

$$\frac{}{\Gamma \vdash \texttt{str}[s] : \texttt{str}} \tag{8.2b}$$

$$\frac{}{\Gamma \vdash \texttt{num}[n] : \texttt{num}} \tag{8.2c}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash \texttt{plus}(e_1, e_2) : \texttt{num}} \tag{8.2d}$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash \texttt{cat}(e_1, e_2) : \texttt{str}} \tag{8.2e}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1, x.e_2) : \tau_2} \tag{8.2f}$$

## 8.2 Structural Properties

The rules defining the static semantics are chosen so as to make the structural rules given in Chapter 4 admissible. We combine the substitution and transitivity properties into a single, simplified form in view of the fact that if $x$ is not declared in $\Gamma$, then $[a/x]\Gamma = \Gamma$.

**Lemma 8.1** (Admissibility of Structural Rules). *1. If $\mathcal{X} \mid \Gamma \vdash e' : \tau'$ and $x \# \mathcal{X}$, then for any $\tau$ type, $\mathcal{X}, x \text{ exp} \mid \Gamma, x : \tau \vdash e' : \tau'$.*

*2. If $\mathcal{X}, x \text{ exp} \mid \Gamma, x : \tau \vdash e' : \tau'$ and $\mathcal{X} \mid \Gamma \vdash e : \tau$, then $\mathcal{X} \mid \Gamma \vdash [e/x]e' : \tau'$.*

*Proof.* 1. By induction on the derivation of $\mathcal{X} \mid \Gamma \vdash e' : \tau'$. We will give one case here, for rule (8.1f). We have that $e' = \texttt{let}(e_1, z.e_2)$, where we may assume $z$ is chosen such that $z \# \mathcal{X}$ and $z \# x$. By induction we have

(a) $\mathcal{X}, x \text{ exp} \mid \Gamma, x : \tau \vdash e_1 : \tau_1$,

(b) $\mathcal{X}, x \text{ exp}, z \text{ exp} \mid \Gamma, x : \tau, z : \tau_1 \vdash e_2 : \tau'$,

from which the result follows by Rule (8.1f).

2. By induction on the derivation of $\mathcal{X}, x \text{ exp} \mid \Gamma, x : \tau \vdash e' : \tau'$. We again consider only rule (8.1f). As in the preceding case, $e' = \texttt{let}(e_1, z.e_2)$, where $z$ is chosen so that $z \# x$, $z \# \mathcal{X}$, and $z \# e$. We have by induction

(a) $\mathcal{X} \mid \Gamma \vdash [e/x]e_1 : \tau_1$,

(b) $\mathcal{X}, z \text{ exp} \mid \Gamma, z : \tau_1 \vdash [e/x]e_2 : \tau'$.

Since we have chosen $z$ such that $z \# e$, we have

$$[e/x]\texttt{let}(e_1, z.e_2) = \texttt{let}([e/x]e_1, z.[e/x]e_2).$$

It follows by Rule (8.1f) that $\mathcal{X} \mid \Gamma \vdash [e/x]\texttt{let}(e_1, z.e_2) : \tau$, as desired. □

The typing rules are *syntax-directed* in the sense that there is exactly one rule for each form of expression. Consequently, we obtain the following inversion properties for typing, which state that the typing rules are necessary, as well as sufficient, for each form of expression.

**Lemma 8.2** (Inversion for Typing). *Suppose that $\mathcal{X} \mid \Gamma \vdash e : \tau$.*

*1. if $e = x$, then $\mathcal{X} \vdash x \text{ exp}$ and $\mathcal{X} \mid \Gamma \vdash x : \tau$.*

2. *if* $e = \texttt{num[}n\texttt{]}$, *then* $\tau = \texttt{num}$.
3. *if* $e = \texttt{str[}s\texttt{]}$, *then* $\tau = \texttt{str}$.
4. *if* $e = \texttt{plus}(e_1, e_2)$ *or* $e = \texttt{times}(e_1, e_2)$, *then* $\tau = \texttt{num}$, $\mathcal{X} \mid \Gamma \vdash e_1 : \texttt{num}$, *and* $\mathcal{X} \mid \Gamma \vdash e_2 : \texttt{num}$.
5. *if* $e = \texttt{cat}(e_1, e_2)$, *then* $\tau = \texttt{str}$, $\mathcal{X} \mid \Gamma \vdash e_1 : \texttt{str}$, *and* $\mathcal{X} \mid \Gamma \vdash e_2 : \texttt{str}$.
6. *if* $e = \texttt{len}(e_1)$, *then* $\tau = \texttt{num}$ *and* $\mathcal{X} \mid \Gamma \vdash e_1 : \texttt{str}$.
7. *if* $e = \texttt{let}(e_1, x.e_2)$, *then* $\mathcal{X} \mid \Gamma \vdash e_1 : \tau_1$ *for some* $\tau_1$ type, *and* $\mathcal{X}, x$ exp $\mid \Gamma, x : \tau_1 \vdash e_2 : \tau$.

*Proof.* These may all be proved by induction on the derivation of the typing judgement $\mathcal{X} \mid \Gamma \vdash e : \tau$. □

## 8.3 Exercises

1. Show that the expression $e = \texttt{plus(num[7],str[}abc\texttt{])}$ is ill-typed in that there is no $\tau$ such that $e : \tau$.

# Chapter 9

# Dynamic Semantics

The *dynamic semantics* of a language specifies how programs are to be executed. One important method for specifying dynamic semantics is called *structural semantics*, which consists of a collection of rules defining a transition system whose states are closed expressions. *Contextual semantics* may be viewed as an alternative presentation of the structural semantics of a language. Another important method for specifying dynamic semantics, called *evaluation semantics*, is the subject of Chapter 11.

## 9.1 Structural Semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$

A structural semantics for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ consists of a transition system whose states are closed expressions. Every closed expression is an initial state. The final states are the *closed values*, as defined by the following rules:

$$\frac{}{\texttt{num[}n\texttt{]}\ \mathsf{val}} \tag{9.1a}$$

$$\frac{}{\texttt{str[}s\texttt{]}\ \mathsf{val}} \tag{9.1b}$$

The transition judgement, $e \mapsto e'$, is also inductively defined.

$$\frac{n_1 + n_2 = n\ \mathsf{nat}}{\texttt{plus(num[}n_1\texttt{],num[}n_2\texttt{])} \mapsto \texttt{num[}n\texttt{]}} \tag{9.2a}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{plus(}e_1\texttt{,}e_2\texttt{)} \mapsto \texttt{plus(}e_1'\texttt{,}e_2\texttt{)}} \tag{9.2b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{plus}(e_1, e_2) \mapsto \texttt{plus}(e_1, e_2')} \tag{9.2c}$$

$$\frac{s_1\,\hat{}\,s_2 = s\ \mathsf{str}}{\texttt{cat}(\texttt{str}[s_1], \texttt{str}[s_2]) \mapsto \texttt{str}[s]} \tag{9.2d}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{cat}(e_1, e_2) \mapsto \texttt{cat}(e_1', e_2)} \tag{9.2e}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{cat}(e_1, e_2) \mapsto \texttt{cat}(e_1, e_2')} \tag{9.2f}$$

$$\frac{e_1\ \mathsf{val}}{\texttt{let}(e_1, x.e_2) \mapsto [e_1/x]e_2} \tag{9.2g}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1, x.e_2) \mapsto \texttt{let}(e_1', x.e_2)} \tag{9.2h}$$

We have omitted rules for multiplication and computing the length of a string, which follow a similar pattern. Rules (9.2a), (9.2d), and (9.2g) are *instruction transitions*, since they correspond to the primitive steps of evaluation. The remaining rules are *search transitions* that determine the order in which instructions are executed.

When defined using structural semantics, a derivation sequence has a "two-dimensional" structure, with the number of steps in the sequence being its "width" and the derivation tree for each step being its "depth." For example, consider the following evaluation sequence.

$$\begin{aligned}
&\texttt{let(plus(num[1],num[2]),}x\texttt{.plus(plus(}x\texttt{,num[3]),num[4]))}\\
&\quad\mapsto\ \texttt{let(num[3],}x\texttt{.plus(plus(}x\texttt{,num[3]),num[4]))}\\
&\quad\mapsto\ \texttt{plus(plus(num[3],num[3]),num[4])}\\
&\quad\mapsto\ \texttt{plus(num[6],num[4])}\\
&\quad\mapsto\ \texttt{num[10]}
\end{aligned}$$

Each step in this sequence of transitions is justified by a derivation according to Rules (9.2). For example, the third transition in the preceding example is justified by the following derivation:

$$\frac{\dfrac{}{\texttt{plus(num[3],num[3])} \mapsto \texttt{num[6]}}\ (9.2a)}{\texttt{plus(plus(num[3],num[3]),num[4])} \mapsto \texttt{plus(num[6],num[4])}}\ (9.2b)$$

The other steps are similarly justified by a composition of rules.

Since the transition judgement is inductively defined, we may reason about it using rule induction. Specifically, to show that $P(e, e')$ holds whenever $e \mapsto e'$, it is sufficient to show that $P$ is closed under the rules defining the transition judgement. For example, it is a simple matter to show by rule induction that the transition judgement for evaluation of expressions is deterministic: if $e \mapsto e'$ and $e \mapsto e''$, then $e'$ is $e''$.

## 9.2 Contextual Semantics of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$

A variant of structural semantics, called *contextual semantics*, is sometimes useful. There is no fundamental difference between the two approaches, only a difference in the style of presentation. The main idea is to isolate instruction steps as a special form of judgement, called *instruction transition*, and to formalize the process of locating the next instruction using a device called an *evaluation context*. The judgement, $e\ \mathsf{val}$, defining whether an expression is a value, remains unchanged.

The instruction transition judgement, $e_1 \rightsquigarrow e_2$, for $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ is defined by the following rules, together with similar rules for multiplication of numbers and the length of a string.

$$\frac{m + n = p\ \mathsf{nat}}{\mathtt{plus(num[}m\mathtt{],num[}n\mathtt{])} \rightsquigarrow \mathtt{num[}p\mathtt{]}} \tag{9.3a}$$

$$\frac{s \hat{} t = u\ \mathsf{str}}{\mathtt{cat(str[}s\mathtt{],str[}t\mathtt{])} \rightsquigarrow \mathtt{str[}u\mathtt{]}} \tag{9.3b}$$

$$\frac{e_1\ \mathsf{val}}{\mathtt{let}(e_1, x\,.\,e_2) \rightsquigarrow [e_1/x]e_2} \tag{9.3c}$$

The left-hand side of each instruction is called a *redex*, and the corresponding right-hand side is called its *contractum*.

The judgement $\mathcal{E}\ \mathsf{ectxt}$ determines the location of the next instruction to execute in a larger expression. The position of the next instruction step is specified by a "hole", written $\circ$, into which the next instruction is placed, as we shall detail shortly.

$$\frac{}{\circ\ \mathsf{ectxt}} \tag{9.4a}$$

$$\frac{\mathcal{E}_1\ \mathsf{ectxt}}{\mathtt{plus}(\mathcal{E}_1, e_2)\ \mathsf{ectxt}} \tag{9.4b}$$

$$\frac{e_1\ \mathsf{val} \quad \mathcal{E}_2\ \mathsf{ectxt}}{\mathtt{plus}(e_1, \mathcal{E}_2)\ \mathsf{ectxt}} \tag{9.4c}$$

$$\frac{\mathcal{E}_1\ \mathsf{ectxt}}{\texttt{cat}(\mathcal{E}_1, e_2)\ \mathsf{ectxt}} \tag{9.4d}$$

$$\frac{e_1\ \mathsf{val} \quad \mathcal{E}_2\ \mathsf{ectxt}}{\texttt{cat}(e_1, \mathcal{E}_2)\ \mathsf{ectxt}} \tag{9.4e}$$

$$\frac{\mathcal{E}_1\ \mathsf{ectxt}}{\texttt{let}(\mathcal{E}_1, x\,.\,e_2)\ \mathsf{ectxt}} \tag{9.4f}$$

The first rule for evaluation contexts specifies that the next instruction may occur "here", at the point of the occurrence of the hole. The remaining rules correspond one-for-one to the search rules of the structural semantics. For example, Rule (9.4c) states that in an expression $\texttt{plus}(e_1, e_2)$, if the first principal argument, $e_1$, is a value, then the next instruction step, if any, lies at or within the second principal argument, $e_2$.

An evaluation context is to be thought of as a template that is instantiated by replacing the hole with an instruction to be executed. The judgement $e' = \mathcal{E}\{e\}$ states that the expression $e'$ is the result of filling the hole in the evaluation context $\mathcal{E}$ with the expression $e$. It is inductively defined by the following rules:

$$\frac{}{e = \circ\{e\}} \tag{9.5a}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\texttt{plus}(e_1, e_2) = \texttt{plus}(\mathcal{E}_1, e_2)\{e\}} \tag{9.5b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 = \mathcal{E}_2\{e\}}{\texttt{plus}(e_1, e_2) = \texttt{plus}(e_1, \mathcal{E}_2)\{e\}} \tag{9.5c}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\texttt{plus}(e_1, e_2) = \texttt{cat}(\mathcal{E}_1, e_2)\{e\}} \tag{9.5d}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 = \mathcal{E}_2\{e\}}{\texttt{plus}(e_1, e_2) = \texttt{cat}(e_1, \mathcal{E}_2)\{e\}} \tag{9.5e}$$

$$\frac{e_1 = \mathcal{E}_1\{e\}}{\texttt{let}(e_1, x\,.\,e_2) = \texttt{let}(\mathcal{E}_1, x\,.\,e_2)\{e\}} \tag{9.5f}$$

There is one rule for each form of evaluation context. Filling the hole with $e$ results in $e$; otherwise we proceed inductively over the structure of the evaluation context.

Finally, the dynamic semantics for $\mathcal{L}\{\texttt{num str}\}$ is defined using contextual semantics by a single rule:

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \rightsquigarrow e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \mapsto e'} \tag{9.6}$$

In words, a transition from $e$ to $e'$ consists of (1) decomposing $e$ into an evaluation context and an instruction, (2) execution of that instruction, and (3) replacing the instruction by the result of its execution in the same spot within $e$ to obtain $e'$.

The structural and contextual semantics define the same transition relation. For the sake of the proof, let us write $e \mapsto_{\mathsf{s}} e'$ for the transition relation defined by the structural semantics (Rules (9.2)), and $e \mapsto_{\mathsf{c}} e'$ for the transition relation defined by the contextual semantics (Rules (9.6)).

**Theorem 9.1.** *$e \mapsto_{\mathsf{s}} e'$ if, and only if, $e \mapsto_{\mathsf{c}} e'$.*

*Proof.* From left to right, proceed by rule induction on Rules (9.2). It is enough in each case to exhibit an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \rightsquigarrow e_0'$. For example, for Rule (9.2a), take $\mathcal{E} = \circ$, and observe that $e \rightsquigarrow e'$. For Rule (9.2b), we have by induction that there exists an evaluation context $\mathcal{E}_1$ such that $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_0 \rightsquigarrow e_0'$. Take $\mathcal{E} = \texttt{plus}(\mathcal{E}_1, e_2)$, and observe that $e = \texttt{plus}(\mathcal{E}_1, e_2)\{e_0\}$ and $e' = \texttt{plus}(\mathcal{E}_1, e_2)\{e_0'\}$ with $e_0 \rightsquigarrow e_0'$.

From right to left, observe that if $e \mapsto_{\mathsf{c}} e'$, then there exists an evaluation context $\mathcal{E}$ such that $e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \rightsquigarrow e_0'$. We prove by induction on Rules (9.5) that $e \mapsto_{\mathsf{s}} e'$. For example, for Rule (9.5a), $e_0$ is $e$, $e_0'$ is $e'$, and $e \rightsquigarrow e'$. Hence $e \mapsto_{\mathsf{s}} e'$. For Rule (9.5b), we have that $\mathcal{E} = \texttt{plus}(\mathcal{E}_1, e_2)$, $e_1 = \mathcal{E}_1\{e_0\}$, $e_1' = \mathcal{E}_1\{e_0'\}$, and $e_1 \mapsto_{\mathsf{s}} e_1'$. Therefore $e$ is $\texttt{plus}(e_1, e_2)$, $e'$ is $\texttt{plus}(e_1', e_2)$, and therefore by Rule (9.2b), $e \mapsto_{\mathsf{s}} e'$.

□

Since the two transition judgements coincide, contextual semantics may be seen as an alternative way of presenting a structural semantics. It has two advantages over structural semantics, one relatively superficial, one rather less so. The superficial advantage stems from writing Rule (9.6) in the simpler form

$$\frac{e_0 \rightsquigarrow e_0'}{\mathcal{E}\{e_0\} \mapsto \mathcal{E}\{e_0'\}}\ . \tag{9.7}$$

This formulation is simpler insofar as it leaves implicit the definition of the decomposition of the left- and right-hand sides. The deeper advantage, which we will exploit in Chapter 15, is that the transition judgement

in contextual semantics applies only to closed expressions of a fixed type, whereas in structural semantics transition is defined over expressions of every type.

## 9.3 Exercises

1. Prove that if $e \mapsto e_1$ and $e \mapsto e_2$, then $e_1 =_\alpha e_2$.

2. Formulate a variation of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with both a by-name and a by-value `let` construct.

# Chapter 10

# Type Safety

Most contemporary programming languages are *safe* (or, *type safe*, or *strongly typed*). Informally, this means that certain kinds of mismatches cannot arise during execution. For example, type safety for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ states that it will never arise that a number is to be added to a string, or that two numbers are to be concatenated, neither of which is meaningful.

In general type safety expresses the coherence between the static and the dynamic semantics. The static semantics may be seen as predicting that the value of an expression will have a certain form so that the dynamic semantics of that expression is well-defined. Consequently, evaluation cannot "get stuck" in a state for which no transition is possible, corresponding in implementation terms to the absence of "illegal instruction" errors at execution time. This is proved by showing that each step of transition preserves typability and by showing that typable states are well-defined. Consequently, evaluation can never "go off into the weeds," and hence can never encounter an illegal instruction.

More precisely, type safety for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ may be stated as follows:

**Theorem 10.1** (Type Safety)**.** *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

*2. If $e : \tau$, then either $e$ $\mathsf{val}$, or there exists $e'$ such that $e \mapsto e'$.*

The first part, called *preservation*, says that the steps of evaluation preserve typing; the second, called *progress*, ensures that well-typed expressions are either values or can be further evaluated. Safety is the conjunction of preservation and progress.

We say that an expression, $e$, is *stuck* iff it is not a value, yet there is no $e'$ such that $e \mapsto e'$. It follows from the safety theorem that a stuck state is

necessarily ill-typed. Or, putting it the other way around, that well-typed states do not get stuck.

## 10.1 Preservation

The preservation theorem for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ defined in Chapters 8 and 9 is proved by rule induction on the transition system (rules (9.2)).

**Theorem 10.2** (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

*Proof.* We will consider two cases, leaving the rest to the reader. Consider rule (9.2b),

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

Assume that $\texttt{plus}(e_1, e_2) : \tau$. By inversion for typing, we have that $\tau = \texttt{num}$, $e_1 : \texttt{num}$, and $e_2 : \texttt{num}$. By induction we have that $e_1' : \texttt{num}$, and hence $\texttt{plus}(e_1', e_2) : \texttt{num}$. The case for concatenation is handled similarly.

Now consider rule (9.2g),

$$\frac{e_1\ \mathsf{val}}{\texttt{let}(e_1, x.e_2) \mapsto [e_1/x]e_2}\ .$$

Assume that $\texttt{let}(e_1, x.e_2) : \tau_2$. By the inversion lemma 8.2 on page 67, $e_1 : \tau_1$ for some $\tau_1$ such that $x : \tau_1 \vdash e_2 : \tau_2$. By the substitution lemma 8.1 on page 67 $[e_1/x]e_2 : \tau_2$, as desired. □

The proof of preservation must proceed by rule induction on the rules defining the transition judgement. It cannot, for example, proceed by induction on the structure of $e$, for in most cases there is more than one transition rule for each expression form. Nor can it be proved by induction on the typing rules, for in the case of the `let` rule, the context is enriched to consider an open term, to which no dynamic semantics is assigned.

## 10.2 Progress

The progress theorem captures the idea that well-typed programs cannot "get stuck". The proof depends crucially on the following lemma, which characterizes the values of each type.

**Lemma 10.3** (Canonical Forms)**.** *If $e\ \mathsf{val}$ and $e : \tau$, then*

1. *If $\tau = \texttt{num}$, then $e = \texttt{num[}n\texttt{]}$ for some number $n$.*
2. *If $\tau = \texttt{str}$, then $e = \texttt{str[}s\texttt{]}$ for some string $s$.*

*Proof.* By induction on rules (8.1) and (9.1). □

Progress is proved by rule induction on rules (8.1) defining the static semantics of the language.

**Theorem 10.4** (Progress)**.** *If $e : \tau$, then either $e$ $\mathsf{val}$, or there exists $e'$ such that $e \mapsto e'$.*

*Proof.* The proof proceeds by induction on the typing derivation. We will consider only one case, for rule (8.1d),

$$\frac{e_1 : \texttt{num} \quad e_2 : \texttt{num}}{\texttt{plus(}e_1\texttt{,}e_2\texttt{)} : \texttt{num}},$$

where the context is empty because we are considering only closed terms.

By induction we have that either $e_1$ $\mathsf{val}$, or there exists $e_1'$ such that $e_1 \mapsto e_1'$. In the latter case it follows that $\texttt{plus(}e_1\texttt{,}e_2\texttt{)} \mapsto \texttt{plus(}e_1'\texttt{,}e_2\texttt{)}$, as required. In the former we also have by induction that either $e_2$ $\mathsf{val}$, or there exists $e_2'$ such that $e_2 \mapsto e_2'$. In the latter case we have that $\texttt{plus(}e_1\texttt{,}e_2\texttt{)} \mapsto \texttt{plus(}e_1\texttt{,}e_2'\texttt{)}$, as required. In the former, we have, by the Canonical Forms Lemma 10.3 on the facing page, $e_1 = \texttt{num[}n_1\texttt{]}$ and $e_2 = \texttt{num[}n_2\texttt{]}$, and hence

$$\texttt{plus(num[}n_1\texttt{],num[}n_2\texttt{])} \mapsto \texttt{num[}n_1 + n_2\texttt{]}.$$

□

Since the typing rules for expressions are syntax-directed, the progress theorem could equally well be proved by induction on the structure of $e$, appealing to the inversion theorem at each step to characterize the types of the parts of $e$. But this approach breaks down when the typing rules are not syntax-directed, that is, when there may be more than one rule for a given expression form. No difficulty arises if the proof proceeds by induction on the typing rules.

Summing up, the combination of preservation and progress together constitute the proof of safety. The progress theorem ensures that well-typed expressions do not "get stuck" in an ill-defined state, and the preservation theorem ensures that if a step is taken, the result remains well-typed (with the same type). Thus the two parts work hand-in-hand to ensure that the static and dynamic semantics are coherent, and that no ill-defined states can ever be encountered while evaluating a well-typed expression.

## 10.3 Run-Time Errors

Suppose that we wish to extend $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with, say, a quotient operation that is undefined for a zero divisor. The natural typing rule for quotients is given by the following rule:

$$\frac{e_1 : \texttt{num} \quad e_2 : \texttt{num}}{\texttt{div}(e_1, e_2) : \texttt{num}}\,.$$

But the expression `div(num[3],num[0])` is well-typed, yet stuck! We have two options to correct this situation:

1. Enhance the type system, so that no well-typed program may divide by zero.
2. Add dynamic checks, so that division by zero signals an error as the outcome of evaluation.

Either option is, in principle, viable, but the most common approach is the second. The first requires that the type checker prove that an expression be non-zero before permitting it to be used in the denominator of a quotient. It is difficult to do this without ruling out too many programs. For now we consider the second option, which is widely used.

The general idea is to distinguish *checked* from *unchecked* errors. An unchecked error is one that is ruled out by the type system. No run-time checking is performed to ensure that such an error does not occur, because the type system rules out the possibility of it arising. For example, the dynamic semantics need not check, when performing an addition, that its two arguments are, in fact, numbers, as opposed to strings, because the type system ensures that this is the case. On the other hand the dynamic semantics for quotient *must* check for a zero divisor, because the type system does not rule out the possibility.

Checked errors are modeled by adding to $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ a new construct, `error`, which signals the occurrence of a checked error. The typing rule for a checked error permits it to be regarded as having any type at all, because it aborts evaluation, and hence cannot return to the calling context.

$$\frac{}{\texttt{error} : \tau} \tag{10.1}$$

The dynamic semantics is augmented with rules that provoke a checked error (such as division by zero), plus rules that propagate the error through other language constructs.

$$\frac{}{\texttt{div}(v_1, \texttt{num[0]}) \mapsto \texttt{error}} \tag{10.2a}$$

$$\frac{}{\texttt{plus(error,}e_2\texttt{)} \mapsto \texttt{error}} \tag{10.2b}$$

$$\frac{}{\texttt{plus(}v_1\texttt{,error)} \mapsto \texttt{error}} \tag{10.2c}$$

There are similar error propagation rules for the other constructs of the language. Finally, we define $e$ err to hold exactly when $e = \texttt{error}$.

The preservation theorem remains the same, and is proved similarly, bearing in mind that `error` has any type we like. The statement (and proof) of progress is modified to permit the possibility of a checked error as the outcome of evaluation.

**Theorem 10.5** (Progress With Error)**.** *If $e : \tau$, then either $e$* err*, or $e$* val*, or there exists $e'$ such that $e \mapsto e'$.*

*Proof.* The proof is by induction on typing, and proceeds similarly to the proof given earlier, except that there are now three cases to consider at each point in the proof. □

## 10.4 Exercises

1. Complete the proof of preservation.
2. Complete the proof of progress.

# Chapter 11

# Evaluation Semantics

In Chapter 9 we defined the dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ using the method of structural semantics. This approach is useful as a foundation for proving properties of a language, but other methods are often more appropriate for other purposes, such as writing user manuals. Another method, called *evaluation semantics*, or *ES*, presents the dynamic semantics as a relation between a phrase and its value, without detailing how it is to be determined in a step-by-step manner. Two variants of evaluation semantics are also considered, namely *environment semantics*, which delays substitution, and *cost semantics*, which records the number of steps that are required to evaluation an expression.

## 11.1 Evaluation Semantics

Another method for defining the dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, called *evaluation semantics*, consists of an inductive definition of the evaluation judgement, $e \Downarrow v$, specifying the value, $v$, of a closed expression, $e$.

$$\frac{}{\texttt{num[}n\texttt{]} \Downarrow \texttt{num[}n\texttt{]}} \tag{11.1a}$$

$$\frac{}{\texttt{str[}s\texttt{]} \Downarrow \texttt{str[}s\texttt{]}} \tag{11.1b}$$

$$\frac{e_1 \Downarrow \texttt{num[}n_1\texttt{]} \quad e_2 \Downarrow \texttt{num[}n_2\texttt{]} \quad n_1 + n_2 = n\ \mathsf{nat}}{\texttt{plus(}e_1\texttt{,}e_2\texttt{)} \Downarrow \texttt{num[}n\texttt{]}} \tag{11.1c}$$

$$\frac{e_1 \Downarrow \texttt{str[}s_1\texttt{]} \quad e_2 \Downarrow \texttt{str[}s_2\texttt{]} \quad s_1\,\hat{}\,s_2 = s\ \mathsf{str}}{\texttt{cat(}e_1\texttt{,}e_2\texttt{)} \Downarrow \texttt{str[}s\texttt{]}} \tag{11.1d}$$

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v_2}{\texttt{let}(e_1, x\,.\,e_2) \Downarrow v_2} \tag{11.1e}$$

The value of a `let` expression is determined by the value of its binding, and the value of the corresponding substitution instance of its body. Since the substitution instance is not a sub-expression of the `let`, the rules are not syntax-directed.

Since the evaluation judgement is inductively defined, it has associated with it a principle of proof by rule induction. Specifically, to show that the property $P(e, v)$ holds, it is enough to show that $P$ is closed under the rules defining the evaluation judgement. Specifically, our proof obligations are:

1. Show that $P(\texttt{num}[n], \texttt{num}[n])$.
2. Show that $P(\texttt{str}[s], \texttt{str}[s])$.
3. Show that $P(\texttt{plus}(e_1, e_2), \texttt{num}[n])$, assuming $n_1 + n_2 = n$ nat, $P(e_1, \texttt{num}[n_1])$ and $P(e_2, \texttt{num}[n_2])$.
4. Show that $P(\texttt{cat}(e_1, e_2), \texttt{str}[s])$, assuming $s_1 \hat{} s_2 = s$ str, $P(e_1, \texttt{str}[s_1])$ and $P(e_2, \texttt{str}[s_2])$.
5. Show that $P(\texttt{let}(e_1, x\,.\,e_2), v_2)$, assuming $P(e_1, v_1)$ and $P([v_1/x]e_2, v_2)$.

This induction principle is *not* the same as structural induction on $e$, because the evaluation rules are not syntax-directed.

It is possible to show by a straightforward rule induction on evaluation that if $e \Downarrow v$, then $v$ val.

## 11.2 Relating Transition and Evaluation Semantics

We have given two different forms of dynamic semantics for $\mathcal{L}\{\texttt{num str}\}$. It is natural to ask whether they are equivalent, but to do so first requires that we consider carefully what we mean by equivalence. The transition semantics describes a step-by-step process of execution, whereas the evaluation semantics suppresses the intermediate states, focussing attention on the initial and final states alone. This suggests that the appropriate correspondence is between *complete* execution sequences in the transition semantics and the evaluation judgement in the evaluation semantics. (We will consider only numeric expressions, but analogous results hold also for string-valued expressions.)

**Theorem 11.1.** *For all closed expressions $e$ and natural numbers $n$, $e \mapsto^* \mathtt{num[}n\mathtt{]}$ iff $e \Downarrow \mathtt{num[}n\mathtt{]}$.*

How might we prove such a theorem? We will consider each direction separately. We consider the easier case first.

**Lemma 11.2.** *If $e \Downarrow \mathtt{num[}n\mathtt{]}$, then $e \mapsto^* \mathtt{num[}n\mathtt{]}$.*

*Proof.* By induction on the definition of the evaluation judgement. For example, suppose that $\mathtt{plus}(e_1, e_2) \Downarrow \mathtt{num}[n]$ by the rule for evaluating additions. By induction we know that $e_1 \mapsto^* \mathtt{num}[n_1]$ and $e_2 \mapsto^* \mathtt{num}[n_2]$. We reason as follows:

$$\begin{array}{rcl} \mathtt{plus}(e_1, e_2) & \mapsto^* & \mathtt{plus}(\mathtt{num}[n_1], e_2) \\ & \mapsto^* & \mathtt{plus}(\mathtt{num}[n_1], \mathtt{num}[n_2]) \\ & \mapsto & \mathtt{num}[n_1 + n_2] \end{array}$$

Therefore $\mathtt{plus}(e_1, e_2) \mapsto^* \mathtt{num}[n_1 + n_2]$, as required. The other cases are handled similarly. □

For the converse, recall from Chapter 5 the definitions of multi-step evaluation and complete evaluation. Since $\mathtt{num}[n] \Downarrow \mathtt{num}[n]$, it suffices to show that evaluation is closed under head expansion.

**Lemma 11.3.** *If $e \mapsto e'$ and $e' \Downarrow \mathtt{num[}n\mathtt{]}$, then $e \Downarrow \mathtt{num[}n\mathtt{]}$.*

*Proof.* By induction on the definition of the transition judgement. For example, suppose that $\mathtt{plus}(e_1, e_2) \mapsto \mathtt{plus}(e_1', e_2)$, where $e_1 \mapsto e_1'$. Suppose further that $\mathtt{plus}(e_1', e_2) \Downarrow \mathtt{num}[n]$, so that $e_1' \Downarrow \mathtt{num}[n_1]$, and $e_2 \Downarrow \mathtt{num}[n_2]$ and $n_1 + n_2 = n$ nat. By induction $e_1 \Downarrow \mathtt{num}[n_1]$, and hence $\mathtt{plus}(e_1, e_2) \Downarrow \mathtt{num}[n]$, as required. □

## 11.3 Environment Semantics

Both the transition semantics and the evaluation semantics given earlier rely on substitution to replace `let`-bound variables by their bindings during evaluation. This approach maintains the invariant that only closed expressions are ever considered. However, in practice, we do not perform substitution, but rather record the bindings of variables in some sort of data structure. In this section we show how this can be elegantly modeled using hypothetical judgements.

The basic idea is to consider hypotheses of the form $x \Downarrow v$, where $x$ is a variable and $v$ is a value, such that no two hypotheses govern the same

variable. Let $\Theta$ range over finite sets of such hypotheses, which we call an *environment*. We will consider judgements of the form $\mathcal{X} \mid \Theta \vdash e \Downarrow v$, where $\mathcal{X}$ is the finite set of variables appearing on the left of a hypothesis in $\Theta$. As usual, we will suppress explicit mention of the parameter set $\mathcal{X}$, and simply write $\Theta \vdash e \Downarrow v$.

$$\frac{}{\Theta, x \Downarrow v \vdash x \Downarrow v} \tag{11.2a}$$

$$\frac{\Theta \vdash e_1 \Downarrow \mathtt{num}[n_1] \quad \Theta \vdash e_2 \Downarrow \mathtt{num}[n_2]}{\Theta \vdash \mathtt{plus}(e_1, e_2) \Downarrow \mathtt{num}[n_1 + n_2]} \tag{11.2b}$$

$$\frac{\Theta \vdash e_1 \Downarrow \mathtt{str}[s_1] \quad \Theta \vdash e_2 \Downarrow \mathtt{str}[s_2]}{\Theta \vdash \mathtt{cat}(e_1, e_2) \Downarrow \mathtt{str}[s_1 \hat{} s_2]} \tag{11.2c}$$

$$\frac{\Theta \vdash e_1 \Downarrow v_1 \quad \Theta, x \Downarrow v_1 \vdash e_2 \Downarrow v_2}{\Theta \vdash \mathtt{let}(e_1, x.e_2) \Downarrow v_2} \tag{11.2d}$$

The variable rule is an instance of the reflexivity rule for hypothetical judgements, and therefore need not be explicitly stated. We nevertheless include it here for clarity. The `let` rule augments the environment with a new assumption governing the bound variable (which, by $\alpha$-conversion, may be chosen to be distinct from any other variable currently in $\Theta$ to preserve the invariant that no two assumptions govern the same variable).

The environment semantics is related to the evaluation semantics by the following theorem:

**Theorem 11.4.** *$x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$ iff $[v_1, \ldots, v_n / x_1, \ldots, x_n]e \Downarrow v$.*

*Proof.* The left to right direction is proved by induction on the rules defining the evaluation semantics, making use of the definition of substitution and the definition of the evaluation semantics for closed expressions. The converse is proved by induction on the structure of $e$, again making use of the definition of substitution. Note that we must induct on $e$ in order to detect occurrences of variables $x_i$ in $e$, which are governed by a hypothesis in the environment semantics. □

## 11.4 Cost Semantics

A structural semantics provides a natural notion of *time complexity* for programs, namely the number of steps required to reach a final state. An evaluation semantics, on the other hand, does not provide such a direct notion of

complexity. Since the individual steps required to complete an evaluation are suppressed, we cannot directly read off the number of steps required to evaluate to a value. Instead we must augment the evaluation relation with a cost measure, resulting in a *cost semantics*.

Evaluation judgements have the form $e \Downarrow^n v$, with the meaning that $e$ evaluates to $v$ in $n$ steps.

$$\frac{}{\texttt{num[}n\texttt{]} \Downarrow^0 \texttt{num[}n\texttt{]}} \tag{11.3a}$$

$$\frac{e_1 \Downarrow^{k_1} \texttt{num[}n_1\texttt{]} \quad e_2 \Downarrow^{k_2} \texttt{num[}n_2\texttt{]}}{\texttt{plus(}e_1\texttt{,}e_2\texttt{)} \Downarrow^{k_1+k_2+1} \texttt{num[}n_1 + n_2\texttt{]}} \tag{11.3b}$$

$$\frac{}{\texttt{str[}s\texttt{]} \Downarrow^0 \texttt{str[}s\texttt{]}} \tag{11.3c}$$

$$\frac{e_1 \Downarrow^{k_1} s_1 \quad e_2 \Downarrow^{k_2} s_2}{\texttt{cat(}e_1\texttt{,}e_2\texttt{)} \Downarrow^{k_1+k_2+1} \texttt{str[}s_1 \texttt{\^{}} s_2\texttt{]}} \tag{11.3d}$$

$$\frac{e_1 \Downarrow^{k_1} v_1 \quad [v_1/x]e_2 \Downarrow^{k_2} v_2}{\texttt{let(}e_1\texttt{,}x\texttt{.}e_2\texttt{)} \Downarrow^{k_1+k_2+1} v_2} \tag{11.3e}$$

**Theorem 11.5.** *For any closed expression $e$ and closed value $v$ of the same type, $e \Downarrow^k v$ iff $e \mapsto^k v$.*

*Proof.* From left to right proceed by rule induction on the definition of the cost semantics. From right to left proceed by induction on $k$, with an inner rule induction on the definition of the transition semantics. □

## 11.5 Type Safety, Revisited

The type safety theorem for $\mathcal{L}\{\texttt{num str}\}$ (Theorem 10.1 on page 75) states that a language is safe iff it satisfies both preservation and progress. This formulation depends critically on the use of a transition system to specify the dynamic semantics. But what if had instead specified the dynamic semantics as an evaluation relation, instead of using a transition system? Can we state and prove safety in such a setting?

The answer, unfortunately, is that we cannot. While there is an analogue of the preservation property for an evaluation semantics, there is no clear analogue of the progress property. Preservation may be stated as saying

that if $e \Downarrow v$ and $e : \tau$, then $v : \tau$. This can be readily proved by induction on the evaluation rules. But what is the analogue of progress? One might be tempted to phrase progress as saying that if $e : \tau$, then $e \Downarrow v$ for some $v$. While this is property true for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, it demands much more than just progress — it requires that every expression evaluate to a value. But if $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ were extended to admit operations that may result in an error (as discussed in Section 10.3 on page 78), or to admit non-terminating expressions, then this property would fail, even though progress would remain valid for this language.

The standard work-around for this difficulty is to instrument the evaluation semantics with additional rules that explicitly check for expressions that would be "stuck" according to a transition semantics, yielding a special, ill-typed expression, `wrong`, in the case that these occur. If $e \Downarrow \texttt{wrong}$, then we say that the expression *e goes wrong*. We then prove preservation for the augmented evaluation semantics, from which we may conclude that *well-typed expression do not go wrong*. Before detailing this methodology, let us point us two reasons why it is inadequate:

1. Which checks to include can only be determined by consideration of the progress theorem for a transition semantics. If we neglect to check for any, of the "stuck" states, then the safety property so obtained is weaker than progress. In the limit if we omit all such checks, then no safety guarantees are provided at all!

2. As the name implies, dynamic checks are performed at run-time. Yet a dynamic check for a type error can never fail for a well-typed program, only for an ill-typed program. Thus we are paying the overhead of dynamic checking, even though it is unnecessary!

These shortcomings suggest that evaluation semantics is not a suitable basis for proving the safety of a programming language.

Nevertheless, let us detail for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ what is involved in instrumenting the evaluation semantics to support a proof of type safety. The evaluation judgement takes the form $e \Downarrow a$, where $a$ is an *answer*, either `wrong`, or $\texttt{ok}(v)$ for some $v$ such that $v$ val.[1] We define the judgement $a : \tau$ to hold iff $a = \texttt{ok}(v)$ with $v : \tau$. Importantly, the answer `wrong` is ill-typed.[2]

[1] In the presence of checked errors such as division by zero, one would also include `error` as a possible answer, distinct from `wrong`.

[2] But, as before, we would have $\texttt{error} : \tau$, for any $\tau$, if we were to include checked errors in the language.

We then prove preservation in the form that if $e : \tau$ and $e \Downarrow a$, then $a : \tau$. This ensures, in particular, that $a \neq \texttt{wrong}$.

The instrumented evaluation rules for $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ are given as follows:

$$\frac{}{\texttt{num[}n\texttt{]} \Downarrow \texttt{ok(num[}n\texttt{])}} \tag{11.4a}$$

$$\frac{}{\texttt{str[}s\texttt{]} \Downarrow \texttt{ok(str[}s\texttt{])}} \tag{11.4b}$$

$$\frac{e_1 \Downarrow \texttt{ok(num[}n_1\texttt{])} \quad e_2 \Downarrow \texttt{ok(num[}n_2\texttt{])} \quad n_1 + n_2 = n \text{ nat}}{\texttt{plus(}e_1\texttt{,}e_2\texttt{)} \Downarrow \texttt{ok(num[}n\texttt{])}} \tag{11.4c}$$

$$\frac{e_1 \Downarrow \texttt{ok(str[}s_1\texttt{])}}{\texttt{plus(}e_1\texttt{,}e_2\texttt{)} \Downarrow \texttt{wrong}} \tag{11.4d}$$

$$\frac{e_1 \Downarrow \texttt{ok(num[}n_1\texttt{])} \quad e_2 \Downarrow \texttt{ok(str[}s_2\texttt{])}}{\texttt{plus(}e_1\texttt{,}e_2\texttt{)} \Downarrow \texttt{wrong}} \tag{11.4e}$$

$$\frac{e_1 \Downarrow \texttt{wrong}}{\texttt{plus(}e_1\texttt{,}e_2\texttt{)} \Downarrow \texttt{wrong}} \tag{11.4f}$$

$$\frac{e_1 \Downarrow \texttt{num[}n_1\texttt{]} \quad e_2 \Downarrow \texttt{wrong}}{\texttt{plus(}e_1\texttt{,}e_2\texttt{)} \Downarrow \texttt{wrong}} \tag{11.4g}$$

$$\frac{e_1 \Downarrow \texttt{ok}(v_1) \quad [v_1/x]e_2 \Downarrow a_2}{\texttt{let(}e_1\texttt{,}x\texttt{.}e_2\texttt{)} \Downarrow a_2} \tag{11.4h}$$

$$\frac{e_1 \Downarrow \texttt{wrong}}{\texttt{let(}e_1\texttt{,}x\texttt{.}e_2\texttt{)} \Downarrow \texttt{wrong}} \tag{11.4i}$$

(The other evaluation rules are omitted for the sake of concision.)

As should be evident, the instrumented semantics is rather verbose, requiring explicit checks for stuck states that cannot arise in a well-typed expression, and also propagating errors upwards from subexpressions.

## 11.6 Exercises

1. Prove that if $e \Downarrow v$, then $v$ val.
2. Prove that if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 = v_2$.
3. Complete the proof of equivalence of evaluation and transition semantics.

4. Prove preservation for the instrumented evaluation semantics, and conclude that well-typed programs cannot go wrong.

5. Is it possible to use environments in a structural semantics? What difficulties do you encounter?

# Chapter 12

# Types and Languages

The static and dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ illustrates several fundamental organizing principles of language design on which we shall rely throughout this book. Chief among these is the central role of *types* in programming languages. The informal concept of a language "feature" is formally analyzed as a manifestation of type structure. For example, in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ the type `nat` comprises the numeric literals and the arithmetic operations of addition and subtraction, and the type `str` comprises the string literals and the string operations of concatenation and length. Thus these types account for nearly all of the "features" of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, apart from the generic concepts of variable binding and reference. These arise from the structural properties of the hypothetico-general typing judgement, and are not tied to any particular types.

The language $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ illustrates a number of important themes that recur throughout the text. In this chapter we summarize the main concepts that will be used throughout the remainder of the text.

## 12.1 Phase Distinction

The semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ maintains a *phase distinction* between the *static phase* and the *dynamic phase* of processing. The static semantics, or typing rules, impose constraints on the formation of programs that are sufficient to ensure that the dynamic semantics, or evaluation rules, are well-behaved. The static phase occurs prior to, and independently of, the dynamic phase. The static phase may be seen as predicting the form of the value of an expression computed during the dynamic phase. For example, by assigning the type `nat` to the addition of two expressions of the same type, the static

semantics is predicting that the result of the sum will be a number. Consequently, it can be used as the argument to multiplication, for example, without fear of error.

The type safety theorem may be seen as stating that the predictions of the static semantics are true of the dynamic semantics, for otherwise the dynamic semantics would "get stuck." A counterexample to safety is a call for revision to either the static semantics—to ensure that the example is barred from consideration—or the dynamic semantics—to ensure that the error condition is checked at run-time. The purpose of proving safety is to ensure the coherence of the static and dynamic semantics.

The phase distinction also manifests itself in the syntax of a language. In most cases the syntax of types does not involve expressions, but the syntax of expressions may well involve types. This is consistent with the idea that the static phase of processing (type checking) usually occurs prior to execution, and hence is independent of it. Languages that do not respect the phase distinction usually do not maintain a clear separation between types and expressions, and consequently intermix some aspects of the dynamic and static phases of processing.

## 12.2 Introduction and Elimination

The primitive operations associated with a type may generally be classified as either *introduction* or *elimination forms*. The introduction operators determine the values of the type, and the elimnation operators determine the instructions for computing with those values. For example in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, the introduction forms for the type `nat` are the numerals, and those for the type `str` are the string literals. The elimination forms for the type `nat` are addition and multiplication, and those for the type `str` are concatenation and length.

The dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is based on the *inversion*, or *conservation*, *principle*, which, roughly speaking, states that the elimination forms are post-inverse to the introduction forms. This is a kind of conservation principle stating that what comes out of an elimination form is determined by what goes into the creation of its arguments. For example, in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ the addition function is post-inverse to the numerals in the sense that the dynamic semantics specifies that the value of an addition, a numeral, is obtained from the values of its arguments, which must both be numerals.

The type safety theorem may be seen as validating the inversion prin-

ciple for the language. Returning to the addition example, the type preservation theorem ensures that the values of the arguments of addition must themselves be of type `nat`, and hence by the canonical forms theorem must be numerals. This ensures that addition can make progress, yielding a numeral, which is a value of type `nat`. Had the type safety theorem failed, say by assigning the type `nat` to a string, then the addition function would be required to be post-inverse to the introduction form for another type, namely `str`, in violation of the inversion principle.

The inversion principle serves as a guide for deriving the dynamic semantics of a language, but in most cases does not fully determine it. For example, suppose that we added the conditional expression $\texttt{ifz}(e, e_1, e_2)$ to $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, thought of as an elimination form for the type `nat`. The expression $e$ of type `nat` is tested, resulting in $e_1$, if $e$ evaluates to zero, and in $e_2$, otherwise. The dynamic semantics of this construct must specify that $e$ is to be evaluated to determine how to proceed, but we do not wish to evaluate $e_1$ or $e_2$ in advance, only once the decision about $e$ has been made. We say that $e$ is a *major*, or *principal*, argument of the conditional elimination form, and that $e_1$ and $e_2$ are *minor*, or *non-principal* arguments. As a rule, the major arguments of an elimination form *must* be evaluated, but the minor arguments need not be. In the case of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ all arguments to the arithmetic and string elimination forms are principal, and hence must be evaluated before evaluation of the operation itself.

Another opportunity for discretion in the dynamic semantics in the evaluation of the arguments of an introduction form. Suppose that we replace the numerals in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ with two new primitives, `z` and $\texttt{s}(e)$, which represent zero and successor, respectively. These are both introductory forms of type `nat`, but this classification does not determine whether $\texttt{s}(e)$ should be considered a value regardless of the form of $e$, or only if $e$ is itself a value. If an argument to an introduction form is required to be a value, then there is an associated search rule in the dynamic semantics to evaluate that argument; the operator is said to be *eager*, or *strict*, in that position. If an argument to an introduction form is not required to be a value, then the operator is said to be *lazy*, or *non-strict*, in that position. When all arguments of all introduction forms are eager, then the language itself is said to be eager, and similarly when all arguments of all introduction forms are lazy, then the language itself is said to be lazy.

Finally, recall that the dynamic semantics of $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ is defined *only* for closed expressions, *i.e.*, those with no free variables. This implies that we may *never* evaluate under the scope of a binder. For example, evaluation of $\texttt{let}(e_1, x.e_2)$ cannot require evaluation of $e_2$ prior to execution of the

`let`, since the variable $x$ is bound within $e_2$. The presence of the binder sequentializes evaluation, requiring $e_1$ to be fully evaluated and its value bound to $x$ before evaluaton of $e_2$ commences.

## 12.3 Variables and Binding

The meaning of the hypothetico-general judgement

$$\mathcal{X}, x\ \mathsf{exp} \mid \Gamma, x : \tau \vdash e' : \tau'$$

is that every instance of this judgement of the form

$$\mathcal{X} \mid \Gamma \vdash [e/x]e' : \tau'$$

obtained by replacing the variable $x$ by an expression $e$ is valid, provided that $\mathcal{X} \mid \Gamma \vdash e : \tau$.

However, if we examine the dynamic semantics of the only variable-binding construct in $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$, we observe that variables are only ever bound to (closed) values, and not to general expressions. This is a consequence of the fact that the binding of a variable in a `let` expression is evaluated before it is bound. Alternatively, we could simply substitute the binding for the variable in the body:

$$\mathtt{let}(e_1, x.e_2) \mapsto [e_1/x]e_2,$$

with no restriction on $e_1$. The difference is in the evaluation order: we may choose whether *variables stand for values* or *variables stand for computations*. If variables stand for values, then the language is said to be *call-by-value*; otherwise, it is *call-by-name*.[1]

This distinction may be imposed in the static semantics by adding an additional hypothesis governing each value variable, demanding that it be bound only to a value:

$$\mathcal{X}, x\ \mathsf{exp} \mid \Gamma, x\ \mathsf{val}, x : \tau \vdash e' : \tau'$$

Now the possible substitutions for $x$ are limited to those $e$ exp such that $\mathcal{X} \mid \Gamma \vdash e : \tau$ and, moreover, $\mathcal{X} \mid \Gamma \vdash e$ val. Such hypotheses give rise to

[1] The justification for this terminology is unclear. The "call-by" part arises from the association of variable declarations with function calls in many languages. Given this, the phrase "call by value" makes good sense, but the origin of the phrase "call by name" for the opposite case remains obscure.

the concept of an *open value*, namely an expression, possibly involving free variables, such that

$$x_1 \ \mathsf{exp}, \ldots, x_k \ \mathsf{exp} \mid x_1 \ \mathsf{val}, \ldots, x_k \ \mathsf{val} \vdash e \ \mathsf{val}.$$

For example, in the extension of $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$ with an eager successor operation, we have

$$x \ \mathsf{exp} \mid x \ \mathsf{val} \vdash \mathtt{s(s}(x)\mathtt{)} \ \mathsf{val}.$$

There is no reason to consider only value variables or only computation variables; we can mix-and-match as we see fit, adding a hypothesis $x$ val to constrain a variable to values. In most, but not all, cases the typing rules are insensitive to whether a variable is limited to values or not. In such situations we omit an explicit declaration of the value status of a variable.

## 12.4 Compositionality

The combined structural properties of substitution and transitivity for typing, which we repeat here for reference, captures an essential feature of a type system, called *compositionality*, or *modularity*.

$$\frac{\mathcal{X} \mid \Gamma \vdash e : \tau \quad \mathcal{X}, x \ \mathsf{exp} \mid \Gamma, x : \tau \vdash e' : \tau'}{\mathcal{X} \mid \Gamma \vdash [e/x]e' : \tau'}$$

This rule captures the essence of *linking*. The expression $e'$, with a free variable $x$ of type $\tau$, represents a client of a separately compiled component, $e$, which is referenced by the variable, $x$. The job of the linker is to combine $e$ with $e'$, by substitution for $x$, to obtain a complete compilation unit, albeit one with further free references to other units to be linked later. The result is *composed* from the shared component and the client, which gives rise to the terminology.

It is important that the client, $e'$, is type checked independently of the implementation of the shared component, $e$. All that is propagated from the library to its clients is its type, and not the details of its implementation. This means, in particular, that a revised implementation of the library can be linked with the *same* client, without requiring any re-writing or other modification to the client code, so long as the type, $\tau$, remains the same. This is the foundation for modular program development, the process of decomposing a large program into separable parts whose interactions are mediated by a specification, or type, that serves as a contract between the client and the implementor. In other words, *types provide the foundation for modularity*.

## 12.5 Exercises

# Part IV

# Functions

# Chapter 13

# Functions

In $\mathcal{L}\{\texttt{num str}\}$ it is possible to express doubling of any given expression of type `nat`, but it is not possible to express the concept of doubling in general. For this we need *functions*, which capture patterns of computation that can be instantiated to obtain specific computations. To pass from particular instances of doubling, of the form $e$+$e$ for some expression $e$, to the general case, we replace occurrences of a fixed expression, $e$, by a variable, $x$, and then mark that variable as subject to variation using $\lambda$-*abstraction*. Thus the general pattern of doubling can be captured by the function

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

The variable, $x$, is called the *parameter* of the function, and the expression $x$+$x$ is its *body*. The parameter is bound by the $\lambda$-abstraction, and, consequently, may be renamed freely in accordance with the rules of $\alpha$-equivalence. We may *apply* this function to any argument, $e$, of type `nat` to obtain an instance of the doubling function for that choice of expression $e$ to be doubled.

To ensure type consistency the type of the parameter of the $\lambda$-abstraction is given explicitly, and instances are restricted to arguments of that type. The type of the result is arbitrary, since we are free to use the parameter, $x$, in any way at all, provided only that it is type-correct. In general, the type of a $\lambda$-abstraction has the form $\sigma \to \tau$, where $\sigma$ is the *domain type*, the type of its parameter, and $\tau$ is the *range type*, the type of its body. Thus, $d$ : `nat` $\to$ `nat`, so that if $e$ : `nat`, then $p$`(`$e$`)` : `nat` as well.

Since function types are themselves types, we have *higher-order functions* with types such as

1. `nat` $\to$ `(nat` $\to$ `nat)`,

2. $\texttt{(nat} \to \texttt{nat)} \to \texttt{nat}$,

3. $\texttt{(nat} \to \texttt{nat)} \to \texttt{(nat} \to \texttt{nat)}$.

These are, respectively,

1. the type of functions that assign a function on the natural numbers to each natural number,

2. the type of functions that assign a natural number to each function on the natural numbers,

3. the type of functions that assign a function on the natural numbers to each function on the natural numbers.

It is a good exercise to think of mathematical functions with these types.

In this chapter we study the function type in isolation from other language features using a language called $\mathcal{L}\{\to\}$. Not much can be done with $\mathcal{L}\{\to\}$ itself; its interest lies in its combination with other language features. As a simple example it is a straightforward exercise to integrate $\mathcal{L}\{\to\}$ with $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$ to obtain a simple language of arithmetic and string expressions augmented with function definitions. In Chapters 14 and 15 we will consider two different ways of defining a language with functions and natural numbers.

## 13.1 Syntax

The abstract syntax of the language $\mathcal{L}\{\to\}$ is specified as follows:

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \texttt{arr}(\tau_1, \tau_2) \\ \text{Expr's} & e & ::= & x \mid \texttt{lam}[\tau](x.e) \mid \texttt{ap}(e_1, e_2) \end{array}$$

The concrete syntax conventions that we shall use for $\mathcal{L}\{\to\}$ are summarized in the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\texttt{arr}(\tau_1, \tau_2)$ | $\tau_1 \to \tau_2$ |
| $\texttt{lam}[\tau](x.e)$ | $\lambda(x{:}\tau.e)$ |
| $\texttt{ap}(e_1, e_2)$ | $e_1(e_2)$ |

As we mentioned in the introduction to this chapter, the expression $\lambda(x{:}\tau.e)$ is called a *λ-abstraction*. The variable $x$ is the *parameter* of the

abstraction, and $e$ is its *body*. It represents the function mapping $e_0 : \tau$ to (the value of) $[e_0/x]e$. The expression $e_1(e_2)$ is called an *application*, with *function* $e_1$ and *argument* $e_2$. If $e_1$ evaluates to a $\lambda$-abstraction $\lambda(x:\tau.\, e)$, then the application $e_1(e_2)$ evaluates to the value of $[e_1/x]e_2$, the instance of the body obtained by replacing the parameter by the argument.

The `let` expression, which was introduced in $\mathcal{L}\{\texttt{num}\,\texttt{str}\}$, is definable in $\mathcal{L}\{\rightarrow\}$ by taking $\texttt{let}[\tau](e_1, x.e_2)$ to stand for the expression

$$\texttt{ap}(\texttt{lam}[\tau](x.e_2), e_1). \tag{13.1}$$

Thus, in the presence of functions, we do not need to treat this as a primitive notion.

## 13.2 Static Semantics

The static semantics of $\mathcal{L}\{\rightarrow\}$ is defined by a generalized inductive definition of judgements of the form $\Gamma \vdash e : \tau$, where $\Gamma$ is a finite set of assumptions of the form $x : \tau$, where each $x$ is a variable.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{13.2a}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}[\tau_1](x.e) : \texttt{arr}(\tau_1, \tau_2)} \tag{13.2b}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arr}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}(e_1, e_2) : \tau} \tag{13.2c}$$

**Lemma 13.1** (Inversion)**.** *Suppose that* $\Gamma \vdash e : \tau$.

1. *If* $e = x$, *then* $\Gamma = \Gamma', x : \tau$.
2. *If* $e = \texttt{lam}[\tau_1](x.e)$, *then* $\tau = \texttt{arr}(\tau_1, \tau_2)$ *and* $\Gamma, x : \tau_1 \vdash e : \tau_2$.
3. *If* $e = \texttt{ap}(e_1, e_2)$, *then there exists* $\tau_2$ *such that* $\Gamma \vdash e_1 : \texttt{arr}(\tau_2, \tau)$ *and* $\Gamma \vdash e_2 : \tau_2$.

*Proof.* The proof proceeds by rule induction on the typing rules. Observe that for each rule, exactly one case applies, and that the premises of the rule in question provide the required result. □

The structural property of substitution holds for the typing judgement defined by the above rules.

**Lemma 13.2** (Substitution). *If* $\Gamma, x : \tau \vdash e' : \tau'$, *and* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

*Proof.* By rule induction on the derivation of the first judgement. □

## 13.3 Dynamic Semantics

The dynamic semantics of $\mathcal{L}\{\rightarrow\}$ is given by a structural operational semantics on closed expressions. The judgement $e$ val, where $e$ is a closed expression, is inductively defined.

$$\frac{}{\texttt{lam[}\tau\texttt{](}x\texttt{.}e\texttt{)}\ \mathsf{val}} \tag{13.3a}$$

Observe that no restriction is placed on the form of $e$, the body of the function.

There are two forms of dynamic semantics for functions, *call-by-value* and *call-by-name*.[1] Under call-by-value, the argument is evaluated *before* it is passed to the function by substitution; under call-by-name, the argument is passed in *unevaluated* form, deferring evaluation until it is actually needed. This can save work in the case that the value is never required, but can replicate work in the case that the value is required more than once. (See Chapter 37 for further discussion.)

The call-by-value dynamic semantics is defined by the following rules:

$$\frac{e_1 \mapsto e_1'}{\texttt{ap(}e_1\texttt{,}e_2\texttt{)} \mapsto \texttt{ap(}e_1'\texttt{,}e_2\texttt{)}} \tag{13.4a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{ap(}e_1\texttt{,}e_2\texttt{)} \mapsto \texttt{ap(}e_1\texttt{,}e_2'\texttt{)}} \tag{13.4b}$$

$$\frac{e_2\ \mathsf{val}}{\texttt{ap(lam[}\tau_2\texttt{](}x\texttt{.}e_1\texttt{),}e_2\texttt{)} \mapsto [e_2/x]e_1} \tag{13.4c}$$

The call-by-name semantics is, instead, defined by the following rules:

$$\frac{e_1 \mapsto e_1'}{\texttt{ap(}e_1\texttt{,}e_2\texttt{)} \mapsto \texttt{ap(}e_1'\texttt{,}e_2\texttt{)}} \tag{13.5a}$$

$$\frac{}{\texttt{ap(lam[}\tau_2\texttt{](}x\texttt{.}e_1\texttt{),}e_2\texttt{)} \mapsto [e_2/x]e_1} \tag{13.5b}$$

In contrast to Rule (13.4c) there is no requirement on Rule (13.5b) that the argument be a value.

[1] The justification for the terminology is lost in the mists of time, but is so well-established as to be unchangeable.

## 13.4 Safety

**Theorem 13.3** (Preservation). *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

*Proof.* The proof is by induction on rules (13.4), which define the dynamic semantics of the language.

Consider rule (13.4c),

$$\frac{e_2 \ \mathsf{val}}{\mathtt{ap(lam[}\tau_2\mathtt{]}(x\mathtt{.}e_1\mathtt{)},e_2\mathtt{)} \mapsto [e_1/x]e_2}\ .$$

Suppose that $\mathtt{ap(lam[}\tau_2\mathtt{]}(x\mathtt{.}e_1\mathtt{)},e_2\mathtt{)} : \tau_1$. By Lemma 13.1 on page 99 $e_2 : \tau_2$ and $x : \tau_2 \vdash e_1 : \tau_1$, so by Lemma 13.2 on the facing page $[e_2/x]e_1 : \tau_1$.

The other rules governing application are handled similarly. □

**Lemma 13.4** (Canonical Forms). *If $e$ $\mathsf{val}$ and $e : \mathtt{arr}(\tau_1, \tau_2)$, then $e = \mathtt{lam[}\tau_1\mathtt{]}(x\mathtt{.}e_2)$ for some $x$ and $e_2$ such that $x : \tau_1 \vdash e_2 : \tau_2$.*

*Proof.* By induction on the typing rules, using the assumption $e$ $\mathsf{val}$. □

**Theorem 13.5** (Progress). *If $e : \tau$, then either $e$ is a value, or there exists $e'$ such that $e \mapsto e'$.*

*Proof.* The proof is by induction on rules (13.2). Note that since we consider only closed terms, there are no hypotheses on typing derivations.

Consider rule (13.2c). By induction either $e_1$ $\mathsf{val}$ or $e_1 \mapsto e_1'$. In the latter case we have $\mathtt{ap}(e_1,e_2) \mapsto \mathtt{ap}(e_1',e_2)$. Otherwise we have by induction either $e_2$ $\mathsf{val}$ or $e_2 \mapsto e_2'$. In the latter case we have $\mathtt{ap}(e_1,e_2) \mapsto \mathtt{ap}(e_1,e_2')$ (bearing in mind $e_1$ $\mathsf{val}$). Otherwise, by Lemma 13.4, we have $e_1 = \mathtt{lam[}\tau_2\mathtt{]}(x\mathtt{.}e)$ for some $x$ and $e$. But then $\mathtt{ap}(e_1,e_2) \mapsto [e_2/x]e$, again bearing in mind that $e_2$ $\mathsf{val}$.

□

## 13.5 Evaluation Semantics

An inductive definition of the evaluation judgement $e \Downarrow v$ for $\mathcal{L}\{\rightarrow\}$ is given by the following rules:

$$\frac{}{\mathtt{lam[}\tau\mathtt{]}(x\mathtt{.}e) \Downarrow \mathtt{lam[}\tau\mathtt{]}(x\mathtt{.}e)} \tag{13.6a}$$

$$\frac{e_1 \Downarrow \mathtt{lam[}\tau\mathtt{]}(x\mathtt{.}e) \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{\mathtt{ap}(e_1,e_2) \Downarrow v} \tag{13.6b}$$

It is easy to check that if $e \Downarrow v$, then $v$ $\mathsf{val}$, and that if $e$ $\mathsf{val}$, then $e \Downarrow e$.

**Theorem 13.6.** *1. If $e \Downarrow v$, then $e \mapsto^* v$.*

*2. If $e \mapsto^* v$, where $v$ $\mathsf{val}$, then $e \Downarrow v$.*

## 13.6 Environment Semantics

The environment semantics of Chapter 11 uses hypothetical judgements of the form

$$x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v$$

stating that the expression $e$ evaluates to the value $v$, under the assumption that the variables $x_i$ evaluate to $v_i$. Let us naïvely extend this semantics to $\mathcal{L}\{\rightarrow\}$ using the following rules for functions and applications:

$$\frac{}{\Theta \vdash \mathtt{lam[\tau](x.e)} \Downarrow \mathtt{lam[\tau](x.e)}} \tag{13.7a}$$

$$\frac{\Theta \vdash e_1 \Downarrow \mathtt{lam[\tau](x.e)} \quad \Theta \vdash e_2 \Downarrow v_2 \quad \Theta, x \Downarrow v_2 \vdash e \Downarrow v \quad x \# \Theta}{\Theta \vdash \mathtt{ap}(e_1, e_2) \Downarrow v} \tag{13.7b}$$

When applying a function to an argument, the parameter of the function is bound to the argument value for the duration of the evaluation of the body. The requirement in Rule (13.7b) that the variable $x$ lie apart from $\Theta$ may always be met by choosing the bound variable of the function appropriately.

This semantics seems to make good sense, but, surprisingly, it is incorrect in that it does not agree with the substitution semantics. The reason is that the use of hypotheses to record the bindings of free variables does not work properly when the returned value of an evaluation can involve such a free variable.

Consider the expression

$$e = \mathtt{ap(lam[nat](x.lam[nat](y.x)),num[3])}.$$

According to Rules (13.6), the expression $e$ evaluates to

$$\mathtt{lam[nat](y.num[3])},$$

as the reader may readily verify. Now, if we evaluate the expression

$$e' = \mathtt{let}(e, f.\mathtt{ap}(f, \mathtt{num[4]})),$$

which involves $e$, using the same rules we obtain `num[3]`. The variable $f$ is bound to the value of $e$, so that the application of $f$ to `num[4]` evaluates to `num[3]`.

But now consider the evaluation of $e$ using the proposed environment semantics, Rules (13.7). Its value is determined by evaluating the body of the outer $\lambda$ under the hypothesis $x \Downarrow$ `num[3]`. According to Rule (13.7a) this evaluates to the *open* expression

$$\texttt{lam[nat]}(y.x),$$

in which the variable $x$ occurs free. So far so good, because we have assumed that $x \Downarrow$ `num[3]`. However, if we now consider the evaluation of the expression $e'$, we observe that the application `ap(`$f$`,num[4])` is evaluated under the hypothesis $f \Downarrow$ `lam[nat]`$(y.x)$, but *without any assumption for the variable* $x$. This quickly leads to trouble, since to evaluate the application of $f$ to `num[4]`, we assume $y \Downarrow$ `num[4]`, and evaluate the body of the inner $\lambda$-abstraction, namely $x$. But then evaluation gets stuck for lack of a binding for $x$!

The difficulty is that the variable $x$ occurring in the inner $\lambda$-abstraction of *e escapes its scope* when it is returned as the value of the body of the outer $\lambda$-abstraction. Consequently, the variable $x$ is *unbound* in the hypothesis list, which causes evaluation to get stuck whenever that variable is used. In essence the stack-like behavior of hypotheses in evaluation judgements does not cohere with the heap-like behavior of variables in a higher-order language. If an expression such as $e$ is to be assigned its proper meaning, the bindings for the free variables in its value must be retained after the call, something that cannot be done using the hypothetical judgement.

## 13.7 Closures

To give a proper semantics to variable binding using environment, we must ensure that the free variables of a $\lambda$-abstraction are not dissociated from their bindings in the environment. This is achieved by treating the environment as an *explicit substitution*, a data structure that records what is to be substituted for a variable without actually doing it. Only when the variable is encountered do we replace it by its binding in the environment, effectively *delaying* substitution as long as possible. To avoid the confusions described in the preceding section, we attach the environment to a $\lambda$-abstraction at the point where the abstraction is evaluated, resulting in a

configuration of the form

$$\texttt{clo[}\tau\texttt{]}(E, x\texttt{.}e),$$

which is called a *closure*. The idea is that the environment "closes" the free variables of the $\lambda$-abstraction by providing bindings for them. These are the bindings that are used when the function body is evaluated, not those in the ambient environment at the point of application.

To give a proper environment semantics for $\mathcal{L}\{\rightarrow\}$ we introduce two new syntactic categories, *values* and *environments*.

$$\begin{array}{llll} \textit{Values} & V & ::= & \texttt{clo[}\tau\texttt{]}(E, x\texttt{.}e) \\ \textit{Env's} & E & ::= & \bullet \mid E, x{=}V \end{array}$$

In this setting a value is no longer a form of expression, but is instead drawn from a syntactic category of its own. Furthermore, an environment is no longer a set of assumptions in a hypothetical evaluation judgement, but is now a data structure that can appear within a closure.

Correspondingly, the environment semantics sketched in the preceding section is revised to employ the *categorical* judgement $[E]\ e \Downarrow V$, which states that the expression $e$, whose free variables are governed by the environment $E$, evaluates to the value $V$. The evaluation rules for $\lambda$-abstraction and application are formulated as follows:

$$\frac{}{[E]\ \texttt{lam[}\tau\texttt{]}(x\texttt{.}e) \Downarrow \texttt{clo[}\tau\texttt{]}(E, x\texttt{.}e)} \tag{13.8a}$$

$$\frac{[E]\ e_1 \Downarrow \texttt{clo[}\tau\texttt{]}(E', x\texttt{.}e) \quad [E]\ e_2 \Downarrow V_2 \quad [E', x{=}V_2]\ e \Downarrow V \quad x \# E'}{[E]\ \texttt{ap}(e_1, e_2) \Downarrow V} \tag{13.8b}$$

Rule (13.8b) switches environments from the *ambient environment*, $E$, of the application to the *closure environment*, $E'$, associated with the function. This ensures that the free variables of the body are governed by the environment in effect at the point where the function is created, not at the point where the function is applied. In addition to these rules we also need a rule to look up a variable in the environment to recover its binding:

$$\frac{E(x) = V}{[E]\ x \Downarrow V} \tag{13.9}$$

We leave it as an exercise to define the judgement $E(x) = V$, which holds iff $x$ is bound to $V$ in $E$.

To state the equivalence of the environment and substitution semantics it is necessary to introduce two auxiliary judgements. The first, $V^*{=}v$ states that the value $V$ in the sense of the environment semantics corresponds to the value $v$ in the substitution semantics. It is defined by the following rule:

$$\frac{E[e]{=}e'}{\texttt{clo[}\tau\texttt{]}(E,x\texttt{.}e)^*{=}\texttt{lam[}\tau\texttt{]}(x\texttt{.}e')} \tag{13.10}$$

The second, $E[e]{=}e'$, states that the result of substituting the expanded value of $E(x)$ for each variable $x$ in $e$ is the expression $e'$. It is defined by the following rule:

$$\frac{E[e]{=}e'' \quad [V^*/x]e'' = e'}{(E, x \Downarrow V)[e]{=}e'} \tag{13.11}$$

It is easy to verify that the former judgement has mode $(\forall, \exists!)$ and that the latter has mode $(\forall, \forall, \exists!)$.

**Theorem 13.7** (Equivalence)**.** *$[E]\ e \Downarrow V$ iff $E[e] \Downarrow V^*$.*

## 13.8 Exercises

1. Complete the definition of the environment semantics for $\mathcal{L}\{\rightarrow\}$.
2. Prove the equivalence theorem.
3. Re-formulate the transition semantics for $\mathcal{L}\{\rightarrow\}$ in terms of environments. What difficulties do you encounter? How might they be overcome?

# Chapter 14

# Gödel's T

The language $\mathcal{L}\{\mathtt{nat} \to\}$, better known as *Gödel's* ***T***, is the combination of function types with the type of natural numbers. In contrast to $\mathcal{L}\{\mathtt{num}\,\mathtt{str}\}$, which equips the naturals with some arbitrarily chosen arithmetic primitives, the language $\mathcal{L}\{\mathtt{nat} \to\}$ provides a general mechanism, called *primitive recursion*, for defining functions on the natural numbers. Primitive recursion captures the essential inductive character of the natural numbers, from which we may define a wide range of functions, including elementary arithmetic.

A chief characteristic of $\mathcal{L}\{\mathtt{nat} \to\}$ is that it permits the definition only of *total* functions, *i.e.*, those that assign a value in the range type to every element of the domain type. This means that programs written in $\mathcal{L}\{\mathtt{nat} \to\}$ may be considered to "come equipped" with their own termination proof, in the form of typing annotations to ensure that it is well-typed. But only certain forms of proof are codifiable in this manner, with the inevitable result that some well-defined total functions on the natural numbers cannot be programmed in $\mathcal{L}\{\mathtt{nat} \to\}$.

## 14.1 Static Semantics

The syntax of $\mathcal{L}\{\mathtt{nat} \to\}$ is given by the following grammar:

| | | | |
|---|---|---|---|
| Type | $\tau$ | ::= | $\mathtt{nat} \mid \mathtt{arr}(\tau_1, \tau_2)$ |
| Expr | $e$ | ::= | $\mathtt{z} \mid \mathtt{s}(e) \mid \mathtt{rec}[\tau](e, e_0, x.y.e_1) \mid \mathtt{lam}[\tau](x.e) \mid \mathtt{ap}(e_1, e_2)$ |

We write $\overline{n}$ for the expression $\mathtt{s}(\ldots\mathtt{s}(\mathtt{z}))$, in which the successor is applied $n \geq 0$ times to zero. The expression

$$\mathtt{rec}[\tau](e, e_0, x.y.e_1)$$

is called *primitive recursion*. It represents the $e$-fold iteration of the transformation $x, y.e_1$ starting from $e_0$. The bound variable $x$ represents the predecessor and the bound variable $y$ represents the result of the $x$-fold iteration.

Sometimes *iteration*, written $\mathtt{iter}[\tau](e, e_0, y.e_1)$, is considered as an alternative to primitive recursion. It has essentially the same meaning as primitive recursion, except that only the result of the recursive call is bound to $y$ in $e_1$, and no binding is made for the predecessor. Clearly iteration is a special case of primitive recursion, since we can always ignore the predecessor binding. Conversely, primitive recursion is definable from iteration, provided that we have product types (Chapter 16) at our disposal. To define primitive recursion from iteration we simultaneously compute the predecessor while iterating the specified computation.

The following chart summarizes the concrete syntax conventions for $\mathcal{L}\{\mathtt{nat} \to\}$:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\mathtt{arr}(\tau_1, \tau_2)$ | $\tau_1 \to \tau_2$ |
| $\mathtt{lam}[\tau](x.e)$ | $\lambda(x{:}\tau.e)$ |
| $\mathtt{ap}(e_1, e_2)$ | $e_1(e_2)$ |
| $\mathtt{rec}[\tau](e, e_0, x.y.e_1)$ | $\mathtt{rec}\, e\, \{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\}$ |

The static semantics of $\mathcal{L}\{\mathtt{nat} \to\}$ is given by the following typing rules:

$$\frac{}{\Gamma, x : \mathtt{nat} \vdash x : \mathtt{nat}} \tag{14.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{14.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{14.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}[\tau](e, e_0, x.y.e_1) : \tau} \tag{14.1d}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau \quad x \# \Gamma}{\Gamma \vdash \mathtt{lam}[\sigma](x.e) : \mathtt{arr}(\sigma, \tau)} \tag{14.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arr}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1, e_2) : \tau} \tag{14.1f}$$

As usual, admissibility of the structural rule of substitution is crucially important.

**Lemma 14.1.** *If* $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

## 14.2 Dynamic Semantics

We will adopt a *lazy* semantics for the successor operation, and a *call-by-name* semantics for function applications. Variables range over computations, which are not necessarily values. These choices are not forced on us, but are natural and convenient in a language in which (as we shall see) every closed expression has a value.

The closed values of $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ are determined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{14.2a}$$

$$\frac{}{\mathtt{s}(e)\ \mathsf{val}} \tag{14.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{14.2c}$$

The dynamic semantics of $\mathcal{L}\{\mathtt{nat} \rightarrow\}$ is given by the following rules:

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1, e_2) \mapsto \mathtt{ap}(e_1', e_2)} \tag{14.3a}$$

$$\frac{}{\mathtt{ap}(\mathtt{lam}[\tau](x.e), e_2) \mapsto [e_2/x]e} \tag{14.3b}$$

$$\frac{e \mapsto e'}{\mathtt{rec}[\tau](e, e_0, x.y.e_1) \mapsto \mathtt{rec}[\tau](e', e_0, x.y.e_1)} \tag{14.3c}$$

$$\frac{}{\mathtt{rec}[\tau](\mathtt{z}, e_0, x.y.e_1) \mapsto e_0} \tag{14.3d}$$

$$\frac{}{\mathtt{rec}[\tau](\mathtt{s}(e), e_0, x.y.e_1) \mapsto [e, \mathtt{rec}[\tau](e, e_0, x.y.e_1)/x, y]e_1} \tag{14.3e}$$

Rules (14.3d) and (14.3e) specify the behavior of the recursor on $\mathtt{z}$ and $\mathtt{s}(e)$. In the former case the recursor evaluates $e_0$, and in the latter case the variable $x$ is bound to $e$ and the variable $y$ is bound to a recursive call on $e$ before evaluating $e_1$. In the case that $e_1$ does not refer to $y$, the result of the recursive call is not computed.

**Theorem 14.2** (Safety)**.** *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

*2. If $e : \tau$, then either $e$ $\mathsf{val}$ or $e \mapsto e'$ for some $e'$*

## 14.3 Definability of Numeric Functions

A mathematical function $f : \mathbb{N} \to \mathbb{N}$ is *definable* in $\mathcal{L}\{\mathtt{nat} \to\}$ iff there exists an expression $e_f$ of type $\mathtt{nat} \to \mathtt{nat}$ such that for every $n \in \mathbb{N}$,

$$e_f(\overline{m}) \equiv \overline{f(m)} : \mathtt{nat} \tag{14.4}$$

That is, the numeric function $f : \mathbb{N} \to \mathbb{N}$ is definable iff there is a expression $e_f$ of type $\mathtt{nat} \to \mathtt{nat}$ that accurately mimics the behavior of $f$ on all possible inputs.

But what do we mean by equivalence? We shall have much more to say about this in Chapter 49, but for the time being we will assume given an equivalence judgement $\Gamma \vdash e_1 \equiv e_2 : \tau$, where $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$, with the following properties:

1. It is a *congruence*, meaning that replacing a sub-expression of an expression by an equivalent sub-expression results in an equivalent expression.

2. It contains *symbolic execution*, meaning that all of the rules of the dynamic semantics are valid equivalences, even when the expressions involved have free variables.

3. It is *consistent* in that $\overline{m} \equiv \overline{n}$ iff $m$ is $n$. That is, equivalence does not equate distinct numerals.

We often omit the typing assumptions and the type of the expressions from the equivalence judgement for the sake of concision.

For example, the successor function is obviously definable in $\mathcal{L}\{\mathtt{nat} \to\}$ by the expression $\mathtt{succ} = \lambda(x{:}\mathtt{nat}.\,\mathtt{s}(x))$. The doubling function, $d(n) = 2 \times n$, is definable by the expression

$$e_d = \lambda(x{:}\mathtt{nat}.\,\mathtt{rec}\,x\,\{\mathtt{z} \Rightarrow \mathtt{z} \,|\, \mathtt{s}(u)\ \mathtt{with}\ v \Rightarrow \mathtt{s}(\mathtt{s}(v))\}).$$

To see this, observe that $e_d(\overline{0}) \equiv \overline{0}$, and, assuming that $e_d(\overline{n}) \equiv \overline{d(n)}$, check that

$$\begin{aligned} e_d(\overline{n+1}) &\equiv \mathtt{s}(\mathtt{s}(e_d(\overline{n}))) && (14.5)\\ &\equiv \mathtt{s}(\mathtt{s}(\overline{2 \times n})) && (14.6)\\ &= \overline{2 \times (n+1)} && (14.7)\\ &= \overline{d(n+1)} && (14.8) \end{aligned}$$

## 14.4 Ackermann's Function

Consider the following function, called *Ackermann's function*, which is defined by the following equations:

$$\begin{aligned} A(0,n) &= n+1 && (14.9)\\ A(m+1,0) &= A(m,1) && (14.10)\\ A(m+1,n+1) &= A(m,A(m+1,n)) && (14.11) \end{aligned}$$

This function grows very quickly! For example, $A(4,2) = 2^{65,536} - 3$, which is often cited as being much larger than the number of atoms in the physical universe! Yet we can show that the Ackermann function is total by a lexicographic induction on the pair of argument $(m,n)$. On each recursive call, either $m$ decreases, or else $m$ remains the same, and $n$ decreases, so inductively the recursive calls are well-defined, and hence so is $A(m,n)$.

A *primitive recursive function* is a function of type $\mathtt{nat} \to \mathtt{nat}$ that is defined using primitive recursion, but without using any higher order functions. Ackermann's function is defined so as to grow more quickly than any primitive recursive function, and hence cannot itself be primitive recursive. But if we permit ourselves to use higher-order functions, then we may give a definition of Ackermann's function using primitive recursion. This is not a contradiction! Everything depends on whether we may use higher-order functions in the definition. If we are permitted to do so, then the Ackermann function may be defined using primitive recursion, otherwise it cannot.

The key is to observe that $A(m,n)$ iterates the function $A(m,-)$ for $n$ times, starting with $A(m,1)$. As an auxiliary, let us define the higher-order function

$$\mathtt{it} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$$

to be the $\lambda$-abstraction

$$\lambda(f{:}\mathtt{nat} \to \mathtt{nat}.\, \lambda(n{:}\mathtt{nat}.\, \mathtt{rec}\, n\, \{\mathtt{z} \Rightarrow \mathtt{id} \mid \mathtt{s}(\_)\ \mathtt{with}\ g \Rightarrow f \circ g\})),$$

where $\mathtt{id} = \lambda(x{:}\mathtt{nat}.\, x)$ is the identity, and $f \circ g = \lambda(x{:}\mathtt{nat}.\, f(g(x)))$ is the composition of $f$ and $g$. It is easy to check that

$$\mathtt{it}(f)(\overline{n})(\overline{m}) \equiv f^{(n)}(\overline{m}),$$

where the latter expression is the $n$-fold composition of $f$ starting with $\overline{m}$.

With this in hand we may define the Ackermann function

$$a : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$$

to be the $\lambda$-abstraction

$$\lambda(m{:}\mathtt{nat}.\, \mathtt{rec}\, m\, \{\mathtt{z} \Rightarrow \mathtt{succ} \mid \mathtt{s}(\_)\ \mathtt{with}\ f \Rightarrow \lambda(n{:}\mathtt{nat}.\, \mathtt{it}(f)(n)(f(\overline{1})))\}).$$

It is instructive to check that the following equivalences, which show that the Ackermann function is definable, are valid:

$$a(\overline{0})(\overline{n}) \equiv \mathtt{s}(\overline{n}) \tag{14.12}$$

$$a(\overline{m+1})(\overline{0}) \equiv a(m)(\overline{1}) \tag{14.13}$$

$$a(\overline{m+1})(\overline{n+1}) \equiv a(\overline{m})(a(\mathtt{s}(\overline{m}))(\overline{n})). \tag{14.14}$$

## 14.5 Termination

We state without proof that every closed expression in $\mathcal{L}\{\mathtt{nat} \to\}$ evaluates to a value.

**Theorem 14.3.** *If $e : \tau$, then there exists $v$ $\mathsf{val}$ such that $e \mapsto^* v$.*

The proof of this theorem relies on a technique called *logical relations*, which is employed in Chapter 49 to analyze equivalence of expressions.

It follows directly from Theorem 14.3 that all functions in $\mathcal{L}\{\mathtt{nat} \to\}$ are *total* in the sense that if $f : \sigma \to \tau$ and $e : \sigma$, then $f(e)$ evaluates to a value of type $\tau$. Using this, we can show that there are total functions on the natural numbers that are not definable in the language.

First, it can be shown that there is a one-to-one correspondence between the natural numbers, $\mathbb{N}$, and closed expressions $e$ of type $\mathtt{nat} \to \mathtt{nat}$. If $e : \mathtt{nat} \to \mathtt{nat}$, we write $\ulcorner e \urcorner$ for the unique $n \in \mathbb{N}$ corresponding to $e$. Using this, it is not hard to define (mathematically!) the function $E : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that $E(\ulcorner e \urcorner)(m) = n$ iff $e(\overline{m}) \equiv \overline{n}$. When given the numeric code of an expression $e : \mathtt{nat} \to \mathtt{nat}$ and an input $n \in \mathbb{N}$, the function $E$ computes the numeric value of $e(\overline{n})$. Theorem 14.3 on the preceding page,

together with consistency of equivalence, ensures that $E$ is indeed a well-defined function.

Using $E$, we may define another mathematical function $F : \mathbb{N} \to \mathbb{N}$ by the equation $F(m) = E(m)(m)$, so that $F(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \equiv \overline{n}$. We will show that the function $F$ is not definable in $\mathcal{L}\{\mathtt{nat} \to\}$. Suppose for a contradiction that $F$ were defined by the expression $e_F$, which means that $e_F(\overline{\ulcorner e \urcorner}) \equiv e(\overline{\ulcorner e \urcorner})$. Let $e_D$ be the expression $\lambda(x{:}\mathtt{nat}.\,\mathtt{s}(e_F(x)))$. We then have

$$e_D(\overline{\ulcorner e_D \urcorner}) \equiv \mathtt{s}(e_F(\overline{\ulcorner e_D \urcorner})) \tag{14.15}$$

$$\equiv \mathtt{s}(e_D(\overline{\ulcorner e_D \urcorner})), \tag{14.16}$$

which is a contradiction (why?).

## 14.6 Exercises

# Chapter 15

# Plotkin's PCF

The language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, also known as *Plotkin's PCF*, integrates functions and natural numbers using *general recursion*, a means of defining self-referential expressions. In contrast to $\mathcal{L}\{\texttt{nat} \rightarrow\}$ expressions in $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ may not terminate when evaluated; consequently, functions are *partial*, rather than *total*. The "broken arrow" notation for function types emphasizes this fact. Compared to $\mathcal{L}\{\texttt{nat} \rightarrow\}$, the language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ moves the termination proof from the expression itself to the mind of the programmer. The type system no longer ensures termination, which permits a wider range of functions to be defined in the system, but at the cost of admitting infinite loops when the termination proof is either incorrect or absent.

We will consider two important variants of the language, an *eager* version and a *lazy* version. In the eager version values of numeric type are numerals, and a function is applied to the value of its argument. In the lazy version values of numeric type need not be numerals, and a function is applied to its argument in unevaluated form.

## 15.1 Static Semantics

The abstract binding syntax of PCF is given by the following grammar:

| | | | |
|---|---|---|---|
| Type | $\tau$ | ::= | $\texttt{nat} \mid \texttt{parr}(\tau_1, \tau_2)$ |
| Expr | $e$ | ::= | $x \mid \texttt{z} \mid \texttt{s}(e) \mid \texttt{ifz}(e, e_0, x.e_1) \mid \texttt{lam[}\tau\texttt{]}(x.e) \mid$ |
| | | | $\texttt{ap}(e_1, e_2) \mid \texttt{fix[}\tau\texttt{]}(x.e)$ |

The expression $\texttt{fix[}\tau\texttt{]}(x.e)$ is called *general recursion*; it is discussed in more detail below.

Concrete syntax conventions for PCF are given by the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\mathtt{parr}(\tau_1, \tau_2)$ | $\tau_1 \rightharpoonup \tau_2$ |
| $\mathtt{ifz}(e, e_0, x.e_1)$ | $\mathtt{ifz}\, e\, \{\mathtt{z} \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\}$ |
| $\mathtt{fix}[\tau](x.e)$ | $\mathtt{fix}\, x{:}\tau\, \mathtt{is}\, e$ |

The static semantics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is inductively defined by the following rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{15.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{15.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{15.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat} \vdash e_1 : \tau}{\Gamma \vdash \mathtt{ifz}(e, e_0, x.e_1) : \tau} \tag{15.1d}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e) : \mathtt{parr}(\tau_1, \tau_2)} \tag{15.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{parr}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1, e_2) : \tau} \tag{15.1f}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}[\tau](x.e) : \tau} \tag{15.1g}$$

Rule (15.1g) captures the self-referential nature of general recursion: we assume that $x : \tau$ while checking that $e : \tau$ in order to determine whether $\mathtt{fix}[\tau](x.e) : \tau$.

The structural rules are admissible for the static semantics, including in particular substitution.

**Lemma 15.1.** *If* $\Gamma, x : \tau \vdash e' : \tau'$, $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

## 15.2 Dynamic Semantics

The judgement $e$ val determines which expressions are (closed) values. The definition of this judgement varies according to whether we adopt an eager or lazy interpretation of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. This judgement is defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{15.2a}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{s}(e)\ \mathsf{val}} \tag{15.2b}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}} \tag{15.2c}$$

The bracketed premise is to be omitted for the lazy variant, and included for the eager variant.

The dynamic semantics of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ is defined by the following rules:

$$\left\{\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')}\right\} \tag{15.3a}$$

$$\frac{e \mapsto e'}{\mathtt{ifz}(e, e_0, x.e_1) \mapsto \mathtt{ifz}(e', e_0, x.e_1)} \tag{15.3b}$$

$$\frac{}{\mathtt{ifz}(\mathtt{z}, e_0, x.e_1) \mapsto e_0} \tag{15.3c}$$

$$\frac{}{\mathtt{ifz}(\mathtt{s}(e), e_0, x.e_1) \mapsto [e/x]e_1} \tag{15.3d}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1, e_2) \mapsto \mathtt{ap}(e_1', e_2)} \tag{15.3e}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathtt{ap}(e_1, e_2) \mapsto \mathtt{ap}(e_1, e_2')}\right\} \tag{15.3f}$$

$$\frac{\{e_2\ \mathsf{val}\}}{\mathtt{ap}(\mathtt{lam}[\tau](x.e), e_2) \mapsto [e_2/x]e} \tag{15.3g}$$

$$\frac{}{\mathtt{fix}[\tau](x.e) \mapsto [\mathtt{fix}[\tau](x.e)/x]e} \tag{15.3h}$$

The bracketed rules and premises are to be omitted for the lazy variant, and included for the eager variant of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

Rule (15.3h) implements self-reference by substituting the recursive expression itself for the variable $x$ in its body. This is called *unwinding* the recursion. Observe that the variable, $x$, in the expression $\mathtt{fix}[\tau](x.e)$ ranges over computations, and not just values.

## 15.3 Recursive Functions and Primitive Recursion

One use of general recursion is to define recursive functions. A recursive function has the form $\texttt{fun}[\tau_1,\tau_2](x.y.e)$, where $x$ is a variable standing for the function itself, and $y$ is its argument. The static semantics of $\mathcal{L}\{\rightharpoonup\}$ is obtained from that of $\mathcal{L}\{\rightarrow\}$ by replacing the rule for $\lambda$-abstractions with following generalization:

$$\frac{\Gamma, x : \texttt{parr}(\tau_1,\tau_2), y : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}[\tau_1,\tau_2](x.y.e) : \texttt{parr}(\tau_1,\tau_2)}\ . \tag{15.4}$$

A recursive function has type $\texttt{parr}(\tau_1,\tau_2)$ if its body has type $\tau_2$ under the assumption that the function itself has the type $\texttt{parr}(\tau_1,\tau_2)$ and that the argument has type $\tau_1$. Since the variable $x$ stands for the function itself, we assume that which we are trying to prove! This form of "circular" reasoning is typical of self-referential constructs such as recursive functions.

The dynamic semantics rule for applying a recursive function to an argument is given as follows:

$$\frac{\{e_1\ \mathsf{val}\} \quad e = \texttt{fun}[\tau_1,\tau_2](x.y.e')}{\texttt{ap}(e,e_1) \mapsto [e,e_1/x,y]e'} \tag{15.5}$$

At the call site the function itself is substituted for $x$, the name that it has given itself, within its body.

Recursive functions are definable from non-recursive functions and general recursion by taking $\texttt{fun}[\tau_1,\tau_2](x.y.e)$ to stand for the expression

$$\texttt{fix}[\texttt{parr}(\tau_1,\tau_2)](x.\texttt{lam}[\tau_1](y.e)),$$

which is written in concrete syntax in the form

$$\texttt{fix}\, x{:}\tau_1 \rightharpoonup \tau_2\, \texttt{is}\, \lambda(y{:}\tau_1.e).$$

It is easy to check that the static and dynamic semantics of recursive functions are derivable from this definition.

We may define primitive recursion from general recursion by taking $\texttt{rec}[\tau](e,e_0,x.y.e_1)$ to stand for the expression $\texttt{ap}(e',e)$, where $e'$ is the recursive function

$$\texttt{fun}[\texttt{nat},\tau](f.u.\texttt{ifz}(u,e_0,x.[\texttt{ap}(f,x)/y]e_1)).$$

It is easy to check that the static and dynamic semantics of primitive recursion are derivable in $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$ using this definition.

## 15.4 Contextual Semantics

In the next section we will make essential use of a contextual semantics for $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. Recall from Chapter 9 that a contextual semantics has only one transition rule,

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \rightsquigarrow e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \mapsto_{\mathsf{c}} e'} \tag{15.6}$$

This rule is defned in terms of the decomposition of an expression into an evaluation context and a redex, which is then replaced by its contractum.

The instruction steps for $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ are inductively defined by the following inference rules:

$$\frac{}{\mathtt{ifz}(\mathtt{z}, e_0, x\,.\,e_1) \rightsquigarrow e_0} \tag{15.7a}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{ifz}(\mathtt{s}(e), e_0, x\,.\,e_1) \rightsquigarrow [e/x]e_1} \tag{15.7b}$$

$$\frac{\{e_2\ \mathsf{val}\}}{\mathtt{ap}(\mathtt{lam}[\tau_2](x\,.\,e), e_2) \rightsquigarrow [e_2/x]e} \tag{15.7c}$$

$$\frac{}{\mathtt{fix}[\tau](x\,.\,e) \rightsquigarrow [\mathtt{fix}[\tau](x\,.\,e)/x]e} \tag{15.7d}$$

The bracketed premises are to be omitted for the lazy variant, and included for the eager variant.

The evaluation contexts are inductively defined by the following rules:

$$\frac{}{\circ\ \mathsf{ectxt}} \tag{15.8a}$$

$$\left\{\frac{\mathcal{E}\ \mathsf{ectxt}}{\mathtt{s}(\mathcal{E})\ \mathsf{ectxt}}\right\} \tag{15.8b}$$

$$\frac{\mathcal{E}\ \mathsf{ectxt}}{\mathtt{ifz}(\mathcal{E}, e_0, x\,.\,e_1)\ \mathsf{ectxt}} \tag{15.8c}$$

$$\frac{\mathcal{E}_1\ \mathsf{ectxt}}{\mathtt{ap}(\mathcal{E}_1, e_2)\ \mathsf{ectxt}} \tag{15.8d}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad \mathcal{E}_2\ \mathsf{ectxt}}{\mathtt{ap}(e_1, \mathcal{E}_2)\ \mathsf{ectxt}}\right\} \tag{15.8e}$$

The bracketed rules are to be omitted for the lazy variant of the language.

It is a straightforward exercise to define the judgement $e = \mathcal{E}\{e_0\}$, which states that the result of replacing the "hole" in $\mathcal{E}$ by $e_0$ is $e$.

Let us write $e \mapsto_{\mathsf{s}} e'$ for the transition relation defined by the structural operational semantics, and $e \mapsto_{\mathsf{c}} e'$ for the transition relation defined by the contextual semantics.

**Theorem 15.2.** *For any expression $e : \mathtt{nat}$ and any $v$ $\mathsf{val}$, $e \mapsto_{\mathsf{s}}^* v$ iff $e \mapsto_{\mathsf{c}}^* v$*

*Proof.* It suffices to that $e \mapsto_{\mathsf{s}} e'$ iff $e \mapsto_{\mathsf{c}} e'$, as in the proof of Theorem 9.1 on page 73. □

## 15.5 Compactness

An important property of general recursion is called *compactness*, which, roughly speaking, states that only finitely many unwindings of a recursive expression are every necessary to complete the evaluation of a program. While intuitively obvious (one cannot complete infinitely many recursive calls in a finite computation), it is rather tricky to state and prove rigorously. To get a feel for what is involved, we consider two motivating examples.

Consider the familiar factorial function, $f$, in $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$:

$$\mathtt{fix}\, f\mathtt{:nat} \rightharpoonup \mathtt{nat}\, \mathtt{is}\, \lambda(x\mathtt{:nat}.\, \mathtt{ifz}\, x\, \{\mathtt{z} \Rightarrow \mathtt{s(z)} \mid \mathtt{s}(x') \Rightarrow x * f(x')\}).$$

Obviously evaluation of $f(\overline{n})$ requires $n$ recursive calls to the function itself. This means that, for a given input, $n$, we may place a *bound*, $k$, on the recursion that is sufficient to ensure termination of the computation. This can be expressed formally using the $k$-bounded form of factorial, $f^{(k)}$, is written

$$\mathtt{fix}^k\, f\mathtt{:nat} \rightharpoonup \mathtt{nat}\, \mathtt{is}\, \lambda(x\mathtt{:nat}.\, \mathtt{ifz}\, x\, \{\mathtt{z} \Rightarrow \mathtt{s(z)} \mid \mathtt{s}(x') \Rightarrow x * f(x')\}).$$

The superscript $k$ limits the recursion to at most $k$ unwindings, after which the computation diverges. Thus, if $f(\overline{n})$ terminates, then for some $k \geq 0$ (in fact, $k = n$ for this simple case), $f^{(k)}(\overline{n})$ also terminates.

One might expect something even stronger, namely that there is a bound that ensures termination with the *same* value. But this is not always the case! For example, consider the addition function, $a$, of type $\tau = \mathtt{nat} \rightharpoonup (\mathtt{nat} \rightharpoonup \mathtt{nat})$, given by the expression

$$\mathtt{fix}\, p\mathtt{:}\tau\, \mathtt{is}\, \lambda(x\mathtt{:nat}.\, \mathtt{ifz}\, x\, \{\mathtt{z} \Rightarrow id \mid \mathtt{s}(x') \Rightarrow s \circ (p(x'))\}),$$

where $id = \lambda(y{:}\mathtt{nat}.\, y)$ is the identity, $e' \circ e = \lambda(x{:}\tau.\, e'(e(x)))$ is composition, and $s = \lambda(x{:}\mathtt{nat}.\, \mathtt{s}(x))$ is the successor function. The application $a(\overline{m})$ terminates after three transitions, regardless of the value of $m$, resulting in a $\lambda$-abstraction. When $m$ is positive, the result contains a *residual* copy of $a$ itself, which is applied to the predecessor of $m$ as a recursive call. The corresponding $k$-bounded version of $a$, written $a^{(k)}$, also terminates in three steps, provided that $k > 0$. But the result in the case of a positive argument, $m$, is a $\lambda$-abstraction that contains a residual copy of $a^{(k-1)}$, not of $a^{(k)}$ or of $a$ itself.

The proof of compactness is given in terms of the contextual semantics given in Section 15.4 on page 119. This greatly simplifies the proof, because contextual semantics permits us to restrict attention to transitions between complete programs, whereas subsidiary deductions in structural semantics must be defined for expressions of arbitrary type.

As a technical convenience we will enrich the syntax of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ with *bounded recursion*, written $\mathtt{fix}^k\, x{:}\tau\, \mathtt{is}\, e$, where $k \geq 0$. The static semantics is the same as for general recursion, the parameter $k$ playing no role in typing. The dynamic semantics is defined by the following primitive instruction rules:

$$\frac{}{\mathtt{fix}^0[\tau](x.e) \rightsquigarrow \mathtt{fix}^0[\tau](x.e)} \tag{15.9a}$$

$$\frac{}{\mathtt{fix}^{k+1}[\tau](x.e) \rightsquigarrow [\mathtt{fix}^k[\tau](x.e)/x]e} \tag{15.9b}$$

If $k$ is positive, the recursive bound is decremented so that subsequent uses of it will be limited to one fewer unrolling. If $k$ reaches zero, the expression steps to itself so that computation with it diverges with no result.

Let $f^{(\omega)} = \mathtt{fix}\, x{:}\tau\, \mathtt{is}\, e_x$ be an arbitrarily chosen recursive expression such that $f^{(\omega)} : \tau$, and let $f^{(k)} = \mathtt{fix}^k\, x{:}\tau\, \mathtt{is}\, e_x$ be the corresponding $k$-bounded recursive expression, for which we also have $f^{(k)} : \tau$. Observe that by inversion of the static semantics of recursive expressions, we have $x : \tau \vdash e_x : \tau$.

**Lemma 15.3.** *If $e_1 \neq y$ and $e_0 = [f^{(\omega)}/y]e_1 \rightsquigarrow e_0'$, then $e_0' = [f^{(\omega)}/y]e_1'$ for some $e_1'$.*

*Proof.* Immediate, by inspection of Rules (15.7). □

**Lemma 15.4.** *If $[f^{(k)}/y]e \downarrow$, then $[f^{(k+1)}/y]e \downarrow$.*

*Proof.* It is enough to prove that if

$$[f^{(k)}/y]\mathcal{E}\{[f^{(k)}/y]e_0\} \downarrow,$$

then

$$[f^{(k+1)}/y]\mathcal{E}\{[f^{(k+1)}/y]e_0\}\downarrow.$$

□

**Theorem 15.5** (Compactness)**.** *Suppose that* $y : \tau \vdash e : \tau'$*, where* $y \# f^{(\omega)}$*. If* $[f^{(\omega)}/y]e\downarrow$*, then there exists* $k \geq 0$ *such that* $[f^{(k)}/y]e\downarrow$*.*

*Proof.* If $[f^{(\omega)}/y]e$ val, then since $e$ cannot be the variable $y$, it must itself be a value, independently of whether $y$ val. The result then follows immediately, choosing $k$ arbitrarily. Otherwise, suppose that $[f^{(\omega)}/y]e \mapsto_{\mathsf{c}} e'\downarrow$. That is, $[f^{(\omega)}/y]e = \mathcal{E}\{e_0\}$, $e' = \mathcal{E}\{e_0'\}$, and $e_0 \rightsquigarrow e_0'$. Therefore $\mathcal{E}$ has the form $[f^{(\omega)}/y]\mathcal{E}_1$ for some evaluation context $\mathcal{E}_1$, $e_0 = [f^{(\omega)}/y]e_1$ for some closed expression $e_1$, and so $e' = [f^{(\omega)}/y]\mathcal{E}_1\{e_0'\}$.

We proceed by case analysis on whether or not $e_1$ is the distinguished variable, $y$. If so, the instruction step under consideration is precisely the unrolling of the distinguished recursive expression $f^{(\omega)}$. Consequently, $e_0 = f^{(\omega)}$, and, therefore, $e_0' = [f^{(\omega)}/x]e_x$, where we may assume without loss of generality $x \# y$. Now

$$[f^{(\omega)}/y]\mathcal{E}_1\{[f^{(\omega)}/x]e_x\} = [f^{(\omega)}/y](\mathcal{E}_1\{[y/x]e_x\}),$$

and so by induction there exists $k \geq 0$ such that

$$\begin{aligned}[f^{(k)}/y](\mathcal{E}_1\{[y/x]e_x\}) &= [f^{(k)}/y]\mathcal{E}_1\{[y/x]e_x\}\\ &= [f^{(k)}/y]\mathcal{E}_1\{[f^{(k)}/x]e_x\}\downarrow.\end{aligned}$$

Hence, by Lemma 15.4 on the preceding page, noting that $y \# [f^{(k)}/x]e_x$, and applying the dynamic semantics of bounded recursion, we have

$$\begin{aligned}[f^{(k+1)}/y](\mathcal{E}_1\{y\}) &= [f^{(k+1)}/y]\mathcal{E}_1\{f^{(k+1)}\}\\ &\mapsto_{\mathsf{c}} [f^{(k+1)}/y]\mathcal{E}_1\{[f^{(k)}/x]e_x\}\downarrow.\end{aligned}$$

This completes the proof for the case $e_1 = y$.

Otherwise, it follows from Lemma 15.3 on the previous page that $e_0' = [f^{(\omega)}/y]e_1'$ for some expression $e_1'$. That is, we have

$$\begin{aligned}[f^{(\omega)}/y](\mathcal{E}_1\{e_0\}) &= [f^{(\omega)}/y]\mathcal{E}_1\{[f^{(\omega)}/y]e_1\}\\ &\mapsto_{\mathsf{c}} [f^{(\omega)}/y]\mathcal{E}_1\{[f^{(\omega)}/y]e_1'\}\\ &= [f^{(\omega)}/y](\mathcal{E}_1\{e_1'\})\downarrow.\end{aligned}$$

Therefore, by induction, there exists $k \geq 0$ such that

$$[f^{(k)}/y]\mathcal{E}_1\{[f^{(k)}/y]e_1'\} = [f^{(k)}/y](\mathcal{E}_1\{e_1'\}) \downarrow .$$

It follows that

$$[f^{(k)}/y]\mathcal{E}_1\{[f^{(k)}/y]e_1\} \mapsto_{\mathsf{c}} [f^{(k)}/y]\mathcal{E}_1\{[f^{(k)}/y]e_1'\} \downarrow .$$

□

## 15.6 Exercises

# Part V

# Products and Sums

# Chapter 16

# Product Types

The *binary product* of two types consists of *ordered pairs* of values, one from each type in the order specified. The associated eliminatory forms are *projections*, which select the first and second component of a pair. The *nullary product*, or *unit*, type consists solely of the unique "null tuple" of no values, and has no associated eliminatory form.

More generally, the *general*, or *n-ary*, *product* of $n \geq 0$ types consists of the *ordered n-tuples* of values, with the eliminatory forms being the $j$th projection, where $0 \leq j < n$.

The *labelled product*, or *record*, type consists of *labelled n-tuples* in which the components are *labelled* by names. The eliminatory forms access the field of a specified name.

The *object type* is the type consists of records equipped with a name for themselves to permit recursive self-reference. Such types are used to model *objects* (in the sense of "object-oriented programming"), which permit methods to refer to object in which they are contained.

## 16.1 Nullary and Binary Products

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

| | | | |
|---|---|---|---|
| Type | $\tau$ | $::=$ | $\texttt{unit} \mid \texttt{prod}(\tau_1, \tau_2)$ |
| Expr | $e$ | $::=$ | $\texttt{triv} \mid \texttt{pair}(e_1, e_2) \mid \texttt{fst}(e) \mid \texttt{snd}(e)$ |

The concrete syntax is summarized in the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| `unit` | `unit` |
| `triv` | $\langle\rangle$ |
| `prod`$(\tau_1, \tau_2)$ | $\tau_1 \times \tau_2$ |
| `pair`$(e_1, e_2)$ | $\langle e_1, e_2 \rangle$ |
| `fst`$(e)$ | `fst`$(e)$ |
| `snd`$(e)$ | `snd`$(e)$ |

The type `prod`$(\tau_1, \tau_2)$ is sometimes called the *binary product* of the types $\tau_1$ and $\tau_2$, and the type `unit` is correspondingly called the *nullary product* (of no types). We sometimes speak loosely of *product types* in such as way as to cover both the binary and nullary cases. The introductory form for the product type is called *pairing*, and its eliminatory forms are called *projections*. For the unit type the introductory form is called the *unit element*, or *null tuple*. There is no eliminatory form, there being nothing to extract from a null tuple!

The static semantics of product types is given by the following rules.

$$\frac{}{\Gamma \vdash \texttt{triv} : \texttt{unit}} \tag{16.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{pair}(e_1, e_2) : \texttt{prod}(\tau_1, \tau_2)} \tag{16.1b}$$

$$\frac{\Gamma \vdash e : \texttt{prod}(\tau_1, \tau_2)}{\Gamma \vdash \texttt{fst}(e) : \tau_1} \tag{16.1c}$$

$$\frac{\Gamma \vdash e : \texttt{prod}(\tau_1, \tau_2)}{\Gamma \vdash \texttt{snd}(e) : \tau_2} \tag{16.1d}$$

The dynamic semantics of product types is specified by the following rules:

$$\frac{}{\texttt{triv}\ \mathsf{val}} \tag{16.2a}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad \{e_2\ \mathsf{val}\}}{\texttt{pair}(e_1, e_2)\ \mathsf{val}} \tag{16.2b}$$

$$\left\{ \frac{e_1 \mapsto e_1'}{\texttt{pair}(e_1, e_2) \mapsto \texttt{pair}(e_1', e_2)} \right\} \tag{16.2c}$$

$$\left\{ \frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{\texttt{pair}(e_1, e_2) \mapsto \texttt{pair}(e_1, e_2')} \right\} \tag{16.2d}$$

$$\frac{e \mapsto e'}{\texttt{fst}(e) \mapsto \texttt{fst}(e')} \tag{16.2e}$$

$$\frac{e \mapsto e'}{\texttt{snd}(e) \mapsto \texttt{snd}(e')} \tag{16.2f}$$

$$\frac{\{e_1 \text{ val}\} \quad \{e_2 \text{ val}\}}{\texttt{fst}(\texttt{pair}(e_1, e_2)) \mapsto e_1} \tag{16.2g}$$

$$\frac{\{e_1 \text{ val}\} \quad \{e_2 \text{ val}\}}{\texttt{snd}(\texttt{pair}(e_1, e_2)) \mapsto e_2} \tag{16.2h}$$

The bracketed rules and premises are to be omitted for a lazy semantics, and included for an eager semantics of pairing.

**Theorem 16.1** (Safety). *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

*2. If $e : \tau$ then either $e$ val or there exists $e'$ such that $e \mapsto e'$.*

## 16.2 Tuples

The syntax of tuple types is given by the following grammar:

$$\begin{array}{lrcl} \text{Type} & \tau & ::= & \texttt{tpl}(\tau_0, \ldots, \tau_{n-1}) \\ \text{Expr} & e & ::= & \texttt{tpl}(e_0, \ldots, e_{n-1}) \mid \texttt{prj}[i](e) \end{array}$$

The concrete syntax for tuples is given by the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\texttt{tpl}(\tau_0, \ldots, \tau_{n-1})$ | $\langle \tau_0, \ldots, \tau_{n-1} \rangle$ |
| $\texttt{tpl}(e_0, \ldots, e_{n-1})$ | $\langle e_0, \ldots, e_{n-1} \rangle$ |
| $\texttt{prj}[i](e)$ | $e \cdot i$ |

Formally, this grammar is indexed by the size, $n$, of the general product type under consideration. In addition the projections are indexed by a natural number constant, $0 \leq i < n$, indicating the position to select from the $n$-tuple. The re-use of the operator tpl for both a type constructor and a term constructor should cause no confusion, but formally there are two operators of arity $n$, one for forming types, the other for forming expressions.

We may either take these constructs as primitives, treating products as special cases, or define these constructs in terms of products, as follows:

$$\texttt{tpl}(\tau_0,\ldots,\tau_{n-1}) = \begin{cases} \texttt{unit} & \text{if } n = 0 \\ \texttt{prod}(\tau_0,\texttt{tpl}(\tau_1,\ldots,\tau_{n-1})) & \text{if } n > 0 \end{cases} \tag{16.3a}$$

$$\texttt{tpl}(e_0,\ldots,e_{n-1}) = \begin{cases} \texttt{triv} & \text{if } n = 0 \\ \texttt{pair}(e_0,\texttt{tpl}(e_1,\ldots,e_{n-1})) & \text{if } n > 1 \end{cases} \tag{16.3b}$$

$$\texttt{prj}[j]\,(e) = \begin{cases} \texttt{fst}(e) & \text{if } j = 0 \\ \texttt{prj}[j-1]\,(\texttt{snd}(e)) & \text{if } j > 0 \end{cases} \tag{16.3c}$$

These definitions are a bit tricky. The definitions of the $n$-ary product type and the $n$-tuple expression are defined for $n > 0$ in terms of their definition for $n-1$. Moreover, the projections are further parameterized by a constant $0 \leq j < n$ indicating the position to project; these are defined for $j > 0$ in terms of their definitions for $j-1$.

We leave it to the reader to derive the static and dynamic semantics for general product types implied by these definitions.

## 16.3 Labelled Products

*Labelled product*, or *record*, types are a useful generalization of product types in which the components are accessed by name, rather than by position. The benefits of this should be clear: one cannot be expected to remember the meaning of the 7th component of a 13-tuple!

The syntax for records is quite similar to that for $n$-tuples:

$$\begin{array}{lrcl} \text{Type} & \tau & ::= & \texttt{rcd}[l_1,\ldots,l_n]\,(\tau_1,\ldots,\tau_n) \\ \text{Expr} & e & ::= & \texttt{rcd}[l_1,\ldots,l_n]\,(e_1,\ldots,e_n) \mid \texttt{prj}[l]\,(e) \end{array}$$

We use the meta-variable $l$ to range over *labels*, an infinite set of names disjoint from variable names, which label the *fields*, or components, of the tuple. The abstract syntax is defined so that to each choice of $n \geq 0$ and each sequence of labels $l_1,\ldots,l_n$ there is associated an $n$-argument type constructor and an $n$-argument term constructor that attaches the label $l_i$ to the $i$th field of the tuple. Fields are accessed by name, using the familiar *dot notation* to extract the value of a field with a specified label.

Since the fields of a record are accessed by name, it makes sense to identify two types that differ only in the order of their fields. Specifically, if $\pi$ is

a permutation of $\{1,\ldots,n\}$, the we treat the record type $\texttt{rcd}[l_1,\ldots,l_n](\tau_1,\ldots,\tau_n)$ as identical to the record type

$$\texttt{rcd}[l_{\pi(1)},\ldots,l_{\pi(n)}](\tau_{\pi(1)},\ldots,\tau_{\pi(n)}).$$

If we assume given a linear ordering, $l \leq l'$, on labels, and then we may choose the permutation $\pi$ so that if $i \leq j$, then $l_{\pi(i)} \leq l_{\pi(j)}$. This permutation may then be regarded as determining a canonical representative of each such equivalence class; two record types are equal iff they are the same when their fields are re-ordered according to such a permutation $\pi$.

The correspondence between abstract and concrete syntax is given by the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\texttt{rcd}[l_1,\ldots,l_n](\tau_1,\ldots,\tau_n)$ | $\langle l_1:\tau_1,\ldots,l_n:\tau_n\rangle$ |
| $\texttt{rcd}[l_1,\ldots,l_n](e_1,\ldots,e_n)$ | $\langle l_1=e_1,\ldots,l_n=e_n\rangle$ |
| $\texttt{prj}[l](e)$ | $e\cdot l$ |

The static semantics of records is given by the following rules:

$$\frac{\Gamma\vdash e_1:\tau_1 \quad\ldots\quad \Gamma\vdash e_n:\tau_n}{\Gamma\vdash \texttt{rcd}[l_1,\ldots,l_n](e_1,\ldots,e_n):\texttt{rcd}[l_1,\ldots,l_n](\tau_1,\ldots,\tau_n)} \tag{16.4a}$$

$$\frac{\Gamma\vdash e:\texttt{rcd}[l_1,\ldots,l_n](\tau_1,\ldots,\tau_n)}{\Gamma\vdash \texttt{prj}[l_i](e):\tau_i} \tag{16.4b}$$

The dynamic semantics is specified by these rules:

$$\frac{\{e_1\ \mathsf{val}\} \quad\ldots\quad \{e_n\ \mathsf{val}\}}{\texttt{rcd}[l_1,\ldots,l_n](e_1,\ldots,e_n)\ \mathsf{val}} \tag{16.5a}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad\ldots\quad e_{i-1}\ \mathsf{val} \quad e_i\mapsto e_i' \quad e_{i+1}'=e_{i+1} \quad\ldots\quad e_n'=e_n}{\texttt{rcd}[l_1,\ldots,l_n](e_1,\ldots,e_n)\mapsto\texttt{rcd}[l_1,\ldots,l_n](e_1',\ldots,e_n')}\right\} \tag{16.5b}$$

$$\frac{e\mapsto e'}{\texttt{prj}[l](e)\mapsto\texttt{prj}[l](e')} \tag{16.5c}$$

$$\frac{\{e_1\ \mathsf{val}\} \quad\ldots\quad \{e_n\ \mathsf{val}\}}{\texttt{prj}[l_i](\texttt{rcd}[l_1,\ldots,l_n](e_1,\ldots,e_n))\mapsto e_i} \tag{16.5d}$$

The bracketed rules and premises are to be omitted for a lazy interpretation, and included for an eager interpretation.

**Theorem 16.2** (Safety for Records)**.** *1. If $e:\tau$ and $e\mapsto e'$, then $e:\tau'$.*

*2. If $e:\tau$, then either $e\ \mathsf{val}$ or $e\mapsto e'$ for some $e'$.*

## 16.4 Object Types

An *object* (in the sense of "object-oriented programming") is a record of functions, called *methods*, that may refer to one another by selection from a variable standing for the object itself. This variables is often called `this`, or `self`, for the obvious reason that when used within a record epression, it refers to the object itself.

The abstract syntax of object types is given by the following grammar.

$$\begin{array}{lrcl} \text{Type} & \tau & ::= & \texttt{obj}[l_1,\ldots,l_n](\tau_1,\ldots,\tau_n) \\ \text{Expr} & e & ::= & \texttt{prj}[l](e) \mid \texttt{obj}[\tau; l_1,\ldots,l_n](x.e_1,\ldots,x.e_n) \end{array}$$

The following chart summarizes the corresponding concrete syntax:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\texttt{obj}[l_1,\ldots,l_n](\tau_1,\ldots,\tau_n)$ | $\texttt{obj}\langle l_1:\tau_1,\ldots,l_n:\tau_n\rangle$ |
| $\texttt{prj}[l](e)$ | $e\cdot l$ |
| $\texttt{obj}[\tau; l_1,\ldots,l_n](x.e_1,\ldots,x.e_n)$ | $\texttt{obj}\, x{:}\tau\, \texttt{is}\, \langle l_1 = e_1,\ldots,l_n = e_n\rangle$ |

In the object expression $\texttt{obj}\, x{:}\tau\, \texttt{is}\, \langle l_1 = e_1,\ldots,l_n = e_n\rangle$ the variable $x$ stands for the object itself, and is bound in *each* of the sub-expressions $e_1, \ldots, e_n$. The binding structure is clarified in the abstract syntax, in which there is one bound variable for each component of the object.

The static semantics is a straightforward generalization of that of ordinary (non-self-referential) records.

$$\frac{(\tau = \texttt{obj}[l_1,\ldots,l_n](\tau_1,\ldots,\tau_n)) \quad \Gamma, x:\tau \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma, x:\tau \vdash e_n : \tau_n}{\Gamma \vdash \texttt{obj}[\tau; l_1,\ldots,l_n](x.e_1,\ldots,x.e_n) : \tau} \tag{16.6a}$$

$$\frac{\Gamma \vdash e : \texttt{obj}[l_1,\ldots,l_n](\tau_1,\ldots,\tau_n)}{\Gamma \vdash \texttt{prj}[l_i](e) : \tau_i} \tag{16.6b}$$

Each field is type checked under the assumption that the variable, $x$, which stands for the object itself, has the type of the entire object.

The dynamic semantics is defined to "unroll the recursion" when a field is selected from the object.

$$\frac{}{\texttt{obj}[\tau; l_1,\ldots,l_n](x.e_1,\ldots,x.e_n)\ \mathsf{val}} \tag{16.7a}$$

$$\frac{e \mapsto e'}{\texttt{prj}[l](e) \mapsto \texttt{prj}[l](e')} \tag{16.7b}$$

$$\frac{(e = \texttt{obj}[\tau; l_1, \ldots, l_n](x.e_1, \ldots, x.e_n))}{\texttt{prj}[l_i](e) \mapsto [e/x]e_i} \tag{16.7c}$$

The last rule ensures that the bound variable, $x$, stands for the object itself whenever a field is projected from an object. There is no distinction between an eager and a lazy semantics for objects, because each of an object lies within the scope of the bound variable standing for the object itself.

These rules suggest that objects may be viewed as self-referential records obtained through the combination of record types and general recursion. Specifically, we may regard the object expression

$$\texttt{obj}[\tau; l_1, \ldots, l_n](x.e_1, \ldots, x.e_n)$$

as standing for the recursive record expression

$$\texttt{fix}[\tau](x.\texttt{rcd}[l_1, \ldots, l_n](e_1, \ldots, e_n)).$$

The use of general recursion means that the representation of an object expression is not a value. In particular, the components of the object can be computed at object creation time, and any of these computations might diverge when the object expression is evaluated. But when self-referential records are used to represent objects, each of the components $e_i$ is a method represented as an explicit $\lambda$-abstraction, which is, of course, a value. Consequently, the fixed point operation evaluates in one step to a record of $\lambda$'s, all of which are values.

## 16.5 Exercises

1. State and prove the canonical forms lemma for unit and product types.
2. Prove the safety theorem for unit and product types under either the.
3. State the static and dynamic semantics for general products implied by the definitions given in Section 16.1 on page 127.
4. Functional update, concatenation, restriction, other record operations?
5. Define recursor from iterator using products.

# Chapter 17

# Sum Types

Most data structures involve alternatives such as the distinction between a leaf and an interior node in a tree, or a choice in the outermost form of a piece of abstract syntax. Importantly, the choice determines the structure of the value. For example, nodes have children, but leaves do not, and so forth. These concepts are expressed by *sum types*, specifically the *binary sum*, which offers a choice of two things, and the *nullary sum*, which offers a choice of no things. These generalize to $n$-ary sums, a choice among $n$ things, and to *labelled sums*, in which the selection is governed by a label.

## 17.1 Binary and Nullary Sums

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

$$\begin{array}{llll} \text{Type} & \tau & ::= & \mathtt{void} \mid \mathtt{sum}(\tau_1, \tau_2) \\ \text{Expr} & e & ::= & \mathtt{abort}[\tau](e) \mid \mathtt{inl}[\tau](e) \mid \mathtt{inr}[\tau](e) \mid \\ & & & \mathtt{case}[\tau_1, \tau_2](e, x_1.e_1, x_2.e_2) \end{array}$$

The corresponding concrete syntax is summarized in the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\mathtt{void}$ | $\mathtt{void}$, $\bot$ |
| $\mathtt{abort}[\tau](e)$ | $\mathtt{abort}_\tau\, e$ |
| $\mathtt{sum}(\tau_1, \tau_2)$ | $\tau_1 + \tau_2$ |
| $\mathtt{inl}[\tau](e)$ | $\mathtt{inl}_\tau(e)$ |
| $\mathtt{inr}[\tau](e)$ | $\mathtt{inr}_\tau(e)$ |
| $\mathtt{case}[\tau_1, \tau_2](e, x_1.e_1, x_2.e_2)$ | $\mathtt{case}\, e\, \{\mathtt{inl}(x_1) \Rightarrow e_1 \mid \mathtt{inr}(x_2) \Rightarrow e_2\}$ |

The type `void` is the *nullary sum* type, whose values are selected from a choice of zero alternatives — there are no values of this type, and so no introductory forms. The eliminatory form, $\texttt{abort}[\tau](e)$, aborts the computation in the event that $e$ evaluates to a value, which it cannot do. The type $\tau = \texttt{sum}(\tau_1, \tau_2)$ is the *binary sum*. Its introductory forms have the form $\texttt{inl}[\tau](e)$ or $\texttt{inl}[\tau](e)$, indicating which of the two possible choices by *tagging* a value of the left or right summand as being a value of the sum type. The eliminatory form performs a case analysis on the tag of a value, decomposing it into its constituent parts.

The static semantics of sum types is given by the following rules.

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{abort}[\tau](e) : \tau} \tag{17.1a}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \texttt{sum}(\tau_1, \tau_2)}{\Gamma \vdash \texttt{inl}[\tau](e) : \tau} \tag{17.1b}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \tau = \texttt{sum}(\tau_1, \tau_2)}{\Gamma \vdash \texttt{inr}[\tau](e) : \tau} \tag{17.1c}$$

$$\frac{\Gamma \vdash e : \texttt{sum}(\tau_1, \tau_2) \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}[\tau_1, \tau_2](e, x_1.e_1, x_2.e_2) : \tau} \tag{17.1d}$$

Just as for the conditional expression considered in Chapter 13, both branches of the case analysis must have the same type. Since the type expresses a static "prediction" on the form of the value of an expression, and since a value of sum type could evaluate to either form at run-time, we must insist that both branches yield the same type.

The dynamic semantics of sums is given by the following rules:

$$\frac{\{e \ \mathsf{val}\}}{\texttt{inl}[\tau](e) \ \mathsf{val}} \tag{17.2a}$$

$$\frac{\{e \ \mathsf{val}\}}{\texttt{inl}[\tau](e) \ \mathsf{val}} \tag{17.2b}$$

$$\left\{\frac{e \mapsto e'}{\texttt{inl}[\tau](e) \mapsto \texttt{inl}[\tau](e')}\right\} \tag{17.2c}$$

$$\left\{\frac{e \mapsto e'}{\texttt{inr}[\tau](e) \mapsto \texttt{inr}[\tau](e')}\right\} \tag{17.2d}$$

$$\frac{e \mapsto e'}{\texttt{case}[\tau_1, \tau_2](e, x_1.e_1, x_2.e_2) \mapsto \texttt{case}[\tau_1, \tau_2](e', x_1.e_1, x_2.e_2)} \tag{17.2e}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{case}[\tau_1,\tau_2](\mathtt{inl}[\tau](e),x_1.e_1,x_2.e_2) \mapsto [e/x_1]e_1} \tag{17.2f}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{case}[\tau_1,\tau_2](\mathtt{inr}[\tau](e),x_1.e_1,x_2.e_2) \mapsto [e/x_2]e_2} \tag{17.2g}$$

The bracketed premises and rules are to be included for an eager semantics, and excluded for a lazy semantics.

The coherence of the static and dynamic semantics is stated and proved as usual.

**Theorem 17.1** (Safety)**.** *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

*2. If $e : \tau$, then either $e$ $\mathsf{val}$ or $e \mapsto e'$ for some $e'$.*

One use of sum types is to define the *Boolean* type, which has the following syntax:

$$\begin{array}{lrcl} \text{Type} & \tau & ::= & \mathtt{bool} \\ \text{Expr} & e & ::= & \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}(e,e_1,e_2) \end{array}$$

This type is definable in the presence of sums and nullary products according to the following equations:

$$\mathtt{bool} = \mathtt{sum}(\mathtt{unit},\mathtt{unit}) \tag{17.3a}$$

$$\mathtt{tt} = \mathtt{inl}[\mathtt{bool}](\mathtt{triv}) \tag{17.3b}$$

$$\mathtt{ff} = \mathtt{inr}[\mathtt{bool}](\mathtt{triv}) \tag{17.3c}$$

$$\mathtt{if}(e,e_1,e_2) = \mathtt{case}[\mathtt{unit},\mathtt{unit}](e,x_1.e_1,x_2.e_2) \tag{17.3d}$$

The variables $x_1$ and $x_2$ are dummies, since their type, `unit`, determines their value, `triv`, and, moreover, they do not occur freely in $e_1$ or $e_2$.

Another use of sums is to define the *option* types, which have the following syntax:

$$\begin{array}{lrcl} \text{Type} & \tau & ::= & \mathtt{opt}(\tau) \\ \text{Expr} & e & ::= & \mathtt{null} \mid \mathtt{just}(e) \mid \mathtt{ifnull}[\tau](e,e_1,x.e_2) \end{array}$$

The type $\mathtt{opt}(\tau)$ represents the type of "optional" values of type $\tau$. The introductory forms are `null`, corresponding to "no value", and $\mathtt{just}(e)$, corresponding to a specified value of type $\tau$. The elimination form discriminates between the two possibilities.

The option type is definable from sums and nullary products according to the following equations:

$$\texttt{opt}(\tau) = \texttt{sum}(\texttt{unit}, \tau) \tag{17.4a}$$
$$\texttt{null} = \texttt{inl}[\texttt{opt}(\tau)](\texttt{triv}) \tag{17.4b}$$
$$\texttt{just}(e) = \texttt{inr}[\texttt{opt}(\tau)](e) \tag{17.4c}$$
$$\texttt{ifnull}[\tau](e, e_1, x_2.e_2) = \texttt{case}[\texttt{unit}, \tau](e, x_1.e_1, x_2.e_2) \tag{17.4d}$$

We leave it to the reader to examine the static and dynamic semantics implied by these definitions.

It is important to understand the difference between the types `unit` and `void`, which are often confused. The type `unit` has exactly one element, `triv`, whereas the type `void` has no elements at all. Consequently, if $e$ : `unit`, then if $e$ evaluates to a value, it must be `unit` — in other words, $e$ has *no interesting value* (but it could diverge). On the other hand, if $e$ : `void`, then *e must not return a value*, because if it were to have a value, it would have to be a value of type `void`, of which there are none. This shows that the `void` type in Java and related languages is really the type `unit`, because it indicates that an expression of that type has no interesting result, not that it must not return!

## 17.2 Labelled Sums

Binary and nullary sums are sufficient to define generalized $n$-ary sums, in a manner analogous to the definition of $n$-ary products from nullary and binary products in Chapter 16. We leave the details of this derivation to the reader, and concentrate instead on *labelled sums*, or *labelled variants*. Labelled sums are a form of $n$-ary sum in which the alternatives are labelled by names, rather than by positions.

The abstract syntax of labelled sums is given by the following grammar:

| | | | |
|---|---|---|---|
| Type | $\tau$ | ::= | $\texttt{var}[l_1, \ldots, l_n](\tau_1, \ldots, \tau_n)$ |
| Expr | $e$ | ::= | $\texttt{inj}[\tau; l](e) \mid \texttt{case}[\tau_1, \ldots, \tau_n; l_1, \ldots, l_n](e, x_1.e_1, \ldots, x_n.e_n)$ |

The corresponding concrete syntax is given by the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\texttt{var}[l_1, \ldots, l_n](\tau_1, \ldots, \tau_n)$ | $[l_1 : \tau_1, \ldots, l_n : \tau_n]$ |
| $\texttt{inj}[\tau; l](e)$ | $[l = e]_\tau$ |
| $\texttt{case}[\tau_1, \ldots, \tau_n; l_1, \ldots, l_n](e, x_1.e_1, \ldots, x_n.e_n)$ | $\texttt{case}\, e\, \{[l_1{=}x_1] \Rightarrow e_1 \mid \ldots \mid [l_n{=}x_n] \Rightarrow e_n\}$ |

It is an awkwardness of the syntax that injections must be marked with the sum type into which the injection is being made. This is to ensure that every expression has a unique type, since we cannot recover the entire sum type from the type of one of its variants.

The static semantics is given by the following rules:

$$\frac{\tau = \mathtt{var}[l_1,\ldots,l_n](\tau_1,\ldots,\tau_n) \qquad \Gamma \vdash e : \tau_i}{\Gamma \vdash \mathtt{inj}[\tau;l_i](e) : \tau} \tag{17.5a}$$

$$\frac{\Gamma \vdash e : \mathtt{var}[l_1,\ldots,l_n](\tau_1,\ldots,\tau_n) \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \ldots \quad \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \mathtt{case}[\tau_1,\ldots,\tau_n;l_1,\ldots,l_n](e, x_1.e_1, \ldots, x_n.e_n) : \tau} \tag{17.5b}$$

These rules are a straightforward generalization of those for binary sums to permit an arbitrary number of labelled variants.

We leave as an exercise to formulate the dynamic semantics of labelled sums and to prove this extension sound.

## 17.3 Exercises

1. Formulate general $n$-ary sums in terms of nullary and binary sums.
2. Explain why is makes little sense to consider self-referential sum types.

# Part VI

# Recursive Types

# Chapter 18

# Inductive and Co-Inductive Types

The *inductive* and the *coinductive* types are two important classes of recursive types. Inductive types correspond to *least*, or *initial*, solutions of certain type isomorphism equations, and coinductive types correspond to their *greatest*, or *final*, solutions. Intuitively, inductive types are considered to be the "smallest" types containing their introduction forms; the elimination form is then a form of recursion over the introduction forms. Dually, coinductive types are considered to be the "greatest" types consistent with their elimination forms; the introduction forms are a means of presenting elements as required by the elimination forms.

The motivating example of an inductive type is the type of natural numbers. It is the least type containing the introductory forms `z` and `s(`$e$`)`, where $e$ is again an introductory form. To compute with a number we define a recursive procedure that returns a specified value on `z`, and, for `s(`$e$`)`, returns a value defined in terms of the recursive call to itself on $e$. Other examples of inductive types are strings, lists, trees, and any other type that may be thought of as finitely generated from its introductory forms.

The motivating example of a coinductive type is the type of streams of natural numbers. Every stream may be thought of as being in the process of generation of pairs consisting of a natural number (its head) and another stream (its tail). To create a stream we define a generator that, when prompted, produces such a natural number and a co-recursive call to the generator. Other examples of coinductive types include infinite regular trees, and the so-called lazy natural numbers, which include a "point at infinity" consisting of an infinite stack of successors.

We will consider inductive and coinductive types in a setting similar to $\mathcal{L}\{\texttt{nat}\rightarrow\}$, in which all functions are total. The type `nat` is replaced by a general inductive type constructor, written $\mu_{\mathsf{i}}(t.\tau)$, and we also consider a coinductive type constructor, written $\mu_{\mathsf{f}}(t.\tau)$. The resulting language is called $\mathcal{L}\{\mu_{\mathsf{i}}\mu_{\mathsf{f}}\rightarrow\}$.

## 18.1 Static Semantics

### 18.1.1 Types and Operators

The syntax of inductive and coinductive types involves *type variables*, which are, of course, variables ranging over the class of types. The abstract syntax of types is given by the following grammar:

$$\text{Type} \qquad \tau \quad ::= \quad t \mid \texttt{arr}(\tau_1,\tau_2) \mid \texttt{ind}(t.\tau) \mid \texttt{coi}(t.\tau).$$

The concrete syntax conventions are summarized by the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\texttt{arr}(\tau_1,\tau_2)$ | $\tau_1 \rightarrow \tau_2$ |
| $\texttt{ind}(t.\tau)$ | $\mu_{\mathsf{i}}(t.\tau)$ |
| $\texttt{coi}(t.\tau)$ | $\mu_{\mathsf{f}}(t.\tau)$ |

The subscripts on the inductive and coinductive types are intended to indicate "initial" and "final", respectively.

We will consider *type formation* judgements of the form

$$t_1\ \mathsf{type},\ldots,t_n\ \mathsf{type} \mid \tau\ \mathsf{type},$$

where $t_1,\ldots,t_n$ are type names. We let $\Delta$ range over finite sets of hypotheses of the form $t\ \mathsf{type}$, where $t\ \mathsf{name}$ is a type name. The type formation judgement is inductively defined by the following rules:

$$\frac{}{\Delta, t\ \mathsf{type} \mid t\ \mathsf{type}} \tag{18.1a}$$

$$\frac{\Delta \mid \tau_1\ \mathsf{type} \quad \Delta \mid \tau_2\ \mathsf{type}}{\Delta \mid \texttt{arr}(\tau_1,\tau_2)\ \mathsf{type}} \tag{18.1b}$$

$$\frac{\Delta, t\ \mathsf{type} \mid \tau\ \mathsf{type} \quad \Delta \mid t.\tau\ \mathsf{pos}}{\Delta \mid \texttt{ind}(t.\tau)\ \mathsf{type}} \tag{18.1c}$$

$$\frac{\Delta, t \text{ type} \mid \tau \text{ type} \quad \Delta \mid t.\tau \text{ pos}}{\Delta \mid \mathtt{coi}(t.\tau) \text{ type}} \tag{18.2}$$

The premises on Rules (18.1c) and (18.2) involve a judgement of the form $t.\tau$ pos, which will be explained in Section 18.2 on the next page.

A *type operator* is an abstractor of the form $t.\tau$ such that $t$ type $\mid$ $\tau$ type. Thus a type operator may be thought of as a type, $\tau$, with a distinguished free variable, $t$, possibly occurring in it. It follows from the meaning of the hypothetical judgement that if $t.\tau$ is a well-formed type operator, and $\sigma$ type, then $[\sigma/t]\tau$ type. Thus, a type operator may also be thought of as a mapping from types to types given by substitution.

As an example of a type operator, consider the abstractor $t.\mathtt{unit} + t$, which will be used in the definition of the natural numbers as an inductive type. Other examples include $t.\mathtt{unit} + (\mathtt{nat} \times t)$, which underlies the definition of the inductive type of lists of natural numbers, and $t.\mathtt{nat} \times t$, which underlies the coinductive type of streams of natural numbers.

### 18.1.2 Expressions

The abstract syntax of expressions for inductive and coinductive types is given by the following grammar:

$$\begin{array}{rrcl} \text{Expr} & e & ::= & \mathtt{in}[t.\tau](e) \mid \mathtt{rec}[t.\tau](x.e, e') \mid \\ & & & \mathtt{out}[t.\tau](e) \mid \mathtt{gen}[t.\tau](x.e, e') \end{array}$$

The concrete syntax is summarized by the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\mathtt{in}[t.\tau](e)$ | $\mathtt{in}_{t.\tau}(e)$ |
| $\mathtt{rec}[t.\tau](x.e, e')$ | $\mathtt{rec}_{t.\tau}(x.e, e')$ |
| $\mathtt{out}[t.\tau](e)$ | $\mathtt{out}_{t.\tau}(e)$ |
| $\mathtt{gen}[t.\tau](x.e, e')$ | $\mathtt{gen}_{t.\tau}(x.e, e')$ |

There is a pleasing symmetry between inductive and coinductive types that arises from the underlying duality of their semantics.

The static semantics for inductive and coinductive types is given by the following typing rules:

$$\frac{\Gamma \vdash e : [\mathtt{ind}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{in}[t.\tau](e) : \mathtt{ind}(t.\tau)} \tag{18.3a}$$

$$\frac{\Gamma \vdash e' : \mathtt{ind}(t.\tau) \quad \Gamma, x : [\rho/t]\tau \vdash e : \rho}{\Gamma \vdash \mathtt{rec}[t.\tau](x.e, e') : \rho} \tag{18.3b}$$

$$\frac{\Gamma \vdash e : \mathtt{coi}(t.\tau)}{\Gamma \vdash \mathtt{out}[t.\tau](e) : [\mathtt{coi}(t.\tau)/t]\tau} \tag{18.3c}$$

$$\frac{\Gamma \vdash e' : \rho \quad \Gamma, x : \rho \vdash e : [\rho/t]\tau}{\Gamma \vdash \mathtt{gen}[t.\tau](x.e, e') : \mathtt{coi}(t.\tau)} \tag{18.3d}$$

The dynamic semantics of these constructs is given in terms of the action of a positive type operator, which we now define.

## 18.2 Positive Type Operators

The formation of inductive and coinductive types is restricted to a special class of type operators, called the *positive type operators*.[1] These are type operators of the form $t.\tau$ in which $t$ is restricted so that its occurrences within $\tau$ do not lie within the domain of a function type. The prototypical example of a type operator that is *not* positive is the operator $t.t \to t$, in which $t$ occurs in both the domain and the range of a function type. On the other hand, the type operator $t.\mathtt{nat} \to t$ is positive, as is $t.u \to t$, where $u$ type is some type variable other than $t$. Thus, the positivity restriction applies only to the bound type variable of the abstractor, and not to any other type variable.

The judgement $\Delta \mid t.\tau$ pos is inductively defined by the following rules:

$$\frac{}{\Delta \mid t.t \text{ pos}} \tag{18.4a}$$

$$\frac{\Delta \mid \tau \text{ type}}{\Delta \mid t.\tau \text{ pos}} \tag{18.4b}$$

$$\frac{\Delta \mid \tau_1 \text{ type} \quad \Delta \mid t.\tau_2 \text{ pos}}{\Delta \mid t.\tau_1 \to \tau_2 \text{ pos}} \tag{18.4c}$$

$$\frac{\Delta, u \text{ type} \mid t.\tau \text{ pos}}{\Delta \mid t.\mu_{\mathsf{i}}(u.\tau) \text{ pos}} \tag{18.4d}$$

[1] These are, in fact, *strictly positive* type operators; there is a more permissive notion of positive operator that we shall not consider here.

$$\frac{\Delta, u \text{ type} \mid t.\tau \text{ pos}}{\Delta \mid t.\mu_{\mathsf{f}}(u.\tau) \text{ pos}} \tag{18.4e}$$

In the latter two rules we assume that $u \# t$, which is always achievable up to $\alpha$-equivalence. Notice that in Rule (18.4c), the type variable $t$ is not permitted to occur in $\tau_1$, the domain type of the function type.

Positivity is preserved under substitution.

**Lemma 18.1.** *If $t.\sigma$ pos and $t.\tau$ pos, then $t.[\sigma/u]\tau$ pos.*

The *covariant action* of a positive type operator $t.\tau$ consists of two parts, an action on types and an action on abstractors. The action on types is given by substitution: $(t.\tau)^*(\sigma) := [\sigma/t]\tau$. The action on abstractors

$$(t.\tau)^*(x.e) = x'.e'$$

is defined by the following rules:

$$\frac{}{(t.t)^*(x.e) = x.e} \tag{18.5a}$$

$$\frac{}{(t.\tau)^*(x.e) = x.x} \tag{18.5b}$$

$$\frac{(t.\tau_2)^*(x.e) = x_2.e_2}{(t.\tau_1 \to \tau_2)^*(x.e) = x'.\lambda(x_1:\tau_1.[x'(x_1)/x_2]e_2)} \tag{18.5c}$$

$$\frac{(t.[\mu_{\mathsf{i}}(u.[\sigma'/t]\tau)/u]\tau)^*(x.e) = x'.e'}{(t.\mu_{\mathsf{i}}(u.\tau))^*(x.e) = y.\mathtt{rec}_{u.[\sigma/t]\tau}(x'.\mathtt{in}_{u.[\sigma'/t]\tau}(e'), y)} \tag{18.5d}$$

$$\frac{(t.[\mu_{\mathsf{f}}(u.[\sigma/t]\tau)/u]\tau)^*(x.e) = x'.e'}{(t.\mu_{\mathsf{f}}(u.\tau))^*(x.e) = y.\mathtt{gen}_{u.[\sigma'/t]\tau}(x'.[\mathtt{out}_{u.[\sigma/t]\tau}(x')/x']e', y)} \tag{18.5e}$$

The covariant action on abstractors is type-consistent with its action on types in the following sense.

**Lemma 18.2.** *If $x : \sigma \vdash e : \sigma'$, then $x' : (t.\tau)^*(\sigma) \vdash e' : (t.\tau)^*(\sigma')$.*

## 18.3 Dynamic Semantics

The dynamic semantics of inductive and coinductive types is given in terms of the covariant action of the associated type operator. Specifically, we take the following axioms as the primitive steps of our semantics:

$$\frac{(t.\tau)^*(x'.\mathtt{rec}_{t.\tau}(x.e, x')) = x''.e''}{\mathtt{rec}_{t.\tau}(x.e, \mathtt{in}_{t.\tau}(e')) \mapsto [[e'/x'']e''/x]e} \tag{18.6a}$$

$$\frac{(t.\tau)^*(x'.\mathtt{gen}_{t.\tau}(x.e, x')) = x''.e''}{\mathtt{out}_{t.\tau}(\mathtt{gen}_{t.\tau}(x.e, e')) \mapsto [[e'/x]e/x'']e''} \tag{18.6b}$$

The remaining rules of the dynamic semantics are specified as follows:

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{in}_{t.\tau}(e)\ \mathsf{val}} \tag{18.7a}$$

$$\frac{\{e'\ \mathsf{val}\}}{\mathtt{gen}_{t.\tau}(x.e, e')\ \mathsf{val}} \tag{18.7b}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{in}_{t.\tau}(e) \mapsto \mathtt{in}_{t.\tau}(e')}\right\} \tag{18.7c}$$

$$\frac{e' \mapsto e''}{\mathtt{rec}_{t.\tau}(x.e, e') \mapsto \mathtt{rec}_{t.\tau}(x.e, e'')} \tag{18.7d}$$

$$\frac{e \mapsto e'}{\mathtt{out}_{t.\tau}(e) \mapsto \mathtt{out}_{t.\tau}(e')} \tag{18.7e}$$

$$\left\{\frac{e' \mapsto e''}{\mathtt{gen}_{t.\tau}(x.e, e') \mapsto \mathtt{gen}_{t.\tau}(x.e, e'')}\right\} \tag{18.7f}$$

**Lemma 18.3.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

**Lemma 18.4.** *If $e : \tau$, then either $e$ $\mathsf{val}$ or there exists $e'$ such that $e \mapsto e'$.*

Although we shall not give a proof here, the language $\mathcal{L}\{\mu_i\mu_f\to\}$ is terminating, and all functions defined within it are total (defined for every argument).

**Theorem 18.5.** *If $e : \tau$, then there exists $v$ $\mathsf{val}$ such that $e \mapsto^* v$.*

## 18.4 Fixed Point Properties

Inductive and coinductive types enjoy an important property that will play a prominent role in Chapter 19, called a *fixed point property*, that characterizes them as solutions to recursive type equations. Specifically, the inductive type $\mu_{\mathsf{i}}(t.\tau)$ is isomorphic to its unrolling,

$$\mu_{\mathsf{i}}(t.\tau) \cong [\mu_{\mathsf{i}}(t.\tau)/t]\tau,$$

and, dually, the coinductive type is isomorphic to its unrolling,

$$\mu_{\mathsf{f}}(t.\tau) \cong [\mu_{\mathsf{f}}(t.\tau)/t]\tau$$

The isomorphism arises from the invertibility of $\mathtt{in}_{t.\tau}(-)$ in the inductive case and of $\mathtt{out}_{t.\tau}(-)$ in the coinductive case, with the required inverses given as follows:

$$x.\mathtt{in}^{-1}_{t.\tau}(x) = x.\mathtt{rec}_{t.\tau}((t.\tau)^*(y.\mathtt{in}_{t.\tau}(y)), x) \tag{18.8}$$

$$x.\mathtt{out}^{-1}_{t.\tau}(x) = x.\mathtt{gen}_{t.\tau}((t.\tau)^*(y.\mathtt{out}_{t.\tau}(y)), x) \tag{18.9}$$

(We are not yet in a position to prove that these are, respectively, inverses to $\mathtt{in}_{t.\tau}(-)$ and $\mathtt{out}_{t.\tau}(-)$, but see Chapter 49 for more on equational reasoning.)

Thus, both the inductive and the coinductive type are solutions (in $X$) to the type isomorphism

$$X \cong [X/t]\tau.$$

What distinguishes the two solutions, in general, is that the inductive type is the *initial*, or *least*, solution, whereas the coinductive type is the *final*, or *greatest*, solution to the isomorphism equation. This implies, in particular, that there is an abstractor $x.e$ such that

$$x : \mu_{\mathsf{i}}(t.\tau) \vdash e : \mu_{\mathsf{f}}(t.\tau),$$

but there is, in general, no such abstractor in the opposite direction.

## 18.5 Exercises

1. Extend the covariant action to nullary and binary products and sums.

2. Prove progress and preservation.

3. Show that the required abstractor mapping the inductive to the coinductive type associated with a type operator is given by the equation

$$x.\mathtt{gen}_{t.\tau}(y.\mathtt{in}_{t.\tau}^{-1}(y), x).$$

Characterize the behavior of this term when $x$ is replaced by an element of the inductive type.

# Chapter 19

# Recursive Types

Inductive and coinductive types provide solutions to type isomorphism equations of the form

$$X \cong [X/t]\tau,$$

where $t.\tau$ is a positive type operator. For such operators the inductive type, $\mu_{\mathsf{i}}(t.\tau)$ provides the least, or initial, solution to the isomorphism equation, and the coinductive type $\mu_{\mathsf{f}}(t.\tau)$ provides the greatest, or final, solution to it. They are therefore both fixed points (up to isomorphism) of the specified type operator. For example, if $N$ is the type operator $t.\mathtt{unit} + t$, then $\mu_{\mathsf{i}}(N)$ is isomorphic to $\mathtt{unit} + \mu_{\mathsf{i}}(N)$, and $\mu_{\mathsf{f}}(N)$ is isomorphic to $\mathtt{unit} + \mu_{\mathsf{f}}(N)$.

The restriction to positive type operators is essential if we are to preserve termination. The language $\mathcal{L}\{\mu_{\mathsf{i}}\mu_{\mathsf{f}}{\to}\}$ defined in Chapter 18 is a natural generalization of $\mathcal{L}\{\mathtt{nat}\to\}$ in that all expressions terminate, and functions remain total. In this chapter we relax the restriction to positive operators, and permit formation of fixed points for arbitrary type operators. As we shall see below, this introduces non-termination and, indeed, permits definition of general recursion for expressions of arbitrary type. It is therefore best understood as a variant of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$, called $\mathcal{L}\{\mu{\rightharpoonup}\}$, in which the built-in type $\mathtt{nat}$ is replaced by the *recursive type* $\mu(t.\tau)$.

## 19.1 Solving Type Equations

A *recursive type* has the form $\mu(t.\tau)$, where $t.\tau$ is any type operator, without restriction. It denotes the fixed point (up to isomorphism) of the given type operator, and hence provides a solution to the isomorphism equation $X \cong [X/t]\tau$. The isomorphism is witnessed by the terms $\mathtt{fold}(e)$ and

`unfold(`$e$`)` that mediate between the recursive type, $\mu$`(`$t$`.`$\tau$`)`, and its unrolling, $[\mu$`(`$t$`.`$\tau$`)`$/t]\tau$ In this sense the parameter, $t$, of the type operator is *self-referential* in that it may be considered to stand for the recursive type itself.

Recursive types are formalized by the following abstract syntax:

$$\begin{array}{lrcl} \text{Type} & \tau & ::= & t \mid \texttt{rec(}t\texttt{.}\tau\texttt{)} \\ \text{Expr} & e & ::= & \texttt{fold[}t\texttt{.}\tau\texttt{](}e\texttt{)} \mid \texttt{unfold(}e\texttt{)} \end{array}$$

The meta-variable $t$ ranges over a class of *type names*, which serve as names for types. The *unrolling* of `rec(`$t$`.`$\tau$`)` is the type $[$`rec(`$t$`.`$\tau$`)`$/t]\tau$ obtained by substituting the recursive type for $t$ in $\tau$.

The introduction form, `fold[`$t$`.`$\tau$`](`$e$`)`, introduces a value of recursive type in terms of an element of its unrolling, and the elimination form, `unfold(`$e$`)`, evaluates to a value of the unrolling from an element of the recursive type. In implementation terms the operation `fold[`$t$`.`$\tau$`](`$e$`)` may be thought of as an abstract "pointer" to a value of the unrolled type, and the operation `unfold(`$e$`)` "chases" the pointer to obtain that value from a value of the corresponding rolled type.

The static semantics of this extension of $\mathcal{L}\{\rightarrow\}$ consists of two forms of judgement, $\tau$ type, and $e : \tau$. The type formation judgement is inductively defined by a set of rules for deriving general judgements of the form

$$\Delta \mid \tau \text{ type},$$

where $\Delta$ is a finite set of assumptions of the form $t_i$ type for some type variable $t_i$.

$$\frac{}{\Delta, t \text{ type} \mid t \text{ type}} \tag{19.1a}$$

$$\frac{\Delta \mid \tau_1 \text{ type} \quad \Delta \mid \tau_2 \text{ type}}{\Delta \mid \texttt{arr(}\tau_1\texttt{,}\tau_2\texttt{)} \text{ type}} \tag{19.1b}$$

$$\frac{\Delta, t \text{ type} \mid \tau \text{ type}}{\Delta \mid \texttt{rec(}t\texttt{.}\tau\texttt{)} \text{ type}} \tag{19.1c}$$

Note that, in contrast to Chapter 18, there is no positivity restriction on the formation of a recursive type.

Typing judgements have the form

$$\Delta \mid \Gamma \vdash e : \tau$$

where $\Delta$ is as above, and $\Gamma$ is, as usual, a finite set of typing assumptions of the form $x_i : \tau_i$ such that $\Delta \vdash \tau_i$ type for each $1 \leq i \leq n$. The typing rules for $\mathcal{L}\{\mu\rightharpoonup\}$ are as follows:

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \mid \Gamma \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Delta \mid \Gamma \vdash \mathtt{fold}[t.\tau](e) : \mathtt{rec}(t.\tau)} \tag{19.2a}$$

$$\frac{\Delta \mid \Gamma \vdash e : \mathtt{rec}(t.\tau)}{\Delta \mid \Gamma \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{19.2b}$$

These rules express an inverse relationship stating that a recursive type is *isomorphic* to its unrolling, with the operations `fold` and `unfold` being the witnesses to the isomorphism.

Operationally, this is expressed by the following dynamic semantics rules:

$$\frac{\{e \text{ val}\}}{\mathtt{fold}[t.\tau](e) \text{ val}} \tag{19.3a}$$

$$\left\{ \frac{e \mapsto e'}{\mathtt{fold}[t.\tau](e) \mapsto \mathtt{fold}[t.\tau](e')} \right\} \tag{19.3b}$$

$$\frac{e \mapsto e'}{\mathtt{unfold}(e) \mapsto \mathtt{unfold}(e')} \tag{19.3c}$$

$$\frac{\{e \text{ val}\}}{\mathtt{unfold}(\mathtt{fold}[t.\tau](e)) \mapsto e} \tag{19.3d}$$

As usual, the bracketed rules and premises are to be omitted for a lazy semantics, and included for an eager semantics.

It is straightforward to prove the safety of this extension to $\mathcal{L}\{\rightharpoonup\}$:

**Theorem 19.1** (Safety). *1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

*2. If $e : \tau$, then either $e$ val, or there exists $e'$ such that $e \mapsto e'$.*

## 19.2 General Recursion, Revisited

In Chapter 15 we introduced the concept of general recursion as a means of defining self-referential expressions. In Chapter 13 we showed how to use general recursion to implement recursive functions. Here we show that general recursion is definable in $\mathcal{L}\{\mu\rightharpoonup\}$ using recursive types.

The trick is to augment a self-referential expression with an implicit argument that will always be bound to the expression itself, thereby effecting the recursion. The way we will do this is to introduce a type of self-referential computations yielding a value of type $\tau$, which we will write abstractly as $\texttt{self}(\tau)$ and concretely as $\tau\ \texttt{self}$. To effect self-reference we will arrange that $\tau\ \texttt{self}$ is isomorphic to $\tau\ \texttt{self} \to \tau$, the argument representing the computation itself. This may, of course, be achieved by defining $\tau\ \texttt{self}$ to be the recursive type $\mu(t.t \to \tau)$.

We may then define $\texttt{fix}\,x{:}\tau\,\texttt{is}\,e$ of type $\tau$ to be $\texttt{unfold}(e')(e')$, where

$$e' = \texttt{fold}(\lambda(y{:}\tau\ \texttt{self}.\,[\texttt{unfold}(y)(y)/x]e)).$$

Observe that

$$\begin{aligned}\texttt{fix}\,x{:}\tau\,\texttt{is}\,e &= \texttt{unfold}(e')(e')\\ &\mapsto^* [\texttt{unfold}(e')(e')/x]e\\ &= [\texttt{fix}\,x{:}\tau\,\texttt{is}\,e/x]e,\end{aligned}$$

which is sufficient to simulate self-reference. It is easy to check that the derived static semantics is as expected for general recursion.

This example shows that adding recursive types has a *global* effect on the type system — the meaning of every type is changed when recursive types are admitted. For in the derivation just give, the type $\tau$ is arbitrary, so that *every* type contains self-referential expressions. In particular, in the presence of recursive types, there is a non-terminating expression of every type, even if there were no such expressions in their absence. Thus recursive types are a *non-conservative extension* in that they change the meaning of types that were present before the extension, as well as adding new types that were not present beforehand.

## 19.3 Exercises

# Part VII

# Dynamic Typing

# Chapter 20

# Untyped Languages

It is customary to distinguish between *typed* and *untyped* languages, as though they were incompatible alternatives. But we shall argue that well-defined, or *safe*, type-free languages are, in fact, just a particular mode of use of types. As we shall see, so-called *untyped* languages may be seen as *uni-typed* languages.

## 20.1 Untyped $\lambda$-Calculus

The premier example of an untyped language is the *(untyped) $\lambda$-calculus*, a very elegant language devised by Alonzo Church in the 1930's. Its chief characteristic is that the language consists of nothing but functions! Functions take functions as arguments and yield functions as results, and all data structures must be represented as functions. Surprisingly, this tiny language is sufficiently powerful to express any computable function!

The abstract syntax of the untyped $\lambda$-calculus is given by the following grammar:

$$\lambda\text{-terms} \qquad u \quad ::= \quad x \mid \lambda(x.u) \mid \mathtt{ap}(u_1, u_2)$$

In concrete syntax these two forms are written $\lambda x.\, u$ and $u_1(u_2)$. The former is called a *$\lambda$-abstraction*, and the latter *application*. The entire language consists of these two constructs, plus variables that range over untyped $\lambda$-terms.

The basic form of execution in the untyped $\lambda$-calculus is defined by the following transition rules:

$$\frac{}{\mathtt{ap}(\lambda(x.u_1), u_2) \mapsto [u_2/x]u_1} \tag{20.1a}$$

$$\frac{u_1 \mapsto u_2}{\mathtt{ap}(u_1, u_2) \mapsto \mathtt{ap}(u_1', u_2)} \tag{20.1b}$$

In the $\lambda$-calculus literature this judgement is called *head reduction*. The first rule is called $\beta$*-reduction*; it defines the meaning of function application in terms of substitution. It is also possible to define a call-by-value variant of head reduction by insisting that the $\beta$-reduction step apply only when the argument is an explicit $\lambda$, and adding the following argument-evaluation rule:

$$\frac{u_2 \mapsto u_2'}{\mathtt{ap}(\lambda(x.u_1), u_2) \mapsto \mathtt{ap}(\lambda(x.u_1), u_2')} \tag{20.2}$$

## 20.2 Expressiveness

Interest in the untyped $\lambda$-calculus stems from its surprising expressive power: it is a Turing-complete language in the sense that it has the same capability to expression computations on the natural numbers as does any other known programming language. The Church-Turing Thesis states that any conceivable notion of computable function on the natural numbers is equivalent to the $\lambda$-calculus. This is certainly true for all *known* means of defining computable functions on the natural numbers. The force of the Church-Turing Thesis is that it postulates that all future notions of computation will be equivalent in expressive power (measured by definability of functions on the natural numbers) to the $\lambda$-calculus. The Church-Turing Thesis is therefore a *scientific law* in the same sense as, say, Newton's Law of Universal Gravitation makes a prediction about all future measurements of the acceleration due to the gravitational field of a massive object.

We will sketch a proof that the untyped $\lambda$-calculus is as powerful as the language PCF described in Chapter 15. The main idea is to show that the PCF primitives for manipulating the natural numbers are definable in the untyped $\lambda$-calculus. This means, in particular, that we must show that the natural numbers are definable as $\lambda$-terms in such a way that case analysis, which discriminates between zero and non-zero numbers, is definable. The principal difficulty is with computing the predecessor of a number, which requires a bit of cleverness. Finally, we show how to represent general recursion, completing the proof.

The first task is to represent the natural numbers as certain $\lambda$-terms,

called the *Church numerals*.

$$\overline{0} = \lambda b.\, \lambda s.\, b \tag{20.3a}$$

$$\overline{n+1} = \lambda b.\, \lambda s.\, s((\overline{n}(b)(s))) \tag{20.3b}$$

It follows that

$$\overline{n}(u_1)(u_2) \mapsto^* u_2((\ldots(u_2(u_1))))(,)$$

the $n$-fold application of $u_2$ to $u_1$. That is, $\overline{n}$ iterates its second argument (the induction step) $n$ times, starting with its first argument (the basis).

Using this definition it is not difficult to define the basic functions of arithmetic. For example, successor, addition, and multiplication are defined by the following untyped $\lambda$-terms:

$$\texttt{succ} = \lambda x.\, \lambda b.\, \lambda s.\, s((x(b)(s))) \tag{20.4}$$

$$\texttt{plus} = \lambda x.\, \lambda y.\, y(x)(\texttt{succ}) \tag{20.5}$$

$$\texttt{times} = \lambda x.\, \lambda y.\, y(\overline{0})((\texttt{plus}(x))) \tag{20.6}$$

It is easy to check that $\texttt{succ}(\overline{n}) \mapsto^* \overline{n+1}$, and that similar correctness conditions hold for the representations of addition and multiplication.

We may readily define $\texttt{ifz}(u, u_0, u_1)$ to be the application $u(u_0)((\lambda x.\, u_1))$, where $x$ is chosen arbitrarily such that $x \# u_1$. We can use this to define $\texttt{ifz}(u, u_0, x\,.\,u_1)$, provided that we can compute the predecessor of a natural number. Doing so requires a bit of ingenuity. We wish to find a term `pred` such that

$$\texttt{pred}(\overline{0}) \mapsto^* \overline{0} \tag{20.7}$$

$$\texttt{pred}(\overline{n+1}) \mapsto^* \overline{n}. \tag{20.8}$$

To compute the predecessor using Church numerals, we must show how to compute the result for $\overline{n+1}$ as a function of its value for $\overline{n}$. At first glance this seems straightforward—just take the successor—until we consider the base case, in which we define the predecessor of $\overline{0}$ to be $\overline{0}$. This invalidates the obvious strategy of taking successors at inductive steps, and necessitates some other approach.

What to do? A useful intuition is to think of the computation in terms of a pair of "shift registers" satisfying the invariant that on the $n$th iteration the registers contain the predecessor of $n$ and $n$ itself, respectively. Given the result for $n$, namely the pair $(n-1, n)$, we pass to the result for $n+1$ by

"shifting left" and "incrementing" to obtain $(n, n+1)$. For the base case, we initialize the registers with $(0,0)$, reflecting the *ad hoc* condition that the predecessor of zero be zero. Now to compute the predecessor of $n$ we compute the pair $(n-1, n)$ by this method, and return the first component.

To make this precise, we must first define a Church-style representation of ordered pairs.

$$\langle u_1, u_2 \rangle = \lambda f.\, f\,(u_1)\,(u_2) \tag{20.9}$$
$$\mathtt{fst}(u) = u\,((\lambda x.\, \lambda y.\, x)) \tag{20.10}$$
$$\mathtt{snd}(u) = u\,((\lambda x.\, \lambda y.\, y)) \tag{20.11}$$

It is easy to check that under this encoding $\mathtt{fst}(\langle u_1, u_2 \rangle) \mapsto^* u_1$, and similarly for the second projection. We may now define the required term $u$ representing the predecessor:

$$u_p' = \lambda x.\, x\,(\langle \overline{0}, \overline{0} \rangle)\,(\lambda y.\, \langle \mathtt{snd}(y), \mathtt{s}(\mathtt{snd}(y)) \rangle) \tag{20.12}$$
$$u_p = \lambda x.\, \mathtt{fst}(u(x)) \tag{20.13}$$

It is then easy to check that this gives us the required behavior.

This gives us all the apparatus of PCF, apart from general recursion. But this is also definable using a *fixed point combinator*. There are many choices of fixed point combinator, of which it is most convenient to use the so-called *Turing fixed point combinator*,

$$\Theta = \theta\,(\theta) \tag{20.14}$$
$$\theta = \lambda f.\, \lambda x.\, x\,((f\,(f)\,(x))) \tag{20.15}$$

Observe that

$$\Theta(u) = \theta\,(\theta)\,(u) \tag{20.16}$$
$$\mapsto^* u\,((\theta\,(\theta)\,(u))) \tag{20.17}$$
$$= u\,((\Theta(u))) \tag{20.18}$$

This provides the basis for general recursion, writing $\Theta((\lambda x.\, u))$ for self-reference to $u$ within $u$ via the variable $x$.

## 20.3 Untyped Means Uni-Typed

The untyped $\lambda$-calculus may be *faithfully embedded* in the typed language $\mathcal{L}\{\mu{\rightharpoonup}\}$, enriched with recursive types. This means that every untyped $\lambda$-term has a representation as an expression in $\mathcal{L}\{\mu{\rightharpoonup}\}$ in such a way that

execution of the representation of a $\lambda$-term corresponds to execution of the term itself. If the execution model of the $\lambda$-calculus is call-by-name, this correspondence holds for the call-by-name variant of $\mathcal{L}\{\mu\rightharpoonup\}$, and similarly for call-by-value.

It is important to understand that this form of embedding is *not* a matter of writing an interpreter for the $\lambda$-calculus in $\mathcal{L}\{\mu\rightharpoonup\}$ (which we could surely do), but rather a direct representation of untyped $\lambda$-terms as certain typed expressions of $\mathcal{L}\{\mu\rightharpoonup\}$. It is for this reason that we say that untyped languages are just a special case of typed languages with recursive types. Thus the supposed opposition between typed and untyped languages is nothing of the kind. Rather, the issue is simply that recursive types greatly increase the expressive power of typed languages, permitting styles of programming that are otherwise impossible.

The key observation is that *untyped* really means *uni-typed*. That is, it is not that the "untyped" $\lambda$-calculus has *zero* types, but rather that it has *one* type! This type is the recursive type

$$D = \mathtt{rec}(t.\mathtt{arr}(t,t)).$$

A value of type $D$ is of the form $\mathtt{fold}[D](e)$ where $e$ is a value of type $\mathtt{arr}(D,D)$ — a function whose domain and range are both $D$. Any such function can be regarded as a value of type $D$ by "rolling", and any value of type $D$ can be turned into a function by "unrolling". As usual, a recursive type may be seen as a solution to a type isomorphism equation, which in the present case is the equation

$$D \cong \mathtt{arr}(D,D).$$

This specifies that $D$ is a type that is isomorphic to the space of functions on $D$ itself, something that is impossible in conventional set theory, but is feasible in the computationally-based setting of the $\lambda$-calculus.

This isomorphism leads to the following embedding, $u^\dagger$, of $u$ into $\mathcal{L}\{\mu\rightharpoonup\}$:

$$x^\dagger = x \tag{20.19a}$$

$$\lambda(x.u)^\dagger = \mathtt{fold}[D](\mathtt{lam}[D](x.u^\dagger)) \tag{20.19b}$$

$$\mathtt{ap}(u_1,u_2)^\dagger = \mathtt{ap}(\mathtt{unfold}(u_1^\dagger),u_2^\dagger) \tag{20.19c}$$

Observe that the embedding of a $\lambda$-abstraction is a value, and that the embedding of an application exposes the function being applied by un-

rolling the recursive type. Consequently,

$$\begin{aligned}\texttt{ap}(\lambda(x.u_1), u_2)^\dagger &= \texttt{ap}(\texttt{unfold}(\texttt{fold}[D](\texttt{lam}[D](x.u_1^\dagger))), u_2^\dagger)\\ &\mapsto \texttt{ap}(\texttt{lam}[D](x.u_1^\dagger), u_2^\dagger)\\ &\mapsto [u_2^\dagger/x]u_1^\dagger\\ &= ([u_2/x]u_1)^\dagger.\end{aligned}$$

The last step, stating that the embedding commutes with substitution, is easily proved by induction on the structure of $u_1$. Thus $\beta$-reduction is faithfully implemented by evaluation of the embedded terms. It is also easy to show that if $u_1^\dagger \mapsto^* v_1^\dagger$, then $\texttt{ap}(u_1, u_2)^\dagger \mapsto^* \texttt{ap}(v_1, u_2)^\dagger$. Consequently, head reduction in the $\lambda$-calculus is faithfully implemented by evaluation in $\mathcal{L}\{\mu\rightharpoonup\}$.

## 20.4 Exercises

# Chapter 21

# Dynamic Typing

We saw in Chapter 20 that an untyped language may be viewed as a unityped language in which the so-called untyped terms are terms of a distinguished recursive type. In the case of the untyped $\lambda$-calculus this recursive type has a particularly simple form, expressing that every term is isomorphic to a function. Consequently, no run-time errors can occur due to the misuse of a value—the only elimination form is application, and its first argument can only be a function. Obviously this property breaks down once more than one class of value is permitted into the language. For example, if we add natural numbers as a primitive concept to the untyped $\lambda$-calculus (*i.e.*, rather than defining them via Church encodings), then it is possible to incur a run-time error arising from attempting to apply a number to an argument, or to add a function to a number.

One school of thought in language design is to turn this vice into a virtue by embracing a model of computation that has multiple classes of value of a single type. Such languages are said to be *dynamically typed*, in supposed opposition to the *statically typed* languages we have studied thus far. In this chapter we show that the supposed opposition between static and dynamic languages is fallacious: dynamic typing is but a mode of use of static typing, and, moreover, it is profitably seen as such. Dynamic typing can hardly be in opposition to that of which it is a special case!

## 21.1 Dynamically Typed PCF

To illustrate dynamic typing we formulate a dynamically typed version of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, called $\mathcal{L}\{\mathit{dyn}\}$. The abstract syntax of $\mathcal{L}\{\mathit{dyn}\}$ is given by the

following grammar:

$$\begin{array}{lrcl} \text{Expr} & d & ::= & x \mid \mathtt{num}(\overline{n}) \mid \mathtt{succ}(d) \mid \mathtt{ifz}(d, d_0, x.d_1) \mid \\ & & & \mathtt{fun}(\lambda(x.d)) \mid \mathtt{ap}(d_1, d_2) \mid \mathtt{fix}(x.d) \mid \mathtt{error} \end{array}$$

The syntax is similar to that of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, the chief difference being that each value is tagged with its *class*, either `num` or `fun`. The notation $\overline{n}$ stands for the $n$th numeral, $n$ applications of successor to zero. Numerals are not themselves forms of expression, but rather must be tagged with the class `num` to be considered as expressions. Similarly, "bare" $\lambda$-abstractions (with no type attached to the bound variable) of the form $\mathtt{fun}(\lambda(x.d))$ are not expressions, rather they must be tagged with the class `fun` to be considered as such. The expression `error` models checked errors in the manner discussed in Section 10.3 on page 78.

The concrete syntax conventions for $\mathcal{L}\{dyn\}$ are summarized by the following chart. Apart from formatting, the concrete syntax avoids mentioning the class tags attached to values of the language.

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\mathtt{num}(\overline{n})$ | $\overline{n}$ |
| $\mathtt{ifz}(d, d_0, x.d_1)$ | $\mathtt{ifz}\, d\, \{\mathtt{z} \Rightarrow d_0 \mid \mathtt{s}(x) \Rightarrow d_1\}$ |
| $\mathtt{fun}(\lambda(x.d))$ | $\lambda x.d$ |
| $\mathtt{ap}(d_1, d_2)$ | $d_1(d_2)$ |
| $\mathtt{fix}(x.d)$ | $\mathtt{fix}\, x\, \mathtt{is}\, d$ |

The omission of class tags on values means that they must be inserted by the parser on passage from concrete to abstract syntax. Unfortunately this fosters the misunderstanding that the expressions of $\mathcal{L}\{dyn\}$ are exactly the same as those of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, but this is not the case. The class tags are essential for the dynamic semantics of $\mathcal{L}\{dyn\}$, but are entirely unnecessary for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

There is no static semantics for $\mathcal{L}\{dyn\}$; every expression is eligible for evaluation. However, the dynamic semantics must check for errors that would never arise in a safe statically typed language. For example, function application must ensure that its first argument is a function, signaling an error in the case that it is not, and similarly the case analysis construct must ensure that its first argument is a number, signaling an error if not. For such checks to be possible at run-time, each value is explicitly tagged with its class.

The a value judgement, $d$ val, states that $d$ is a fully evaluated (closed) expression:

$$\frac{}{\mathtt{num}(\overline{n})\ \mathsf{val}} \tag{21.1a}$$

$$\frac{}{\texttt{fun}(\lambda(x.d))\ \mathsf{val}} \tag{21.1b}$$

The judgement $d$ err states that $d$ represents a run-time error. It is defined by the single rule

$$\frac{}{\texttt{error}\ \mathsf{err}} \tag{21.2}$$

The dynamic semantics makes use of judgements that check the class of a value, and recover the underlying $\lambda$-abstraction in the case of a function.

$$\frac{}{\texttt{num}(\overline{n})\ \mathsf{is\_num}\ \overline{n}} \tag{21.3a}$$

$$\frac{}{\texttt{fun}(\lambda(x.d))\ \mathsf{is\_fun}\ \lambda(x.d)} \tag{21.3b}$$

The second argument of each of these judgements has a special status—it is not an expression of $\mathcal{L}\{dyn\}$, but rather just a special piece of syntax used internally to the transition rules given below.

We also will need the "negations" of the class-checking judgements in order to detect run-time type errors.

$$\frac{}{\texttt{num(\_)}\ \mathsf{isnt\_fun}} \tag{21.4a}$$

$$\frac{}{\texttt{fun(\_)}\ \mathsf{isnt\_num}} \tag{21.4b}$$

We may now define the dynamic semantics of $\mathcal{L}\{dyn\}$ by the following rules:

$$\frac{d \mapsto d'}{\texttt{succ}(d) \mapsto \texttt{succ}(d')} \tag{21.5a}$$

$$\frac{d\ \mathsf{is\_num}\ n}{\texttt{succ}(d) \mapsto \texttt{num}(\texttt{s}(\overline{n}))} \tag{21.5b}$$

$$\frac{d\ \mathsf{isnt\_num}}{\texttt{succ}(d) \mapsto \texttt{error}} \tag{21.5c}$$

$$\frac{d \mapsto d'}{\texttt{ifz}(d, d_0, x.d_1) \mapsto \texttt{ifz}(d', d_0, x.d_1)} \tag{21.5d}$$

$$\frac{d\ \mathsf{is\_num}\ \texttt{z}}{\texttt{ifz}(d, d_0, x.d_1) \mapsto d_0} \tag{21.5e}$$

$$\frac{d \text{ is\_num } \mathtt{s}(\overline{n})}{\mathtt{ifz}(d, d_0, x.d_1) \mapsto [\mathtt{num}(\overline{n})/x]d_1} \tag{21.5f}$$

$$\frac{d \text{ isnt\_num}}{\mathtt{ifz}(d, d_0, x.d_1) \mapsto \mathtt{error}} \tag{21.5g}$$

$$\frac{d_1 \mapsto d_1'}{\mathtt{ap}(d_1, d_2) \mapsto \mathtt{ap}(d_1', d_2)} \tag{21.5h}$$

$$\frac{d_1 \text{ val} \quad d_2 \mapsto d_2'}{\mathtt{ap}(d_1, d_2) \mapsto \mathtt{ap}(d_1, d_2')} \tag{21.5i}$$

$$\frac{d_1 \text{ is\_fun } \lambda(x.d) \quad d_2 \text{ val}}{\mathtt{ap}(d_1, d_2) \mapsto [d_2/x]d} \tag{21.5j}$$

$$\frac{d_1 \text{ isnt\_fun}}{\mathtt{ap}(d_1, d_2) \mapsto \mathtt{error}} \tag{21.5k}$$

$$\frac{}{\mathtt{fix}(x.d) \mapsto [\mathtt{fix}(x.d)/x]d} \tag{21.5l}$$

Note that in Rule (21.5f) the tagged numeral $\mathtt{num}(\overline{n})$ is bound to $x$ to maintain the invariant that variables are bound to forms of expression.

Although it lacks a static semantics, the language $\mathcal{L}\{dyn\}$ nevertheless is safe in the same sense as languages that do have a static semantics.

**Theorem 21.1.** *For every $d$ either $d$ val, or $d$ err, or there exists $d'$ such that $d \mapsto d'$.*

The safety of $\mathcal{L}\{dyn\}$ is often promoted as an *advantage* of dynamic over static typing: every parseable expression is capable of execution. But this can also be seen as a *disadvantage*: errors that would be ruled out at compile time by a static type system are not signalled until execution time.

## 21.2 Critique of Dynamic Typing

The dynamic semantics of $\mathcal{L}\{dyn\}$ exhibits considerable run-time overhead compared to that of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$. Suppose that we define addition by the $\mathcal{L}\{dyn\}$ expression

$$\lambda x.\, (\mathtt{fix}\, p\, \mathtt{is}\, \lambda y.\, \mathtt{ifz}\, y\, \{\mathtt{z} \Rightarrow x \mid \mathtt{s}(y') \Rightarrow \mathtt{succ}(p(y'))\}).$$

By carefully examining the dynamics semantics, we may observe some of the hidden costs of dynamic typing.

First, observe that the body of the fixed point expression is a $\lambda$-abstraction, which is tagged by the parser with class `fun`. The semantics of the fixed point construct binds $p$ to this (tagged) $\lambda$-abstraction, so the dynamic tag check incurred by the recursive call is guaranteed to succeed. The check is redundant, but there is no way within the language to avoid it.

Second, the result of applying the inner $\lambda$-abstraction is either $x$, the parameter of the outer $\lambda$-abstraction, or the result of a recursive call. The semantics of the successor operation ensures that the result of the recursive call is tagged with class `num`, so the only way the tag check performed by the successor operation could fail is if $x$ is not bound to a number at the initial call. In other words, it is a loop invariant that the result is of class `num`, so there is no need for this check within the loop, only at the initial call.

Third, the argument, $y$, to the inner $\lambda$-abstraction arises either at the initial call, or as a result of a recursive call. But if the initial call binds $y$ to a number, then so must the recursive call, because the dynamic semantics ensures that the predecessor of a number is also a number. Once again we have an unnecessary dynamic check in the inner loop of the function, but there is no way to avoid it.

Tag checking and creation is not free—storage is required for the tag itself, and the marking of a value with a tag takes time as well as space. While the overhead is not asymptotically significant (it slows down the program only by a constant factor), it is nevertheless non-negligible, and should be eliminated whenever possible. But within $\mathcal{L}\{dyn\}$ itself there is no way to avoid the overhead, because there are no "unchecked" operations in the language—for these to be safe requires a static type system!

## 21.3 Hybrid Typing

Let us consider the language $\mathcal{L}\{\texttt{nat}\,\texttt{dyn}\rightharpoonup\}$, whose syntax extends that of the language $\mathcal{L}\{\texttt{nat}\rightharpoonup\}$ defined in Chapter 15 with the following additional constructs:

| | | | |
|---|---|---|---|
| Type | $\tau$ | ::= | `dyn` |
| Expr | $e$ | ::= | `error` $\mid$ `tag[`$t$`](`$e$`)` $\mid$ `cast[`$t$`](`$e$`)` |
| Tag | $t$ | ::= | `num` $\mid$ `fun` |

The type `dyn` represents the type of tagged values. Here we have only two classes of data object, numbers and functions, and hence only two forms of

tag. (In a richer language we would have more classes of data, and correspondingly more forms of tags.)

The concrete syntax of these constructs is give by the following correspondences:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\texttt{tag}[t]\,(e)$ | $t\,\texttt{!}\,e$ |
| $\texttt{cast}[t]\,(e)$ | $e\,\texttt{?}\,t$ |

Observe that the `cast` operation takes as argument a tag, not a type. That is, casting is concerned with an object's *class*, which is indicated by a tag, not with its *type*, which is always `dyn`.

The static semantics for $\mathcal{L}\{\texttt{nat dyn} \rightharpoonup\}$ is the extension of that of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with the following rules governing the type `dyn`.

$$\frac{}{\Gamma \vdash \texttt{error} : \tau} \tag{21.6a}$$

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{tag[num]}\,(e) : \texttt{dyn}} \tag{21.6b}$$

$$\frac{\Gamma \vdash e : \texttt{parr(dyn,dyn)}}{\Gamma \vdash \texttt{tag[fun]}\,(e) : \texttt{dyn}} \tag{21.6c}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast[num]}\,(e) : \texttt{nat}} \tag{21.6d}$$

$$\frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast[fun]}\,(e) : \texttt{parr(dyn,dyn)}} \tag{21.6e}$$

The static semantics ensures that tags are only applied to objects of the appropriate type, `num` to natural numbers, and `fun` to functions defined over tagged values.

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

$$\frac{e\ \mathsf{val}}{\texttt{tag}[t]\,(e)\ \mathsf{val}} \tag{21.7a}$$

$$\frac{e \mapsto e'}{\texttt{tag}[t]\,(e) \mapsto \texttt{tag}[t]\,(e')} \tag{21.7b}$$

$$\frac{e \mapsto e'}{\texttt{cast}[t]\,(e) \mapsto \texttt{cast}[t]\,(e')} \tag{21.7c}$$

$$\frac{\texttt{tag}[t]\,(e)\ \mathsf{val}}{\texttt{cast}[t]\,(\texttt{tag}[t]\,(e)) \mapsto e} \tag{21.7d}$$

$$\frac{\texttt{tag}[t']\,(e)\ \mathsf{val} \quad t \# t'}{\texttt{cast}[t]\,(\texttt{tag}[t']\,(e)) \mapsto \texttt{error}} \tag{21.7e}$$

Casting compares the tag of the object to the required tag, returning the underlying object if these coincide, and signalling an error otherwise.

**Lemma 21.2** (Canonical Forms)**.** *If $e : \mathit{dyn}$ and $e$ $\mathsf{val}$, then $e = \mathit{tag}[t]\,(e')$ for some tag $t$ and some $e'$ $\mathsf{val}$. If $t = \mathit{num}$, then $e' : \mathit{nat}$, and if $t = \mathit{fun}$, then $e' : \mathit{parr}(\mathit{dyn}, \mathit{dyn})$.*

**Theorem 21.3** (Safety)**.** *The language $\mathcal{L}\{\mathit{nat}\ \mathit{dyn} \rightharpoonup\}$ is safe:*

1. *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*
2. *If $e : \tau$, then either $e$ $\mathsf{val}$, or $e$ $\mathsf{err}$, or $e \mapsto e'$ for some $e'$.*

## 21.4 Optimization of Dynamic Typing

The type dyn—whether primitive or derived—supports the smooth integration of dynamic with static typing. This means that we can take full advantage of the expressive power of static types whenever possible, while permitting the flexibility of dynamic typing whenever desirable.

One application of the hybrid framework is that it permits the optimization of dynamically typed programs by taking advantage of statically evident typing constraints. Let us examine how this plays out in the case of the addition function, which is rendered in $\mathcal{L}\{\texttt{nat}\ \texttt{dyn} \rightharpoonup\}$ as follows:

$$\texttt{fun}\,!\,\lambda(x{:}\texttt{dyn}.\,\texttt{fix}\,p{:}\texttt{dyn}\,\texttt{is}\,\texttt{fun}\,!\,\lambda(y{:}\texttt{dyn}.\,e_{x,p,y})),$$

where $e_{x,p,y}$ of type dyn is the expression

$$\texttt{ifz}\,(y\,?\,\texttt{num})\,\{\texttt{z} \Rightarrow x \mid \texttt{s}(y') \Rightarrow \texttt{num}\,!\,(\texttt{s}(((p\,?\,\texttt{fun})((\texttt{num}\,!\,y'))\,?\,\texttt{num}))\}.$$

This expression, which has type dyn, is essentially an explicit form of the dynamically typed addition function given in Section 21.2 on page 166. This formulation in $\mathcal{L}\{\texttt{nat}\ \texttt{dyn} \rightharpoonup\}$ makes explicit the checking of class tags that is implicit in $\mathcal{L}\{\mathit{dyn}\}$.

One important consequence of embedding dynamic typing in a statically typed language is that it permits us to express optimizations that are not expressible in a purely dynamic language. We will illustrate these by a sequence of transformations on the addition function defined above.

The first optimization is to note that the body of the `fix` expression is an explicitly tagged function. This means that when the recursion is unwound, the variable $p$ is bound to this value of type `dyn`. Consequently, the check that $p$ is tagged with class `fun` is redundant, and can be eliminated. This is achieved by re-writing the function as follows:

$$\texttt{fun}\,!\,\lambda(x\!:\!\texttt{dyn}.\,\texttt{fun}\,!\,\texttt{fix}\,p\!:\!\texttt{dyn}\rightharpoonup\texttt{dyn}\,\texttt{is}\,\lambda(y\!:\!\texttt{dyn}.\,e'_{x,p,y})),$$

where $e'_{x,p,y}$ is the expression

$$\texttt{ifz}\,(y\,?\,\texttt{num})\,\{\texttt{z}\Rightarrow x\mid\texttt{s}(y')\Rightarrow\texttt{num}\,!\,(\texttt{s}((p((\texttt{num}\,!\,y'\!))\,?\,\texttt{num}))\}.$$

We have "hoisted" the function tag out of the loop, and suppressed the cast inside the loop. Correspondingly, the type of $p$ has changed from `dyn` to $\texttt{dyn}\rightharpoonup\texttt{dyn}$, reflecting that the body is now a "bare function", rather than a tagged value of type `dyn`.

Next, observe that the parameter $y$ of type `dyn` is cast to a number on each iteration of the loop before it is tested for zero. Since this function is recursive, the bindings of $y$ arise in one of two ways, at the initial call to the addition function, and on each recursive call. But the recursive call is made on the predecessor of $y$, which is a true natural number that is tagged with `num` at the call site, only to be removed by the tag check at the conditional on the next iteration. This suggests that we hoist the check on $y$ outside of the loop, and avoid tagging the argument to the recursive call. Doing so changes the type of the function, however, from $\texttt{dyn}\rightharpoonup\texttt{dyn}$ to $\texttt{nat}\rightharpoonup\texttt{dyn}$. Consequently, further changes are required to ensure that the entire function remains well-typed.

Before doing so, let us make another observation. The result of the recursive call is checked to ensure that it has class `num`, and, if it does, the underlying value is incremented, and tagged with class `num`. If the result of the recursive call came from an earlier use of this branch of the conditional, then obviously the tag check is redundant, because we know that it must have class `num`. But what if the result came from the other branch of the conditional? In that case the function returns $x$, which need not be of class `num`! However, one might reasonably insist that this is only a theoretical possibility—after all, we are defining the addition function, and its arguments might reasonably be restricted to numbers. This can be achieved by replacing $x$ by the cast $x\,?\,\texttt{num}$, which checks that $x$ is of class `num`, and returns the underlying number.

Combining these optimizations we obtain the inner loop $e''_x$ defined as follows:

$$\texttt{fix}\,p\!:\!\texttt{nat}\rightharpoonup\texttt{nat}\,\texttt{is}\,\lambda(y\!:\!\texttt{nat}.\,\texttt{ifz}\,y\,\{\texttt{z}\Rightarrow x\,?\,\texttt{num}\mid\texttt{s}(y')\Rightarrow\texttt{s}(p(y'))\}).$$

This function has type $\texttt{nat} \rightharpoonup \texttt{nat}$, and runs at full speed when applied to a natural number—all checks have been hoisted out of the inner loop.

Returning to the definition of addition on arguments of type `dyn`, we require a function of type $\texttt{dyn} \rightharpoonup \texttt{dyn}$, not one of type $\texttt{nat} \rightharpoonup \texttt{nat}$. This can be achieved as follows:

$$\texttt{fun}\,!\,\lambda(x\!:\!\texttt{dyn}.\,\texttt{fun}\,!\,\lambda(y\!:\!\texttt{dyn}.\,\texttt{num}\,!\,(e''_x(y\,?\,\texttt{num})))).$$

The innermost $\lambda$-abstraction converts the function $e''_x$ from type $\texttt{nat} \rightharpoonup \texttt{nat}$ to type $\texttt{dyn} \rightharpoonup \texttt{dyn}$ by composing it with a tag check that ensures that $y$ is a natural number at the initial call site, and applies a tag to the result to restore it to type `dyn`.

## 21.5 Static "Versus" Dynamic Typing

There have been many attempts to explain the distinction between dynamic and static typing, most of which are misleading or even incorrect. For example, it is often said that static type systems associate types with variables, but dynamic type systems associate types with values. This explanation appears to stem from the absence of type information on $\lambda$-abstractions in $\mathcal{L}\{dyn\}$, and the tagging of values with their class. Part of the confusion here is between classes and types—the *class* of a value (`num` or `fun`) is not the same as its *type*. Another confusion is that static type disciplines assign types to values just as surely as they do any other form of expression, including variables.

Another common explanation of the difference is to say that dynamic languages check types at run-time, whereas static languages check types at compile-time. The latter statement is true, but tautologous, but the former is simply incorrect. Dynamic languages perform run-time *class checking*, not run-time *type checking*. In particular, application merely checks that its first argument is tagged with `fun`; it does not type check the body of the function. Here again the purported contrast is largely a matter of terminology, and not a fundamental semantic difference between the languages.

Another explanation asserts that data structures in a dynamic language are *heterogeneous*, whereas in static languages they are *homogeneous*. To understand this explanation requires first that we consider how to add, say, lists to $\mathcal{L}\{dyn\}$. Briefly, one would add two constructs, `nil` and $\texttt{cons}(d_1, d_2)$, representing the empty list and a non-empty list with head $d_1$ and tail $d_2$, respectively. Since each data value in $\mathcal{L}\{dyn\}$ is tagged with its class, the

successive elements of the list might be of different classes. For example, one might form the list

$$\texttt{cons(s(z),cons(}\lambda x.\, x\texttt{,nil)),}$$

whose first element is a number, and whose second element is a function. Such a list is said to be heterogeneous. By contrast static languages are said to permit only homogeneous lists, precluding formation of such a data structure. But this again is false, because in a hybrid language dynamic values are simply static values of type dyn—so the above list is, in fact, homogeneous after all! Again, the issue is largely a matter of terminology: lists are *type homogeneous*, but may be *class heterogeneous*, even in a static language.

## 21.6 Hybrid Typing Via Recursive Types

The type dyn codifies the use of dynamic typing within a static language. Its introduction form attaches a tag to an object of the appropriate type, and its elimination form is a (possibly undefined) casting operation. Rather than treating dyn as primitive, we may derive it as a particular use of recursive types, according to the following definitions:

$$\texttt{dyn} := \mu(t.\,[\texttt{num}:\texttt{nat},\texttt{fun}:t \rightharpoonup t]) \tag{21.8}$$

$$\texttt{dyn}' := [\texttt{num}:\texttt{nat},\texttt{fun}:\texttt{dyn} \rightharpoonup \texttt{dyn}] \tag{21.9}$$

$$\texttt{num}\,!\,e := \texttt{fold}([\texttt{num}=e]_{\texttt{dyn}'}) \tag{21.10}$$

$$\texttt{fun}\,!\,e := \texttt{fold}([\texttt{fun}=e]_{\texttt{dyn}'}) \tag{21.11}$$

$$e\,?\,\texttt{num} := \texttt{case}\,\texttt{unfold}(e)\,\{[\texttt{num}{=}x] \Rightarrow x \mid [\texttt{fun}{=}x] \Rightarrow \texttt{error}\} \tag{21.12}$$

$$e\,?\,\texttt{fun} := \texttt{case}\,\texttt{unfold}(e)\,\{[\texttt{num}{=}x] \Rightarrow \texttt{error} \mid [\texttt{fun}{=}x] \Rightarrow x\} \tag{21.13}$$

One may readily check that the static and dynamic semantics for the type dyn are derivable according to these definitions.

Tagging a value with its class is merely injection into a sum type, and casting a value to a class is definable in terms of case analysis for labelled sum types. Thus dynamically typed languages are really just a mode of use of statically typed languages, provided that the type system is sufficiently rich to include recursive sum types. Consequently, dynamic typing can hardly be put into *opposition* to static typing, when it is in fact simply a special case of it!

## 21.7 Exercises

# Chapter 22

# Type Dynamic

# Part VIII

# Polymorphism

# Chapter 23

# Polymorphism

The languages $\mathcal{L}\{\mathtt{nat} \to\}$ and $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, and their various extensions, have the property that every expression has at most one type. In particular, a function has uniquely determined domain and range types. Consequently, there is a distinct identity function for each type, $id_\tau = \lambda(x:\tau.\, x)$, and a distinct composition function for each triple of types,

$$\circ_{\tau_1,\tau_2,\tau_3} = \lambda(f:\tau_2 \to \tau_3.\, \lambda(g:\tau_1 \to \tau_2.\, \lambda(x:\tau_1.\, f(g(x))))).$$

And yet every identity function and every composition function "works the same way", regardless of the choice of types! It quickly gets tedious to write the "same" program over and over, with the sole difference being the types involved. It would clearly be advantageous to capture the underlying computation once and for all, with specific instances arising by specifying the types involved.

What is needed is a way to capture the pattern of a computation in a way that is *generic*, or *parametric*, in the types involved. This is called *polymorphism*.

## 23.1 Polymorphic $\lambda$-Calculus

The *polymorphic $\lambda$-calculus*, or $\mathcal{L}\{\to\forall\}$, is a minimal functional language that illustrates the core concepts of polymorphic typing, and permits us to examine its surprising expressive power in isolation from other language features. The abstract syntax of the polymorphic $\lambda$-calculus is given as follows:

| | | | |
|---|---|---|---|
| Type | $\tau$ | ::= | $t \mid \mathtt{arr}(\tau_1,\tau_2) \mid \mathtt{all}(t.\,\tau)$ |
| Expr | $e$ | ::= | $x \mid \mathtt{lam}[\tau](x.e) \mid \mathtt{ap}(e_1,e_2) \mid \mathtt{Lam}(t.e) \mid \mathtt{App}[\tau](e)$ |

The meta-variable $t$ ranges over a class of *type names* (also called *type variables*), and $x$ ranges over a class of *expression names* (also called *expression variables*). The *type abstraction*, `Lam(`$t$`.`$e$`)`, defines a *generic*, or *polymorphic*, function with *type parameter* $t$ standing for an unpspecified type within $e$. The *type application*, or *instantiation*, `App[`$\tau$`](`$e$`)`, applies a polymorphic function to a specified type, which is then plugged in for the type parameter to obtain the result. Polymorphic functions are classified by the *universal type*, `all(`$t$`.`$\tau$`)`, that determines the type, $\tau$, of the result as a function of the argument, $t$.

In examples we use the following mathematical and concrete syntax for these constructs:

| Abstract Syntax | Concrete Syntax |
|---|---|
| `all(`$t$`.`$\tau$`)` | $\forall(t.\tau)$ |
| `Lam(`$t$`.`$e$`)` | $\Lambda(t.e)$ |
| `App[`$\tau$`](`$e$`)` | $e[\tau]$ |

The static semantics of $\mathcal{L}\{\rightarrow\forall\}$ consists of two categorical judgement forms, $\tau$ type, stating that $\tau$ is a well-formed type, and $e : \tau$, stating that $e$ is a well-formed expression of type $\tau$. The definitions of these judgements make use of judgements of the form

$$t_1 \text{ type}, \ldots, t_n \text{ type} \mid \tau \text{ type}$$

and

$$t_1 \text{ type}, \ldots, t_n \text{ type} \mid x_1 : \tau_1, \ldots, x_k : \tau_k \vdash e : \tau.$$

As usual we suppress the variable set on the turnstile for the sake of clarity. The meta-variable $\Delta$ ranges over finite sets of type variable formation hypotheses, and $\Gamma$ ranges over finite sets of expression variable typing hypotheses.

The rules for type formation are as follows:

$$\frac{}{\Delta, t \text{ type} \mid t \text{ type}} \tag{23.1a}$$

$$\frac{\Delta \mid \tau_1 \text{ type} \quad \Delta \mid \tau_2 \text{ type}}{\Delta \mid \texttt{arr}(\tau_1, \tau_2) \text{ type}} \tag{23.1b}$$

$$\frac{\Delta, t \text{ type} \mid \tau \text{ type}}{\Delta \mid \texttt{all}(t.\tau) \text{ type}} \tag{23.1c}$$

The rules for typing expressions are as follows:

$$\frac{\Delta, t \text{ type} \mid \Gamma \vdash e : \tau}{\Delta \mid \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)} \tag{23.2a}$$

$$\frac{\Delta \mid \Gamma \vdash e : \mathtt{all}(t.\tau') \quad \Delta \vdash \tau \text{ type}}{\Delta \mid \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'} \tag{23.2b}$$

As an example, the polymorphic composition function is written as follows:

$$\Lambda(t_1.\Lambda(t_2.\Lambda(t_3.\lambda(f:t_2 \to t_3.\lambda(g:t_1 \to t_2.\lambda(x:t_1.f(g(x)))))))).$$

This expression has the polymorphic type

$$\forall(t_1.\forall(t_2.\forall(t_3.(t_2 \to t_3) \to (t_1 \to t_2) \to (t_1 \to t_3)))).$$

Typing is closed under substitution of types for type variables and terms for term variables.

**Lemma 23.1** (Substitution). *1. If* $\Delta, t \text{ type} \vdash \tau' \text{ type}$ *and* $\Delta \vdash \tau \text{ type}$, *then* $\Delta \vdash [\tau/t]\tau' \text{ type}$.

*2. If* $\Delta, t \text{ type} \mid \Gamma \vdash e' : \tau'$ *and* $\Delta \vdash \tau \text{ type}$, *then* $\Delta \mid [\tau/t]\Gamma \vdash [\tau/t]e' : [\tau/t]\tau'$.

*3. If* $\Delta \mid \Gamma, x : \tau \vdash e' : \tau'$ *and* $\Delta \mid \Gamma \vdash e : \tau$, *then* $\Delta \mid \Gamma \vdash [e/x]e' : \tau'$.

Notice that the second part of the lemma requires substitution into the context, $\Gamma$, as well as into the term and its type, because the type variable $t$ may occur freely in any of these positions.

## Dynamic Semantics

The dynamic semantics of $\mathcal{L}\{\to\forall\}$ is a simple extension of $\mathcal{L}\{\to\}$. We need only add the following two rules to the structure semantics:

$$\frac{}{\mathtt{Lam}(t.e) \text{ val}} \tag{23.3a}$$

$$\frac{}{\mathtt{App}[\tau](\mathtt{Lam}(t.e)) \mapsto [\tau/t]e} \tag{23.3b}$$

$$\frac{e \mapsto e'}{\mathtt{App}[\tau](e) \mapsto \mathtt{App}[\tau](e')} \tag{23.3c}$$

It is then a simple matter to prove safety for this language, using by-now familiar methods.

**Lemma 23.2** (Canonical Forms)**.** *Suppose that $e : \tau$ and $e$ $\mathsf{val}$, then*

1. *If $\tau = \mathtt{arr}(\tau_1, \tau_2)$, then $e = \mathtt{lam}[\tau_1](x.e_2)$ with $x : \tau_1 \vdash e_2 : \tau_2$.*
2. *If $\tau = \mathtt{all}(t.\tau')$, then $e = \mathtt{Lam}(t.e')$ with $t$ $\mathsf{type} \vdash e' : \tau'$.*

**Theorem 23.3** (Preservation)**.** *If $e : \sigma$ and $e \mapsto e'$, then $e' : \sigma$.*

**Theorem 23.4** (Progress)**.** *If $e : \sigma$, then either $e$ $\mathsf{val}$ or there exists $e'$ such that $e \mapsto e'$.*

## 23.2 Polymorphic Definability

Although we will not give a proof here, it is possible to show that every well-typed expression in $\mathcal{L}\{\to\forall\}$ evaluates to a value — there is no possibility of writing an infinite loop. It might seem, at first glance, that this is obviously the case, because there is, apparently, no form of iteration or recursion available in the language. After all, the entire language consists solely of function types and polymorphic types, and nothing else, not even a base type!

Surprisingly, though, it *is* possible to define loops in $\mathcal{L}\{\to\forall\}$, albeit ones that always terminate. For example, it is possible to define within $\mathcal{L}\{\to\forall\}$ a type of natural numbers whose elimination form is essentially the iterator described in Chapter 14. More generally, any inductively defined type may be represented in $\mathcal{L}\{\to\forall\}$ in such a way that its associated iterator is definable as well.

Let us begin by showing that the type, $\mathtt{nat}$, is definable in $\mathcal{L}\{\to\forall\}$. This means that we can fill in the following chart in such a way that the static and dynamic semantics are preserved:

$$\begin{aligned} \mathtt{nat} &= \ldots \\ \mathtt{z} &= \ldots \\ \mathtt{s}(e) &= \ldots \\ \mathtt{iter}[\tau](e_0, e_1, x.e_2) &= \ldots \end{aligned}$$

The key to understanding how this is achieved is to focus attention on the iterator.

Recall that the typing rule for the iterator is as follows:

$$\frac{e_0 : \mathtt{nat} \quad e_1 : \tau \quad x : \tau \vdash e_2 : \tau}{\mathtt{iter}[\tau](e_0, e_1, x.e_2) : \tau}\,.$$

Since the type $\tau$ is completely arbitrary, this means that if we have an iterator, then it can be used to define a polymorphic function of type

$$\mathtt{nat} \to \forall(t.t \to (t \to t) \to t).$$

This function, when applied to an argument $n$, yields a polymorphic function that, for any result type, $t$, if given the initial result for $\mathtt{z}$, and if given a function transforming the result for $x$ into the result for $\mathtt{s}(x)$, then it returns the result of iterating the transformer $n$ times starting with the initial result.

Since the *only* operation we can perform on a natural number is to iterate up to it in this manner, we may simply *identify* a natural number, $n$, with the polymorphic iterate-up-to-$n$ function just described. This means that the chart sketched above may be completed as follows:

$$\begin{aligned}
\mathtt{nat} &= \forall(t.t \to (t \to t) \to t)\\
\mathtt{z} &= \Lambda(t.\lambda(z{:}t.\lambda(s{:}t \to t.z)))\\
\mathtt{s}(e) &= \Lambda(t.\lambda(z{:}t.\lambda(s{:}t \to t.s(e[t](z)(s)))))\\
\mathtt{iter}[\tau](e_0, e_1, x.e_2) &= e_0[\tau](e_1)(\lambda(x{:}t.e_2))
\end{aligned}$$

It is a simple matter to check that the static semantics of these constructs is correctly derived from these definitions. We turn, then, to the dynamic semantics.

The number $\mathtt{z}$ iterates a given $s$ zero times starting from $z$, which means that it merely returns $z$. The successor, $\mathtt{s}(e)$, of $e$ iterates $s$ from $z$ for $e$ iterations, then iterates $s$ one more time, as required. Letting $\overline{n}$ stand for the $n$-fold composition $\mathtt{s}(\ldots\mathtt{s}(\mathtt{z})\ldots)$, and assuming a call-by-value semantics for function application, we may show by induction on $n$ that

$$\overline{n}[\tau]e_1e_2 \mapsto^* e_2(\ldots e_2(e_1)\ldots).$$

That is, $\overline{n}$ is indeed the polymorphic iterator specialized to the number $n$.

Since we are identifying natural numbers with their associated iterators, it follows that we should define the iterator from $e_0$ to simply instantiate and apply $e_0$ to the result type, initial result, and result transformer associated with the iterator. Observe that

$$\mathtt{z}[\tau](e_1)(e_2) \mapsto^* e_1$$

and that if

$$\overline{n}[\tau](e_1)(e_2) \mapsto^* e,$$

then

$$\mathtt{s}(\overline{n})\,[\tau]\,(e_1)\,(e_2) \mapsto^* e_2(e).$$

Thus the dynamic semantics is correctly simulated by these definitions.

As an example, here is the addition function defined in terms of this representation of natural numbers:

$$\lambda(x{:}\mathtt{nat}.\,\lambda(y{:}\mathtt{nat}.\,y\,[\mathtt{nat}]\,(x)\,(\lambda(x{:}\mathtt{nat}.\,\mathtt{s}(x))))).$$

Given $x$ and $y$ of type `nat`, this function iterates the successor function (defined above) $y$ times starting with $x$ — that is, it computes the sum of $x$ and $y$.

Following a similar pattern of reasoning, we may define product and sum types, and other, more general, recursive types. Here is a chart of the type definitions:

$$\begin{aligned}
\mathtt{unit} &= \forall(t.\,t \to t)\\
\tau_1 \times \tau_2 &= \forall(t.\,(\tau_1 \to \tau_2 \to t) \to t)\\
\mathtt{void} &= \forall(t.\,t)\\
\tau_1 + \tau_2 &= \forall(t.\,(\tau_1 \to t) \to (\tau_2 \to t) \to t)\\
\tau\,\mathtt{list} &= \forall(t.\,t \to (\tau \to t \to t) \to t)
\end{aligned}$$

We leave it as an exercise to define the introduction and elimination forms for these types according to the same pattern as we did for natural numbers. Remember that the main idea is to represent each introduction form as the elimination form applied to that introduction form.

## 23.3 Restricted Forms of Polymorphism

The remarkable expressive power of the language $\mathcal{L}\{\to\forall\}$ stems from the ability to instantiate a polymorphic type with another polymorphic type. For example, if we let $\tau$ be the type $\forall(t.\,t \to t)$, and, assuming that $e : \tau$, we may apply $e$ to its own type, obtaining the expression $e\,[\tau]$ of type $\tau \to \tau$. Written out in full, this is the type

$$(\forall(t.\,t \to t)) \to (\forall(t.\,t \to t)),$$

which is obviously much "larger" than the type of $e$ itself. In fact, this type is "large enough" that we can go ahead and apply $e\,[\tau]$ to $e$ again, obtaining the expression $e\,[\tau]\,(e)$, which is again of type $\tau$ — the very type of $e$!

Contrast this behavior with the situation in $\mathcal{L}\{\rightarrow\}$, in which the type of an application of a function is evidently "smaller" than the type of the function itself. For if $e : \tau_1 \rightarrow \tau_2$, and $e_1 : \tau_1$, then we have $e(e_1) : \tau_2$, a smaller type than the type of $e$. For this reason $\mathcal{L}\{\rightarrow\}$ is not powerful enough to permit types such as the natural numbers to be defined in terms of function spaces alone — such types have to be built in as primitives.

The source of the expressive power of $\mathcal{L}\{\rightarrow\forall\}$ is that it permits polymorphic types to be instantiated with other polymorphic types, so that we may instantiate $\tau = \forall(t.t \rightarrow t)$ with itself to obtain a "larger" type as result. This property of $\mathcal{L}\{\rightarrow\forall\}$ is called *impredicativity*[1], and we say that $\mathcal{L}\{\rightarrow\forall\}$ permits *impredicative (type) quantification*.

The alternative, called *predicative*[2] *quantification*, is to restrict the quantifier to range only over *un-quantified* types. (For a formalization of this fragment, please see Section 25.3 on page 201.) Under this restriction we may, for example, instantiate the type $\tau$ given above with the type $u \rightarrow u$ to obtain the type $(u \rightarrow u) \rightarrow (u \rightarrow u)$. This type is "larger" than $\tau$ in one sense (it has more symbols), but is "smaller" in another sense (it has fewer quantifiers). For this reason the predicative fragment of the language is substantially less expressive than the impredicative part.

## Prenex Fragment

An even more restricted form of polymorphism, called the *prenex fragment*, further restricts polymorphism to occur only at the outermost level — not only is quantification predicative, but quantifiers are not permitted to occur within the arguments to any other type constructors. This restriction, called *prenex quantification*, is imposed in ML for the sake of *type inference*. Type inference permits the programmer to omit type information entirely from expressions in the knowledge that the compiler can always reconstruct the *most general*, or *principal*, type of an expression. We will not discuss type inference here, but we will give a formulation of the prenex fragment of $\mathcal{L}\{\rightarrow\forall\}$ because it plays such an important role in the design of ML.

The prenex fragment of $\mathcal{L}\{\rightarrow\forall\}$ is obtained by *stratifying* types into two classes, the *monotypes* and the *polytypes*. The monotypes are those that do not involve any quantification, and are thus eligible for instantiation of polymorphic quantifiers. The polytypes include the monotypes, and also

[1] pronounced *im-PRED-ic-a-tiv-it-y*
[2] pronounced *PRED-i-ca-tive*

permit quantification over monotypes to obtain another polytype.

$$\begin{array}{lrcl} \text{Monotype} & \tau & ::= & t \mid \mathtt{arr}(\tau_1, \tau_2) \\ \text{Polytype} & \sigma & ::= & \tau \mid \mathtt{all}(t.\sigma) \end{array}$$

Base types, such as `nat` (as a primitive), or other type constructors, such as sums and products, would be added to the language as monotypes. The polytypes are always of the form

$$\forall(t_1 \ldots \forall(t_n.\tau)\ldots),$$

were $\tau$ is a monotype. We often abbreviate this to just $\forall(t_1,\ldots,t_n.\tau)$.

The static semantics of this fragment of $\mathcal{L}\{\to\forall\}$ is given as follows.

$$\frac{\Delta \mid \Gamma, x:\tau_1 \vdash e_2 : \tau_2}{\Delta \mid \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) : \mathtt{arr}(\tau_1,\tau_2)} \tag{23.4a}$$

$$\frac{\Delta, t\ \mathsf{type} \mid \Gamma \vdash e : \sigma}{\Delta \mid \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\sigma)} \tag{23.4b}$$

$$\frac{\Delta \vdash \tau\ \mathsf{type} \quad \Delta \mid \Gamma \vdash e : \mathtt{all}(t.\sigma)}{\Delta \mid \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\sigma} \tag{23.4c}$$

Expressions are classified by monotypes. We may *generalize* with respect to any free type variable, and *instantiate* any quantified polytype. Since every monotype is also a polytype, these rules "get started" by assigning a monotype to an expression, then generalizing on its free type variables.

This type discipline may then be combined with the `let` construct to obtain the core of the ML type system:

$$\frac{\Delta \mid \Gamma \vdash e_1 : \sigma_1 \quad \Delta \mid \Gamma, x:\sigma_1 \vdash e_2 : \tau_2}{\Delta \mid \Gamma \vdash \mathtt{let}[\sigma_1](e_1, x.e_2) : \tau_2}\ . \tag{23.5}$$

Note that this rule requires that we consider hypotheses of the form $x:\sigma$, which include those of the form $x:\tau$ as a special case. This corresponds to the policy in ML that only variables can have polymorphic type — if you wish to use a function polymorphically, you must bind it to a variable so that it can be assigned a polytype. Each use of a variable must then be instantiated to obtain a monotype so that it can appear in another expression.

The following expression exemplifies the ML type discipline in action. The expression

$$\mathtt{let}\ I{:}\forall(t.t\to t)\ \mathtt{be}\ \Lambda(t.\lambda(x{:}t.x))\ \mathtt{in}\ I[u\to u](I[u])$$

has type $u \to u$, where $u$ is a free type variable. The ML type inference mechanism permits us to suppress mention of types, writing only

$$\texttt{let}\ I\ \texttt{be}\ \lambda x.\, x\ \texttt{in}\ I(I).$$

The type inference process fills in the missing type abstractions and type applications in the most general way possible, with the result being as just illustrated.

## 23.4 Exercises

# Chapter 24

# Data Abstraction

Data abstraction is perhaps the most fundamental technique for structuring programs. The fundamental idea of data abstraction is to separate a *client* from the *implementor* of an abstraction by an *interface*. The interface forms a "contract" between the client and implementor that specifies those properties of the abstraction on which the client may rely, and, correspondingly, those properties that the implementor must satisfy. This ensures that the client is insulated from the details of the implementation of an abstraction so that the implementation can be modified, without changing the client's behavior, provided only that the interface remains the same. This property is called *representation independence* for abstract types.

Data abstraction may be formalized by extending the language $\mathcal{L}\{\rightarrow\forall\}$ with *existential types*. Interfaces are modelled as existential types that provide a collection of operations acting on an unspecified, or abstract, type. Implementations are modelled as packages, the introductory form for existentials, and clients are modelled as uses of the corresponding elimination form. It is remarkable that the programming concept of data abstraction is modelled so naturally and directly by the logical concept of existential type quantification.

Existential types are closely connected with universal types, and hence are often treated together. The superficial reason is that both are forms of type quantification, and hence both require the machinery of type variables. The deeper reason is that existentials are *definable* from universals — surprisingly, data abstraction is actually just a form of polymorphism!

## 24.1 Existential Types

The syntax of $\mathcal{L}\{\rightarrow\forall\exists\}$ is the extension of $\mathcal{L}\{\rightarrow\forall\}$ with the following constructs:

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \texttt{some}(t.\tau) \\ \textit{Expr's} & e & ::= & \texttt{pack}[t.\tau;\rho](e) \mid \texttt{open}[t.\tau](e_1;t,x.e_2) \end{array}$$

The following chart shows the correspondence between concrete and abstract syntax for these constructs.

| *Abstract* | *Concrete* |
|---|---|
| $\texttt{some}(t.\tau)$ | $\exists(t.\tau)$ |
| $\texttt{pack}[t.\tau;\rho](e)$ | $\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$ |
| $\texttt{open}[t.\tau](e_1;t,x.e_2)$ | $\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\texttt{:}\tau\,\texttt{in}\,e_2$ |

The introductory form for the existential type $\sigma = \exists(t.\tau)$ is a *package* of the form $\texttt{pack}\,\rho\,\texttt{with}\,e\,\texttt{as}\,\exists(t.\tau)$, where $\rho$ is a type and $e$ is an expression of type $[\rho/t]\tau$. The type $\rho$ is called the *representation type* of the package, and the expression $e$ is called the *implementation* of the package. The eliminatory form for existentials is the expression $\texttt{open}\,e_1\,\texttt{as}\,t\,\texttt{with}\,x\texttt{:}\tau\,\texttt{in}\,e_2$, which *opens* the package $e_1$ for use within the *client* $e_2$ by binding its representation type to $t$ and its implementation to $x$ for use within $e_2$. Crucially, the typing rules ensure that the client is type-correct independently of the actual representation type used by the implementor, so that it may be varied without affecting the type correctness of the client.

The abstract syntax of the `open` construct specifies that the type variable, $t$, and the expression variable, $x$, are bound within the client. They may be renamed at will by $\alpha$-equivalence without affecting the meaning of the construct, provided, of course, that the names are chosen so as not to conflict with any others that may be in scope. In other words the type, $t$, may be thought of as a "new" type, one that is distinct from all other types, when it is introduced. This is sometimes called *generativity* of abstract types: the use of an abstract type by a client "generates" a "new" type within that client. This behavior is simply a consequence of identifying terms up to $\alpha$-equivalence, and is not particularly tied to data abstraction.

### 24.1.1 Static Semantics

The static semantics of existential types is specified by rules defining when an existential is well-formed, and by giving typing rules for the associated

introductory and eliminatory forms.

$$\frac{\Delta, t \text{ type} \mid \tau \text{ type}}{\Delta \mid \texttt{some}(t.\tau) \text{ type}} \tag{24.1a}$$

$$\frac{\Delta \mid \rho \text{ type} \quad \Delta, t \text{ type} \mid \tau \text{ type} \quad \Delta \mid \Gamma \vdash e : [\rho/t]\tau}{\Delta \mid \Gamma \vdash \texttt{pack}[t.\tau;\rho]\,(e) : \texttt{some}(t.\tau)} \tag{24.1b}$$

$$\frac{\Delta \mid \Gamma \vdash e_1 : \texttt{some}(t.\tau) \quad \Delta, t \text{ type} \mid \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \mid \tau_2 \text{ type}}{\Delta \mid \Gamma \vdash \texttt{open}[t.\tau]\,(e_1; t, x.e_2) : \tau_2} \tag{24.1c}$$

Rule (24.1c) is complex, so study it carefully! There are two important things to notice:

1. The type of the client, $\tau_2$, must not involve the abstract type $t$. This restriction prevents the client from attempting to export a value of the abstract type outside of the scope of its definition.

2. The body of the client, $e_2$, is type checked without knowledge of the representation type, $t$. The client is, in effect, polymorphic in the type variable $t$.

### 24.1.2 Dynamic Semantics

The dynamic semantics of existential types is specified as follows:

$$\frac{\{e \text{ val}\}}{\texttt{pack}[t.\tau;\rho]\,(e) \text{ val}} \tag{24.2a}$$

$$\left\{ \frac{e \mapsto e'}{\texttt{pack}[t.\tau;\rho]\,(e) \mapsto \texttt{pack}[t.\tau;\rho]\,(e')} \right\} \tag{24.2b}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{open}[t.\tau]\,(e_1; t, x.e_2) \mapsto \texttt{open}[t.\tau]\,(e_1'; t, x.e_2)} \tag{24.2c}$$

$$\frac{e \text{ val}}{\texttt{open}[t.\tau]\,(\texttt{pack}[t.\tau;\rho]\,(e); t, x.e_2) \mapsto [\rho, e/t, x]e_2} \tag{24.2d}$$

The bracketed premises and rules are to be omitted for a lazy semantics, and included for an eager semantics.

Observe that *there are no abstract types at run time*! The representation type is fully exposed to the client during evaluation. Data abstraction is a compile-time discipline that imposes no run-time overhead.

### 24.1.3 Safety

The safety of the extension is stated and proved as usual. The argument is a simple extension of that used for $\mathcal{L}\{\rightarrow\forall\}$ to the new constructs.

**Theorem 24.1** (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

**Lemma 24.2** (Canonical Forms)**.** *If $e : \texttt{some}(t.\tau)$ and $e$ val, then $e = \texttt{pack}[t.\tau;\rho](e')$ for some type $\rho$ and some $e'$ val such that $e' : [\rho/t]\tau$.*

**Theorem 24.3** (Progress)**.** *If $e : \tau$ then either $e$ val or there exists $e'$ such that $e \mapsto e'$.*

## 24.2 Data Abstraction Via Existentials

To illustrate the use of existentials for data abstraction, we consider an abstract type of (persistent) queues supporting three operations:

1. Formation of the empty queue.
2. Inserting an element at the tail of the queue.
3. Remove the head of the queue.

This is clearly a bare-bones interface, but is sufficient to illustrate the main ideas of data abstraction. Queue elements may be taken to be of any type, $\tau$, of our choosing; we will not be specific about this choice, since nothing depends on it.

The crucial property of this description is that nowhere do we specify what queues actually *are*, only what we can *do* with them. This is captured by the following existential type, $\exists(t.\sigma)$, which serves as the interface of the queue abstraction:[1]

$$\exists(t.\langle \texttt{emp}:t, \texttt{ins}:\tau\times t\rightarrow t, \texttt{rem}:t\rightarrow\tau\times t\rangle).$$

The representation type, $t$, of queues is *abstract* — all that is specified about it is that it supports the operations `emp`, `ins`, and `rem`, with the specified types.

An implementation of queues consists of a package specifying the representation type, together with the implementation of the associated operations in terms of that representation. Internally to the implementation,

[1] For the sake of illustration, we assume that type constructors such as products, records, and lists are also available in the language.

the representation of queues is known and relied upon by the operations. Here is a very simple implementation, $e_l$, in which queues are represented as lists:

$$\texttt{pack}\,\tau\,\texttt{list}\,\texttt{with}\,\langle\texttt{emp}=\texttt{nil},\texttt{ins}=e_i,\texttt{rem}=e_r\rangle\,\texttt{as}\,\exists(t.\sigma),$$

where

$$e_i : \tau \times \tau\,\texttt{list} \to \tau\,\texttt{list} = \lambda(x{:}\tau \times \tau\,\texttt{list}.\, e_i'),$$

and

$$e_r : \tau\,\texttt{list} \to \tau \times \tau\,\texttt{list} = \lambda(x{:}\tau\,\texttt{list}.\, e_r').$$

Here the expression $e_i'$ conses the first component of $x$, the element, onto the second component of $x$, the queue. Correspondingly, the expression $e_r'$ reverses its argument, and returns the head element paired with the reversal of the tail. These operations "know" that queues are represented as values of type $\tau\,\texttt{list}$, and are programmed accordingly.

It is also possible to give another implementation, $e_p$, of the same interface, $\exists(t.\sigma)$, but in which queues are represented as pairs of lists, consisting of the "back half" of the queue paired with the reversal of the "front half". This representation avoids the need for reversals on each call, and, as a result, achieves amortized constant-time behavior:

$$\texttt{pack}\,\tau\,\texttt{list} \times \tau\,\texttt{list}\,\texttt{with}\,\langle\texttt{emp}=\langle\texttt{nil},\texttt{nil}\rangle,\texttt{ins}=e_i,\texttt{rem}=e_r\rangle\,\texttt{as}\,\exists(t.\sigma).$$

In this case $e_i$ has type

$$\tau \times (\tau\,\texttt{list} \times \tau\,\texttt{list}) \to (\tau\,\texttt{list} \times \tau\,\texttt{list}),$$

and $e_r$ has type

$$(\tau\,\texttt{list} \times \tau\,\texttt{list}) \to \tau \times (\tau\,\texttt{list} \times \tau\,\texttt{list}).$$

These operations "know" that queues are represented as values of type

$$\tau\,\texttt{list} \times \tau\,\texttt{list},$$

and are implemented accordingly.

Clients of the queue abstraction are shielded from the implementation details by the open construct. If $e$ is *any* implementation of $\exists(t.\sigma)$, then a client of the abstraction has the form

$$\texttt{open}\,e\,\texttt{as}\,t\,\texttt{with}\,x{:}\sigma\,\texttt{in}\,e' : \tau',$$

where the type, $\tau'$, of $e'$ does not involve the abstract type $t$. Within $e'$ the variable $x$ has type

$$\langle \mathtt{emp}:t, \mathtt{ins}:\tau \times t \to t, \mathtt{rem}:t \to \tau \times t \rangle,$$

in which $t$ is unspecified — or, as is often said, *held abstract*.

Observe that *only* the type information specified in $\exists(t.\sigma)$ is propagated to the client, $e'$, and nothing more. Consequently, the `open` expression above type checks properly regardless of whether $e$ is $e_l$ (the implementation of $\exists(t.\sigma)$ in terms of lists) or $e_p$ (the implementation in terms of pairs of lists), or, for that matter, any other implementation of the same interface. This property is called *representation independence*, because the client is guaranteed to be independent of the representation of the abstraction.

## 24.3 Definability of Existentials

Strictly speaking, it is not necessary to extend $\mathcal{L}\{\to\forall\}$ with existential types in order to model data abstraction, because they are definable in terms of universals! Before giving the details, let us consider why this should be possible. The key is to observe that the client of an abstract type is *polymorphic* in the representation type. The typing rule for

$$\mathtt{open}\, e\, \mathtt{as}\, t\, \mathtt{with}\, x{:}\tau\, \mathtt{in}\, e' : \tau',$$

where $e : \exists(t.\tau)$, specifies that $e' : \tau'$ under the assumptions $t$ type and $x : \tau$. In essence, the client is a polymorphic function of type

$$\forall(t.\tau \to \tau'),$$

where $t$ may occur in $\tau$ (the type of the operations), but not in $\tau'$ (the type of the result).

This suggests the following encoding of existential types:

$$\begin{aligned} \exists(t.\sigma) &= \forall(t'.\forall(t.\sigma \to t') \to t') \\ \mathtt{pack}\,\rho\,\mathtt{with}\,e\,\mathtt{as}\,\exists(t.\tau) &= \Lambda(t'.\lambda(x{:}\forall(t.\tau \to t').x[\rho](e))) \\ \mathtt{open}\,e\,\mathtt{as}\,t\,\mathtt{with}\,x{:}\tau\,\mathtt{in}\,e' &= e[\tau'](\Lambda(t.\lambda(x{:}\tau.e'))) \end{aligned}$$

An existential is encoded as a polymorphic function taking the overall result type, $t'$, as argument, followed by a polymorphic function representing the client with result type $t'$, and yielding a value of type $t'$ as overall result. Consequently, the `open` construct simply packages the client as such a

polymorphic function, instantiates the existential at the result type, $\tau'$, and applies it to the polymorphic client. (The translation therefore depends on knowing the overall result type, $\tau'$, of the `open` construct.) Finally, a package consisting of a representation type $\tau$ and an implementation $e$ is a polymorphic function that, when given the result type, $t'$, and the client, $x$, instantiates $x$ with $\tau$ and passes to it the implementation $e$.

It is then a straightforward exercise to show that this translation correctly reflects the static and dynamic semantics of existential types.

## 24.4 Exercises

# Chapter 25

# Constructors and Kinds

In Chapters 23 and 24 we introduced the concept of a *type variable*, which stands for a fixed, but unspecified type. These formed the foundation for *type quantification*, both universal and existential, which forms the foundation for polymorphism and data abstraction, respectively. When scaling to more realistic languages, it quickly becomes apparent that type quantification is not enough. For example, suppose we wish to introduce an abstract *type constructor*, such as an abstract type $\tau$ `set` of sets of values of type $\tau$? Existential type quantification is not sufficient to express such an abstraction in general—only specific instances of it for specific choices of type $\tau$, exactly the sort of replication we sought to avoid by introducing type abstraction in the first place!

The solution is to generalize the framework to permit a richer class of variables than just type variables, including (at least) type constructor variables, and variables ranging over tuples of types. This is achieved by enriching $\mathcal{L}\{\rightarrow\forall\exists\}$ with *constructors* and *kinds*. Constructors are a form of "static value" classified by kinds, just as expressions are "dynamic values" classified by types. In this setting types arise as constructors of the *kind of types*, and type constructors are constructors of so-called *higher kind*.

In this chapter we will present the basic infrastructure of constructors and kinds in the language $\mathcal{L}\{\rightarrow\forall_\kappa,\exists_\kappa\}$ of universal and existential quantification over higher kinds. Later we will see further applications and generalizations of the kind structure to model subtyping and type definitions.

## 25.1 Syntax of Constructors and Kinds

The abstract syntax of $\mathcal{L}\{\rightarrow\forall_\kappa,\exists_\kappa\}$ is given by the following grammar:

| | | | |
|---|---|---|---|
| Kind | $\kappa$ | ::= | $\mathtt{Type} \mid \mathtt{Prod}(\kappa_1,\kappa_2) \mid \mathtt{Arr}(\kappa_1,\kappa_2)$ |
| Cons | $c$ | ::= | $t \mid \mathtt{arr} \mid \mathtt{all}[\kappa] \mid \mathtt{some}[\kappa] \mid \mathtt{pair}(c_1,c_2) \mid \mathtt{fst}(c) \mid$ $\mathtt{snd}(c) \mid \mathtt{lambda}[\kappa](t.c) \mid \mathtt{app}(c_1,c_2)$ |
| Type | $\tau$ | ::= | $c$ |
| Expr | $e$ | ::= | $x \mid \mathtt{lam}[\tau](x.e) \mid \mathtt{ap}(e_1,e_2) \mid \mathtt{Lam}[\kappa](t.e) \mid$ $\mathtt{App}[c](e) \mid \mathtt{pack}[\kappa,c,c']\,(e) \mid$ $\mathtt{open}[\kappa,c](e_1,t,x.e_2)$ |

The corresponding concrete syntax is given by the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\mathtt{Prod}(\kappa_1,\kappa_2)$ | $\kappa_1 \times \kappa_2$ |
| $\mathtt{Arr}(\kappa_1,\kappa_2)$ | $\kappa_1 \rightarrow \kappa_2$ |
| $\mathtt{app}(\mathtt{app}(\mathtt{arr},c_1),c_2)$ | $c_1 \rightarrow c_2$ |
| $\mathtt{app}(\mathtt{all}[\kappa],\mathtt{lambda}[\kappa](t.c))$ | $\forall_\kappa(\lambda(t::\kappa.c))$ or $\forall_\kappa(t::\kappa.c)$ |
| $\mathtt{app}(\mathtt{some}[\kappa],\mathtt{lambda}[\kappa](t.c))$ | $\exists_\kappa(\lambda(t::\kappa.c))$ or $\exists_\kappa(t::\kappa.c)$ |
| $\mathtt{pair}(c_1,c_2)$ | $\langle c_1,c_2\rangle$ |
| $\mathtt{fst}(c)$ | $\mathtt{fst}(c)$ |
| $\mathtt{snd}(c)$ | $\mathtt{snd}(c)$ |
| $\mathtt{lambda}[\kappa](t.c)$ | $\lambda(t::\kappa.c)$ |
| $\mathtt{app}(c_1,c_2)$ | $c_1(c_2)$ |
| $\mathtt{Lam}[\kappa](t.e)$ | $\Lambda(t::\kappa.e)$ |
| $\mathtt{App}[c](e)$ | $e[c]$ |
| $\mathtt{pack}[\kappa,c,c']\,(e)$ | $\mathtt{pack}\,c'\,\mathtt{with}\,e\,\mathtt{as}\,\exists_\kappa(c)$ |
| $\mathtt{open}[\kappa,c](e_1,t,x.e_2)$ | $\mathtt{open}\,e_1\,\mathtt{as}\,t::\kappa\,\mathtt{with}\,x{:}\mathtt{app}(c,t)\,\mathtt{in}\,e_2$ |

The abstract representations of arrow types and quantified types arises from the interpretation of the function type constructor and the two quantifiers as constants of higher kind, as will become clear shortly.

## 25.2 Static Semantics

The static semantics of $\mathcal{L}\{\rightarrow\forall_\kappa,\exists_\kappa\}$ consists of hypothetico-general judgements of the following forms:

| | |
|---|---|
| $\mathcal{T} \mid \Delta \vdash c :: \kappa$ | $c$ is a constructor of kind $\kappa$ |
| $\mathcal{T} \mid \Delta \vdash c_1 \equiv c_2 :: \kappa$ | $c_1$ and $c_2$ are equivalent constructors of kind $\kappa$ |
| $\mathcal{T} \mid \Delta \vdash \tau\ \mathtt{type}$ | synonymous with $\mathcal{T} \mid \Delta \vdash \tau :: \mathtt{Type}$ |
| $\mathcal{T}\,\mathcal{X} \mid \Delta\,\Gamma \vdash e : \tau$ | $e$ is an expression of type $\tau$ |

Here $\mathcal{T}$ stands for a finite set of *constructor name* declarations of the form $t$ cons, and $\mathcal{X}$ stands for a finite set of *expression name* declarations of the form $t$ exp. The hypotheses $\Delta$ have the form $t :: \kappa$, where $\mathcal{T} \vdash t$ cons, and the hypotheses $\Gamma$ have the form $x : \tau$ where $\mathcal{X} \vdash x$ exp and $\mathcal{T} \mid \Delta \vdash \tau$ type. As usual, we omit explicit declaration of $\mathcal{T}$ and $\mathcal{X}$, since they can be recovered from the assumptions in $\Delta$ and $\Gamma$, respectively.

The constructor formation judgement, $\Delta \vdash c :: \kappa$, is inductively defined by the following rules:

$$\frac{}{\Delta, t :: \kappa \vdash t :: \kappa} \tag{25.1a}$$

$$\frac{}{\Delta \vdash \mathtt{arr} :: \mathtt{Arr(Type,Arr(Type,Type))}} \tag{25.1b}$$

$$\frac{}{\Delta \vdash \mathtt{all}[\kappa] :: \mathtt{Arr(Arr}(\kappa,\mathtt{Type}),\mathtt{Type})} \tag{25.1c}$$

$$\frac{}{\Delta \vdash \mathtt{some}[\kappa] :: \mathtt{Arr(Arr}(\kappa,\mathtt{Type}),\mathtt{Type})} \tag{25.1d}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \mathtt{pair}(c_1,c_2) :: \mathtt{Prod}(\kappa_1,\kappa_2)} \tag{25.1e}$$

$$\frac{\Delta \vdash c :: \mathtt{Prod}(\kappa_1,\kappa_2)}{\Delta \vdash \mathtt{fst}(c) :: \kappa_1} \tag{25.1f}$$

$$\frac{\Delta \vdash c :: \mathtt{Prod}(\kappa_1,\kappa_2)}{\Delta \vdash \mathtt{snd}(c) :: \kappa_2} \tag{25.1g}$$

$$\frac{\Delta, t :: \kappa_1 \vdash c_2 :: \kappa_2}{\Delta \vdash \mathtt{lambda}[\kappa_1](t.c_2) :: \mathtt{Arr}(\kappa_1,\kappa_2)} \tag{25.1h}$$

$$\frac{\Delta \vdash c_1 :: \mathtt{Arr}(\kappa_2,\kappa) \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \mathtt{app}(c_1,c_2) :: \kappa} \tag{25.1i}$$

There is an evident correspondence between these rules and the rules for functions and products given in Chapters 14 and 16, except that we are working at the static level of constructors and kinds, rather than the dynamic level of expressions and types.

Observe that the constants `arr`, `all`$[\kappa]$, and `some`$[\kappa]$ all have functional kinds. The kind of `arr` is a curried function kind that specifies that the function type constructor takes two types as arguments, yielding a type. It could just as well have been specified in uncurried form to have the kind `Arr(Prod(Type,Type),Type)`, in which case it is to be applied to a pair of

types, rather than successively applied to two types, to form a type. The kind of the quantifiers, `all[`$\kappa$`]` and `some[`$\kappa$`]`, namely `Arr(Arr(`$\kappa$`,Type),Type)`, specifies that the body of the quantifier is a function mapping constructors of kind $\kappa$ to types (*i.e.*, constructors of kind `Type`). Instantiation of the quantifiers is achieved by application, as we shall see in the typing rules for expressions given below.

The equivalence judgement relating constructors (in particular, types) expresses *definitional equality* of constructors. Roughly speaking, two constructors are definitionally equivalent iff they are considered to be identical by virtue of their form and kind, and this may be seen immediately by inspection. For example, `fst(`$\langle c_1, c_2 \rangle$`)` and $c_1$ are definitionally equivalent (when they are well-formed), because we can see immediately that the first component of an explicitly given pair is the left-hand side of that pair. The typing relation is required to respect definitional equivalence in that definitionally equivalent types classify the same expressions.

Definitional equality of well-formed constructors is defined to be the least congruence closed under the following rules:[1]

$$\frac{\Delta \vdash \texttt{pair}(c_1, c_2) :: \texttt{Prod}(\kappa_1, \kappa_2)}{\Delta \vdash \texttt{fst}(\texttt{pair}(c_1, c_2)) \equiv c_1 :: \kappa_1} \tag{25.2a}$$

$$\frac{\Delta \vdash \texttt{pair}(c_1, c_2) :: \texttt{Prod}(\kappa_1, \kappa_2)}{\Delta \vdash \texttt{snd}(\texttt{pair}(c_1, c_2)) \equiv c_2 :: \kappa_2} \tag{25.2b}$$

$$\frac{\Delta \vdash c :: \texttt{Prod}(\kappa_1, \kappa_2)}{\Delta \vdash \texttt{pair}(\texttt{fst}(c), \texttt{snd}(c)) \equiv c :: \texttt{Prod}(\kappa_1, \kappa_2)} \tag{25.2c}$$

$$\frac{\Delta, t :: \kappa_1 \vdash c_2 :: \kappa_2 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \texttt{app}(\texttt{lambda}[\kappa](t.c_1), c_2) \equiv [c_2/t]c_1 :: \kappa_2} \tag{25.2d}$$

$$\frac{\Delta \vdash c_2 :: \texttt{Arr}(\kappa_1, \kappa_2) \quad t \# \Delta}{\Delta \vdash \texttt{lambda}[\kappa_1](t.\texttt{app}(c_2, t)) \equiv c_2 :: \texttt{Arr}(\kappa_1, \kappa_2)} \tag{25.2e}$$

The rules for typing expressions are a straightforward generalization of those given in Chapters 23 and 24.

$$\frac{\Delta\,\Gamma \vdash e : \tau' \quad \Delta \vdash \tau \equiv \tau' :: \texttt{Type}}{\Delta\,\Gamma \vdash e : \tau} \tag{25.3a}$$

[1] A congruence is an equivalence relation that is compatible with the introductory and eliminatory constructors of the language, so that we may "replace equals by equals" anywhere within a constructor.

$$\frac{}{\Delta\,\Gamma, x : \tau \vdash x : \tau} \tag{25.3b}$$

$$\frac{\Delta\,\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\,\Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) : \mathtt{app}(\mathtt{app}(\mathtt{arr}, \tau_1), \tau_2)} \tag{25.3c}$$

$$\frac{\Delta\,\Gamma \vdash e_1 : \mathtt{app}(\mathtt{app}(\mathtt{arr}, \tau_2), \tau) \quad \Delta\,\Gamma \vdash e_2 : \tau_2}{\Delta\,\Gamma \vdash \mathtt{ap}(e_1, e_2) : \tau} \tag{25.3d}$$

$$\frac{\Delta, t :: \kappa\,\Gamma \vdash e : \tau}{\Delta\,\Gamma \vdash \mathtt{Lam}[\kappa](t.e) : \mathtt{app}(\mathtt{all}[\kappa], \mathtt{lambda}[\kappa](t.\tau))} \tag{25.3e}$$

$$\frac{\Delta\,\Gamma \vdash e : \mathtt{app}(\mathtt{all}[\kappa], c') \quad \Delta \vdash c :: \kappa}{\Delta\,\Gamma \vdash \mathtt{App}[c](e) : \mathtt{app}(c', c)} \tag{25.3f}$$

$$\frac{\Delta \vdash c' :: \mathtt{Arr}(\kappa, \mathtt{Type}) \quad \Delta \vdash c :: \kappa \quad \Delta\,\Gamma \vdash e : \mathtt{app}(c', c)}{\Delta\,\Gamma \vdash \mathtt{pack}[\kappa, c', c](e) : \mathtt{app}(\mathtt{some}[\kappa], c')} \tag{25.3g}$$

$$\frac{\Delta\,\Gamma \vdash e_1 : \mathtt{app}(\mathtt{some}[\kappa], c) \quad \Delta, t :: \kappa\,\Gamma, x : \mathtt{app}(c, t) \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta\,\Gamma \vdash \mathtt{open}[\kappa, c](e_1, t, x.e_2) : \tau_2} \tag{25.3h}$$

## 25.3 Alternate Formulations

The formulation of $\mathcal{L}\{\rightarrow \forall_\kappa, \exists_\kappa\}$ identifies types with constructors of kind `Type`. Consequently, constructors of kind `Type` have a dual role:

1. As static data values that may be passed as arguments to polymorphic functions or bound into packages of existential type.

2. As classifiers of dynamic data values according to the typing rules for expressions.

The dual role of such constructors is apparent in our use of the meta-variables $c$ and $\tau$ for constructors of kind `Type` in the static semantics. These two roles can be separated, and in some presentations are separated in a manner that we shall now outline briefly.

The main idea is to make explicit the inclusion of types-as-constructors into the collection of types-as-classifiers. This forces us to distinguish the dual role of type constructors as functions acting on types-as-constructors

and as combining forms for types-as-classifiers. Syntactically, this means that the abstract syntax of constructors and types takes the following form:

$$\begin{array}{llll} \text{Cons} & c & ::= & t \mid \texttt{arr} \mid \texttt{all}[\kappa] \mid \texttt{some}[\kappa] \mid \texttt{pair}(c_1, c_2) \mid \texttt{fst}(c) \mid \\ & & & \texttt{snd}(c) \mid \texttt{lambda}[\kappa](t.c) \mid \texttt{app}(c_1, c_2) \\ \text{Type} & \tau & ::= & \texttt{typ}(c) \mid \texttt{arr}(\tau_1, \tau_2) \mid \forall_\kappa(t::\kappa.\tau) \mid \exists_\kappa(t::\kappa.\tau) \end{array}$$

There is a certain redundancy in having a constructor and a type for each construct, but we shall see shortly that the duplication is mitigated when we restrict to the predicative fragment.

The two levels are linked by the following definitional equalities, stated without explicit contexts for the sake of clarity:

$$\overline{\texttt{typ}(\texttt{app}(\texttt{app}(\texttt{arr}, c_1), c_2)) \equiv \texttt{arr}(\texttt{typ}(c_1), \texttt{typ}(c_2)) :: \texttt{Type}} \tag{25.4a}$$

$$\overline{\texttt{typ}(\texttt{app}(\texttt{all}[\kappa], c)) \equiv \forall_\kappa(t::\kappa.\texttt{typ}(\texttt{app}(c, t))) :: \texttt{Type}} \tag{25.4b}$$

$$\overline{\texttt{typ}(\texttt{app}(\texttt{some}[\kappa], c)) \equiv \exists_\kappa(t::\kappa.\texttt{typ}(\texttt{app}(c, t))) :: \texttt{Type}} \tag{25.4c}$$

One advantage of distinguishing constructors (of kind Type) from types is that it facilitates formalization of the predicative fragment of $\mathcal{L}\{\rightarrow\forall\}$ described informally in Chapter 23. According to this view, a constructor, $c$, of kind Type may be thought of as a static *representative* of the type $\texttt{typ}(c)$, whose meaning is determined by the axioms of definitional equality. Thus, every constructor of kind Type designates some type. However, not every type need have a representative as such a constructor! In particular, the predicative fragment of $\mathcal{L}\{\rightarrow\forall\}$ is obtained by simply omitting the constants $\texttt{all}[\kappa]$ and $\texttt{some}[\kappa]$, which serve to provide representatives for the quantified types. Without these, the kind Type excludes quantified types, and hence type quantification becomes predicative. (Note, however, that the prenex fragment is not as easily characterized in this manner, since it requires restriction on the form of quantified types-as-classifiers, in addition to the predicative restriction on the meanings of the quantifiers.)

## 25.4 Exercises

# Part IX

# Control Flow

# Chapter 26

# An Abstract Machine for Control

The technique of specifying the dynamic semantics as a transition system is very useful for theoretical purposes, such as proving type safety, but is too high level to be directly usable in an implementation. One reason is that the use of "search rules" requires the traversal and reconstruction of an expression in order to simplify one small part of it. In an implementation we would prefer to use some mechanism to record "where we are" in the expression so that we may "resume" from that point after a simplification. This can be achieved by introducing an explicit mechanism, called a *control stack*, that keeps track of the context of an instruction step for just this purpose. By making the control stack explicit the transition rules avoid the need for any premises — every rule is an axiom! This is the formal expression of the informal idea that no traversals or reconstructions are required to implement it.

By making the control stack explicit we move closer to an actual implementation of the language. As we expose more and more of the mechanisms required in an implementation we get closer and closer to the physical machine. At each step along the way, starting with the structural semantics and continuing down towards an assembly-level description, we are working with a particular *abstract*, or *virtual*, *machine*. The closer we get to the physical machine, the less "abstract" and the more "concrete" it becomes. But there is no clear dividing line between the levels, rather it is a matter of progressive exposure of implementation details. After all, even machine instructions are implemented using gates, and gates are implemented using transistors, and so on down to the level of fundamental

physics.

Nevertheless, some abstract machines are more concrete than others, and recently there has been a resurgence of interest in using them to provide a hardware-independent computing platform. The idea is to define a low-enough level abstract machine such that (a) it is easily implementable on typical hardware platforms, and (b) higher-level languages can be translated (compiled) to it. In this way it is hoped that most software can be freed of dependence on specific hardware platforms.[1] It is of paramount importance that the abstract machine be precisely defined, for otherwise it is not clear how to translate to it, nor is it clear how to implement it on a given platform.

In this chapter we introduce an abstract machine, $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$, for the language $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. The purpose of this machine is to make control flow explicit by introducing a control stack that maintains a record of the pending sub-computations of a computation. We then prove the equivalence of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ with the structural operational semantics of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$.

## 26.1 Machine Definition

A state, $s$, of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ consists of a *control stack*, $k$, and a closed expression, $e$. States may take one of two forms:

1. An *evaluation* state of the form $k \triangleright e$ corresponds to the evaluation of a closed expression, $e$, relative to a control stack, $k$.
2. A *return* state of the form $k \triangleleft e$, where $e$ $\mathsf{val}$, corresponds to the evaluation of a stack, $k$, relative to a closed value, $e$.

As an aid to memory, note that the separator "points to" the focal entity of the state, the expression in an evaluation state and the stack in a return state.

The control stack represents the context of evaluation. It records the "current location" of evaluation, the context into which the value of the current expression is to be returned. Formally, a control stack is a list of *frames*:

$$\frac{}{\varepsilon\ \mathsf{stack}} \tag{26.1a}$$

$$\frac{f\ \mathsf{frame} \quad k\ \mathsf{stack}}{f; k\ \mathsf{stack}} \tag{26.1b}$$

[1] This is much easier said than done; it remains an active area of research and development.

The definition of frame depends on the language we are evaluating. The frames of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are inductively defined by the following rules:

$$\frac{}{\mathtt{s}(-)\ \mathsf{frame}} \tag{26.2a}$$

$$\frac{}{\mathtt{ifz}(-, e_1, x\,.\,e_2)\ \mathsf{frame}} \tag{26.2b}$$

$$\frac{}{\mathtt{ap}(-, e_2)\ \mathsf{frame}} \tag{26.2c}$$

$$\frac{}{\mathtt{ap}(e_1, -)\ \mathsf{frame}} \tag{26.2d}$$

The frames correspond to rules with transition premises in the dynamic semantics of $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. Thus, instead of relying on the structure of the transition derivation to maintain a record of pending computations, we make an explicit record of them in the form of a frame on the control stack.

The transition judgement between states of the $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ is inductively defined by a set of inference rules. We begin with the rules for natural numbers.

$$\frac{}{k \triangleright \mathtt{z} \mapsto k \triangleleft \mathtt{z}} \tag{26.3a}$$

$$\frac{}{k \triangleright \mathtt{s}(e) \mapsto \mathtt{s}(-); k \triangleright e} \tag{26.3b}$$

$$\frac{}{\mathtt{s}(-); k \triangleleft e \mapsto k \triangleleft \mathtt{s}(e)} \tag{26.3c}$$

To evaluate $\mathtt{z}$ we simply return it. To evaluate $\mathtt{s}(e)$, we push a frame on the stack to record the pending successor, and evaluate $e$; when that returns with $e'$, we return $\mathtt{s}(e')$ to the stack.

Next, we consider the rules for case analysis.

$$\frac{}{k \triangleright \mathtt{ifz}(e, e_1, x\,.\,e_2) \mapsto \mathtt{ifz}(-, e_1, x\,.\,e_2); k \triangleright e} \tag{26.4a}$$

$$\frac{}{\mathtt{ifz}(-, e_1, x\,.\,e_2); k \triangleleft \mathtt{z} \mapsto k \triangleright e_1} \tag{26.4b}$$

$$\frac{}{\mathtt{ifz}(-, e_1, x\,.\,e_2); k \triangleleft \mathtt{s}(e) \mapsto k \triangleleft [e/x]e_2} \tag{26.4c}$$

First, the test expression is evaluated, recording the pending case analysis on the stack. Once the value of the test expression has been determined,

we branch to the appropriate arm of the conditional, substituting the predecessor in the case of a positive number.

Finally, we consider the rules for functions and recursion.

$$\overline{k \triangleright \mathtt{lam}[\tau](x.e) \mapsto k \triangleleft \mathtt{lam}[\tau](x.e)} \tag{26.5a}$$

$$\overline{k \triangleright \mathtt{ap}(e_1, e_2) \mapsto \mathtt{ap}(-, e_2); k \triangleright e_1} \tag{26.5b}$$

$$\frac{e_1 \text{ val}}{\mathtt{ap}(-, e_2); k \triangleleft e_1 \mapsto \mathtt{ap}(e_1, -); k \triangleright e_2} \tag{26.5c}$$

$$\frac{e_2 \text{ val} \quad e_1 = \mathtt{fun}[\tau_1, \tau_2](f.x.e)}{\mathtt{ap}(e_1, -); k \triangleleft e_2 \mapsto k \triangleright [e_1, e_2/f, x]e} \tag{26.5d}$$

$$\overline{k \triangleright \mathtt{fix}[\tau](x.e) \mapsto k \triangleright [\mathtt{fix}[\tau](x.e)/x]e} \tag{26.5e}$$

These rules ensure that the function is evaluated before the argument, applying the function when both have been evaluated. Note that evaluation of general recursion requires no stack space! (But see Chapter 38 for more on evaluation of general recursion.)

The initial and final states of the $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ are defined by the following rules:

$$\overline{\varepsilon \triangleright e \text{ initial}} \tag{26.6a}$$

$$\frac{e \text{ val}}{\varepsilon \triangleleft e \text{ final}} \tag{26.6b}$$

## 26.2 Safety

To define and prove safety for $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ requires that we introduce a new typing judgement, $k : \tau$, stating that the stack $k$ is well-formed and expects a value of type $\tau$. For a stack to be well-formed means that, when passed a value of appropriate type, it safely transforms that value into an *answer*, which is the final result of a program. The type $\tau_{ans}$ is defined to be the type of complete programs, which should be a type whose values are directly observable. For $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$ we will take $\tau_{ans}$ to be the type $\mathtt{nat}$, but in richer languages other choices are also possible.

Since control stacks represent the work remaining to complete a computation, the typing judgement for control stacks, $k : \tau$, means that the stack

$k$ transforms a value of type $\tau$ into a value of type $\tau_{ans}$. This judgement is inductively defined by the following rules:

$$\frac{}{\varepsilon : \tau_{ans}} \tag{26.7a}$$

$$\frac{k : \tau' \quad f : \tau \Rightarrow \tau'}{f;k : \tau} \tag{26.7b}$$

This definition makes use of an auxiliary judgement, $f : \tau \Rightarrow \tau'$, stating that a frame $f$ transforms a value of type $\tau$ to a value of type $\tau'$.

$$\frac{}{\mathtt{s}(-) : \mathtt{nat} \Rightarrow \mathtt{nat}} \tag{26.8a}$$

$$\frac{e_1 : \tau \quad x : \mathtt{nat} \vdash e_2 : \tau}{\mathtt{ifz}(-, e_1, x.e_2) : \mathtt{nat} \Rightarrow \tau} \tag{26.8b}$$

$$\frac{e_2 : \tau_2}{\mathtt{ap}(-, e_2) : \mathtt{arr}(\tau_2, \tau) \Rightarrow \tau} \tag{26.8c}$$

$$\frac{e_1 : \mathtt{arr}(\tau_2, \tau) \quad e_1 \ \mathsf{val}}{\mathtt{ap}(e_1, -) : \tau_2 \Rightarrow \tau} \tag{26.8d}$$

The two forms of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ state are well-formed provided that their stack and expression components match.

$$\frac{k : \tau \quad e : \tau}{k \triangleright e \ \mathsf{ok}} \tag{26.9a}$$

$$\frac{k : \tau \quad e : \tau \quad e \ \mathsf{val}}{k \triangleleft e \ \mathsf{ok}} \tag{26.9b}$$

We leave the proof of safety of $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ as an exercise.

**Theorem 26.1** (Safety). *1. If $s$ ok and $s \mapsto s'$, then $s'$ ok.*

*2. If $s$ ok, then either $s$ final or there exists $s'$ such that $s \mapsto s'$.*

## 26.3 Correctness of the Control Machine

It is natural to ask whether $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ correctly implements $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$. If we evaluate a given expression, $e$, using $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$, do we get the same result as would be given by $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$, and *vice versa*?

Answering this question decomposes into two propositions relating $\mathcal{K}\{\mathtt{nat}\rightharpoonup\}$ and $\mathcal{L}\{\mathtt{nat}\rightharpoonup\}$:

**Completeness** If $e \mapsto^* e'$, where $e'$ val, then $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft e'$.

**Soundness** If $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft e'$, then $e \mapsto^* e'$ with $e'$ val.

Let us consider, in turn, what is involved in the proof of each part.

For completeness it is natural to consider a proof by induction on the definition of multistep transtion, which reduces the theorem to the following two results:

1. If $e$ val, then $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft e$.
2. If $e \mapsto e'$, then, for every $v$ val, if $\varepsilon \triangleright e' \mapsto^* \varepsilon \triangleleft v$, then $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft v$.

The first can be proved easily by induction on the structure of $e$. The second requires an inductive analysis of the derivation of $e \mapsto e'$, giving rise to two complications that must be accounted for in the proof. The first complication is that we cannot restrict attention to the empty stack, for if $e$ is, say, $\mathtt{ap}(e_1, e_2)$, then the first step of the machine is

$$\varepsilon \triangleright \mathtt{ap}(e_1, e_2) \mapsto \mathtt{ap}(-, e_2); \varepsilon \triangleright e_1,$$

and so we must consider evaluation of $e_1$ on a non-empty stack.

A natural generalization is to prove that if $e \mapsto e'$ and $k \triangleright e' \mapsto^* k \triangleleft v$, then $k \triangleright e \mapsto^* k \triangleleft v$. Consider again the case $e = \mathtt{ap}(e_1, e_2)$, $e' = \mathtt{ap}(e_1', e_2)$, with $e_1 \mapsto e_1'$. We are given that $k \triangleright \mathtt{ap}(e_1', e_2) \mapsto^* k \triangleleft v$, and we are to show that $k \triangleright \mathtt{ap}(e_1, e_2) \mapsto^* k \triangleleft v$. It is easy to show that the first step of the former derivation is

$$k \triangleright \mathtt{ap}(e_1', e_2) \mapsto \mathtt{ap}(-, e_2); k \triangleright e_1'.$$

We would like to apply induction to the derivation of $e_1 \mapsto e_1'$, but to do so we must have a $v_1$ such that $e_1' \mapsto^* v_1$, which is not immediately at hand.

To push the proof through, we must consider the ultimate value of each sub-expression, which is provided by the evaluation semantics described in Chapter 11. Combining the foregoing observations leads leads to the following lemma:

**Lemma 26.2.** *If $e \Downarrow v$, then for every $k$ $\mathsf{stack}$, $k \triangleright e \mapsto^* k \triangleleft v$.*

The desired result follows by the analogue of Theorem 11.1 on page 83 for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$, which states that $e \Downarrow v$ iff $e \mapsto^* v$.

For the proof of soundness, it is awkward to reason inductively about the multistep transition from $\varepsilon \triangleright e \mapsto^* \varepsilon \triangleleft v$, because the intervening steps may involve alternations of evaluation and return states. Instead we regard each $\mathcal{K}\{\mathtt{nat} \rightharpoonup\}$ machine state as encoding an expression, and show that $\mathcal{K}\{\mathtt{nat} \rightharpoonup\}$ transitions are simulated by $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$ transitions under this encoding.

Specifically, we define a judgement, $s \leftarrow\!\!\shortmid e$, stating that state $s$ "unravels to" expression $e$. It will turn out that for initial states, $s = \varepsilon \triangleright e$, and final states, $s = \varepsilon \triangleleft e$, we have $s \leftarrow\!\!\shortmid e$. Then we show that if $s \mapsto^* s'$, where $s'$ $\mathsf{final}$, $s \leftarrow\!\!\shortmid e$, and $s' \leftarrow\!\!\shortmid e'$, then $e'$ $\mathsf{val}$ and $e \mapsto^* e'$. For this it is enough to show the following two facts:

1. If $s \leftarrow\!\!\shortmid e$ and $s$ $\mathsf{final}$, then $e$ $\mathsf{val}$.

2. If $s \mapsto s'$, $s \leftarrow\!\!\shortmid e$, $s' \leftarrow\!\!\shortmid e'$, and $e' \mapsto^* v$, where $v$ $\mathsf{val}$, then $e \mapsto^* v$.

The first is quite simple, we need only observe that the unravelling of a final state is a value. For the second, it is enough to show the following lemma.

**Lemma 26.3.** *If $s \mapsto s'$, $s \leftarrow\!\!\shortmid e$, and $s' \leftarrow\!\!\shortmid e'$, then $e \mapsto^* e'$.*

The remainder of this section is devoted to the proofs of these lemmas.

### 26.3.1 Completeness

*Proof of Lemma 26.2.* The proof is by induction on an evaluation semantics for $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

Consider the evaluation rule

$$\frac{e_1 \Downarrow \mathtt{lam[}\tau_2\mathtt{]}(x.e) \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{\mathtt{ap}(e_1, e_2) \Downarrow v} \tag{26.10}$$

For an arbitrary control stack, $k$, we are to show that $k \triangleright \mathtt{ap}(e_1, e_2) \mapsto^* k \triangleleft v$. Applying each of the three inductive hypotheses in succession, inter-

leaved with steps of the abstract machine, we obtain

$$\begin{aligned} k \triangleright \mathtt{ap}(e_1, e_2) &\mapsto \mathtt{ap}(-, e_2); k \triangleright e_1 && (26.11)\\ &\mapsto^* \mathtt{ap}(-, e_2); k \triangleleft \mathtt{lam}[\tau_2](x.e) && (26.12)\\ &\mapsto \mathtt{ap}(\mathtt{lam}[\tau_2](x.e), -); k \triangleright e_2 && (26.13)\\ &\mapsto^* \mathtt{ap}(\mathtt{lam}[\tau_2](x.e), -); k \triangleleft v_2 && (26.14)\\ &\mapsto k \triangleright [v_2/x]e && (26.15)\\ &\mapsto^* k \triangleleft v. && (26.16) \end{aligned}$$

The other cases of the proof are handled similarly. □

### 26.3.2 Soundness

The judgement $s \not\mapsto e'$, where $s$ is either $k \triangleright e$ or $k \triangleleft e$, is defined in terms of the auxiliary judgement $k \bowtie e = e'$ by the following rules:

$$\frac{k \bowtie e = e'}{k \triangleright e \not\mapsto e'} \tag{26.17a}$$

$$\frac{k \bowtie e = e'}{k \triangleleft e \not\mapsto e'} \tag{26.17b}$$

In words, to unravel a state we wrap the stack around the expression. The latter relation is inductively defined by the following rules:

$$\frac{}{\varepsilon \bowtie e = e} \tag{26.18a}$$

$$\frac{k \bowtie \mathtt{s}(e) = e'}{\mathtt{s}(-); k \bowtie e = e'} \tag{26.18b}$$

$$\frac{k \bowtie \mathtt{ifz}(e_1, e_2, x.e_3) = e'}{\mathtt{ifz}(-, e_2, x.e_3); k \bowtie e_1 = e'} \tag{26.18c}$$

$$\frac{k \bowtie \mathtt{ap}(e_1, e_2) = e}{\mathtt{ap}(-, e_2); k \bowtie e_1 = e} \tag{26.18d}$$

$$\frac{k \bowtie \mathtt{ap}(e_1, e_2) = e}{\mathtt{ap}(e_1, -); k \bowtie e_2 = e} \tag{26.18e}$$

These judgements both define total functions.

**Lemma 26.4.** *The judgement $s \not\mapsto e$ has mode $(\forall, \exists!)$, and the judgement $k \bowtie e = e'$ has mode $(\forall, \forall, \exists!)$.*

That is, each state unravels to a unique expression, and the result of wrapping a stack around an expression is uniquely determined. We are therefore justified in writing $k \bowtie e$ for the unique $e'$ such that $k \bowtie e = e'$.

The following lemma is crucial. It states that unravelling preserves the transition relation.

**Lemma 26.5.** *If $e \mapsto e'$, $k \bowtie e = d$, $k \bowtie e' = d'$, then $d \mapsto d'$.*

*Proof.* The proof is by rule induction on the transition $e \mapsto e'$. The inductive cases, in which the transition rule has a premise, follow easily by induction. The base cases, in which the transition is an axiom, are proved by an inductive analysis of the stack, $k$.

For an example of an inductive case, suppose that $e = \mathtt{ap}(e_1, e_2)$, $e' = \mathtt{ap}(e_1', e_2)$, and $e_1 \mapsto e_1'$. We have $k \bowtie e = d$ and $k \bowtie e' = d'$. It follows from Rules (26.18) that $\mathtt{ap}(-, e_2); k \bowtie e_1 = d$ and $\mathtt{ap}(-, e_2); k \bowtie e_1' = d'$. So by induction $d \mapsto d'$, as desired.

For an example of a base case, suppose that $e = \mathtt{ap}(\mathtt{lam}[\tau_2](x.e), e_2)$ and $e' = [e_2/x]e$ with $e \mapsto e'$ directly. Assume that $k \bowtie e = d$ and $k \bowtie e' = d'$; we are to show that $d \mapsto d'$. We proceed by an inner induction on the structure of $k$. If $k = \varepsilon$, the result follows immediately. Consider, say, the stack $k = \mathtt{ap}(-, c_2); k'$. It follows from Rules (26.18) that $k' \bowtie \mathtt{ap}(e, c_2) = d$ and $k' \bowtie \mathtt{ap}(e', c_2) = d'$. But by the SOS rules $\mathtt{ap}(e, c_2) \mapsto \mathtt{ap}(e', c_2)$, so by the inner inductive hypothesis we have $d \mapsto d'$, as desired. □

We are now in a position to complete the proof of Lemma 26.3 on page 211.

*Proof of Lemma 26.3 on page 211.* The proof is by case analysis on the transitions of $\mathcal{K}\{\mathtt{nat} \rightharpoonup\}$. In each case after unravelling the transition will correspond to zero or one transitions of $\mathcal{L}\{\mathtt{nat} \rightharpoonup\}$.

Suppose that $s = k \triangleright \mathtt{s}(e)$ and $s' = \mathtt{s}(-) \triangleright e$. Note that $k \bowtie \mathtt{s}(e) = e'$ iff $\mathtt{s}(-); k \bowtie e = e'$, from which the result follows immediately.

Suppose that $s = \mathtt{ap}(\mathtt{lam}[\tau](x.e_1), -); k \triangleleft e_2$ and $s' = k \triangleright [e_2/x]e_1$. Let $e'$ be such that $\mathtt{ap}(\mathtt{lam}[\tau](x.e_1), -); k \bowtie e_2 = e'$ and let $e''$ be such that $k \bowtie [e_2/x]e_1 = e''$. Observe that $k \bowtie \mathtt{ap}(\mathtt{lam}[\tau](x.e_1), e_2) = e'$. The result follows from Lemma 26.5.

□

## 26.4 Exercises

# Chapter 27

# Exceptions

Exceptions effects a non-local transfer of control from the point at which the exception is *raised* to a dynamically enclosing *handler* for that exception. This transfer interrupts the normal flow of control in a program in response to unusual conditions. For example, exceptions can be used to signal an error condition, or to indicate the need for special handling in certain circumstances that arise only rarely. To be sure, one could use explicit conditionals to check for and process errors or unusual conditions, but using exceptions is often more convenient, particularly since the transfer to the handler is direct and immediate, rather than indirect via a series of explicit checks. All too often explicit checks are omitted (by design or neglect), whereas exceptions cannot be ignored.

## 27.1 Failures

To begin with let us consider a simple control mechanism, which permits the evaluation of an expression to *fail* by passing control to the nearest enclosing handler, which is said to *catch* the failure. Failures are a simplified form of exception in which no value is associated with the failure. This allows us to concentrate on the control flow aspects, and to treat the associated value separately.

The following grammar describes an extension to $\mathcal{L}\{\rightarrow\}$ to include failures:

$$\text{Expr} \quad e \quad ::= \quad \texttt{fail[}\tau\texttt{]} \mid \texttt{catch(}e_1\texttt{,}e_2\texttt{)}$$

The expression $\texttt{fail[}\tau\texttt{]}$ aborts the current evaluation. The expression $\texttt{catch(}e_1\texttt{,}e_2\texttt{)}$ evaluates $e_1$. If it terminates normally, its value is returned; if it fails, its value is the value of $e_2$.

The static semantics of failures is quite straightforward:

$$\frac{}{\Gamma \vdash \texttt{fail}[\tau] : \tau} \tag{27.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{catch}(e_1, e_2) : \tau} \tag{27.1b}$$

Observe that a failure can have any type, because it never returns to the site of the failure. Both clauses of a handler must have the same type, to allow for either possible outcome of evaluation.

The dynamic semantics of failures uses a technique called *stack unwinding*. Evaluation of a `catch` installs a handler on the control stack. Evaluation of a `fail` unwinds the control stack by popping frames until it reaches the nearest enclosing handler, to which control is passed. The handler is evaluated in the context of the surrounding control stack, so that failures within it propagate further up the stack.

This behavior is naturally specified using the abstract machine $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ from Chapter 26, because it makes the control stack explicit. We introduce a new form of state, $k \blacktriangleleft$ , which passes a failure to the stack, $k$, in search of the nearest enclosing handler. A state of the form $\varepsilon \blacktriangleleft$ is considered final, rather than stuck; it corresponds to an "uncaught failure" making its way to the topic of the stack.

The set of frames is extended with the following additional rule:

$$\frac{e_2 \ \mathsf{exp}}{\texttt{catch}(-, e_2) \ \mathsf{frame}} \tag{27.2}$$

The transition rules of $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ are extended with the following additional rules:

$$\frac{}{k \triangleright \texttt{fail}[\tau] \mapsto k \blacktriangleleft} \tag{27.3a}$$

$$\frac{}{k \triangleright \texttt{catch}(e_1, e_2) \mapsto \texttt{catch}(-, e_2); k \triangleright e_1} \tag{27.3b}$$

$$\frac{}{\texttt{catch}(-, e_2); k \triangleleft v \mapsto k \triangleleft v} \tag{27.3c}$$

$$\frac{}{\texttt{catch}(-, e_2); k \blacktriangleleft \ \mapsto k \triangleright e_2} \tag{27.3d}$$

$$\frac{(f \neq \texttt{catch}(-, e_2))}{f; k \blacktriangleleft \ \mapsto k \blacktriangleleft} \tag{27.3e}$$

Evaluating `fail[`$\tau$`]` propagates a failure up the stack. Evaluating `catch(`$e_1$`,`$e_2$`)` consists of pushing the handler onto the control stack and evaluating $e_1$. If a value is propagated to the handler, the handler is removed and the value continues to propagate upwards. If a failure is propagated to the handler, the stored expression is evaluated with the handler removed from the control stack. All other frames propagate failures.

The definition of initial state remains the same as for $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$, but we change the definition of final state to include these two forms:

$$\frac{e\ \mathsf{val}}{\varepsilon \lhd e\ \mathsf{final}} \tag{27.4a}$$

$$\frac{}{\varepsilon \blacktriangleleft\ \mathsf{final}} \tag{27.4b}$$

The first of these is as before, corresponding to a normal result with the specified value. The second is new, corresponding to an uncaught exception propagating through the entire program.

It is a straightforward exercise the extend the definition of stack typing given in Chapter 26 to account for the new forms of frame. Using this, safety can be proved by standard means. Note, however, that the meaning of the progress theorem is now significantly different: a well-typed program does not get stuck . . . but it may well result in an uncaught failure!

**Theorem 27.1** (Safety)**.** *1. If $s$ $\mathsf{ok}$ and $s \mapsto s'$, then $s'$ $\mathsf{ok}$.*

*2. If $s$ $\mathsf{ok}$, then either $s$ final or there exists $s'$ such that $s \mapsto s'$.*

## 27.2 Exceptions

Let us now consider enhancing the simple failures mechanism of the preceding section with an exception mechanism that permits a value to be associated with the failure, which is then passed to the handler as part of the control transfer. The syntax of exceptions is given by the following grammar:

$$\text{Expr}\quad e \ ::=\ \texttt{raise[}\tau\texttt{](}e\texttt{)} \mid \texttt{handle(}e_1\texttt{,}x\texttt{.}e_2\texttt{)}$$

The argument to `raise` is evaluated to determine the value passed to the handler. The expression `handle(`$e_1$`,`$x$`.`$e_2$`)` binds a variable, $x$, in the handler, $e_2$, to which the associated value of the exception is bound, should an exception be raised during the execution of $e_1$.

The dynamic semantics of exceptions is a mild generalization of that of failures given in Section 27.1 on page 215. The failure state, $k \blacktriangleleft$ , is extended to permit passing a value along with the failure, $k \blacktriangleleft e$, where $e$ val. Stack frames include these two forms:

$$\frac{}{\texttt{raise}[\tau](-)\ \mathsf{frame}} \tag{27.5a}$$

$$\frac{}{\texttt{handle}(-, x.e_2)\ \mathsf{frame}} \tag{27.5b}$$

The rules for evaluating exceptions are as follows:

$$\frac{}{k \triangleright \texttt{raise}[\tau](e) \mapsto \texttt{raise}[\tau](-); k \triangleright e} \tag{27.6a}$$

$$\frac{}{\texttt{raise}[\tau](-); k \triangleleft e \mapsto k \blacktriangleleft e} \tag{27.6b}$$

$$\frac{}{\texttt{raise}[\tau](-); k \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{27.6c}$$

$$\frac{}{k \triangleright \texttt{handle}(e_1, x.e_2) \mapsto \texttt{handle}(-, x.e_2); k \triangleright e_1} \tag{27.6d}$$

$$\frac{}{\texttt{handle}(-, x.e_2); k \triangleleft e \mapsto k \triangleleft e} \tag{27.6e}$$

$$\frac{}{\texttt{handle}(-, x.e_2); k \blacktriangleleft e \mapsto k \triangleright [e/x]e_2} \tag{27.6f}$$

$$\frac{(f \neq \texttt{handle}(-, x.e_2))}{f; k \blacktriangleleft e \mapsto k \blacktriangleleft e} \tag{27.6g}$$

The static semantics of exceptions generalizes that of failures.

$$\frac{\Gamma \vdash e : \tau_{exn}}{\Gamma \vdash \texttt{raise}[\tau](e) : \tau} \tag{27.7a}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau_{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{handle}(e_1, x.e_2) : \tau} \tag{27.7b}$$

These rules are parameterized by the type of values associated with exceptions, $\tau_{exn}$. But what should be the type $\tau_{exn}$?

The first thing to observe is that *all* exceptions should be of the same type, otherwise we cannot guarantee type safety. The reason is that a handler might be invoked by *any* raise expression occurring during the execution of the expression that it guards. If different exceptions could have different associated values, the handler could not predict (statically) what type of value to expect, and hence could not dispatch on it without violating type safety.

Since the data associated with an exception is intended to indicate the reason for the failure, it may seem reasonable to choose $\tau_{exn}$ to be a string that describes the reason for the failure. For example, one might write

```
raise "Division by zero error."
```

to signal the obvious arithmetic fault, or

```
raise "File not found."
```

to indicate failure to open a specified file. While this might be reasonable for exceptions that are not intended to be caught, it is quite unreasonable for those that may be caught by an exception handler — it would have to parse the associated string, according to some conventions, to determine what happened and how to respond! Similar criticisms apply to choosing $\tau_{exn}$ to be, say, `nat`, associating an "error number" with each form of failure, and requiring the handler to dispatch on the number. This, too, is obviously rather primitive and error-prone, and would not permit a natural means of associating other, exception-specific data with the failure.

A much more reasonable choice would be to distinguish a labelled sum type of the form

$$\tau_{exn} = \texttt{[div:unit,fnf:string,...]}.$$

Each variant of the sum specifies the type of data associated with that variant. The handler may perform a case analysis on the tag of the variant, thereby recovering the underlying data value of the appropriate type. For example,

```
try e1 ow x.case x {
  div ⟨⟩ ⇒ e_div
| fnf s ⇒ e_fnf
| ... }
```

This code closely resembles the exception mechanisms found in many languages.

A significant complication remains. The type $\tau_{exn}$ must be specified on a *per-language* basis to ensure that program fragments may be combined sensibly with one another. But having to choose a single, fixed labelled sum type to serve as the type of exceptions for all possible programs is clearly absurd! Although certain low-level exceptions, such as division by zero, might reasonably be included in any program, we expect in general that the choice of exceptions is specific to the task at hand, and ought to be chosen by the programmer. This is something of a dilemma, because we must choose $\tau_{exn}$ once for all programs written in the language, yet we also expect that programmers may declare their own exceptions.

The way out of this dilemma is to define $\tau_{exn}$ to be an *extensible* labelled sum type, rather than a *fixed* labelled sum type. An extensible sum is one that permits new tags to be created *dynamically* so that the collection of possible tags on values of the type is not fixed statically, but only at run-time. The concept of extensible sum has applications beyond their use as the type of values associated with exceptions. We will discuss this type in detail in Chapter 36.

## 27.3 Exercises

# Chapter 28

# Continuations

The semantics of many control constructs (such as exceptions and co-routines) can be expressed in terms of *reified* control stacks, a representation of a control stack as an ordinary value. This is achieved by allowing a stack to be passed as a value within a program and to be restored at a later point, *even if* control has long since returned past the point of reification. Reified control stacks of this kind are called *first-class continuations*, where the qualification "first class" stresses that they are ordinary values with an indefinite lifetime that can be passed and returned at will in a computation. First-class continuations never "expire", and it is always sensible to reinstate a continuation without compromising safety. Thus first-class continuations support unlimited "time travel" — we can go back to a previous point in the computation and then return to some point in its future, at will.

How is this achieved? The key to implementing first-class continuations is to arrange that control stacks are *persistent* data structures, just like any other data structure in ML that does not involve mutable references. By a persistent data structure we mean one for which operations on it yield a "new" version of the data structure without disturbing the old version. For example, lists in ML are persistent in the sense that if we cons an element to the front of a list we do not thereby destroy the original list, but rather yield a new list with an additional element at the front, retaining the possibility of using the old list for other purposes. In this sense persistent data structures allow time travel — we can easily switch between several versions of a data structure without regard to the temporal order in which they were created. This is in sharp contrast to more familiar *ephemeral* data structures for which operations such as insertion of an element irrevocably mutate the data structure, preventing any form of time travel.

Returning to the case in point, the standard implementation of a control stack is as an ephemeral data structure, a pointer to a region of mutable storage that is overwritten whenever we push a frame. This makes it impossible to maintain an "old" and a "new" copy of the control stack at the same time, making time travel impossible. If, however, we represent the control stack as a persistent data structure, then we can easily reify a control stack by simply binding it to a variable, and continue working. If we wish we can easily return to that control stack by referring to the variable that is bound to it. This is achieved in practice by representing the control stack as a list of frames in the heap so that the persistence of lists can be extended to control stacks. While we will not be specific about implementation strategies in this note, it should be born in mind when considering the semantics outlined below.

Why are first-class continuations useful? Fundamentally, they are representations of the control state of a computation at a given point in time. Using first-class continuations we can "checkpoint" the control state of a program, save it in a data structure, and return to it later. In fact this is precisely what is necessary to implement *threads* (concurrently executing programs) — the thread scheduler must be able to checkpoint a program and save it for later execution, perhaps after a pending event occurs or another thread yields the processor.

## 28.1 Informal Overview

We will extend $\mathcal{L}\{\rightarrow\}$ with the type `cont(`$\tau$`)` of continuations accepting values of type $\tau$. The introduction form for `cont(`$\tau$`)` is `letcc[`$\tau$`](`$x$`.`$e$`)`, which binds the *current continuation* (*i.e.*, the current control stack) to the variable $x$, and evaluates the expression $e$. The corresponding elimination form is `throw(`$e_1$`,`$e_2$`)`, which restores the value of $e_1$ to the control stack that is the value of $e_2$.[1]

To illustrate the use of these primitives, consider the problem of multiplying the first $n$ elements of an infinite sequence $q$ of natural numbers, where $q$ is represented by a function of type `nat` $\rightarrow$ `nat`. If zero occurs among the first $n$ elements, we would like to effect an "early return" with the value zero, rather than perform the remaining multiplications. This problem can be solved using exceptions (we leave this as an exercise), but

[1] Close relatives of these primitives are available in SML/NJ in the following forms: for `letcc[`$\tau$`](`$x$`.`$e$`)`, write `SMLofNJ.Cont.callcc (fn` $x$ `=>` $e$`)`, and for `throw(`$e_1$`,`$e_2$`)`, write `SMLofNJ.Cont.throw` $e_2$ $e_1$.

we will give a solution that uses continuations in order to help motivate what follows.

Here is the solution without short-cutting, written using some convenient concrete syntax conventions:

```
fun ms (q : nat -> nat, n : nat) is
  case n {
    zero ⇒ succ(zero) |
    succ (m:nat) ⇒ times (q zero) (ms (q ∘ succ, m)) }
```

The recursive call composes $q$ with the successor function to shift the sequence by one step.

Here is the version with short-cutting:

```
λ (q : nat -> nat, n : nat) in
  letcc ret : nat cont in
    let fun ms (q : nat -> nat, n : nat) be
      case n {
          zero ⇒ succ(zero) |
          succ (m:nat) ⇒
            case (q zero) {
              zero => throw zero to ret |
              succ(p:nat) => times (succ p) (ms (q ∘ succ) m) } }
     in
       ms q n
```

The `letcc` binds the return point of the function to the variable `ret` for use within the main loop of the computation. If zero is encountered, control is thrown to `ret`, effecting an early return with the value zero.

Let's look at another example: given a continuation $k$ of type $\tau$ `cont` and a function $f$ of type $\tau' \to \tau$, return a continuation $k'$ of type $\tau'$ `cont` with the following behavior: throwing a value $v'$ of type $\tau'$ to $k'$ throws the value $f(v')$ to $k$. This is called *composition of a function with a continuation*. We wish to fill in the following template:

`fun compose(f:`$\tau' \to \tau$`,k:`$\tau$ `cont):`$\tau'$ `cont = ....`

The first problem is to obtain the continuation we wish to return. The second problem is how to return it. The continuation we seek is the one in effect at the point of the ellipsis in the expression `throw` $f$`(...) to` $k$. This is the continuation that, when given a value $v'$, applies $f$ to it, and throws the result to $k$. We can seize this continuation using `letcc`, writing

```
throw f(letcc x:τ′ cont in ...) to k
```

At the point of the ellipsis the variable $x$ is bound to the continuation we wish to return. How can we return it? By using the same trick as we used for short-circuiting evaluation above! We don't want to actually throw a value to this continuation (yet), instead we wish to abort it and return it as the result. Here's the final code:

```
fun compose (f:τ′ → τ, k:τ cont):τ′ cont =
    letcc ret:τ′ cont cont in
      throw (f (letcc r in throw r to ret)) to k
```

Notice that the type of `ret` is that of a continuation-expecting continuation!

## 28.2 Semantics of Continuations

We extend the language of $\mathcal{L}\{\rightarrow\}$ expressions with these additional forms:

| | | | |
|---|---|---|---|
| Type | $\tau$ | ::= | $\texttt{cont}(\tau)$ |
| Expr | $e$ | ::= | $\texttt{letcc}[\tau](x.e) \mid \texttt{throw}(e_1, e_2) \mid \texttt{cont}(k)$ |

The expression $\texttt{cont}(k)$ is a reified control stack; they arise during evaluation, but are not available as expressions to the programmer.

The static semantics of this extension is defined by the following rules:

$$\frac{\Gamma, x : \texttt{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{letcc}[\tau](x.e) : \tau} \tag{28.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{cont}(\tau_1)}{\Gamma \vdash \texttt{throw}(e_1, e_2) : \tau'} \tag{28.1b}$$

The result type of a `throw` expression is arbitrary because it does not return to the point of the call.

The static semantics of continuation values is given by the following rule:

$$\frac{k : \tau}{\Gamma \vdash \texttt{cont}(k) : \texttt{cont}(\tau)} \tag{28.2}$$

A continuation value $\texttt{cont}(k)$ has type $\texttt{cont}(\tau)$ exactly if it is a stack accepting values of type $\tau$.

To define the dynamic semantics, we extend $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ stacks with two new forms of frame:

$$\frac{e_2 \ \mathsf{exp}}{\texttt{throw}(-, e_2) \ \mathsf{frame}} \tag{28.3a}$$

$$\frac{e_1 \text{ val}}{\texttt{throw}(e_1, -) \text{ frame}} \tag{28.3b}$$

Every reified control stack is a value:

$$\frac{k \text{ stack}}{\texttt{cont}(k) \text{ val}} \tag{28.4}$$

The transition rules for the continuation constructs are as follows:

$$\overline{k \rhd \texttt{letcc}[\tau]\,(x.e) \mapsto k \rhd [\texttt{cont}(k)/x]e} \tag{28.5a}$$

$$\overline{\texttt{throw}(v, -); k \lhd \texttt{cont}(k') \mapsto k' \lhd v} \tag{28.5b}$$

$$\overline{k \rhd \texttt{throw}(e_1, e_2) \mapsto \texttt{throw}(-, e_2); k \rhd e_1} \tag{28.5c}$$

$$\frac{e_1 \text{ val}}{\texttt{throw}(-, e_2); k \lhd e_1 \mapsto \texttt{throw}(e_1, -); k \rhd e_2} \tag{28.5d}$$

Evaluation of a `letcc` expression duplicates the control stack; evaluation of a `throw` expression destroys the current control stack.

The safety of this extension of $\mathcal{L}\{\rightharpoonup\}$ may be established by a simple extension to the safety proof for $\mathcal{K}\{\texttt{nat}\rightharpoonup\}$ given in Chapter 26.

We need only add typing rules for the two new forms of frame, which are as follows:

$$\frac{e_2 : \texttt{cont}(\tau)}{\texttt{throw}(-, e_2) : \tau \Rightarrow \tau'} \tag{28.6a}$$

$$\frac{e_1 : \tau \quad e_1 \text{ val}}{\texttt{throw}(e_1, -) : \texttt{cont}(\tau) \Rightarrow \tau'} \tag{28.6b}$$

The rest of the definitions remain as in Chapter 26.

**Lemma 28.1** (Canonical Forms)**.** *If $e$ : $\texttt{cont}(\tau)$ and $e$ val, then $e = \texttt{cont}(k)$ for some $k$ such that $k : \tau$.*

**Theorem 28.2** (Safety)**.** 1. *If $s$ ok and $s \mapsto s'$, then $s'$ ok.*

2. *If $s$ ok, then either $s$ final or there exists $s'$ such that $s \mapsto s'$.*

## 28.3 Exceptions from Continuations

The dynamic semantics of failures may be specified by a translation into a language with continuations. The general idea is to pass the current exception handler, represented as a continuation of type $\tau_{exn}$ `cont`, as an implicit argument to each expression. To raise an exception, we throw the exception value to the current handler. To install a handler we must create a continuation to serve as the new exception handler, which we then pass to the protected computation. This handler must ensure that, when invoked, it evaluates the handling code in the proper exception environment.

To make this a bit more precise, let us consider the translation of well-typed expressions from the language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ enriched with exceptions into the same language enriched with continuations. The general idea is to translate an expression $e : \tau$ involving exceptions to an expression $e' : \tau_{exn}\,\texttt{cont} \to \tau$ involving only continuations. To make this fully precise requires a further translation of types, which we will suppress in this informal discussion so as to focus attention on the central ideas.

The translation of `raise[`$\tau$`](`$e$`)` is the function

$$\lambda\,(h : \tau_{exn}\,\texttt{cont.throw}\,e\,\texttt{to}\,h).$$

In words, to raise an exception, we throw the raised value to the current exception handler.

The translation of `try`$\,e_1\,$`ow`$\,x\,$`.`$\,e_2$ is the function of type $\tau_{exn}\,\texttt{cont} \to \tau$ given by the $\lambda$-abstraction

$$\lambda\,(h : \tau_{exn}\,\texttt{cont.letcc}\,r\,\texttt{in let}\,h'\,\texttt{be}\ldots\texttt{in}\,(e_1\,(h'))), \tag{28.7}$$

where the elided portion is the following expression:

$$\texttt{letcc}\,r'\,\texttt{in}\,(\texttt{let}\,x\,\texttt{be}\,(\texttt{letcc}\,h'\,\texttt{in throw}\,h'\,\texttt{to}\,r')\,\texttt{in}\,(\texttt{throw}\,e_2\,(h)\,\texttt{to}\,r)). \tag{28.8}$$

When applied to the current exception handler, $h$, the expression (28.7) binds $r$ to the return continuation, binds $h'$ to a new exception handler, and evaluates $e_1\,(h')$. The continuation bound to $h'$ transfers control to $e_2$ with $x$ bound to the raised value, in a manner to be described shortly. The application of $e_1$ to $h'$ ensures that if $e_1$ raises an exception, it is passed to this continuation. The binding of $h'$, which makes reference to the return continuation, $r$, is given by the expression (28.8). This code works by setting up an inner return continuation, $r'$, then establishes the required continuation, which is then thrown to $r'$, and hence bound to $h'$.

The required continuation is the one which, when thrown an exception value, binds that value to the variable $x$ and evaluates $e_2$ with the *surrounding* exception handler installed. This suggests that the continuation we seek is described by the following expression template:

$$\texttt{let}\ x\ \texttt{be}\ \ldots\ \texttt{in}\ (e_2(h)), \tag{28.9}$$

which can be returned to $r'$ by writing

$$\texttt{let}\ x\ \texttt{be}\ \texttt{letcc}\ k\ \texttt{in}\ \texttt{throw}\ k\ \texttt{to}\ r'\ \texttt{in}\ (e_2(h)), \tag{28.10}$$

following the pattern we used to compose a continuation with a function. However, this is not quite right! The difficulty is that the application $e_2(h)$ must be evaluated as the *overall result* of the handler, which is accomplished by throwing its value to the outer return continuation, $r$. This leads to the expression (28.8).

## 28.4 Exercises

1. Study the short-circuit multiplication example carefully to be sure you understand why it works!

2. Attempt to solve the problem of composing a continuation with a function yourself, before reading the solution.

3. Simulate the evaluation of `compose (`$f$`,` $k$`)` on the empty stack. Observe that the control stack substituted for $x$ is

   $$\texttt{ap}(f,-);\texttt{throw}(-,k);\varepsilon$$

   This stack is returned from `compose`. Next, simulate the behavior of throwing a value $v'$ to this continuation. Observe that the stack is reinstated and that $v'$ is passed to it.

4. Verify the type preservation theorem for the exception translation.

# Part X

# Propositions and Types

# Chapter 29

# The Curry-Howard Isomorphism

The *Curry-Howard Isomorphism* is a central organzing principle of type theory. Roughly speaking, the Curry-Howard Isomorphism states that there is a correspondence between *propositions* and *types* such that *proofs* correspond to *programs*. To each proposition, $\phi$, there is an associated type, $\tau$, such that to each proof $p$ of $\phi$, there is a corresponding expression $e$ of type $\tau$. Among other things, this correspondence tells us that *proofs have computational content* and that *programs are a form of proof*. It also suggests that programming language features may be expected to give rise to concepts of logic, and conversely that concepts from logic give rise to programming language features. It is a remarkable fact that this correspondence, which began as a rather modest observation about types and logics, has developed into a central principle of language design whose implications are still being explored.

This informal discussion leaves open what we mean by proposition and proof. The original isomorphism observed by Curry and Howard pertains to a particular branch of logic called *constructive logic*, of which we will have more to say in the next section. However, the observation has since been extended to an impressive array of logics, all of which are, by virtue of the correspondence, "constructive", but which extend the interpretation to richer notions of proposition and proof. Thus one might say that there are *many* Curry-Howard Isomorphisms, of which the original is but one!

We will focus our attention on constructive propositional logic, which involves a minimum of technical machinery to motivate and explain. We will concentrate on the "big picture", and make only glancing reference to

the considerable technical details involved in fully working out the correspondence between propositions and types.

## 29.1 Constructive Logic

### 29.1.1 Constructive Semantics

Constructive logic is concerned with two judgement forms, $\phi$ prop, stating that $\phi$ expresses a proposition, and $\phi$ true, stating that $\phi$ is a true proposition. In constructive logic a proposition is a *specification* describing a *problem to be solved*. The *solution* to the problem posed by a proposition is a *proof*. If a proposition has a proof (*i.e.*, it specifies a soluble problem), then the proposition is said to be *true*. The characteristic feature of constructive logic is that *there is no other criterion of truth than the existence of a proof.*

In a contructive setting the notion of falsity of a proposition is not primitive. Rather, to say that a proposition is false is simply to say that the assumption that it is true (*i.e.*, that it has a proof) is contradictory. In other words, for a proposition to be false, constructively, means that there is a *refutation* of it, which consists of a *proof* that assuming it to be true is contradictory. In this sense constructive logic is a logic of *positive*, or *affirmative*, *information* — we must have explicit evidence in the form of a proof in order to affirm the truth or falsity of a proposition.

One consequence is that a given proposition need not be either true or false! While at first this might seem absurd (what else could it be, green?), a moment's reflection on the semantics of propositions reveals that this consequence is quite natural. There are, and always will be, unsolved problems that can be posed as propositions. For a problem to be unsolved means that we are not in possession of a proof of it, nor do we have a refutation of it. Therefore, in an affirmative sense, we cannot say that the proposition is either true or false! As an example, the famous $P \stackrel{?}{=} NP$ problem has neither a proof nor a refutation at the time of this writing, so we cannot at present affirm or deny its truth.

A proposition, $\phi$, for which we possess either a proof or a refutation of it is said to be *decidable*. Any proposition for which we have either a proof or a refutation is, of course, decidable, because we have already "decided" it by virtue of having that information! But we can also make general statements about decidability of propositions. For example, if $\phi$ expresses an inequality between natural numbers, then $\phi$ is decidable, because we can always work out, for given natural numbers $m$ and $n$, whether $m \leq n$ or

$m \not\leq n$ — we can either prove or refute the given inequality. Once we step outside the realm of such immediately checkable conditions, it is not clear whether a given proposition has a proof or a refutation. It's a matter of rolling up one's sleeves and doing some work! And there's no guarantee of success! Life's hard, but we muddle through somehow.

The judgements $\phi$ prop and $\phi$ true are basic, or *categorical*, judgements. These are the building blocks of reason, but they are rarely of interest by themselves. Rather, we are interested in the more general case of the *hypothetical judgement*, or *consequence relation*, of the form

$$\phi_1 \text{ true}, \ldots, \phi_n \text{ true} \vdash \phi \text{ true}.$$

This judgement expresses that the proposition $\phi$ is true (*i.e.*, has a proof), *under the assumptions* that each of $\phi_1, \ldots, \phi_n$ are also true (*i.e.*, have proofs). Of course, when $n = 0$ this is just the same as the categorical judgement $\phi$ true. We let $\Gamma$ range over finite sets of assumptions.

The hypothetical judgement satisfies the following *structural properties*, which characterize what we mean by reasoning under hypotheses:

$$\frac{}{\Gamma, \phi \text{ true} \vdash \phi \text{ true}} \tag{29.1a}$$

$$\frac{\Gamma \vdash \phi \text{ true} \quad \Gamma, \phi \text{ true} \vdash \psi \text{ true}}{\Gamma \vdash \psi \text{ true}} \tag{29.1b}$$

$$\frac{\Gamma \vdash \psi \text{ true}}{\Gamma, \phi \text{ true} \vdash \psi \text{ true}} \tag{29.1c}$$

$$\frac{\Gamma, \phi \text{ true}, \phi \text{ true} \vdash \theta \text{ true}}{\Gamma, \phi \text{ true} \vdash \theta \text{ true}} \tag{29.1d}$$

$$\frac{\Gamma, \psi \text{ true}, \phi \text{ true}, \Gamma' \vdash \theta \text{ true}}{\Gamma, \phi \text{ true}, \psi \text{ true}, \Gamma' \vdash \theta \text{ true}} \tag{29.1e}$$

The last two rules are implicit in that we regard $\Gamma$ as a *set* of hypotheses, so that two "copies" are as good as one, and the order of hypotheses does not matter.

### 29.1.2 Propositional Logic

The connectives of propositional logic (truth, falsehood, conjunction, disjunction, implication, and negation) are given meaning by rules that determine (a) what constitutes a "direct" proof of a proposition formed from a

given connective, and (b) how to exploit the existence of such a proof in an "indirect" proof of another proposition. These are called the *introduction* and *elimination* rules for the connective. The principle of *conservation of proof* states that these rules are inverse to one another — the elimination rule cannot extract more information (in the form of a proof) than was put into it by the introduction rule, and the introduction rules can be used to reconstruct a proof from the information extracted from it by the elimination rules.

The abstract syntax of propositional logic is given by the following rules for deriving judgements of the form $\phi$ prop.

$$\frac{}{\texttt{true prop}} \tag{29.2a}$$

$$\frac{}{\texttt{false prop}} \tag{29.2b}$$

$$\frac{\phi \text{ prop} \quad \psi \text{ prop}}{\texttt{and}(\phi,\psi) \text{ prop}} \tag{29.2c}$$

$$\frac{\phi \text{ prop} \quad \psi \text{ prop}}{\texttt{imp}(\phi,\psi) \text{ prop}} \tag{29.2d}$$

$$\frac{\phi \text{ prop} \quad \psi \text{ prop}}{\texttt{or}(\phi,\psi) \text{ prop}} \tag{29.2e}$$

The following table summarizes the concrete syntax of propositions:

| *Abstract* | *Concrete* |
|---|---|
| `true` | $\top$ |
| `false` | $\bot$ |
| $\texttt{and}(\phi,\psi)$ | $\phi \wedge \psi$ |
| $\texttt{imp}(\phi,\psi)$ | $\phi \supset \psi$ |
| $\texttt{or}(\phi,\psi)$ | $\phi \vee \psi$ |

**Truth** Our first proposition is trivially true. No information goes into proving it, and so no information can be obtained from it.

$$\frac{}{\Gamma \vdash \top \text{ true}} \tag{29.3a}$$

*(no elimination rule)*

(29.3b)

**Conjunction** Conjunction expresses the truth of both of its conjuncts.

$$\frac{\Gamma \vdash \phi \ \mathsf{true} \quad \Gamma \vdash \psi \ \mathsf{true}}{\Gamma \vdash \phi \wedge \psi \ \mathsf{true}} \tag{29.4a}$$

$$\frac{\Gamma \vdash \phi \wedge \psi \ \mathsf{true}}{\Gamma \vdash \phi \ \mathsf{true}} \tag{29.4b}$$

$$\frac{\Gamma \vdash \phi \wedge \psi \ \mathsf{true}}{\Gamma \vdash \psi \ \mathsf{true}} \tag{29.4c}$$

**Implication** Implication states the truth of a proposition under an assumption.

$$\frac{\Gamma, \phi \ \mathsf{true} \vdash \psi \ \mathsf{true}}{\Gamma \vdash \phi \supset \psi \ \mathsf{true}} \tag{29.5a}$$

$$\frac{\Gamma \vdash \phi \supset \psi \ \mathsf{true} \quad \Gamma \vdash \phi \ \mathsf{true}}{\Gamma \vdash \psi \ \mathsf{true}} \tag{29.5b}$$

**Falsehood** Falsehood expresses the trivially false (refutable) proposition.

*(no introduction rule)*

(29.6a)

$$\frac{\Gamma \vdash \bot \ \mathsf{true}}{\Gamma \vdash \phi \ \mathsf{true}} \tag{29.6b}$$

**Disjunction** Disjunction expresses the truth of either (or both) of two propositions.

$$\frac{\Gamma \vdash \phi \ \mathsf{true}}{\Gamma \vdash \phi \vee \psi \ \mathsf{true}} \tag{29.7a}$$

$$\frac{\Gamma \vdash \psi \ \mathsf{true}}{\Gamma \vdash \phi \vee \psi \ \mathsf{true}} \tag{29.7b}$$

$$\frac{\Gamma \vdash \phi \vee \psi \ \mathsf{true} \quad \Gamma, \phi \ \mathsf{true} \vdash \theta \ \mathsf{true} \quad \Gamma, \psi \ \mathsf{true} \vdash \theta \ \mathsf{true}}{\Gamma \vdash \theta \ \mathsf{true}} \tag{29.7c}$$

### 29.1.3 Explicit Proofs

The key to the Curry-Howard Isomorphism is to make explict the forms of proof. The categorical judgement $\phi$ true, which states that $\phi$ has a proof, is replaced by the judgement $p : \phi$, stating that $p$ is a proof of $\phi$. The hypothetical judgement is modified correspondingly, with variables standing for the presumed, but unknown, proofs:

$$x_1 : \phi_1, \ldots, x_n : \phi_n \vdash p : \phi.$$

We again let $\Gamma$ range over such hypothesis lists, subject to the restriction that no variable occurs more than once.

The rules of constructive propositional logic may be restated using proof terms as follows.

$$\frac{}{\Gamma \vdash \texttt{true-i} : \top} \tag{29.8a}$$

$$\frac{\Gamma \vdash p : \phi \quad \Gamma \vdash q : \psi}{\Gamma \vdash \texttt{and-i}(p, q) : \phi \wedge \psi} \tag{29.8b}$$

$$\frac{\Gamma \vdash p : \phi \wedge \psi}{\Gamma \vdash \texttt{and-e-l}(p) : \phi} \tag{29.8c}$$

$$\frac{\Gamma \vdash p : \phi \wedge \psi}{\Gamma \vdash \texttt{and-e-r}(p) : \psi} \tag{29.8d}$$

$$\frac{\Gamma, x : \phi \vdash p : \psi}{\Gamma \vdash \texttt{imp-i}[\phi](x.p) : \phi \supset \psi} \tag{29.8e}$$

$$\frac{\Gamma \vdash p : \phi \supset \psi \quad \Gamma \vdash q : \phi}{\Gamma \vdash \texttt{imp-e}(p, q) : \psi} \tag{29.8f}$$

$$\frac{\Gamma \vdash p : \bot}{\Gamma \vdash \texttt{false-e}[\phi](p) : \phi} \tag{29.8g}$$

$$\frac{\Gamma \vdash p : \phi}{\Gamma \vdash \texttt{or-i-l}[\psi](p) : \phi \vee \psi} \tag{29.8h}$$

$$\frac{\Gamma \vdash p : \psi}{\Gamma \vdash \texttt{or-i-r}[\phi](p) : \phi \vee \psi} \tag{29.8i}$$

$$\frac{\Gamma \vdash p : \phi \vee \psi \quad \Gamma, x : \phi \vdash q : \theta \quad \Gamma, y : \psi \vdash r : \theta}{\Gamma \vdash \texttt{or-e}[\phi, \psi](p, x.q, y.r) : \theta} \tag{29.8j}$$

## 29.2 Propositions as Types

The Curry-Howard Isomorphism amounts to the observation that there is a close correspondence between propositions and their proofs, on the one hand, and types and their elements, on the other. The following chart summarizes the correspondence between propositions, $\phi$, and types, $\phi^*$:

| *Proposition* | *Type* |
|---|---|
| $\top$ | `unit` |
| $\bot$ | `void` |
| $\phi \wedge \psi$ | $\phi^* \times \psi^*$ |
| $\phi \supset \psi$ | $\phi^* \rightarrow \psi^*$ |
| $\phi \vee \psi$ | $\phi^* + \psi^*$ |

The correspondence extends to proofs and programs as well:

| *Proof* | *Program* |
|---|---|
| `true-i` | `triv` |
| `false-e[`$\phi$`](`$p$`)` | `abort[`$\phi^*$`](`$p^*$`)` |
| `and-i(`$p$`,`$q$`)` | `pair(`$p^*$`,`$q^*$`)` |
| `and-e-l(`$p$`)` | `fst(`$p^*$`)` |
| `and-e-r(`$p$`)` | `snd(`$p^*$`)` |
| `imp-i[`$\phi$`](`$x$`.`$p$`)` | `lam[`$\phi^*$`](`$x$`.`$p^*$`)` |
| `imp-e(`$p$`,`$q$`)` | `ap(`$p^*$`,`$q^*$`)` |
| `or-i-l[`$\psi$`](`$p$`)` | `inl[`$\psi^*$`](`$p^*$`)` |
| `or-i-r[`$\phi$`](`$p$`)` | `inr[`$\phi^*$`](`$p^*$`)` |
| `or-e[`$\phi$`,`$\psi$`](`$p$`,`$x$`.`$q$`,`$y$`.`$r$`)` | `case[`$\phi^*$`,`$\psi^*$`](`$p^*$`,`$x$`.`$q^*$`,`$y$`.`$r^*$`)` |

The translations above preserve and reflect formation and membership when viewed as a translation into a typed language with unit, product, void, sum, and function types.

**Theorem 29.1** (Curry-Howard Isomorphism)**.** 1. *If* $\phi$ prop, *then* $\phi^*$ type

2. *If* $\Gamma \vdash p : \phi$, *then* $\Gamma^* \vdash p^* : \phi^*$.

The preceding theorem establishes a *static* correspondence between propositions and types and their associated proofs and programs. It also extends to a *dynamic* correspondence, in which we see that the execution behavior of programs arises from the cancellation of elimination and introduction

rules in the following manner:

$$\begin{array}{rcl}
\texttt{and-e-l(and-i(}p\texttt{,}q\texttt{))} & \mapsto & p \\
\texttt{and-e-r(and-i(}p\texttt{,}q\texttt{))} & \mapsto & q \\
\texttt{imp-e(imp-i[}\phi\texttt{](}x\texttt{.}q\texttt{),}p\texttt{)} & \mapsto & [p/x]q \\
\texttt{or-e[}\phi\texttt{,}\psi\texttt{](or-i-l[}\psi\texttt{](}p\texttt{),}x\texttt{.}q\texttt{,}y\texttt{.}r\texttt{)} & \mapsto & [p/x]q \\
\texttt{or-e[}\phi\texttt{,}\psi\texttt{](or-i-r[}\phi\texttt{](}p\texttt{),}x\texttt{.}q\texttt{,}y\texttt{.}r\texttt{)} & \mapsto & [p/y]r
\end{array}$$

These are precisely the primitive instructions associated with the programs corresponding to these proofs! Indeed, these rules may be understood as the codification of the *computational content* of proofs — the precise sense in which proofs in propositional logic correspond, both statically and dynamically, to programs.

The correspondence given here does not extend to general recursion, which would correspond to admitting a circular proof, one whose justification relies on its own presumed truth. Unsurprisingly, permitting circular proofs renders the logic inconsistent—one can derive a "proof" of any proposition simply by appealing to itself! However, this does not mean that there is no logical account of general recursion. Rather, it simply says that self-reference cannot be permitted as evidence for the *truth* of a proposition. But one could well imagine using self-reference in connection with a relaxed notion of truth corresponding to the isolation of effects in a monad in a programming language.

## 29.3 Exercises

# Chapter 30

# Classical Proofs and Control Operators

In Chapter 29 we saw that constructive logic is a logic of positive information in that the meaning of the judgement $\phi$ true is that there exists a proof of $\phi$. A refutation of a proposition $\phi$ consists of a proof of the hypothetical judgement $\phi$ true $\vdash \bot$ true, asserting that the assumption of $\phi$ leads to a proof of logical falsehood (*i.e.*, a contradiction). Since there are propositions, $\phi$, for which we possess neither a proof nor a refutation, we cannot assert, in general, $\phi \vee \neg\phi$ true.

By contrast classical logic (the one we all learned in school) maintains a complete symmetry between truth and falsehood — that which is not true is false and that which is not false is true. Obviously such an interpretation conflicts with the constructive interpretation, for lack of a proof of a proposition is not a refutation, nor is lack of a refutation a proof.[1] In this sense classical logic is a logic of perfect information, in which all mathematical problems have been resolved, and for each one it is clear whether it is true or false. One might consider this "god's view" of mathematics, in constrast to the "mortal's view" we are stuck with.

Despite this absolutism, classical logic nevertheless has computational content, *albeit* in a somewhat attenuated form compared to constructive logic. Whereas in constructive logic truth is identified with the existence of certain positive information, in classical logic it is identified with the absence of a refutation, a much weaker criterion. Dually, falsehood is identified with the absence of a proof, which is also much weaker than possession

[1] Or, in the words of the brilliant military strategist Donald von Rumsfeld, the absence of evidence is not evidence of absence.

of a refutation. This weaker interpretation is responsible for the pleasing symmetries of classical logic. The drawback is that in classical logic propositions means much less than they do in constructive logic. For example, in classical logic the proposition $\phi \vee \neg\phi$ does not state that we have either a proof of $\phi$ or a refutation of it, rather just that it is impossible that we have both a proof of it and a refutation of it.

## 30.1 Classical Logic

Classical logic is concerned with three categorical judgement forms:

1. $\phi$ true, stating that proposition $\phi$ is true;
2. $\phi$ false, stating that proposition $\phi$ is false;
3. #, stating that a contradiction has been derived.

We will consider hypothetical judgements in which hypotheses have either of the first two forms; we will have no need of a hypothesis of the third form. Up to permutation, then, hypothetical judgements have the form

$$\phi_1 \text{ false}, \ldots, \phi_m \text{ false}; \psi_1 \text{ true}, \ldots, \psi_n \text{ true} \vdash J,$$

where $J$ is any of the three categorical judgement forms.

Rather than axiomatize classical logic directly in terms of these judgement forms, we will instead give an axiomatization in which proof terms are made explicit at the outset. The proof-explicit form of the three categorical judgements of classical logic are as follows:

1. $p : \phi$, stating that $p$ is a proof of $\phi$;
2. $k \div \phi$, stating that $k$ is a refutation of $\phi$;
3. $k \# p$, stating that $k$ and $p$ are contradictory.

We will consider hypothetical judgements of the form (up to permutation of hypotheses)

$$\underbrace{u_1 \div \phi_1, \ldots, u_m \div \phi_m}_{\Delta}; \underbrace{x_1 : \psi_1, \ldots, x_n : \psi_n}_{\Gamma} \vdash J,$$

where $J$ is any of the three categorical judgements in explicit form.

### Statics

A contradiction arises from the conflict between a proof and a refutation:

$$\frac{\Delta;\Gamma \vdash k \div \phi \quad \Delta;\Gamma \vdash p : \phi}{\Delta;\Gamma \vdash k \# p} \tag{30.1a}$$

The reflexivity rules capture the meaning of hypotheses:

$$\frac{}{\Delta, u \div \phi;\Gamma \vdash u \div \phi} \tag{30.1b}$$

$$\frac{}{\Delta;\Gamma, x : \psi \vdash x : \phi} \tag{30.1c}$$

Truth and falsity are complementary:

$$\frac{\Delta, u \div \phi;\Gamma \vdash k \# p}{\Delta;\Gamma \vdash \mathtt{ccr}(u \div \phi.k \# p) : \phi} \tag{30.1d}$$

$$\frac{\Delta;\Gamma, x : \phi \vdash k \# p}{\Delta;\Gamma \vdash \mathtt{ccp}(x : \phi.k \# p) \div \phi} \tag{30.1e}$$

In both of these rules the entire contradiction, $k \# p$, lies within the scope of the abstractor!

The rules for the connectives are organized as introductory rules for truth and for falsity, the latter playing the role of eliminatory rules in constructive logic.

$$\frac{}{\Delta;\Gamma \vdash \langle\rangle : \top} \tag{30.1f}$$

$$\frac{}{\Delta;\Gamma \vdash \mathtt{abort} \div \bot} \tag{30.1g}$$

$$\frac{\Delta;\Gamma \vdash p : \phi \quad \Delta;\Gamma \vdash q : \psi}{\Delta;\Gamma \vdash \langle p, q\rangle : \phi \wedge \psi} \tag{30.1h}$$

$$\frac{\Delta;\Gamma \vdash k \div \phi \wedge \psi}{\Delta;\Gamma \vdash \mathtt{fst};k \div \phi} \tag{30.1i}$$

$$\frac{\Delta;\Gamma \vdash k \div \phi \wedge \psi}{\Delta;\Gamma \vdash \mathtt{snd};k \div \psi} \tag{30.1j}$$

$$\frac{\Delta;\Gamma, x : \phi \vdash p : \psi}{\Delta;\Gamma \vdash \lambda(x : \phi.\, p) : \phi \supset \psi} \tag{30.1k}$$

$$\frac{\Delta;\Gamma \vdash p:\phi \quad \Delta;\Gamma \vdash k \div \psi}{\Delta;\Gamma \vdash \mathtt{app}(p);k \div \phi \supset \psi} \tag{30.1l}$$

$$\frac{\Delta;\Gamma \vdash p:\phi}{\Delta;\Gamma \vdash \mathtt{inl}_{\psi}(p):\phi \vee \psi} \tag{30.1m}$$

$$\frac{\Delta;\Gamma \vdash p:\psi}{\Delta;\Gamma \vdash \mathtt{inr}_{\phi}(p):\phi \vee \psi} \tag{30.1n}$$

$$\frac{\Delta;\Gamma \vdash k \div \phi \quad \Delta;\Gamma \vdash l \div \psi}{\Delta;\Gamma \vdash \mathtt{case}(k,l) \div \phi \vee \psi} \tag{30.1o}$$

$$\frac{\Delta;\Gamma \vdash k \div \phi}{\Delta;\Gamma \vdash \mathtt{not}(k):\neg\phi} \tag{30.1p}$$

$$\frac{\Delta;\Gamma \vdash p:\phi}{\Delta;\Gamma \vdash \mathtt{not}(p) \div \neg\phi} \tag{30.1q}$$

## Dynamics

The dynamic semantics of classical logic may be described as a process of *conflict resolution*. The state of the abstract machine is a contradiction, $k \# p$, between a refutation, $k$, and a proof, $p$, of the same proposition. Execution consists of "simplifying" the conflict based on the form of $k$ and $p$. This process is formalized by an inductive definition of a transition relation between contradictory states.

Here are the rules for each of the logical connectives, which all have the form of resolving a conflict between a proof and a refutation of a proposition formed with that connective.

$$\mathtt{fst};k \# \langle p,q\rangle \mapsto k \# p \tag{30.2a}$$

$$\mathtt{snd};k \# \langle p,q\rangle \mapsto k \# q \tag{30.2b}$$

$$\mathtt{case}(k,l) \# \mathtt{inl}_{\psi}(p) \mapsto k \# p \tag{30.2c}$$

$$\mathtt{case}(k,l) \# \mathtt{inr}_{\phi}(q) \mapsto l \# q \tag{30.2d}$$

$$\mathtt{app}(p);k \# \lambda(x:\phi.q) \mapsto k \# [p/x]q \tag{30.2e}$$

$$\mathtt{not}(p) \# \mathtt{not}(k) \mapsto k \# p \tag{30.2f}$$

The symmetry of the transition rule for negation is particularly elegant.

Here are the rules for the generic primitives relating truth and falsity.

$$\mathtt{ccp}(x:\phi.k \mathbin{\#} p) \mathbin{\#} q \mapsto [q/x]k \mathbin{\#} [q/x]p \tag{30.2g}$$

$$k \mathbin{\#} \mathtt{ccr}(u \div \phi.l \mathbin{\#} p) \mapsto [k/u]l \mathbin{\#} [k/u]p \tag{30.2h}$$

These rules explain the terminology: "ccp" means "call with current proof", and "ccr" means "call with current refutation". The former is a refutation that binds a variable to the current proof and installs the corresponding instance of its constituent state as the current state. The latter is a proof that binds a variable to the current refutation and installs the corresponding instance of its constituent state as the current state.

It is important to observe that the rules (30.2g) to (30.2h) overlap in the sense that there are two possible transitions for a state of the form

$$\mathtt{ccp}(x:\phi.k \mathbin{\#} p) \mathbin{\#} \mathtt{ccr}(u \div \phi.l \mathbin{\#} q).$$

This state may transition either to the state

$$[r/x]k \mathbin{\#} [r/x]p,$$

where $r$ is $\mathtt{ccr}(u \div \phi.l \mathbin{\#} q)$, or to the state

$$[m/u]l \mathbin{\#} [m/u]q,$$

where $m$ is $\mathtt{ccp}(x:\phi.k \mathbin{\#} p)$, and these are not equivalent.

There are two possible attitudes about this. One is to simply accept that classical logic has a non-deterministic dynamic semantics, and leave it at that. But this means that it is difficult to predict the outcome of a computation, since it could be radically different in the case of the overlapping state just described. The alternative is to impose an arbitrary priority ordering among the two cases, either preferring the first transition to the second, or *vice versa*. Preferring the first corresponds, very roughly, to a "lazy" semantics for proofs, because we pass the unevaluated proof, $r$, to the refutation on the left, which is thereby activated. Preferring the second corresponds to an "eager" semantics for proofs, in which we pass the unevaluated refutation, $m$, to the proof, which is thereby activated. Dually, these choices correspond to an "eager" semantics for refutations in the first case, and a "lazy" one for the second. Take your pick.

The final issue is the initial state: how is computation to be started? Or, equivalently, when is it finished? The difficulty is that we need both a

proof and a refutation of the same proposition! While this can easily come up in the "middle" of a proof, it would be impossible to have a finished proof and a finished refutation of the same proposition! The solution for an eager interpretation of proofs (and, correspondingly, a lazy interpretation of refutations) is simply to postulate an initial (or final, depending on your point of view) refutation, `halt`, and to deem a state of the form `halt` # $p$ to be initial, and also final, provided that $p$ is not a "ccr" instruction. The solution for a lazy interpretation of proofs (and an eager interpretation of refutations) is dual, taking $k$ # `halt` as initial, and also final, provided that $k$ is not a "ccp" instruction.

## 30.2 Exercises

# The Wave Front

*Halmos describes the process of mathematical writing as a "spiral" — writing (or, maddeningly, revising) Chapter n demands revision of Chapters* 1 *through* $n - 1$. *The currently propagating wave front of revision has reached this point in the text. From here forward one may expect more discontinuities, discrepancies, incongruities, and errors than, I hope, will be found from here backward.*

# Part XI

# Subtyping

# Chapter 31

# Subtyping

Many languages include a notion of subtyping, which is a relation between types stating, approximately, that a value of the subtype may also be regarded as a value of the supertype. Though intuitively appealing, this formulation is misleading, or at any rate underspecified, because it does not make clear what is meant by regarding a subtype value as a supertype value. In this chapter we formulate a general framework for subtyping, and consider several varieties of subtyping.

## 31.1 Subtyping

A *subtype* relation is a pre-order (reflexive and transitive relation) on types that validates the *subsumption principle*:

> if $\sigma$ is a subtype of $\tau$, then a value of type $\sigma$ may be provided whenever a value of type $\tau$ is required.

The subsumption principle provides a clear criterion for determining whether one type may be considered to be a subtype of another. A requirement for a value of type $\tau$ is induced by the use of an elimination form whose principal argument is of type $\tau$. By the subsumption principle, for $\sigma$ to be a subtype of $\tau$ means that any such requirement for a value of type $\tau$ may be met by supplying a value of type $\sigma$ instead. Roughly speaking, this may be achieved by either showing that the elimination form is well-behaved when given a value of the subtype, or, more generally, that the value of the subtype may be *coerced*, or transformed, into a value of the supertype before passing it to the elimination form, without affecting the intended meaning of the elimination construct.

The coercion interpretation is the more general, because we may always consider the possibility of the null coercion that does nothing whatsoever to its argument. In typical cases, however, the coercion will transform a value of the subtype into a value of the supertype in some non-trivial manner in order to prepare it to be acted on by the elimination form. Whether a coercion preserves the intended meaning of the elimination construct is usually intuitively clear; we will not attempt to formalize this concept, but rather simply argue the plausibility based on informal considerations.

Summarizing, to determine whether $\sigma$ may be regarded as a subtype of $\tau$, we must consider all possible elimination forms associated with the type $\tau$ and ensure that there is a suitable coercion available to permit its action on a value of type $\sigma$ instead. If this requirement cannot be met for some elimination form, then the proposed subtyping cannot be safely permitted. Surprisingly, this simple criterion is often ignored.

The *subtyping judgement*, $\sigma <: \tau$, states that the type $\sigma$ is a subtype of the type $\tau$. This judgement is inductively defined by a set of rules that may be divided into two general categories: *subtyping axioms*, stating primitive subtyping relationships, and *variance rules*, determining how type constructors interact with subtyping. In addition we tacitly include the following two *closure rules*, which ensure that subtyping is reflexive and transitive:

$$\frac{}{\tau <: \tau} \tag{31.1a}$$

$$\frac{\rho <: \sigma \quad \sigma <: \tau}{\rho <: \tau} \tag{31.1b}$$

The examples to follow will illustrate particular forms of subtyping.

## 31.2 Subsumption

The point of a subtyping relation is to enlarge the set of well-typed programs. This is achieved by incorporating the principle of subsumption into the typing rules for a language. This may be achieved in one of two ways. An an *implicit* subtyping system we include the rule of *implicit subsumption*, which permits an expression of the subtype to be silently regarded as an expression of the supertype.

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau} \tag{31.2a}$$

This rule is *not* syntax-directed, because it may be applied to any expression of any form at any time.

An *explicit* subtyping system includes the rule of *explicit subsumption*, which permits an expression of the subtype to be explicitly *cast*, or *coerced*, into a value of the supertype.

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash (\sigma <: \tau)\, e : \tau} \tag{31.2b}$$

The rules remain syntax-directed, at the expense of requiring explicit indications of the use of subtyping.

The relationship between the implicit and explicit formulations may be seen as a problem of type inference. Implicit subsumption provides the convenience of not requiring the uses of subtyping to be explicitly indicated. Rather the type checker must determine when to insert subtyping checks in such a way that no programs are ruled out that have a typing derivation using the implicit subsumption rule. According to this view, it is a simple matter to instrument the type checker so that it *translates* the implicit form of subtyping into the explicit form by inserting coercions wherever subtyping relationships are required. We may therefore consider the explicit form of subtyping to be the primary one, with implicit subtyping arising as a form of type inference.

Another reason to treat the explicit form as primary is that it provides the most general foundation for the dynamic semantics. Since coercions may, but need not, transform their arguments, the dynamic semantics must be sensitive to the use of subtyping in a program. When we specify a subtyping relation, we must specify the dynamic semantics of the associated coercion, subject to the requirement that the result of the coercion applied to a value of the subtype be an expression of the supertype (in the explicit form of the language). We tacitly include the following rule for evaluating coercions:

$$\frac{e \mapsto e'}{(\sigma <: \tau)\, e \mapsto (\sigma <: \tau)\, e'} \tag{31.3a}$$

Further, we require the following *safety requirement* for coercions: if $e$ val and $(\sigma <: \tau)\, e \mapsto e'$, then $e'$ is of type $\tau$ in the explicit system.

Finally, we include the following rule to eliminate identity coercions:

$$\frac{e \text{ val}}{(\sigma <: \sigma)\, e \mapsto e} \tag{31.3b}$$

See Section 31.5 on page 264 for further discussion pertinent to this rule.

In the sequel we will work with variants of $\mathcal{L}\{\rightarrow <:\}$, the extension of $\mathcal{L}\{\rightarrow\}$ with subtyping. Assuming that the dynamic semantics of coercion satisfies the safety requirement for coercions, we may prove type safety for each such extension.

## 31.3 Varieties of Subtyping

In this section we will explore several different forms of subtyping in the context of extensions of $\mathcal{L}\{\rightarrow\}$.

### 31.3.1 Numeric Subtyping

In informal mathematics we treat the set of integers, $\mathbb{Z}$, as a subset of the set of real numbers, $\mathbb{R}$, on the grounds that there is an inclusion $\iota : \mathbb{Z} \hookrightarrow \mathbb{R}$ that treats every integer as a real number. Similarly, we regard $\mathbb{R}$ as a subset of $\mathbb{C}$, and so forth.

It is tempting, therefore, to consider a similar interpretation in a programming context, by treating `int` as a subtype of `float`, where `int` is the type of machine integers and `float` is the type of IEEE floating point numbers. This may be achieved by postulating that there is a coercion operation converting each integer, $n$, to the corresponding floating point number, $x_n$:

$$\frac{(n \in \mathbb{Z})}{\texttt{(int<:float)}\ n \mapsto x_n}\ . \tag{31.4}$$

But this interpretation is unrealistic, because not every machine integer has an exact representation as a floating point number. Some representations overflow the capacity of floating point representations, and others have only inexact representations due to the limitations of floating point.

An alternative in the same spirit is to consider the type `int` to consist of *arbitrary* integers (*i.e.*, with no fixed bound imposed by machine word size), and to replace `float` by the type of *exact real numbers*, `real`, represented in any number of ways, say by Cauchy sequences of rationals. While this is in some ways more theoretically satisfying, it is quite unrealistic because computation on the exact real numbers is rather inefficient in practice when compared to floating point arithmetic.

On the other hand, a related, better-behaved, form of subtyping arises when considering multiple "sizes" of machine integers, say `int` and `long`, representing single-word and double-word integers. In this case there is no difficulty postulating `int` $<:$ `long`, since every single-word integer has an exact correlate as a double-word integer (by sign extension).

### 31.3.2 Function Subtyping

Suppose, for the sake of discussion, that $\texttt{int} <: \texttt{float}$.[1] What subtyping relationships, if any, should hold among the following four types?

1. $\texttt{int} \to \texttt{int}$
2. $\texttt{int} \to \texttt{float}$
3. $\texttt{float} \to \texttt{int}$
4. $\texttt{float} \to \texttt{float}$

To determine the answer, keep in mind the subsumption principle, which says that a value of the subtype should be usable in a supertype context.

Suppose $f : \texttt{int} \to \texttt{int}$. If we apply $f$ to $x : \texttt{int}$, the result has type `int`, and hence, by the arithmetic subtyping axiom, has type `float`. This suggests that

$$\texttt{int} \to \texttt{int} <: \texttt{int} \to \texttt{float}$$

is a valid subtype relationship. By similar reasoning, we may derive that

$$\texttt{float} \to \texttt{int} <: \texttt{float} \to \texttt{float}$$

is also valid.

Now suppose that $f : \texttt{float} \to \texttt{int}$. If $x : \texttt{int}$, then $x : \texttt{float}$ by subsumption, and hence we may apply $f$ to $x$ to obtain a result of type `int`. This suggests that

$$\texttt{float} \to \texttt{int} <: \texttt{int} \to \texttt{int}$$

is a valid subtype relationship. Since $\texttt{int} \to \texttt{int} <: \texttt{int} \to \texttt{float}$, it follows that

$$\texttt{float} \to \texttt{int} <: \texttt{int} \to \texttt{float}$$

is also valid.

Subtyping rules that specify how a type constructor interacts with subtyping are called *variance* principles. If a type constructor *preserves* subtyping in a given argument position, it is said to be *covariant* in that position. If, instead, it *inverts* subtyping in a given position it is said to be *contravariant* in that position. The discussion suggests that the function space constructor

[1] Nothing depends on this particular choice; any other specified subtyping relationship would suffice.

is covariant in the range position and contravariant in the domain position. This is expressed by the following rule:

$$\frac{\tau_1 <: \sigma_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2} \tag{31.5a}$$

Note well the inversion of subtyping in the domain, where the function constructor is contravariant, and the preservation of subtyping in the range, where the function constructor is covariant.

To ensure safety, we may define the dynamic semantics of casting to a function type by the following rule:

$$\frac{e \text{ val}}{(\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2)\, e \mapsto \lambda(x:\tau_1.\, (\sigma_2 <: \tau_2)\, e((\tau_1 <: \sigma_1)\, x))} \tag{31.5b}$$

Here $e$ has type $\sigma_1 \to \sigma_2$, $\tau_1 <: \sigma_1$, and $\sigma_2 <: \tau_2$. The argument is cast to the domain type of the function prior to the call, and its result is cast to the intended type of the application.

### 31.3.3 Product and Record Subtyping

What are the variance principles associated with binary product types? Suppose that $e : \sigma_1 \times \sigma_2$. There are two elimination forms associated with products, the projections, with which we may form $\texttt{fst}(e)$ and $\texttt{snd}(e)$, of types $\sigma_1$ and $\sigma_2$, respectively. If $\sigma_1 <: \tau_1$ and $\sigma_2 <: \tau_2$, then these expressions are also of types $\tau_1$ and $\tau_2$, respectively. But these are the types of projections from a value of type $\tau_1 \times \tau_2$. This leads to the following *covariance* principle of subtyping for product types:

$$\frac{\sigma_1 <: \tau_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \times \sigma_2 <: \tau_1 \times \tau_2}\ . \tag{31.6a}$$

The covariance of product types is also called *depth subtyping*, since it applies subtyping *within* the components of the product.

The dynamic semantics of covariance is given by the following transition:

$$\frac{e_1 \text{ val} \quad e_2 \text{ val}}{(\sigma_1 \times \sigma_2 <: \tau_1 \times \tau_2)\, \langle e_1, e_2\rangle \mapsto \langle (\sigma_1 <: \tau_1)\, e_1, (\sigma_2 <: \tau_2)\, e_2\rangle} \tag{31.6b}$$

In other words, we may cast $e$ from $\sigma_1 \times \sigma_2$ by producing the pair consisting of the cast of its two components from $\sigma_1$ to $\tau_1$ and $\sigma_2$ to $\tau_2$, respectively.

Similar covariance principles apply to $n$-tuples and records, for essentially the same reasons.

Another form of subtyping for products is called *width* subtyping; it applies to the $n$-ary form of product type, $\langle \tau_1, \ldots, \tau_n \rangle$. Width subtyping states that a wider tuple may be regarded as a narrower tuple by simply "forgetting" the additional components:

$$\frac{m \geq n}{\langle \tau_1, \ldots, \tau_m \rangle <: \langle \tau_1, \ldots, \tau_n \rangle} \cdot \tag{31.6c}$$

There are two possible dynamic interpretations of this subtyping principle, according to whether projections are sensitive to the exact size of the tuple, or only to those components that are actually needed to be present. If $e \cdot i$ is well-defined for any tuple of size $n \geq i$, then no coercion is necessary. If, on the other hand, projection requires full knowledge of the width of the tuple from which it is projecting, we may coerce as follows:

$$\frac{e \ \mathsf{val} \quad m \geq n}{(\langle \sigma_1, \ldots, \sigma_m \rangle <: \langle \tau_1, \ldots, \tau_n \rangle)\ \langle e_1, \ldots, e_m \rangle \mapsto \langle e_1, \ldots, e_n \rangle} \cdot \tag{31.6d}$$

That is, we simply create a new tuple consisting of the first $n \leq m$ fields of the subject of the coercion.

Width subtyping also applies to records (labelled tuples), but the semantic interpretation is subtlely different. The reason is that the order of fields in a record type is immaterial — any permutation of the fields results in an equivalent record type.[2] The width subtyping principle for records may be stated as follows:

$$\frac{m \geq n}{\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \tag{31.7a}$$

This seems superficially similar to prefix subtyping for tuples, except that since the order of fields does not matter, the dynamic meaning is more complicated.

To see what is going on, let us consider the behavior of the field selection operation, $e \cdot l$, where $e : \langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle$. If the type of $e$ determines precisely the fields present in its value, then projection can be implemented by a simple offset calculation. That is, if the typing assumption tells us that the value of $e$ has the form $\langle l_1 = e_1, \ldots, l_m = e_m \rangle$, then by maintaining the fields in sorted order, we can determine the offset of the field $l$ in the value of $e$ *statically*, and implement projection accordingly. If, on the other hand, the typing only provides a *view* of the shape of $e$, then we can no longer

[2] It is certainly possible to consider records in which order matters, but the issues that arise are not significantly different from those for ordered $n$-tuples.

make such a prediction statically, but must determine it *dynamically* by, say, looking up $l$ in a dictionary associated with $e$'s value that determines the location of field $l$ in that value.

Turning these conditions around, if field selection is performed by a dynamic search for the location of a field in the record value, then record width subtyping need have no dynamic effect. If, on the other hand, field selection is to be resolved statically, then we have no choice but to implement width subtyping by the following coercion:

$$\frac{e \text{ val} \quad m \geq n}{\begin{array}{c}(\langle l_1 : \sigma_1, \ldots, l_m : \sigma_m \rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle)\ \langle l_1 = e_1, \ldots, l_m = e_m \rangle \\ \mapsto \\ \langle l_1 = e_1, \ldots, l_n = e_n \rangle\end{array}} \tag{31.7b}$$

This rule is notationally deceptive, because it assumes that the selected fields are laid out in order at the beginning of the record value. In reality one must select the appropriate fields one-by-one, and reconstruct the record consisting of just those selected fields. If we make the implicit permutation explicit, the coercion takes the following form:

$$\frac{e \text{ val} \quad \pi : \{1, \ldots, m\} \rightarrowtail \{1, \ldots, n\}}{\begin{array}{c}(\langle l_1 : \sigma_1, \ldots, l_m : \sigma_m \rangle <: \langle l_{\pi(1)} : \sigma_{\pi(1)}, \ldots, l_{\pi(m)} : \sigma_{\pi(m)} \rangle)\ \langle l_1 = e_1, \ldots, l_m = e_m \rangle \\ \mapsto \\ \langle l_{\pi(1)} = e_{\pi(1)}, \ldots, l_{\pi(n)} = e_{\pi(n)} \rangle\end{array}} \tag{31.7c}$$

Here $\pi$ is an injective (one-to-one) mapping of the indices of the supertype into the indices of the subtype. This represents the selection of fields from the subtype that remain in the supertype. The coercion copies fields in correspondence with this permutation.

### 31.3.4 Sum and Variant Subtyping

Deriving the rules of subsumption for sum types follows a similar pattern. We must bear in mind that the elimination form for a sum type is a case analysis that covers *all* cases that can possibly arise — but no harm is done if fewer than anticipated are possible.

Binary sum types are covariant in each position:

$$\frac{\sigma_1 <: \tau_1 \quad \sigma_2 <: \tau_2}{\sigma_1 + \sigma_2 <: \tau_1 + \tau_2}\ . \tag{31.8a}$$

This is reasonable, because each branch of a case analysis on a value of the supertype expects a value of type $\tau_1$ and $\tau_2$, respectively, and we may instead supply a value of type $\sigma_1$ and $\sigma_2$, since these may be coerced to the specified supertypes.

The coercion interpretation is given by the following rules, wherein we write $\sigma$ for $\sigma_1 + \sigma_2$, and $\tau$ for $\tau_1 + \tau_2$:

$$\frac{e_1 \text{ val}}{(\sigma_1 + \sigma_2 <: \tau_1 + \tau_2)\ \mathtt{inl}_{\sigma_1+\sigma_2}(e_1) \longmapsto \mathtt{inl}_{\tau_1+\tau_2}((\sigma_1 <: \tau_1)\ e_1)} \tag{31.8b}$$

$$\frac{e_1 \text{ val}}{(\sigma_1 + \sigma_2 <: \tau_1 + \tau_2)\ \mathtt{inr}_{\sigma_1+\sigma_2}(e_2) \longmapsto \mathtt{inr}_{\tau_1+\tau_2}((\sigma_2 <: \tau_2)\ e_2)}\ . \tag{31.8c}$$

That is, we coerce the component expression to the appropriate summand, then re-inject into the supertype.

The analogue of width subtyping for labelled sums states that a *narrower* choice is a subtype of a *wider* choice.

$$\frac{m \leq n}{[l_1 : \tau_1, \ldots, l_m : \tau_m] <: [l_1 : \tau_1, \ldots, l_n : \tau_n]} \tag{31.8d}$$

This is justified by the observation that a case analysis on the supertype is prepared to consider more cases than can actually arise from a value of the subtype.

The dynamic interpretation of this principle is simply to relabel the injection with the supertype.

$$\frac{1 \leq j \leq m}{([l_1 : \sigma_1, \ldots, l_n : \sigma_n] <: [l_1 : \tau_1, \ldots, l_m : \tau_m])\ [l_j = e_j]_{[l_1:\tau_1,\ldots,l_n:\tau_n]} \longmapsto [l_j = e_j]_{[l_1:\tau_1,\ldots,l_n:\tau_n]}}\ . \tag{31.8e}$$

Since the only action of the coercion is to relabel the injection, there is no need for taking explicit account of permutation as there was in the case of records.

### 31.3.5 Reference Subtyping

Let us apply the same principles to determining the variance principles for reference types. Suppose that $r$ has type $\sigma$ `ref`. We can do one of two things with $r$:

1. Retrieve its contents as a value of type $\sigma$.

2. Replace its contents with a value of type $\sigma$.

If $\sigma <: \tau$, then retrieving the contents of $r$ yields a value of type $\tau$, by subsumption. This suggests that references are covariant:

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

On the other hand, if $\tau <: \sigma$, then we may store a value of type $\tau$ into $r$. This suggests that references are contravariant:

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

Given that we may perform either operation on a reference cell, we must insist that reference types are *invariant* (or *nonvariant*):

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\sigma\,\mathtt{ref} <: \tau\,\mathtt{ref}} \tag{31.9a}$$

The premise of the rule is often strengthened to the requirement that $\sigma$ and $\tau$ be equal:

$$\frac{\sigma = \tau}{\sigma\,\mathtt{ref} <: \tau\,\mathtt{ref}} \tag{31.9b}$$

since there are seldom situations where distinct types are mutual subtypes.

A similar analysis may be applied to any mutable data structure. For example, *immutable* sequences may be safely taken to be covariant, but *mutable* sequences (arrays) must be taken to be invariant, lest safety be compromised.

### 31.3.6 Recursive Subtyping

The subtyping principles for recursive types are a bit tricky to state, because of the binding operator. It is tempting to postulate the following covariance principle for recursive types:

$$\frac{\sigma <: \tau}{\mu(t.\sigma) <: \mu(t.\tau)} \quad \textbf{??}$$

The idea is that since the bound variable on both sides has been chosen to be the same, then reflexivity will ensure that $t <: t$.

For example, this rule validates the intuitively plausible subtyping principle stating that the type of lists of integers is a subtype of the type of lists of floating point numbers. Assuming that $\texttt{int} <: \texttt{float}$,

$$\mu(t.\texttt{unit} + (\texttt{int} \times t)) <: \mu(t.\texttt{unit} + (\texttt{float} \times t)).$$

This is easily derived using the covariance of sum and product types, and reflexivity of subtyping to derive $t <: t$.

On the other hand, this rule also validates illegitimate subtyping relationships. Again assuming that $\texttt{int} <: \texttt{float}$, using the above rule we may derive the subtyping

$$\sigma = \mu(t.\texttt{int} \times (t \to \texttt{int})) <: \mu(t.\texttt{float} \times (t \to \texttt{float})) = \tau.$$

But this is unsound! Consider the function $e$ of type $\sigma \to \texttt{int}$ defined by

$$\lambda(x{:}\sigma.\,\texttt{let}\ x'\ \texttt{be}\ \texttt{unfold}(x)\ \texttt{in}\ \texttt{fst}(x')\texttt{+1}).$$

If the value $v = \texttt{fold}(\langle 0, e\rangle)$ of type $\sigma$ is regarded as a value of type $\tau$, then we may form the expression $e'$ of type $\texttt{float}$ given by

$$\texttt{snd}(\texttt{unfold}(v))(\texttt{fold}(\langle 3.1, f\rangle)),$$

where $f$ is any function of type $\tau \to \texttt{float}$ (*e.g.*, a constant function). But

$$e' \mapsto^* e(3.1) \mapsto^* 3.1\texttt{+1},$$

which is type-incorrect (integer addition applied to a floating point number).

What has gone wrong? By choosing the same bound variable on both sides of the questionable subtyping rule, we have tacitly assumed that the two recursive types may be regarded as *equal* when comparing the bodies of the recursive types for subtyping! That is, we may re-phrase the questionable rule above as follows, separating the bound variables and using a hypothetical judgement:

$$\frac{s = t \vdash \sigma <: \tau}{\mu(s.\sigma) <: \mu(t.\tau)}\ \textbf{??}$$

That is, we *assume* that $s$ and $t$ are equal while comparing the bodies of the recursive types.

But this is clearly wrong, because the bound variable of a recursive type stands for the type itself, which on the left is $\mu(s.\sigma)$, and on the right is

$\mu(t.\tau)$, and these are different types, in general! The correct rule is to think self-referentially. While comparing the bodies of recursive types, we *assume* what we are trying to prove:

$$\frac{s <: t \vdash \sigma <: \tau}{\mu(s.\sigma) <: \mu(t.\tau)}\ . \tag{31.10}$$

This is the correct rule. Revisiting our example, we are unable to show that $\sigma <: \tau$, because we *cannot* derive the hypothetical judgement

$$s <: t \vdash s \to \mathtt{int} <: t \to \mathtt{float},$$

precisely because the function type is contravariant in the argument position. This is a good thing, for otherwise the subtyping relation would be unsound!

The dynamic interpretation of this coercion is interesting. An illustrative example will help to see what is going on. Consider the types $\sigma = \mu(s.\mathtt{unit} + (\mathtt{int} \times s))$ and $\tau = \mu(t.\mathtt{unit} + (\mathtt{float} \times t))$. Intuitively, these are the types of integer and floating point lists, respectively. One would expect $\sigma <: \tau$, and indeed this follows directly from the hypothetical judgement

$$s <: t \vdash \mathtt{unit} + (\mathtt{int} \times s) <: \mathtt{unit} + (\mathtt{float} \times t).$$

What is the dynamic interpretation of this subtyping relation? Intuitively, we must coerce every element of the list from `int` to `float`. That is, the coercion between recursive types is a recursive function!

The dynamic interpretation of the recursive subtyping rule may be stated as follows:

$$\frac{e\ \mathsf{val}}{(\mu(s.\sigma)<:\mu(t.\tau))\ \mathtt{fold}(e) \mapsto \mathtt{fold}(([\mu(s.\sigma)/s]\sigma<:[\mu(t.\tau)/t]\tau)\ e)}\ . \tag{31.11}$$

Notice that the coercion is self-referential! The substitution on the right-hand side will, in general, replicate $\mu(t.\tau)$ in the coercion of $e$, leading to a recursive call of the coercion itself. In the case of the list example, this means that the entire list will be reconstructed, applying coercions to every element along the way.

### 31.3.7 Object Subtyping

Similar subtyping principles apply to object types as apply to record types. In particular, object types are covariant in their components, and permit

width subtyping.

$$\frac{\sigma_1 <: \tau_1 \quad \ldots \quad \sigma_n <: \tau_n}{\mathtt{obj}\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle <: \mathtt{obj}\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle} \tag{31.12a}$$

$$\frac{m \leq n}{\mathtt{obj}\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle <: \mathtt{obj}\langle l_1 : \sigma_1, \ldots, l_m : \sigma_m \rangle} \tag{31.12b}$$

As with records, the statement of width subtyping relies on an implicit use of permutation os that the first $m$ components are retained.

The dynamic semantics of object width subtyping is a bit more complex than that for records. The reason is that, because of self-reference, it is not sensible to drop fields that are not present in the supertype. For example, if an object has two fields labelled $a$ and $b$, then the binding of the $a$ field might, via self-reference, refer to field $b$, and *vice versa*. Consequently, if we coerce this object to an object type with only an $a$ field, then we cannot sensibly remove the omitted $b$ field without incurring a violation of type safety — the remaining $a$ field would refer to a non-existent $b$ field at run-time.

This means that field access for objects cannot be performed by direct indexing based on offsets into the object based only on its static type — the position of a field labelled $l$ is not determined by its type when width subtyping is permitted. Instead we must employ an indirection scheme that separates the static view of the type from the object itself. Specifically, the dynamic representation of an object of type $\mathtt{obj}\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle$ consists of an object whose "true type" (at the point of creation) is a subtype $\mathtt{obj}\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n \rangle$, together with an injection $\pi : \{1, \ldots, m\} \rightarrowtail \{1, \ldots, n\}$ indicating the positions of the fields indexed by $1 \leq i \leq m$ in the underlying object whose fields are indexed $1 \leq i \leq n$. The dynamic effect of subtyping is to compose permutations to maintain an accurate mapping of the visible components of the object to its actual components.

## 31.4 Explicit Coercions

The dynamic semantics of the coercions given in the preceding section can be expressed explicitly by defining an explicit coercion function from the subtype to the supertype for each principle of subtyping. This shows that subtyping can be "compiled away" by simply inserting calls to the appropriate coercion functions during type checking. Put in other terms, we may

forego subtyping entirely, if we are willing to simply call the implied coercion functions ourselves. In this sense subtyping is only a matter of convenience; it does not fundamentally increase the expressiveness of the language.

Coercions among primitives types must be taken as primitive. For example, we might postulate a function `to_float`, which coerces an integer to a floating point number, to serve as a witness to the subtyping relation `int` $<:$ `float`. Coercions for compound types are then built up systematically from the coercions at primitive type, as illustrated by the following equations.

$$(\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2) = \lambda(f:\sigma_1 \to \sigma_2.\, \lambda(x:\tau_1.\, (\sigma_2<:\tau_2)\, f((\tau_1<:\sigma_1)\, x))) \tag{31.13a}$$

$$(\sigma_1 \times \sigma_2 <: \tau_1 \times \tau_2) = \lambda(x:\sigma_1 \times \sigma_2.\, \langle (\sigma_1<:\tau_1)\, \mathtt{fst}(x), (\sigma_2<:\tau_2)\, \mathtt{snd}(x)\rangle) \tag{31.13b}$$

$$(\langle\sigma_1,\ldots,\sigma_n\rangle <: \langle\sigma_1,\ldots,\sigma_m\rangle) = \lambda(x:\langle\sigma_1,\ldots,\sigma_n\rangle.\, \langle x\cdot 1,\ldots,x\cdot m\rangle) \tag{31.13c}$$

$$(\mu(s.\sigma) <: \mu(t.\tau)) = \mathtt{fun}\, f(x:\mu(s.\sigma)):\mu(t.\tau)\, \mathtt{is}\, F(f)(x) \tag{31.13d}$$

In the last equation the function $F(f)$ is the interpretation of $(\sigma <: \tau)$ as a function of the assumed coercion $f$ witnessing $s <: t$.

Using this explicit interpretation of coercions, we may then interpret $(\sigma<:\tau)\, e$ as the application of the coercion $(\sigma <: \tau)$ to $e$!

## 31.5 Coherence

In Section 31.2 on page 252 we argued that implicit subsumption is best regarded as a convenient syntax for explicit subsumption in which each implicit typing derivation corresponds to a decoration of the expression with explicit uses of subtyping inserted as casts. But since there are, in general, many ways to perform this translation, we are faced with the question of whether all such ways of inserting casts are equivalent. If not, then the meaning of a program is sensitive to the structure of the derivation of its type correctness, and not just to the program itself. Thus, the program itself has an ambiguous meaning, forcing the type checker to choose one of

many possible interpretations, quite possibly not the one intended by the programmer. If so, then the meaning of the program is independent of the way in which it is type checked, and hence is unambiguous in its implicit form.

When all possible translations of implicit into explicit subtyping are equivalent, the interpretation is said to be *coherent*, and is otherwise *incoherent*. To avoid ambiguity, the implicit form of subsumption should be considered only when the coercion interpretation is coherent. Coherence is guaranteed if all coercions are trivial in the sense that they do not alter the representation of a value when passing from a subtype to a supertype. In the presence of non-trivial coercions, however, coherence is far from obvious.

One source of failure is well-illustrated by considering the numeric subtyping `int` $<:$ `float`. Consider the expression `sqrt(`$e_1$`+`$e_2$`)`, where $e_1$ and $e_2$ are integer expressions, and `sqrt` is an operation on floating point numbers. This expression may be type checked in one of two ways, corresponding to the following two explicit forms of the expression.

1. `sqrt((int<:float)` $e_1$`+(int<:float)` $e_2$`)`.
2. `sqrt((int<:float) (`$e_1$`+`$e_2$`))`.

In the first case the two integers are coerced to floating point numbers, then added and passed to `sqrt`. In the second the integers are added as such, then coerced to floating point before being passed to `sqrt`. *But these two interpretations are not always the same!* The reason is that addition does not commute with floating point representation, so that the floating point form of an integer sum need not coincide with the floating sum of the floating point representations of its arguments.

Another, more subtle, problem of incoherence arises even in the explicit case. When assigning semantics to a cast $(\sigma <: \tau)\ e$, we are tactitly assuming that *all possible ways of coercing $\sigma$ to $\tau$ are equivalent*. The unstated assumption lies in the supposition that we can coerce $e$ from $\sigma$ to $\tau$ knowing only the endpoints, and not the path by which we get from one to the other.[3] If there are two semantically distinct ways of coercing values of type $\sigma$ to values of type $\tau$, then even explicit subsumption is incoherent. For example, one might have the subsumption relations `int` $<:$ `float` $<:$ `double`, in which the type `double` represents double-precision floating point numbers. It is not assured that first coercing an integer to floating point, and

[3]This is the justification for defining $(\sigma <: \sigma)\ e \mapsto e$ in Section 31.2 on page 252: all coercions from $\sigma$ to itself are equivalent to the identity coercion.

then to double-precision floating points, is equivalent to directly converting the integer to double-precision, because the integer may not have an exact representation in floating point, but may have one in double-precision. Therefore subtyping is sensitive to the "path" from `int` to `double`, leading to incoherence.

In the presence of such incoherence, not even explicit subsumption can be regarded as semantically sensible, since the meaning of $(\sigma <: \tau)\, e$ depends on the "reason" for $\sigma <: \tau$, and not just on $\sigma$ and $\tau$ themselves. In such situations it is best to avoid subtyping entirely in favor of simply asking that the programmer write down the intended coercion as a function of type $\sigma \to \tau$ herself, and not rely on automatic generation at all. If, on the other hand, all coercions witnessing the subtype relation $\sigma <: \tau$ are equivalent, then one may consider explicit subsumption in the form described here. Finally, to justify the use of implicit subsumption, one must prove coherence of type checking — that any way of inserting coercions is as good as any other.

## 31.6 Exercises

1. Consider the subtyping issues related to signed and unsigned, fixed precision integer types.

2. Explore the effect of functional update on record and object subtyping.

# Chapter 32

# Bounded Quantification

# Chapter 33

# Singleton and Dependent Kinds

The expression $\texttt{let}\, e_1 : \tau\, \texttt{be}\, x\, \texttt{in}\, e_2$ is a form of abbreviation mechanism by which we may bind $e_1$ to the variable $x$ for use within $e_2$. In the presence of function types this expression is definable as the application $\lambda\,(x:\tau.\, e_2)\,(e_1)$, which accomplishes the same thing. It is natural to consider an analogous form of `let` expression which permits a *type expression* to be bound to a type variable within a specified scope. The expression $\texttt{let}\, t\, \texttt{be}\, \tau\, \texttt{in}\, e$ binds $t$ to $\tau$ within $e$, so that one may write expressions such as

$$\texttt{let}\, t\, \texttt{be}\, \texttt{nat} \times \texttt{nat}\, \texttt{in}\, \lambda\,(x:t.\, \texttt{s}(\texttt{fst}(x))).$$

For this expression to be type-correct the type variable $t$ must be *synonymous* with the type $\texttt{nat} \times \texttt{nat}$, for otherwise the body of the $\lambda$-abstraction is not type correct.

Following the pattern of the ordinary `let`, we might guess that `lettype` is an abbreviation for the polymorphic instantiation $\Lambda\,(t.\, e)\,[\tau]$, which binds $t$ to $\tau$ within $e$. However, this representation is incorrect! The difficulty is that $e$ is typechecked with the type $t$ held abstract, which precisely violates the requirement of synonymity with its binding.

One reaction is to simply take `lettype` as a primitive concept, but this would violate the principle that language features should arise naturally from type structure. What type-theoretic concept gives rise to type abbreviations? The answer to this question is *singleton kinds*, which extend the kind structure of the language with a kind, $\texttt{equiv}(c)$, of constructors of kind `Type` that are definitionally equivalent to $c$. Using singletons we can

define $\texttt{let}\ t\ \texttt{be}\ \tau\ \texttt{in}\ e$ as the application

$$\Lambda(t::\texttt{equiv}(\tau).e)[\tau],$$

which propagates the definition of $t$ as $\tau$ into the scope, $e$, of the abbreviation. For if $t$ has kind $\texttt{equiv}(\tau)$, then $t$ is definitionally equivalent to $\tau$, as required to faithfully model type abbreviations.

# Part XII

# State

# Chapter 34

# Storage Effects

$\mathcal{L}\{\rightarrow\}$ is said to be a *pure* language because the execution model consists entirely of evaluating an expression for its value. ML is an *impure* language because its execution model also includes *effects*, specifically, *control effects* and *store effects*. Control effects are non-local transfers of control; these were studied in Chapters 28 and 27. Store effects are dynamic modifications to mutable storage. This chapter is concerned with store effects.

## 34.1 References

The $\mathcal{L}\{\rightarrow\}$ type language is extended with *reference types* $\texttt{ref}(\tau)$ whose elements are to be thought of as mutable storage cells. We correspondingly extend the expression language with these primitive operations:

$$\text{Expr} \quad e \quad ::= \quad l \mid \texttt{new}(e) \mid \texttt{get}(e) \mid \texttt{set}(e_1, e_2)$$

As in Standard ML, $\texttt{new}(e)$ allocates a "new" reference cell, $\texttt{get}(e)$ retrieves the contents of the cell $e$, and $\texttt{set}(e_1, e_2)$ sets the contents of the cell $e_1$ to the value $e_2$. The variable $l$ ranges over a set of *locations*, an infinite set of names disjoint from variables. These are needed for the dynamic semantics, but are not expected to be notated directly by the programmer.

The typing judgement, $e : \tau$, for the extension of $\mathcal{L}\{\rightarrow\}$ with references must be considered in the context of two forms of assumptions:

1. *Variable assumptions* of the form $x : \tau$, introducing an expression variable $x$ with type $\tau$.
2. *Location assumptions* of the form $l : \tau$, introducing a location $l$ whose contents is of type $\tau$.

The hypothetical typing judgement has the form

$$\Lambda\,\Gamma \vdash e : \tau,$$

where $\Gamma$ stands for a finite set of variable assumptions, and $\Lambda$ stands for a finite set of location assumptions, such that no variable and no location is the subject of more than one assumption.

The typing rules are those of $\mathcal{L}\{\rightarrow\}$ (extended to carry a location typing), plus the following rules governing the new constructs of the language:

$$\frac{}{\Lambda, l : \tau\,\Gamma \vdash l : \mathtt{ref}(\tau)} \tag{34.1a}$$

$$\frac{\Lambda\,\Gamma \vdash e : \tau}{\Lambda\,\Gamma \vdash \mathtt{new}(e) : \mathtt{ref}(\tau)} \tag{34.1b}$$

$$\frac{\Lambda\,\Gamma \vdash e : \mathtt{ref}(\tau)}{\Lambda\,\Gamma \vdash \mathtt{get}(e) : \tau} \tag{34.1c}$$

$$\frac{\Lambda\,\Gamma \vdash e_1 : \mathtt{ref}(\tau_2) \quad \Lambda\,\Gamma \vdash e_2 : \tau_2}{\Lambda\,\Gamma \vdash \mathtt{set}(e_1, e_2) : \tau_2} \tag{34.1d}$$

Notice that the location typing is not extended during type checking! Locations arise only during execution, and are not part of complete programs, which must not have any free locations in them. The role of the location typing will become apparent in the proof of type safety for $\mathcal{L}\{\rightarrow\}$ extended with references.

A *memory* is a finite function mapping locations to closed values. The dynamic semantics of $\mathcal{L}\{\rightarrow\}$ with references is given by an abstract machine with states of the form $(M, e)$, where $M$ is a memory and $e$ is an expression whose locations all lie within the domain of $M$. The locations in the domain of $M$ are bound simultaneously in both components of the state, so that they may be renamed as convenient. All states of the form $(\emptyset, e)$, where $e$ contains no locations, are initial. All states of the form $(M, e)$, where $e$ $\mathsf{val}$, are final.

The dynamic semantics of references is defined by the following rules:

$$\frac{}{l\ \mathsf{val}} \tag{34.2a}$$

$$\frac{(M, e) \mapsto (M', e')}{(M, \mathtt{new}(e)) \mapsto (M', \mathtt{new}(e'))} \tag{34.2b}$$

$$\frac{e\ \mathsf{val} \quad l \,\#\, M}{(M, \mathtt{new}(e)) \mapsto (M[l = e], l)} \tag{34.2c}$$

$$\frac{(M,e) \mapsto (M',e')}{(M,\mathtt{get}(e)) \mapsto (M',\mathtt{get}(e'))} \tag{34.2d}$$

$$\frac{e\ \mathsf{val}}{(M[l=e],\mathtt{get}(l)) \mapsto (M,e)} \tag{34.2e}$$

$$\frac{(M,e_1) \mapsto (M',e_1')}{(M,\mathtt{set}(e_1,e_2)) \mapsto (M',\mathtt{set}(e_1',e_2))} \tag{34.2f}$$

$$\frac{e_1\ \mathsf{val} \quad (M,e_2) \mapsto (M',e_2')}{(M,\mathtt{set}(e_1,e_2)) \mapsto (M',\mathtt{set}(e_1,e_2'))} \tag{34.2g}$$

$$\frac{e\ \mathsf{val}}{(M[l=e'],\mathtt{set}(l,e)) \mapsto (M[l=e],e)} \tag{34.2h}$$

To prove type safety for this extension we will make use of some auxiliary relations. Most importantly, the typing relation between memories and location typings, written $M : \Lambda$, is inductively defined by the following rule:

$$\frac{\forall l \in dom(\Lambda)\ \Lambda \vdash M(l) : \Lambda(l)}{M : \Lambda} \tag{34.3}$$

For each location $l$ in the location typing, we require that the value stored at location $l$ has the type assigned to it by the location typing, relative to the *entire* location typing. This allows for cyclic memories in which the contents of one location may refer, directly or indirectly, to itself.

The typing rule for memories is reminiscent of the typing rule for recursive functions—we assume the very typing that we are trying to prove while trying to prove it. This similarity is no accident. In fact, we can use mutable storage to implement recursion, as is illustrated by the following example:

```
let r be new(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 `nat` $\rightarrow$ `nat` that is obtained by (a) allocating a reference cell initialized arbitrarily with a function of ths type, (b) defining a $\lambda$-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. This technique is called *backpatching*.

The well-formedness of a machine state is defined by the following rule:

$$\frac{M : \Lambda \quad \Lambda \vdash e : \tau}{(M, e)\ \mathsf{ok}} \tag{34.4}$$

That is, $(M, e)$ is well-formed iff there is a location typing for $M$ relative to which $e$ is well-typed.

**Lemma 34.1** (Weakening)**.** *If $\Lambda \vdash e : \tau$ and $\Lambda' \supseteq \Lambda$, then $\Lambda' \vdash e : \tau$.*

*Proof.* By induction on the derivation of typing. The presence of additional assumptions governing locations not in $\Lambda$ does not affect the validity of the derivation. □

**Theorem 34.2** (Preservation)**.** *If $(M, e)$ $\mathsf{ok}$ and $(M, e) \mapsto (M', e')$, then $(M', e')$ $\mathsf{ok}$.*

*Proof.* The trick is to prove a stronger result by induction on evaluation: if $(M, e) \mapsto (M', e')$, $M : \Lambda$, and $\Lambda \vdash e : \tau$, then there exists $\Lambda' \supseteq \Lambda$ such that $M' : \Lambda'$ and $\Lambda' \vdash e' : \tau$. □

**Theorem 34.3** (Progress)**.** *If $(M, e)$ $\mathsf{ok}$ then either $(M, e)$ is a final state or there exists $(M', e')$ such that $(M, e) \mapsto (M', e')$.*

*Proof.* The proof is by induction on typing: if $M : \Lambda$ and $\Lambda \vdash e : \tau$, then either $e$ is a value or there exists $M'$ and $e'$ such that $(M, e) \mapsto (M', e')$. □

## 34.2 Exercises

1. Prove Theorem 34.2. The strengthened form tells us that the location typing, and the memory, increase monotonically during evaluation in the sense that the type of a location never changes once it is established at the point of allocation.

2. Prove Theorem 34.3 by induction on typing of machine states.

3. Sketch the contents of the memory after each step in the backpatching example.

# Chapter 35

# Monadic Storage Effects

As we saw in Chapter 34 one way to combine functional and imperative programming is to add a type of reference cells to $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$. This approach works well for call-by-value languages, because we can easily predict where expressions are evaluated, and hence where references are allocated and assigned. For call-by-name languages this approach is problematic, because in such languages it is much harder to predict when (and how often) expressions are evaluated.

Enriching ML with a type of references has an additional consequence that one can no longer determine from the type alone whether an expression mutates storage. For example, a function of type $\texttt{nat} \to \texttt{nat}$ must take in a natural number as argument and yield a natural number as result, but may or may not allocate new reference cells or mutate existing reference cells. The expressive power of the type system is thereby weakened, because we cannot distinguish *pure* (effect-free) expressions from *impure* (effect-ful) expressions.

Another approach to introducing effects in a purely functional language is to make the possibility of effects explicit in the type system. This is achieved by introducing a *modality*, called a *monad*, that segregates the effect-free fragment of the language from the effect-ful fragment. These two sublanguages are related by two principles: (a) every effect-free expression may be regarded as (vacuously) effect-ful, and (b) an effect-ful expression may be suspended and packaged as an effect-free expression, and, correspondingly, unpackaging and activating such an expression is an effect-ful operation. A packaged effect-ful expression is a value of *monadic type*, $\tau\,\texttt{comp}$, which signals that it classifies an impure computation yielding a value of type $\tau$.

In this setting the type $\texttt{nat} \to \texttt{nat}$ consists only of pure, possibly non-terminating, functions on the natural numbers. Applying such a function can have no effect on the store. However, the type $\texttt{nat} \to \texttt{nat}\,\texttt{comp}$ consists of functions that, when applied to a natural number, yield an effect-ful computation that, when activated, yields a natural number and may also modify the store. Thus, the type distinguishes the possibility of there being an effect.

## 35.1 A Monadic Language

The syntax of a monadic re-formulation of the extension of the language $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with references is given by the following grammar:

$$\begin{array}{llcl}
\text{Type} & \tau & ::= & \texttt{nat} \mid \texttt{parr}(\tau_1, \tau_2) \mid \texttt{ref}(\tau) \mid \texttt{comp}(\tau) \\
\text{Expr} & e & ::= & x \mid l \mid \texttt{z} \mid \texttt{s}(e) \mid \texttt{ifz}(e_0, e_1, e_2) \mid \texttt{fix}[\tau](x.e) \mid \\
 & & & \texttt{lam}[\tau](x.e) \mid \texttt{ap}(e_1, e_2) \mid \texttt{comp}(m) \\
\text{Comm} & m & ::= & \texttt{return}(e) \mid \texttt{letcomp}(e, x.m) \mid \\
 & & & \texttt{new}(e) \mid \texttt{get}(e) \mid \texttt{set}(e_1, e_2)
\end{array}$$

The class of expressions is *pure* in that they do not involve operations on the state, which are relegated to the class of *commands*. The type $\texttt{comp}(\tau)$ is the type of suspended computations yielding a value of type $\tau$. The introductory form is an expression of the form $\texttt{comp}(m)$, and the corresponding eliminatory form is the command $\texttt{letcomp}(e, x.m)$. The command $\texttt{return}(e)$ represents the inclusion of expressions into commands. The other constructs are adapted from Chapters 15 and 34.

The concrete syntax corresponding to the new forms of abstract syntax is given by the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\texttt{comp}(\tau)$ | $\tau\ \texttt{comp}$ |
| $\texttt{return}(e)$ | $\texttt{return}\ e$ |
| $\texttt{letcomp}(e, x.m)$ | $\texttt{let comp}(x)\ \texttt{be}\ e\ \texttt{in}\ m$ |
| $\texttt{comp}(m)$ | $\texttt{comp}(m)$ |

As a convenience for examples, we sometimes employ the *do syntax*, which permits sequential composition of impure computations in a manner reminiscent of imperative programming languages. It is defined in two stages. First, the binary "do" construct, with abstract syntax $\texttt{do}(m_1, x.m_2)$ and concrete syntax $\texttt{do}\,\{x \leftarrow m_1\ ;\ m_2\}$, is defined to be the impure expression

$\texttt{letcomp}(\texttt{comp}(m_1), x.m_2)$. Second, we introduce the concrete syntax

$$\texttt{do}\,\{x_1 \leftarrow m_1\,;\,\ldots\,;\,x_k \leftarrow m_k\,;\,\texttt{return}\,e\}$$

to stand for the abstract syntax

$$\texttt{do}\,\{x_1 \leftarrow m_1\,;\ldots\texttt{do}\,\{x_k \leftarrow m_k\,;\,\texttt{return}\,e\}\ldots\}.$$

The static semantics of this language consists of two forms of typing judgement, $e : \tau$, stating that pure expression $e$ has type $\tau$, and $m \sim \tau$, stating that the impure expression $m$ has type $\tau$. Both of these judgement forms are considered with respect to hypotheses of the form $x_i : \tau_i$, which introduces a variable $x_i$ with type $\tau_i$, and of the form $l_i : \tau_i$, which introduces a location $l_i$ whose contents is to be of type $\tau_i$. We will not have need of hypotheses of the form $u_i \sim \tau_i$, because variables are only ever bound to values, which are always pure (since they are fully evaluated). As in Chapter 34, we will write $\Gamma$ for a finite set of variable assumptions, and $\Lambda$ for a finite set of location assumptions. We will segregate these assumptions from one another when writing hypothetical judgements.

The typing rules for this extension are an extension of those for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, with the following additional rules.

$$\frac{\Lambda\,\Gamma \vdash m \sim \tau}{\Lambda\,\Gamma \vdash \texttt{comp}(m) : \texttt{comp}(\tau)} \tag{35.1a}$$

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

$$\frac{\Lambda\,\Gamma \vdash e : \texttt{comp}(\tau) \quad \Lambda\,\Gamma, x : \tau \vdash m \sim \tau'}{\Lambda\,\Gamma \vdash \texttt{letcomp}(e, x.m) \sim \tau'} \tag{35.1c}$$

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

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

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

$$\frac{\Lambda\,\Gamma \vdash e_1 : \texttt{ref}(\tau) \quad \Lambda\,\Gamma \vdash e_2 : \tau}{\Lambda\,\Gamma \vdash \texttt{set}(e_1, e_2) \sim \texttt{unit}} \tag{35.1g}$$

The dynamic semantics of the monadic formulation of $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$ with references is structured into two parts:

1. A transition relation $e \mapsto e'$ for pure expressions.

2. A transition relation $(M, m) \mapsto (M', m')$ for impure expressions.

The former relation is defined just as it is for $\mathcal{L}\{\texttt{nat} \rightharpoonup\}$, amended to account for the new pure expression forms. The latter is defined similarly to Chapter 34, again amended to account for the extensions with monadic primitives.

There are two additional forms of value at the pure expression level:

$$\frac{}{\texttt{comp}(m)\ \mathsf{val}} \tag{35.2a}$$

$$\frac{}{l\ \mathsf{val}} \tag{35.2b}$$

That is, both suspended computations and locations are values.

The rules governing the monadic primitives are as follows.

$$\frac{e \mapsto e'}{(M, \texttt{return}(e)) \mapsto (M, \texttt{return}(e'))} \tag{35.3a}$$

$$\frac{e \mapsto e'}{(M, \texttt{letcomp}(e, x\,.\,m)) \mapsto (M, \texttt{letcomp}(e', x\,.\,m))} \tag{35.3b}$$

$$\frac{(M, m_1) \mapsto (M', m_1')}{(M, \texttt{letcomp}(\texttt{comp}(m_1), x\,.\,m_2)) \mapsto (M', \texttt{letcomp}(\texttt{comp}(m_1'), x\,.\,m_2))} \tag{35.3c}$$

$$\frac{e\ \mathsf{val}}{(M, \texttt{letcomp}(\texttt{comp}(\texttt{return}(e)), x\,.\,m)) \mapsto (M, [e/x]m)} \tag{35.3d}$$

The evaluation rules for the reference primitives are as follows:

$$\frac{e \mapsto e'}{(M, \texttt{new}(e)) \mapsto (M, \texttt{new}(e'))} \tag{35.3e}$$

$$\frac{e\ \mathsf{val} \quad l \# M}{(M, \texttt{new}(e)) \mapsto (M[l = e], \texttt{return}(l))} \tag{35.3f}$$

$$\frac{e \mapsto e'}{(M, \texttt{get}(e)) \mapsto (M, \texttt{get}(e'))} \tag{35.3g}$$

$$\frac{e\ \mathsf{val} \quad l \# M}{(M[l = e], \texttt{get}(l)) \mapsto (M[l = e], \texttt{return}(e))} \tag{35.3h}$$

$$\frac{e_1 \mapsto e_1'}{(M, \mathtt{set}(e_1, e_2)) \mapsto (M, \mathtt{set}(e_1', e_2))} \tag{35.3i}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \mapsto e_2'}{(M, \mathtt{set}(e_1, e_2)) \mapsto (M, \mathtt{set}(e_1, e_2'))} \tag{35.3j}$$

$$\frac{e \ \mathsf{val} \quad l \ \# \ M}{(M[l = e'], \mathtt{set}(l, e)) \mapsto (M[l = e], \mathtt{return}(e))} \tag{35.3k}$$

The transition rules for the monadic elimination form is somewhat unusual. First, the expression $e$ is evaluated to obtain an encapsulated impure computation. Once such a computation has been obtained, execution continues by evaluating it in the current memory, updating that memory as appropriate during its execution. This process ends once the encapsulated computation is a `return` statement, in which case this value is passed to the body of the `letcomp`.

## 35.2 Explicit Effects

The chief motivation for introducing monads is to make explicit in the types any reliance on computational effects. In the case of storage effects this is not always an advantage. The problem is that any use of storage forces the computation to be within the monad, and there is no way to get back out—once in the monad, always in the monad. This rules out the use of so-called *benign effects*, which may be used internally in some computation that is, for all outward purposes, entirely pure. One example of this is provided by splay trees, which may be used to implement a purely functional dictionary abstraction, but which rely heavily on mutation for their implementation in order to ensure efficiency. A simpler example, which we consider in detail, is provided by the use of backpatching to implement recursion as described in Chapter 34.

When formulated using monads to expose the use of effects, the backpatching implementation, *fact*, of the factorial function is as follows:

```
do {
  r ← new (λ n:nat. comp(return (n)))
; f ← return (λ n:nat. ...)
; _ ← set (r, f)
; return f
}
```

where the elided $\lambda$-abstraction is given as follows:

```
λ(n:nat.
  ifz(n,
      comp(return(1)),
      n'.comp(
           do {
             f'← get(r)
           ; return (n*f'(n'))
           })))
```

Observe that each branch of the conditional test returns a suspended computation. In the case that the argument is zero, the computation simply returns the value 1. Otherwise, it fetches the contents of the associated reference cell, applies this to the predecessor, and returns the result of the appropriate calculation.

We may check that that *fact* $\sim$ `nat` $\to$ `(nat comp)`, which exposes two aspects of this code:

1. The computation that builds the recursive factorial function is impure, because it allocates and assigns to the reference cell used to implement backpatching.

2. The body of the factorial function is itself impure, because it accesses the reference cell to effect the recursive call.

The consequence is that the factorial function may no longer be used as a (pure) function! In particular, we cannot apply *fact* to an argument in a pure expression. In an impure context we must write

```
do {
  f ← fact
; x ← let comp (x:nat) be f(n) in return x
; return x
}
```

in order to bind the function computed by the expression *fact* to the variable `f`; apply this to $n$, yielding the result; and return this to the caller.

The difficulty is that the use of a reference cell to implement recursion is a benign effect, one that does not affect the purity of the function expression itself, nor of its applications. But the type system for effects studied here is incapable of recognizing this fact, and for good reason. For it is extremely

difficult, in general, to determine whether or not the use of effects in some region of a program is benign. As a stop-gap measure, one way around this is to introduce an operation of type $\tau\,\texttt{comp} \to \tau$, which may be used to exit the monad. But this ruins the very distinction we are trying to enforce using monads, namely that between pure and impure code!

## 35.3 Exercises

1. State and prove type safety for the monadic formulation of storage effects.

# Chapter 36

# Extensible Sums

The Standard ML exception mechanism comprises two separable concepts:

1. An exception *control mechanism*, which permits non-local transfers of control from any point in a program to the nearest enclosing exception handler. The exception control mechanism is described in Chapter 27.

2. A *type* of values that are passed to the exception handler when an exception is raised. In Standard ML the type of values associated with exceptions is the type `exn`, but the choice is essentially arbitrary, the only restriction being that a single type must govern the values associated with all exceptions in a program.

The type `exn` is known as an *extensible datatype* because it has many of the same characteristics as a datatype, except that the collection of constructors of that type is both statically and dynamically extensible. The type `exn` is similar to a datatype in that at any point in time we may think of it as a partially constructed, possibly recursive, labelled sum type $l_1, \tau_1, \ldots, l_n, \tau_n$. Each label $l_i$ serves a dual role as a labelled injection mapping a value of type $\tau_i$ into the type `exn`, and as a discriminator that determines whether a value of type `exn` is labelled by a specified $l_i$ and, if so, extracts the injected value. This is sufficient to construct exceptions using the injections and to perform case analysis for specified injections using discriminators.

The crucial difference to sum types, however, is that the collection of labels is *extensible*, meaning that at any point in the program we may introduce a "new" value constructor for the `exn` type, which has the effect of extending the sum type `exn` at that point with an addition summand.

Thus the constructors of the type `exn` are not fully determined until the entire program has been processed—the summands are specified piecemeal throughout the program. But even more so new summands can be added *dynamically*, for example in each iteration of a loop, so that the simple interpretation of `exn` as a partially constructed sum type breaks down in that one cannot determine the full meaning of the `exn` type until after execution is complete! Nevertheless the analogy with recursive sums is informative, and a good guide to what follows.

Confusingly, new value constructors for the `exn` type are introduced in Standard ML using an *exception* declaration. For example, the declaration

```
exception Div
```

introduces a new summand labelled `Div` of `exn` type with an associated value of type `unit`, which is suppressed from the syntax. Similarly, the declaration

```
exception Fail of string
```

introduces a new summand labelled `Fail` of `exn` type with an associated value of type `str`. Finally, one may introduce recursion using an exception declaration such as

```
exception Recursive of exn
```

which introduces a new summand labelled `Recursive` whose associated value is of type `exn`!

Importantly, if two exception declarations introduce an exception with the same name, they nevertheless are distinct exceptions, even though within the scope of one we cannot refer by name to the other. The exception declaration is therefore said to be *generative* in that each use generates a "new" summand of the `exn` type. In particular, if an exception declaration occurs within a loop, then each iteration of the loop introduces a fresh exception, *albeit* each with the same name!

Although the syntax of suggests otherwise, the `exn` type has little to do with the exception control mechanism. In particular, the `exception` declaration has nothing at all to do with exceptions! Rather, it introduces a new summand of the `exn` type, which may be used in whatever way we see fit—the type `exn` is no different than any other type in this respect. To avoid confusion we will study the concept of a dynamically extensible sum type in isolation, using a different notation that makes clear that there is no fundamental connection to the exception control mechanism.

## 36.1 Tags and Tagging

The abstract syntax of our tagging mechanism is given by the following grammar:

| | | | |
|---|---|---|---|
| Type | $\tau$ | ::= | $\mathtt{tagged} \mid \mathtt{tag}(\tau)$ |
| Expr | $e$ | ::= | $l \mid \mathtt{tagwith}[\tau](e_1;e_2) \mid \mathtt{lettag}(e_1;t,x,y.e_2) \mid \mathtt{newtag}[\tau]() \mid$ $\mathtt{istag}[t.\tau](e_1;e_2;e_3;e_4)$ |

The type `tagged` is the type of *tagged values*; it corresponds to the type `exn` in Standard ML. The type $\mathtt{tag}(\tau)$ is the type of *tags* with *associated value* of type $\tau$. A tag is just a label, or name. The expression $\mathtt{tagwith}[\tau](e_1;e_2)$ attaches a tag with associated type $\tau$ to a value of that type. The expression $\mathtt{lettag}(e_1;t,x,y.e_2)$ decomposes a tagged value into its constituent parts for use within a specified scope. A new tag with associated type $\tau$ is created by the expression $\mathtt{newtag}[\tau]()$. Two tags are compared for equality using $\mathtt{istag}[t.\tau](e_1;e_2;e_3;e_4)$.

The corresponding concrete syntax is given by the following chart:

| Abstract Syntax | Concrete Syntax |
|---|---|
| `tagged` | `tagged` |
| $\mathtt{tagwith}[\tau](e_1;e_2)$ | $\mathtt{tag}_\tau\ e_2\ \mathtt{with}\ e_1$ |
| $\mathtt{lettag}(e_1;t,x,y.e_2)$ | $\mathtt{let\ tag}_t\ y\ \mathtt{with}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2$ |
| $\mathtt{newtag}[\tau]()$ | $\mathtt{newtag}[\tau]$ |
| $\mathtt{istag}[t.\tau](e_1;e_2;e_3;e_4)$ | $\mathtt{iftag}_{t.\tau}\ e_1 \mathtt{is}\ e_2\ \mathtt{then}\ e_3\ \mathtt{else}\ e_4$ |

The static semantics consists of an inductive definition of judgements of the form $\Theta\,\Delta\,\Gamma \vdash e : \tau$, where $\Theta$ is a finite set of hypotheses of the form $l : \tau$ assigning an associated type to a tag such that no tag is assigned more than one type. The hypotheses of $\Delta$ are of the form $t$ type, and those of $\Gamma$ are of the form $x : \tau$, as usual.

$$\frac{}{\Theta, l : \tau\,\Delta\,\Gamma \vdash l : \mathtt{tag}(\tau)} \tag{36.1a}$$

$$\frac{\Delta \vdash \tau\ \mathsf{type} \quad \Theta\,\Delta\,\Gamma \vdash e_1 : \mathtt{tag}(\tau) \quad \Theta\,\Delta\,\Gamma \vdash e_2 : \tau}{\Theta\,\Delta\,\Gamma \vdash \mathtt{tagwith}[\tau](e_1;e_2) : \mathtt{tagged}} \tag{36.1b}$$

$$\frac{\Theta\,\Delta\,\Gamma \vdash e_1 : \mathtt{tagged} \quad \Theta\,\Delta, t\ \mathsf{type}\,\Gamma, x : \mathtt{tag}(t), y : t \vdash e_2 : \tau_2}{\Theta\,\Delta\,\Gamma \vdash \mathtt{lettag}(e_1;t,x,y.e_2) : \tau_2} \tag{36.1c}$$

$$\frac{\Delta \vdash \tau\ \mathsf{type}}{\Theta\,\Delta\,\Gamma \vdash \mathtt{newtag}[\tau]() : \mathtt{tag}(\tau)} \tag{36.1d}$$

$$\frac{\Theta\,\Delta\,\Gamma \vdash e_1 : \mathtt{tag}(\tau_1) \qquad \Theta\,\Delta\,\Gamma \vdash e_2 : \mathtt{tag}(\tau_2) \qquad \Theta\,\Delta\,\Gamma \vdash e_3 : [\tau_2/t]\tau \qquad \Theta\,\Delta\,\Gamma \vdash e_4 : [\tau_1/t]\tau \qquad \Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Theta\,\Delta\,\Gamma \vdash \mathtt{istag[}t\,.\,\tau\mathtt{]}\,(e_1; e_2; e_3; e_4) : [\tau_1/t]\tau} \tag{36.1e}$$

Rule (36.1e) is the most interesting. When comparing two tags for equality, we do not know *a priori* whether they have the same type of associated value. Indeed, if the tags are different, they will not, in general, have the same associated type, but if they are the same, then they will. Thinking of $e_1$ as the target of the comparison and $e_2$ as the candidate, then the type of the comparison is always $[\tau_1/t]\tau$. When the two tags are equal, then we propagate the equality of $\tau_1$ and $\tau_2$ by insisting that $e_3$ have type $[\tau_2/t]\tau$, but when they are not equal, we require only that $e_4$ have type $[\tau_1/t]\tau$.

The dynamic semantics is given in terms of an abstract machine whose states have the form $(T, e)$, where $T$ is a finite set of tags, and $e$ is an expression all of whose tags are in $T$.

$$\frac{l\ \mathsf{name}}{l\ \mathsf{val}} \tag{36.2a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\mathtt{tagwith[}\tau\mathtt{]}\,(e_1; e_2)\ \mathsf{val}} \tag{36.2b}$$

$$\frac{}{(\emptyset, e)\ \mathsf{initial}} \tag{36.2c}$$

$$\frac{e\ \mathsf{val}}{(T, e)\ \mathsf{final}} \tag{36.2d}$$

$$\frac{(T, e_1) \mapsto (T', e_1')}{(T, \mathtt{tagwith[}\tau\mathtt{]}\,(e_1; e_2)) \mapsto (T, \mathtt{tagwith[}\tau\mathtt{]}\,(e_1'; e_2))} \tag{36.2e}$$

$$\frac{e_1\ \mathsf{val} \quad (T, e_2) \mapsto (T', e_2')}{(T, \mathtt{tagwith[}\tau\mathtt{]}\,(e_1; e_2)) \mapsto (T', \mathtt{tagwith[}\tau\mathtt{]}\,(e_1; e_2'))} \tag{36.2f}$$

$$\frac{(T, e_1) \mapsto (T', e_1')}{(T, \mathtt{lettag}(e_1; t, x, y\,.\,e_2)) \mapsto (T', \mathtt{lettag}(e_1'; t, x, y\,.\,e_2))} \tag{36.2g}$$

$$\frac{\mathtt{tagwith[}\tau\mathtt{]}\,(e_1; e_2)\ \mathsf{val}}{(T, \mathtt{lettag(tagwith[}\tau\mathtt{]}\,(e_1; e_2); t, x, y\,.\,e)) \mapsto (T, [\tau, e_1, e_2/t, x, y]e)} \tag{36.2h}$$

$$\frac{l \,\#\, T}{(T, \mathtt{newtag[}\tau\mathtt{]}\,()) \mapsto (T \cup \{\, l \,\}, l)} \tag{36.2i}$$

$$\frac{(T, e_1) \mapsto (T', e_1')}{(T, \mathtt{istag}[t.\tau](e_1; e_2; e_3; e_4)) \mapsto (T', \mathtt{istag}[t.\tau](e_1'; e_2; e_3; e_4))} \tag{36.2j}$$

$$\frac{e_1\ \mathsf{val} \quad (T, e_2) \mapsto (T', e_2')}{(T, \mathtt{istag}[t.\tau](e_1; e_2; e_3; e_4)) \mapsto (T', \mathtt{istag}[t.\tau](e_1; e_2'; e_3; e_4))} \tag{36.2k}$$

$$\frac{(l_1 = l_2)}{(T, \mathtt{istag}[t.\tau](l_1; l_2; e_3; e_4)) \mapsto (T, e_3)} \tag{36.2l}$$

$$\frac{(l_1 \neq l_2)}{(T, \mathtt{istag}[t.\tau](l_1; l_2; e_3; e_4)) \mapsto (T, e_4)} \tag{36.2m}$$

The type `tagged` is definable in a language with existential types. Specifically, we may define `tagged` to be the existential type $\exists(t.t\,\mathtt{tag} \times t)$. Values of this type are packages consisting of the associated type, $\tau$, of the tagged value together with a tag for values of type $\tau$ paired with a value of type $\tau$. Consequently, we may define the introduction and elimination forms for the type `tagged` as follows:

$$\mathtt{tagwith}[\tau](e_1; e_2) = \mathtt{pack}[t.t\,\mathtt{tag} \times t; \tau](\mathtt{pair}(e_1, e_2)) \tag{36.3}$$

$$\mathtt{lettag}(e_1; t, x, y.e_2) = \mathtt{open}[t.t\,\mathtt{tag} \times t](e_1; t, z.[\mathtt{fst}(z), \mathtt{snd}(z)/x, y]e_2) \tag{36.4}$$

It is easy to check that the static and dynamic semantics is preserved by these definitions.

## 36.2 Safety

Given the definability of the type `tagged`, the safety of dynamically tagged values reduces to showing progress and preservation for the introduction and elimination forms for the type $\mathtt{tag}(\tau)$ of tags.

First, we define $T : \Theta$ to hold iff $dom(\Theta) = T$, which is to say that $\Theta$ assigns a type to all and only those tags in $T$.

The preservation theorem is similar to that for reference types ( 34.2 on page 276).

**Lemma 36.1** (Preservation)**.** *If $\Theta \vdash e : \tau$, $T : \Theta$, and $(T, e) \mapsto (T', e')$, then there exists $\Theta' \supseteq \Theta$ such that $T' : \Theta'$ and $\Theta \vdash e' : \tau$.*

*Proof.* The proof is by induction on transition. The most interesting case is for Rule (36.2l) in which the tags $l_1$ and $l_2$ are equal. We have by inversion that $\Theta \vdash l_1 : \mathtt{tag}(\tau_1)$ and $\Theta \vdash l_2 : \mathtt{tag}(\tau_2)$. But since $l_1 = l_2$, it follows from inversion and $\Theta$ being a function that $\tau_1 = \tau_2$. By inversion we have $\Theta \vdash e_2 : [\tau_2/t]\tau$, but then this is just $\Theta \vdash e_2 : [\tau_1/t]\tau$, as required. □

The canonical forms lemma characterizes the closed values of tag type.

**Lemma 36.2** (Canonical Forms)**.** *If $\Theta \vdash e : \mathtt{tag}(\tau)$ with $e$ $\mathsf{val}$, then $e = l$ for some $l \in dom(\Theta)$ with $\Theta(l) = \tau$.*

**Lemma 36.3** (Progress)**.** *If $\Theta \vdash e : \tau$ with $T : \Theta$, then either $(T, e)$ final, or there exists $T' \supseteq T$ and $e'$ such that $(T, e) \mapsto (T', e')$.*

*Proof.* By induction on typing, making use of Lemma 36.2. □

## 36.3 Exercises

Show that the operation `iftagof` with typing rule

$$\frac{\Theta\,\Delta\,\Gamma \vdash e_1 : \mathtt{tagged} \quad \Theta\,\Delta\,\Gamma \vdash e_2 : \mathtt{tag}(\tau_2) \quad \Theta\,\Delta\,\Gamma, x : \tau_2 \vdash e_3 : \tau \quad \Theta\,\Delta\,\Gamma \vdash e_4 : \tau}{\Theta\,\Delta\,\Gamma \vdash \mathtt{iftagof}(e_1; e_2; x.e_3; e_4) : \tau} \tag{36.5}$$

and evaluation rule

$$\frac{(l_1 = l_2) \quad \mathtt{tagwith}[\tau_1](l_1; e_1)\ \mathsf{val}}{(T, \mathtt{iftagof}(\mathtt{tagwith}[\tau_1](l_1; e_1); l_2; x.e_3; e_4)) \mapsto (T, [e_1/x]e_3)} \tag{36.6}$$

is definable.

# Part XIII

# Laziness

# Chapter 37

# Eagerness and Laziness

A fundamental distinction between *eager*, or *strict*, and *lazy*, or *non-strict*, evaluation arises in the dynamic semantics of function, product, sum, and recursive types. This distinction is of particular importance in the context of $\mathcal{L}\{\mu\rightharpoonup\}$, which permits the formation of divergent expressions. Quite often eager and lazy evaluation is taken to be a *language design distinction*, but we argue that it is better viewed as a *type distinction*.

## 37.1 Eager and Lazy Dynamics

According to the methodology outlined in Chapter 10, language features are identified with types. The constructs of the language arise as the introductory and eliminatory forms associated with a type. The static semantics specifies how these may be combined with each other and with other language constructs in a well-formed program. The dynamic semantics specifies how these constructs are to be executed, subject to the requirement of type safety. Safety is assured by the conservation principle, which states that the introduction forms are the values of the type, and the elimination forms are inverse to the introduction forms.

Within these broad guidelines there is often considerable leeway in the choice of dynamic semantics for a language construct. For example, consider the dynamic semantics of function types given in Chapter 13. There we specified the $\lambda$-abstractions are values, and that applications are evaluated according to the following rules:

$$\frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)} \tag{37.1a}$$

$$\frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{e_1(e_2) \mapsto e_1(e_2')} \tag{37.1b}$$

$$\frac{e_2 \text{ val}}{\lambda(x:\tau.e)(e_2) \mapsto [e_2/x]e} \tag{37.1c}$$

The first of these states that to evaluate an application $e_1(e_2)$ we must first of all evaluate $e_1$ to determine what function is being applied. The third of these states that application is inverse to abstraction, but is subject to the requirement that the argument be a value. For this to be tenable, we must also include the second rule, which states that to apply a function, we must first evaluate its argument. This is called the *call-by-value*, or *strict*, or *eager*, evaluation order for functions.

Regarding a $\lambda$-abstraction as a value is inevitable so long as we retain the principle that only closed expressions (complete programs) can be executed. Similarly, it is natural to demand that the function part of an application be evaluated before the function can be called. On the other hand it is somewhat arbitrary to insist that the argument be evaluated before the call, since nothing seems to oblige us to do so. This suggests an alternative evaluation order, called *call-by-name*,[1] or *lazy*, which states that arguments are to be passed unevaluated to functions. Consequently, function parameters stand for computations, not values, since the argument is passed in unevaluated form. The following rules define the call-by-name evaluation order:

$$\frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)} \tag{37.2a}$$

$$\frac{}{\lambda(x:\tau.e)(e_2) \mapsto [e_2/x]e} \tag{37.2b}$$

We omit the requirement that the argument to an application be a value.

This example illustrates some general principles governing the dynamic semantics of a language:

1. The conservation principle demands that the elimination forms be inverse to the introduction forms. The elimination forms associated with a type have a distinguished *principal argument*, which is of the type under consideration, to which the elimination form is inverse.

2. The principal argument of an elimination form is necessarily evaluated to an introduction form, thereby exposing an opportunity for cancellation according to the conservation principle.

[1] For obscure historical reasons.

3. It is more or less arbitrary whether the non-principal arguments to an elimination form are evaluated prior to cancellation.

4. Values of the type have introductory form, but may also be chosen to satisfy further requirements such as insisting that certain sub-expressions also be values.

Let us apply these principles to the product type. First, the sole argument to the elimination forms is, of course, principal, and hence must be evaluated. Second, if the argument is a value, it must be a pair (the only introductory form), and the projections extract the appropriate component of the pair.

$$\frac{\langle e_1, e_2\rangle \ \mathsf{val}}{\mathtt{fst}(\langle e_1, e_2\rangle) \mapsto e_1} \tag{37.3}$$

$$\frac{\langle e_1, e_2\rangle \ \mathsf{val}}{\mathtt{snd}(\langle e_1, e_2\rangle) \mapsto e_1} \tag{37.4}$$

$$\frac{e \mapsto e'}{\mathtt{fst}(e) \mapsto \mathtt{fst}(e')} \tag{37.5}$$

$$\frac{e \mapsto e'}{\mathtt{snd}(e) \mapsto \mathtt{snd}(e')} \tag{37.6}$$

Since there is only one introductory form for the product type, a value of product type must be a pair. But this leaves open whether the components of a pair value must themselves be values or not. The *eager* (or *strict*) semantics, which we gave in Chapter 16, evaluates the components of a pair before deeming it to be a value: specified by the following additional rules:

$$\frac{e_1 \ \mathsf{val} \quad e_2 \ \mathsf{val}}{\langle e_1, e_2\rangle \ \mathsf{val}} \tag{37.7}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle} \tag{37.8}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \mapsto e_2'}{\langle e_1, e_2\rangle \mapsto \langle e_1, e_2'\rangle} \tag{37.9}$$

The *lazy* (or *non-strict*) semantics, on the other hand, deems any pair to be a value, regardless of whether its components are values:

$$\frac{}{\langle e_1, e_2\rangle \ \mathsf{val}} \tag{37.10}$$

There are similar alternatives for sum and recursive types, differing according to whether or not the argument of an injection, or to the introductory half of an isomorphism, is evaluated. There is no choice, however, regarding evaluation of the branches of a case analysis, since each branch binds a variable to the injected value for each case. Incidentally, this explains the apparent restriction on the evaluation of the conditional expression, $\texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2$, arising from the definition of `bool` to be the sum type `unit + unit` as described in Chapter 17 — the "then" and the "else" branches lie within the scope of an (implicit) bound variable, and hence are not eligible for evaluation!

## 37.2 Eagerness and Laziness Via Types

Rather than specify a blanket policy for the eagerness or laziness of the various language constructs, it is more expressive to put this decision into the hands of the programmer by a *type distinction*. That is, we can distinguish types of by-value and by-name functions, and of eager and lazy versions of products, sums, and recursive types.

We may give eager and lazy variants of product, sum, function, and recursive types according to the following chart:

| | Eager | Lazy |
|---|---|---|
| Unit | $\mathbf{1}$ | $\top$ |
| Product | $\tau_1 \otimes \tau_2$ | $\tau_1 \times \tau_2$ |
| Void | $\bot$ | $\mathbf{0}$ |
| Sum | $\tau_1 + \tau_2$ | $\tau_1 \oplus \tau_2$ |
| Function | $\tau_1 \circ\!\!\rightarrow \tau_2$ | $\tau_1 \rightarrow \tau_2$ |

We leave it to the reader to formulate the static and dynamic semantics of these constructs using the following grammar of introduction and elimination forms for the unfamiliar type constructors in the foregoing chart:

| | Introduction | Elimination |
|---|---|---|
| $\mathbf{1}$ | $\bullet$ | *(none)* |
| $\tau_1 \otimes \tau_2$ | $e_1 \otimes e_2$ | $\texttt{let}\,x_1 \otimes x_2\,\texttt{be}\,e\,\texttt{in}\,e'$ |
| $\mathbf{0}$ | *(none)* | $\texttt{abort}_\tau(e)$ |
| $\tau_1 \oplus \tau_2$ | $\texttt{lft}_\tau(e)$, $\texttt{rht}_\tau(e)$ | $\texttt{choose}\,e\,\{\texttt{lft}(x_1) \Rightarrow e_1 \mid \texttt{rht}(x_2) \Rightarrow e_2\}$ |
| $\tau_1 \circ\!\!\rightarrow \tau_2$ | $\lambda^\circ(x\!:\!\tau_1.\,e_2)$ | $\texttt{ap}^\circ(e_1, e_2)$ |

The elimination form for the eager product type uses pattern-matching to recover both components of the pair at the same time. The elimination form for the lazy empty sum performs a case analysis among zero choices, and is therefore tantamount to aborting the computation. Finally, the circle adorning the eager function abstraction and application is intended to suggest a correspondence to the eager product and function types.

The notation is chosen to emphasize certain aspects of the eager and lazy interpretations. Specifically, in the lazy column we have standard nullary and binary products, and a standard function types, whereas in the eager column we have only restricted forms of products and a strict function type. Dually, in the call-by-value column we have standard nullary and binary sum types, whereas in the lazy column we have only restricted forms of these. The distinction between "standard" and "restricted" forms of these types derives from considerations that we can only summarize briefly here. Lazy product types are said to be standard in the sense that the expression $\mathtt{fst}(\langle e_1, e_2\rangle)$ is interchangeable with $e_1$, regardless of whether or not $e_2$ terminates, and similarly $\mathtt{snd}(\langle e_1, e_2\rangle)$ is interchangeable with $e_2$, independently of $e_1$. These conditions fail for strict products (or, more properly, hold only in restricted circumstances), and hence they are said to be restricted, or non-standard. Correspondingly, lazy function types are standard in that $\lambda(x{:}\tau_1.\, e_2)(e_1)$ is interchangeable with $[e_1/x]e_2$, whereas the corresponding property fails for strict function types, again because $e_2$ may not terminate and $x$ may not be relevant to the value of $e_1$. Dually, lazy sums fail to satisfy certain interchangeability properties that hold for strict sums, and consequently they are said to be restricted, or non-standard.

## 37.3 Laziness and Self-Reference

We have seen in Chapter 15 that we may use general recursion at the expression level to define recursive functions. In the presence of laziness we may also define other forms of self-referential expression. For example, consider the so-called lazy natural numbers, which are defined by the recursive type $\mathtt{lnat} = \mu(t.\, \top \oplus t)$. The successor operation for the lazy natural numbers is defined by the equation $\mathtt{lsucc}(e) = \mathtt{fold}(\mathtt{rht}(e))$. Using general recursion we may form the lazy natural number

$$\omega = \mathtt{fix}\, x{:}\mathtt{lnat}\, \mathtt{is}\, \mathtt{lsucc}(x),$$

which consists of an infinite stack of successors!

Of course, one could argue (correctly) that $\omega$ is not a natural number at all, and hence should not be regarded as one. So long as we can distinguish the type `lnat` from the type `nat`, there is no difficulty—$\omega$ is the infinite *lazy* natural number, but it is not an *eager* natural number. But if the distinction is not available, then serious difficulties arise. For example, lazy languages provide only lazy product and sum types, and hence are only capable of defining the lazy natural numbers as a recursive types. In such languages $\omega$ is said to be a "natural number", but only for a non-standard use of the term; the true natural numbers are simply unavailable.

It is a significant weakness of lazy languages is that they provide only a paucity of types. One might expect that, dually, eager languages are similarly disadvantaged in providing only eager, but not lazy types. However, in the presence of function types (the common case), we may encode the lazy types as instances of the corresponding eager types, as we described in the next section.

## 37.4 Suspensions

The essence of lazy evaluation is the suspension of evaluation of certain expressions. For example, the lazy product type suspends evaluation of the components of a pair until they are needed, and the lazy sum type suspends evaluation of the injected value until it is required. To encode lazy types as eager types, then, requires only a means of suspending evaluation of an expression using a *suspension*, or *thunk*.[2]

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

| | | | |
|---|---|---|---|
| Type | $\tau$ | $::=$ | $\texttt{susp}(\tau)$ |
| Expr | $e$ | $::=$ | $\texttt{susp}(e) \mid \texttt{letsusp}(e_1, x.e_2)$ |

Concretely, we use the following notation:

| Abstract Syntax | Concrete Syntax |
|---|---|
| $\texttt{susp}(\tau)$ | $\tau\ \texttt{susp}$ |
| $\texttt{susp}(e)$ | $\texttt{susp}(e)$ |
| $\texttt{letsusp}(e_1, x.e_2)$ | $\texttt{let susp}(x)\ \texttt{be}\ e_1\ \texttt{in}\ e_2$ |

In addition we let $\texttt{force}(e)$ stand for the expression $\texttt{letsusp}(e, x.x)$.

The static semantics of suspensions is given as follows:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{susp}(e) : \texttt{susp}(\tau)} \tag{37.11a}$$

[2]The etymology of this term is uncertain, but its usage persists.

$$\frac{\Gamma \vdash e_1 : \mathtt{susp}(\tau_1) \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{letsusp}(e_1, x.e_2) : \tau_2} \tag{37.11b}$$

The following typing rule is derived:

$$\frac{\Gamma \vdash e : \mathtt{susp}(\tau)}{\Gamma \vdash \mathtt{force}(e) : \tau} \tag{37.12}$$

The dynamic semantics is given by the following rules:

$$\frac{}{\mathtt{susp}(e)\ \mathsf{val}} \tag{37.13a}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{letsusp}(e_1, x.e_2) \mapsto \mathtt{letsusp}(e_1', x.e_2)} \tag{37.13b}$$

$$\frac{e \mapsto e'}{\mathtt{letsusp}(\mathtt{susp}(e), x.e_2) \mapsto \mathtt{letsusp}(\mathtt{susp}(e'), x.e_2)} \tag{37.13c}$$

$$\frac{e\ \mathsf{val}}{\mathtt{letsusp}(\mathtt{susp}(e), x.e_2) \mapsto [e/x]e_2} \tag{37.13d}$$

Suspensions in this form are implementable in terms of strict functions, using the following definitions:

$$\tau\,\mathtt{susp} = \mathbf{1} \circ\!\!\to \tau \tag{37.14}$$

$$\mathtt{susp}(e) = \lambda^{\circ}(\_{:}\mathbf{1}.\, e) \tag{37.15}$$

$$\mathtt{let}\,\mathtt{susp}(x)\,\mathtt{be}\,e_1\,\mathtt{in}\,e_2 = \mathtt{ap}^{\circ}((\lambda^{\circ}(x{:}\tau.e_2)), (\mathtt{ap}^{\circ}(e_1, \bullet))) \tag{37.16}$$

It is easy to check that the static semantics of suspensions is derivable, and that the dynamic semantics is weakly simulated, under these definitions.

We may use suspensions to encode the lazy type constructors as instances of the corresponding eager type constructors as follows:

$$\top = \mathbf{1} \tag{37.17a}$$

$$\langle\rangle = \bullet \tag{37.17b}$$

$$\tau_1 \times \tau_2 = \tau_1\,\mathtt{susp} \otimes \tau_2\,\mathtt{susp} \tag{37.18a}$$

$$\langle e_1, e_2\rangle = \mathtt{susp}(e_1) \otimes \mathtt{susp}(e_2) \tag{37.18b}$$

$$\mathtt{fst}(e) = \mathtt{let}\, x \otimes \_\,\mathtt{be}\, e\,\mathtt{in}\,\mathtt{force}(x) \tag{37.18c}$$

$$\mathtt{snd}(e) = \mathtt{let}\, \_ \otimes y\,\mathtt{be}\, e\,\mathtt{in}\,\mathtt{force}(y) \tag{37.18d}$$

$$\mathbf{0} = \bot \tag{37.19a}$$
$$\mathtt{abort}_\tau(e) = \mathtt{abort}_\tau\, e \tag{37.19b}$$

$$\tau_1 \oplus \tau_2 = \tau_1\,\mathtt{susp} + \tau_2\,\mathtt{susp} \tag{37.20a}$$
$$\mathtt{lft}(e) = \mathtt{inl}(\mathtt{susp}(e)) \tag{37.20b}$$
$$\mathtt{rht}(e) = \mathtt{inr}(\mathtt{susp}(e)) \tag{37.20c}$$

$$\begin{aligned}&\mathtt{choose}\, e\, \{\mathtt{lft}(x_1) \Rightarrow e_1 \mid \mathtt{rht}(x_2) \Rightarrow e_2\}\\ &\quad = \mathtt{case}\, e\, \{\mathtt{inl}(y_1) \Rightarrow [\mathtt{force}(y_1)/x_1]e_1 \mid \mathtt{inr}(y_2) \Rightarrow [\mathtt{force}(y_2)/x_2]e_2\}\end{aligned} \tag{37.20d}$$

$$\tau_1 \rightharpoonup \tau_2 = \tau_1\,\mathtt{susp} \circ\!\!\rightarrow \tau_2 \tag{37.21a}$$
$$\lambda(x{:}\tau_1.\, e_2) = \lambda^\circ(x{:}\tau_1\,\mathtt{susp}.\, [\mathtt{force}(x)/x]e_2) \tag{37.21b}$$
$$e_1(e_2) = \mathtt{ap}^\circ(e_1, \mathtt{susp}(e_2)) \tag{37.21c}$$

In the case of lazy case analysis and call-by-name functions we replace occurrences of the bound variable, $x$, with $\mathtt{force}(x)$ to recover the value of the suspension bound to $x$ whenever it is required. Note that $x$ may occur in a lazy context, in which case $\mathtt{force}(x)$ is delayed. In particular, expressions of the form $\mathtt{susp}(\mathtt{force}(x))$ may be safely replaced by $x$, since forcing the former computation simply forces $x$.

## 37.5 Exercises

# Chapter 38

# Lazy Evaluation

*Lazy evaluation* refers to a variety of concepts that seek to avoid evaluation of an expression unless its value is needed, and to share the results of evaluation of an expression among all uses of its, so that no expression need be evaluated more than once. Within this broad mandate, various forms of laziness are considered.

One is the *call-by-need* evaluation strategy for functions. This is a refinement of the *call-by-name* semantics described in Chapter 37 in which arguments are passed unevaluated to functions so that it is only evaluated if needed, and, if so, the value is shared among all occurrences of the argument in the body of the function.

Another is the *lazy* evaluation strategy for data structures, including formation of pairs, injections into summands, and recursive folding. The decisions of whether to evaluate the components of a pair, or the argument to an injection or fold, are independent of one another, and of the decision whether to pass arguments to functions in unevaluated form.

A third aspect of laziness is the ability to form *recursive values*, including as a special case recursive functions. Using general recursion we can create self-referential expressions, but these are only useful if the self-referential expression can be evaluated without needing its own values. Function abstractions provide one such mechanism, but so do lazy data constructors.

These aspects of laziness are often consolidated into a programming language with call-by-need function evaluation, lazy data structures, and unrestricted uses of recursion. Such languages are called *lazy languages*, because they impose the lazy evaluation strategy throughout. These are to be contrasted with *strict languages*, which impose an eager evaluation strategy throughout. This leads to a sense of opposition between two camps, but ex-

perience has shown that this is neither necessary nor desirable. For example, in a lazy language the successor operation on natural numbers is lazy, so that it is possible to form "natural numbers" such as $\omega = \mathtt{fix}\, x\mathtt{:nat\, is\, s}(x)$ consisting of an infinite stack of successors. Obviously $\omega$ is not a natural number in the ordinary sense of the word, but we cannot avoid it in a lazy language, and hence the familiar type of natural numbers is simply unavailable. On the other hand, in an eager language the successor is evaluated eagerly, so the recursive expression above simply engenders an infinite loop when evaluated. It is impossible in an eager language to obtain the unnatural number $\omega$.

Rather than accept these as consequences of language design, it is preferable to put the distinction in the hands of the programmer by introducing a type of suspended computations whose evaluation is memoized so that they are only ever evaluated once. The ambient evaluation strategy remains eager, but we now have a *value* representing an *unevaluated* expression. Moreover, we may confine self-reference to suspensions to avoid the pathologies of laziness while permitting self-referential data structures to be programmed.

## 38.1 Call-By-Need

The distinguishing feature of call-by-need, as compared to call-by-name, is that it records in memory the bindings of all variables so that when the binding of a variable is first needed, it is evaluated and the result is re-bound to that variable. Subsequent demands for the binding simply retrieve the stored value without having to repeat the computation. Of course, if the binding is never needed, it is never evaluated, consistently with the call-by-name semantics.

We will give the dynamic semantics of call-by-need using a transition system with states of the form $(M, e)$, where $M$ is a memory mapping variables to open expressions and $e$ is an open expression. States must satisfy the invariant that a free variable of $e$ or any binding in $M$ must lie within the domain of $M$.

An initial state of the transition system has the form $(\emptyset, e)$, where $e$ is a closed expression. A final state has the form $(M, e)$, where $e$ is an open value — but note well that variables themselves are not values! The transition judgement is inductively defined by the following rules:

$$\frac{e\ \mathsf{val}}{(M[x=e], x) \mapsto (M[x=e], e)} \tag{38.1a}$$

$$\frac{(M[x=\bullet], e) \mapsto (M'[x=\bullet], e')}{(M[x=e], x) \mapsto (M'[x=e'], x)} \tag{38.1b}$$

$$\frac{(M, e_1) \mapsto (M', e_1')}{(M, e_1(e_2)) \mapsto (M', e_1'(e_2))} \tag{38.1c}$$

$$\frac{x \# M}{(M, \lambda(x:\tau.e)(e_2)) \mapsto (M[x=e_2], e)} \tag{38.1d}$$

We omit here the presentation of the rules for the other constructs, which follow a similar pattern.

The crucial rules are those for variables, since application merely binds the unevaluated argument to the parameter of the function before evaluating its body. If the binding of a variable is a value, then that value is returned immediately. Otherwise, the binding must be evaluated to determine its value, which replaces the binding for future reference. This is accomplished in the second rule by performing a transition on the binding, $e$, of the variable, $x$, then replacing the binding with the result, $e'$. During evaluation of $e$ the binding of $x$ is replaced by a special construct, called a *black hole*, which ensures that evaluation would be "stuck" should the binding of $x$ ever be required during evaluation of $e$. (Observe that since the black hole is not a value, and admits no transitions, a state of the form $(M[x=\bullet], x)$ is "stuck".) The main reason to replace the binding of $x$ with a black hole is primarily to do with recursion, which will be discussed in the next section. It *is* important, however, that there be a binding for $x$ in the memory while evaluating its contents, so as to ensure that any "fresh" variables that are added to the memory during this evaluation are different from $x$ so as to avoid confusion.

Type safety for the call-by-need interpretation is proved by methods similar to those used to prove safety for mutable references (see Chapter 34). However, unlike the situation with reference cells, no cyclic dependencies are possible. Moreover, we wish to show that stuck states such as $(M[x=\bullet], x)$ do not arise during evaluation. We define the judgement $M : \Gamma$ iff $M(x) = e$ implies that $x : \tau$ occurs in $\Gamma$ for some $\tau$ such that $\Gamma \vdash e : \tau$. We then define the judgement $(M, e)$ ok iff the following three conditions are met:

1. There exists $\Gamma$ and $\tau$ such that $M : \Gamma$ and $\Gamma \vdash e : \tau$.
2. If $M(x) = e$, then $x \# e$.
3. If $M(y) = \bullet$, then $y \# e$ and $y \# M(x)$ for every $x$.

The first condition captures the typing invariants, the second rules out self-reference, and the third suffices to ensure that evaluation does not get stuck due to black holes.

**Theorem 38.1.** *1. If* $(M,e)$ ok *and* $(M,e) \mapsto (M',e')$*, then* $(M',e')$ ok*.*

*2. If* $(M,e)$ ok*, then either* $(M,e)$ *final, or there exists* $M'$ *and* $e'$ *such that* $(M,e) \mapsto (M',e')$*.*

The first part is proved by rule induction on the definition of the transition judgement. The second part is proved by induction on the definition of $(M,e)$ ok, treating the derivations of typing for $e$ and the bindings in $M$ as sub-derivations.

## 38.2 General Recursion

It is interesting to examine the dynamic semantics of general recursion in a call-by-need setting. The most obvious approach is to simply mimic the semantics given in Chapter 15:

$$\frac{}{(M, \mathtt{fix}\, x{:}\tau\, \mathtt{is}\, e) \mapsto (M, [\mathtt{fix}\, x{:}\tau\, \mathtt{is}\, e/x]e)}\ . \tag{38.2}$$

Substitution for the recursive variable may introduce many different copies of the recursive expression, each of which is re-evaluated whenever it is used. It would be more in the spirit of call-by-need to share these computations among the copies, which may be formalized by the following rules.

$$\frac{x \# M}{(M, \mathtt{fix}\, x{:}\tau\, \mathtt{is}\, e) \mapsto (M[x = e], x)}\ . \tag{38.3}$$

If the computation variable, $x$, is free in $e$, then evaluation of $e$ may well require the binding of $x$, in which case evaluation gets stuck with a state of the form $(M[x = \bullet], x)$. For example, $\mathtt{fix}\, x{:}\tau\, \mathtt{is}\, x$ gets stuck in precisely this manner. Such stuck states correspond to infinite loops under the call-by-name semantics. We may regard this as a "checked error" for certain forms of non-termination, namely those that result from an infinite regress of self-reference.

In the presence of recursion it is no longer possible to rule out self-reference, or dependence of evaluation on a black hole. Therefore we must define $M : \Gamma$ in a manner similar to Chapter 34 to require that $\Gamma \vdash e : \tau$ whenever $M(x) = e$ and $\Gamma(x) = \tau$. Moreover, it is essential to define $\bullet : \tau$,

since a variable of any type may be bound to a black hole during evaluation. With this in mind it is relatively straightforward to prove preservation and progress. The statement of progress must be weakened, however, to regard states of the form $(M[x = \bullet], x)$ as checked errors, rather than stuck states, since typing does not preclude circular dependencies.

## 38.3 Lazy Sums and Products

Call-by-need evaluation addresses only one aspect of laziness, namely deferring evaluation of function arguments until they are needed, and sharing the value among all other uses of it. Other aspects of laziness pertain to product, sum, and recursive types, whose introductory forms may be given a lazy interpretation, in which case it is also sensible to consider a "need" semantics that avoids re-computation whenever possible.

Let us consider a "need" semantics for product types corresponding to the lazy interpretation of pairing. The main idea is to share evaluation of the components of the pair among all uses of that pair. This is achieved by the following rules:

$$\frac{}{\langle x_1, x_2\rangle \ \mathsf{val}} \tag{38.4a}$$

$$\frac{x_1 \# M \quad x_2 \# M}{(M, \langle e_1, e_2\rangle) \mapsto (M[x_1 = e_1][x_2 = e_2], \langle x_1, x_2\rangle)} \tag{38.4b}$$

$$\frac{(M, e) \mapsto (M', e')}{(M, \mathtt{fst}(e)) \mapsto (M', \mathtt{fst}(e'))} \tag{38.4c}$$

$$\frac{}{(M, \mathtt{fst}(\langle x_1, x_2\rangle)) \mapsto (M, x_1)} \tag{38.4d}$$

$$\frac{(M, e) \mapsto (M', e')}{(M, \mathtt{snd}(e)) \mapsto (M', \mathtt{snd}(e'))} \tag{38.4e}$$

$$\frac{}{(M, \mathtt{snd}(\langle x_1, x_2\rangle)) \mapsto (M, x_2)} \tag{38.4f}$$

In this semantics a pair is considered a value only if its arguments are variables, which are introduced when the pair is created. The projections evaluate to one variable or the other, inducing a demand for that component of the pair, causing it to be evaluated and stored in memory. This ensures that another occurrence of the same projection of the same pair will yield the same value without having to recompute it.

## 38.4 Suspension Types

As suggested in Chapter 37 it is more economical, and more expressive, to consolidate all aspects of laziness into a single type, $\texttt{susp}(\tau)$, of memoized, self-referential suspensions. The expression forms associated with this type are given by the following grammar:

$$\text{Expr} \quad e \quad ::= \quad \texttt{susp[}\tau\texttt{]}(x\texttt{.}e) \mid \texttt{letsusp}(e_1, x\texttt{.}e_2)$$

The introductory form is self-referential so as to permit formation of recursive suspensions. We correspondingly drop general recursion from the language, limiting it to values of suspension type.

The static semantics of these constructs is given by the following typing rules:

$$\frac{\Gamma, x : \texttt{susp}(\tau) \vdash e : \tau}{\Gamma \vdash \texttt{susp[}\tau\texttt{]}(x\texttt{.}e) : \texttt{susp}(\tau)} \tag{38.5a}$$

$$\frac{\Gamma \vdash e_1 : \texttt{susp}(\tau_1) \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{letsusp}(e_1, x\texttt{.}e_2) : \tau_2} \tag{38.5b}$$

The dynamic semantics of suspensions is given by a transition system that is reminiscent of that used for call-by-need. Indeed, the same principles of memoization and self-reference apply, *albeit* restricted to the type of suspensions. One important difference, however, is that variables are now regarded as values! Rather than implicitly replacing variables by their values when encountered, we now explicitly force evaluation of a suspension to retrieve its value.

$$\frac{x \# M}{(M, \texttt{susp[}\tau\texttt{]}(x\texttt{.}e)) \mapsto (M[x = e], x)} \tag{38.6a}$$

$$\frac{(M, e_1) \mapsto (M', e_1')}{(M, \texttt{letsusp}(e_1, x\texttt{.}e_2)) \mapsto (M', \texttt{letsusp}(e_1', x\texttt{.}e_2))} \tag{38.6b}$$

$$\frac{e \ \mathsf{val}}{(M[x = e], \texttt{letsusp}(x, y\texttt{.}e_2)) \mapsto (M[x = e], [e/y]e_2)} \tag{38.6c}$$

$$\frac{(M[x = \bullet], e) \mapsto (M[x = \bullet], e')}{(M[x = e], \texttt{letsusp}(x, y\texttt{.}e_2)) \mapsto (M'[x = e'], \texttt{letsusp}(x, y\texttt{.}e_2))} \tag{38.6d}$$

Type safety for suspension types may be proved by means similar to that for call-by-need. Care must be taken in the definition of the judgment

$M : \Gamma$ to account for variables in memory having suspension type. Specifically, we require that $\Gamma \vdash e : \tau$ whenever $M(x) = e$ and $\Gamma(x) = \tau\,\mathtt{susp}$. Moreover, we deem that $\bullet : \tau$ for any type $\tau$, because the static semantics does not preclude circular dependencies.

## 38.5 Exercises

# Part XIV

# Parallelism

# Chapter 39

# Speculative Parallelism

The semantics of call-by-need given in Chapter 38 suggests opportunities for *speculative parallelism*. Evaluation of a delayed binding is initiated as soon as the binding is created, executing simultaneously with the evaluation of the body. Should the variable ever be needed, evaluation of the body synchronizes with the concurrent evaluation of the binding, and proceeds only once the value is available. This form of parallelism is called speculative, because the value of the binding may never be needed, in which case the resources required for its evaluation are wasted. However, in some situations there are available computing resources that would otherwise be wasted, and which can be usefully employed for speculative evaluation.

There is also a speculative version of suspensions, called *futures*, which behave in the same manner, except that the synchronization points are explicit in the form of calls to force the suspension. The suspended computation can be executed in parallel on the hypothesis that its value will eventually be needed to proceed.

## 39.1 Speculative Execution

An interesting variant of the call-by-need semantics is obtained by relaxing the restriction that the bindings of variables be evaluated only once they are needed. Instead, we may permit a step of execution of the binding of any variable to occur at any time. Specifically, we replace the second variable rule given in Section 38.1 on page 302 by the following general rule:

$$\frac{(M[y=\bullet], e) \mapsto (M[y=\bullet], e')}{(M[y=e], e_0) \mapsto (M[y=e'], e_0)} \tag{39.1}$$

This rule permits any variable binding to be chosen at any time as the focus of attention for the next evaluation step. The first variable rule remains as-is, so that, as before, a variable may be evaluated only after the value of its binding has been determined.

This semantics is said to be *non-deterministic* because the transition relation is no longer a partial function on states. That is, for a given state $(M, e)$, there may be many different states $(M', e')$ such that $(M, e) \mapsto (M', e')$, precisely because the foregoing rule permits us to shift attention to any location in memory at any time. The rules abstract away from the specifics of how such "context switches" might be scheduled, permitting them to occur at any time so as to be consistent with any scheduling strategy. In this sense non-determinism models parallel execution by permitting the individual steps of a complete computation to be interleaved in an arbitrary manner.

The non-deterministic semantics is said to be *speculative*, because it permits evaluation of any suspended expression at any time, without regard to whether its value is needed to determine the overall result of the computation. In this sense it is contrary to the spirit of call-by-need, since it may perform work that is not strictly necessary. The benefit of speculation is that it leads to a form of parallel computation, called *speculative parallelism*, which seeks to exploit computing resources that would otherwise be left idle. Ideally one should only use processors to compute results that are needed, but in some situations it is difficult to make full use of available resources without resorting to speculation.

Just as with call-by-need, there is also a speculative version of suspensions, which are called *futures*. Conceptually, a delayed computation in memory is evaluated speculatively "in parallel" while computation along the main thread proceeds. When a suspension is forced, evalation of the main thread is blocked until the suspension has been evaluated, at which point the value is propagated to the main thread and execution proceeds. The semantics of futures is a straightforward modification to the semantics of suspensions given in Chapter 38, modified to permit non-deterministic context switches during execution to evaluate suspended computations.

## 39.2 Speculative Parallelism

The non-deterministic semantics given in Section 39.1 on the previous page captures the idea of speculative execution, but addresses parallelism only indirectly, by avoiding specification of when the focus of evaluation may

shift from one suspended expression to another. Thus the semantics is specified from the point of view of an omniscient observer who sequentializes the parallel execution into a sequence of atomic steps. No paricular sequentialization is enforced; rather, all possible sequentializations are derivable from the rules.

Another approach is to formulate directly a parallel execution semantics that permits multiple expressions to be evaluated simultaneously. To avoid overspecification the parallel semantics imposes no bound on the degree of parallelism. In practice, of course, one has only limited computing resources available, so it is necessary to impose a scheduling discipline that multiplexes parallel computations onto the available processing elements. In the semantics we avoid imposing any particular scheduling regimen, since the number of processors available, and any constraints on their use, are highly sensitive to the execution environment, and are therefore not to be considered properties of the language, but rather of its implementation.

Speculative parallelism may be formalized by introducing an additional judgement form, $M \mapsto_{par} M'$, which expresses one step of parallel evaluation of any number of suspended expressions in memory. Each step of the computation consists of essentially one step of the abstract machine given in Chapter 38, but modified to remove scheduling restrictions. Let us write $(M, e) \mapsto_{seq} (M', e')$ for the sequential transition relation defined in Chapter 38, but modified so that states of the form $(M[x = e], x)$, where $e$ is not a value, are *blocked*, so that no transition is possible. All other transitions are as before. In particular, if $e$ is a value, then we may always make the transition

$$(M[x = e], x) \mapsto_{seq} (M[x = e], e).$$

The main reason to make this change to the sequential semantics is that even if $e_0$ is restricted to be the variable $x$, Rule (39.1) amounts to a decision to schedule evaluation of $e$ ahead of the completion of $e_0$. We prefer instead to confine scheduling to the parallel semantics, which nondeterministically chooses which suspended expressions to execute at any one time.

$$\frac{(M[x_1 = \bullet] \ldots [x_k = \bullet], e_i) \mapsto_{seq} (M_i[x_1 = \bullet] \ldots [x_k = \bullet], e_i') \quad (1 \leq i \leq n)}{M[x_1 = e_1] \ldots [x_k = e_k] \mapsto_{par} M'[x_1 = e_1'] \ldots [x_k = e_k']} \tag{39.2}$$

(The choice of memory, $M'$, in the result of the execution step, and an important side condition governing the inference, will be specified shortly.)

Observe that each step of the parallel semantics may correspond to any number of steps of the sequential semantics, limited only by the number of

evaluation expressions in memory. By our restriction on the sequential semantics, no progress can be made on any expression that requires the value of a variable whose binding is not a value. It is therefore not necessary to "black hole" the variables whose bindings are being evaluated, but we do so to emphasize the similarity with the sequential semantics.

The same restriction on the sequential semantics also implies that no sequential step can modify the binding of an allocated suspension. However, a sequential step may allocate *new* suspensions with names that lie apart from $M$ and the $x_i$'s. It follows that each $M_i$ has the form $M \otimes M_i'$ for some memory $M_i'$ extending the memory $M$.[1] The result memory, $M'$, is the combination $M \otimes M_1' \otimes \cdots \otimes M_k'$, which consolidates the newly allocated suspensions from each of the individual steps as an extension to the memory $M$.

There is one technical difficulty, however: there is no guarantee that the domains of the $M_i'$ are disjoint! That is, two independent executions might seek to allocate the same memory cell, each for its own purpose, resulting in a collision. This is a well-known problem in parallel computing—the parallel steps must somehow synchronize their allocation of memory so as to avoid interference. There are many ways to achieve this in practice. For specification purposes we will simply impose the requirement that the domains of each of the newly allocated memories $M_i'$ be disjoint from one another. This may always be achieved by a suitable choice of variables, of which we assume there are an infinite supply.

## 39.3 Exercises

[1] If $M_1$ and $M_2$ are two memories with disjoint domain, we define the memory $M_1 \otimes M_2$ so that it maps $x \in dom(M_1)$ to $M_1(x)$ and $x \in dom(M_2)$ to $M_2(x)$ and is undefined otherwise.

# Chapter 40

# Implicit Parallelism

In this chapter we study two languages with implicit parallelism, one based on product types, the other based on a type of vectors (tuples of values of unbounded size). In the former case the source of parallelism is the simultaneous evaluation of the components of a tuple, whereas in the latter parallelism arises from the simultaneous computation of the components of a vector.

Parallelism arises naturally in an eager, effect-free language. In such a language it is not possible to determine the order of evaluation of the components of a data structure. This is in sharp contrast to effect-ful languages, for then the order of evaluation, or the use of parallelism, is visible to the programmer. Indeed, dependence on the evaluation order must be carefully guarded against to ensure that the outcome is determinate.

## 40.1 Tuple Parallelism

We begin by considering a parallel semantics for tuples according to which all components of a tuple are evaluated simultaneously. For simplicity we consider only pairs, but the ideas generalize in a straightforward manner to tuples of any size. Since the "widths" of tuples are specified statically as part of their type, the amount of parallelism that can be induced in any one step is bounded by a static constant. In Section 40.3 on page 320 we will extend this to permit a statically unbounded degree of parallelism.

To facilitate comparison, we will consider two operational semantics for products, the *sequential* and the *parallel*. The sequential semantics is as in Chapter 16. However, we now write $e \mapsto_{seq} e'$ for the transition relation to stress that this is the sequential semantics. The sequential evaluation

rules for pairs are as follows:

$$\frac{e_1 \mapsto_{seq} e_1'}{\texttt{pair}(e_1, e_2) \mapsto_{seq} \texttt{pair}(e_1', e_2)} \tag{40.1a}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto_{seq} e_2'}{\texttt{pair}(e_1, e_2) \mapsto_{seq} \texttt{pair}(e_1, e_2')} \tag{40.1b}$$

The parallel semantics is similar, except that we evaluate *both* components of a pair *simultaneously* whenever this is possible. This leads to the following rules:

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2 \mapsto_{par} e_2'}{\texttt{pair}(e_1, e_2) \mapsto_{par} \texttt{pair}(e_1', e_2')} \tag{40.2a}$$

$$\frac{e_1 \mapsto_{par} e_1' \quad e_2\ \mathsf{val}}{\texttt{pair}(e_1, e_2) \mapsto_{par} \texttt{pair}(e_1', e_2)} \tag{40.2b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto_{par} e_2'}{\texttt{pair}(e_1, e_2) \mapsto_{par} \texttt{pair}(e_1, e_2')} \tag{40.2c}$$

The first rule corresponds to a simultaneous step of evaluation on both components of a pair. The second two permit the evaluation of one component to complete before the evaluation of the other.

When presented two semantics for the same language, it is natural to ask whether they are equivalent. They are, in the sense that both semantics deliver the same value for any expression. This is the precise statement of what we mean by "implicit parallelism".

**Lemma 40.1.** *1. If* $\texttt{pair}(e_1, e_2) \mapsto_{seq}^* e'\ \mathsf{val}$, *then* $e' = \texttt{pair}(e_1', e_2')$ *with* $e_1 \mapsto_{seq}^* e_1'\ \mathsf{val}$ *and* $e_2 \mapsto_{seq}^* e_2'\ \mathsf{val}$.

*2. If* $\texttt{pair}(e_1, e_2) \mapsto_{par}^* e'\ \mathsf{val}$, *then* $e' = \texttt{pair}(e_1', e_2')$ *with* $e_1 \mapsto_{par}^* e_1'\ \mathsf{val}$ *and* $e_2 \mapsto_{par}^* e_2'\ \mathsf{val}$.

**Lemma 40.2.** *1. If* $e'' \mapsto_{par}^* e'\ \mathsf{val}$ *and* $e \mapsto_{seq} e''$, *then* $e \mapsto_{par}^* e'$.

*2. If* $e'' \mapsto_{seq}^* e'\ \mathsf{val}$ *and* $e \mapsto_{par} e''$, *then* $e \mapsto_{seq}^* e'$.

*Proof.* For example, suppose we have

$$\begin{aligned} \texttt{pair}(e_1, e_2) &\mapsto_{seq} \texttt{pair}(e_1'', e_2) && (40.3)\\ &\mapsto_{par}^* e'\ \mathsf{val}, && (40.4) \end{aligned}$$

where $e_1 \mapsto_{seq} e_1''$. By Lemma 40.1 on the preceding page $e' = \texttt{pair}(e_1', e_2')$, where $e_1'' \mapsto_{par}^* e_1'$ val and $e_2 \mapsto_{par}^* e_2'$ val. By induction we have $e_1 \mapsto_{par}^* e_1'$ val, and hence that $\texttt{pair}(e_1, e_2) \mapsto_{par}^* \texttt{pair}(e_1', e_2')$.

As an example in the other direction, suppose that we have

$$\texttt{pair}(e_1, e_2) \mapsto_{par} \texttt{pair}(e_1'', e_2'') \tag{40.5}$$
$$\mapsto_{seq}^* e' \text{ val}, \tag{40.6}$$

where $e_1 \mapsto_{par} e_1''$ and $e_2 \mapsto_{par} e_2''$. By Lemma 40.1 on the facing page $e' = \texttt{pair}(e_1', e_2')$ with $e_1'' \mapsto_{seq}^* e_1'$ and $e_2'' \mapsto_{seq}^* e_2'$. By induction we have $e_1 \mapsto_{seq}^* e_1'$ and $e_2 \mapsto_{seq}^* e_2'$, from which the result follows by applying the rules for sequential evaluation of pairs. □

**Theorem 40.3** (Implicit Parallelism)**.** *If $e$ and $e'$ are closed expressions of the same type, $e \mapsto_{seq}^* e'$ val iff $e \mapsto_{par}^* e'$ val.*

*Proof.* In each direction the proof proceeds by induction on the derivation of the multistep transition. The result is immediate when $e$ is $e'$, and follows from Lemma 40.2 on the preceding page in the inductive case. □

One important consequence of this theorem is that parallelism is *semantically invisible*: whether we use parallel or sequential evaluation of pairs, the result is the same. Consequently, parallelism may safely be left *implicit*, at least as far as *correctness* is concerned. However, as one might expect, parallelism effects the *efficiency* of programs.

## 40.2 Work and Depth

An operational semantics for a language induces a measure of time complexity for expressions, namely the number of steps required to evaluate that expression to a value. The *sequential complexity* of an expression is its time complexity relative to the sequential semantics; the *parallel complexity* is its time complexity relative to the parallel semantics. These can, in general, be quite different. Consider, for example, the following naïve implementation of the Fibonacci sequence:

```
fun fib (n:int):int is
  ifz(n, 1, n'.ifz(n', 1, n''.plus(fib n', fib n'')))
```

where `plus` is the evident function of type $\texttt{nat} \times \texttt{nat} \to \texttt{nat}$.

The sequential complexity of $\texttt{fib}(n)$ is $O(2^n)$, whereas the parallel complexity of the same expression is $O(n)$. The reason is that each recursive call

spawns two further recursive calls which, if evaluated sequentially, lead to an exponential number of steps to complete. However, if the two recursive calls are evaluated in parallel, then the number of parallel steps to completion is bounded by $n$, since $n$ is decreased by 1 or 2 on each call. In either case the same number of arithmetic operations is performed. The difference is solely whether or not they are performed simultaneously.

This leads to the important concepts of *work* and *depth*. The *work* of an expression is the total number of primitive instruction steps required to evaluate it to a value. Since each of the rules defining the sequential semantics have at most one transition premise, one step of the sequential semantics represents the execution of one instruction. It follows that the sequential complexity coincides with the overall work required. For example, the work required to evaluate `fib(`$n$`)` is $O(2^n)$.

The *depth* of an expression is the length of the longest chain of dependencies among the evaluation steps in the program. A dependency between two expressions arises whenever the value of one expression depends on the value of another. In the Fibonacci example the two recursive calls have no dependency between them, but the function itself depends on both recursive calls in that it cannot return until both calls have returned. Since the rules of the parallel semantics specify the simultaneous evaluation of the components of a pair, the erecursive calls are evaluated simultaneously, but must complete before the function returns a value. This follows precisely the dependency relationships in the code, and hence the parallel complexity coincides with the depth of the computation. For example, the depth of the expression `fib(`$n$`)` is $O(n)$.

With this in mind, the cost semantics introduced in Chapter 9 may be extended to account for parallelism by specifying both the work and the depth of evaluation. The judgements of the parallel cost semantics have the form $e \Downarrow^{w,d} v$, where $w$ is the work and $d$ the depth. For all cases but evaluation of pairs the work and the depth track one another. The rule for pairs is as follows:

$$\frac{e_1 \Downarrow^{w_1,d_1} v_1 \quad e_2 \Downarrow^{w_2,d_2} v_2}{\texttt{pair}(e_1,e_2) \Downarrow^{w_1+w_2,\max(d_1,d_2)} \texttt{pair}(v_1,v_2)} \tag{40.7}$$

The remaining rules are easily derived from the sequential cost semantics, with both work and depth being combined additively at each step.

The correctness of the cost semantics states that the work and depth costs are consistent with the sequential and parallel complexity, respectively, of the expression.

**Theorem 40.4.** *For any closed, well-typed expression $e$, $e \Downarrow^{w,d} v$ iff $e \mapsto^{w}_{seq} v$ and $e \mapsto^{d}_{par} v$.*

*Proof.* From left to right, we proceed by induction on the cost semantics. For example, we must show that if $e_1 \mapsto^{d_1}_{par} v_1$ and $e_2 \mapsto^{d_2}_{par} v_2$, then

$$\texttt{pair}(e_1, e_2) \mapsto^{d}_{par} \texttt{pair}(v_1, v_2),$$

where $d = \max(d_1, d_2)$. Suppose that $d = d_2$, and let $d' = d - d_1$ (the case $d = d_1$ is handled similarly). We have $e_1 \mapsto^{d_1}_{par} v_1$ and $e_2 \mapsto^{d_1}_{par} e_2' \mapsto^{d'}_{par} v_2$. It follows that

$$\begin{aligned} \texttt{pair}(e_1, e_2) &\mapsto^{d_1}_{par} \texttt{pair}(v_1, e_2') && (40.8)\\ &\mapsto^{d'}_{par} \texttt{pair}(v_1, v_2). && (40.9) \end{aligned}$$

For the converse, we proceed by considering work and depth costs separately. For work, we proceed as in Chapter 9. For depth, it suffices to show that if $e \mapsto_{par} e'$ and $e' \Downarrow^{d} v$, then $e \Downarrow^{d+1} v$.[1] For example, suppose that $\texttt{pair}(e_1, e_2) \mapsto_{par} \texttt{pair}(e_1', e_2')$, with $e_1 \mapsto_{par} e_1'$ and $e_2 \mapsto_{par} e_2'$. Since $\texttt{pair}(e_1', e_2') \Downarrow^{d} v$, we must have $v = \texttt{pair}(v_1, v_2)$, $d = \max(d_1, d_2)$ with $e_1' \Downarrow^{d_1} v_1$ and $e_2' \Downarrow^{d_2} v_2$. By induction $e_1 \Downarrow^{d_1+1} v_1$ and $e_2 \Downarrow^{d_2+1} v_2$ and hence $\texttt{pair}(e_1, e_2) \Downarrow^{d+1} \texttt{pair}(v_1, v_2)$, as desired. □

This theorem relates the work and depth of an expression, which are given by the cost semantics, to the number of transitions required to drive the expression to a value using the sequential and parallel transition systems, respectively. This correspondence explains why the work and depth for evaluation of a pair are $w_1 + w_2$ and $\max(d_1, d_2)$, respectively. But one may argue that these costs are slightly unrealistic, since they do not account for the memory allocation implicit in the creation of a pair, nor for the "fork" and "join" that are implicit in the parallel evaluation of the components of the pair. These issues can be brought out more clearly by refining the transition semantics. For example, to account for allocation, we might distinguish the expression $\texttt{pair}(e_1, e_2)$, which is a primitive operation that allocates a pair, from the expression $\texttt{pr}(e_1, e_2)$, where $e_1$ and $e_2$ are values, which represents the pair once it has been allocated. To model this setup, we add the rule

$$\frac{e_1 \text{ val} \quad e_2 \text{ val}}{\texttt{pair}(e_1, e_2) \mapsto_{seq} \texttt{pr}(e_1, e_2)} \tag{40.10}$$

[1] The work component of the cost is suppressed here for the sake of clarity.

and similarly for the parallel transition semantics. Correspondingly, a pair $\texttt{pair}(e_1,e_2)$ is never a value; instead, we have the rule

$$\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\texttt{pr}(e_1,e_2)\ \mathsf{val}} \tag{40.11}$$

Moreover, we regard the expression $\texttt{pr}(e_1,e_2)$ is an "internal" expression of the semantics, which may not be written down as part of a program, but which only arises in the course of evaluation.

Using this model, the cost assignments for evaluation of a pair are revised so that the work is given by the equation $w = w_1 + w_2 + 1$ and the depth is given by $d = \max(d_1,d_2) + 1$ to account for the allocation of the pair. This implies, in particular, that the evaluation of nested pairs now takes time proportional to the depth of nesting. For example, if $e = \texttt{pair(pair(1,2),pair(3,4))}$ and $v = \texttt{pr(pr(1,2),pr(3,4))}$, then we have

$$e \mapsto^{3}_{seq} v \tag{40.12}$$

$$e \mapsto^{2}_{par} v \tag{40.13}$$

corresponding to work 3 and depth 2 for these expressions under the revised cost semantics (assuming zero cost to evaluate the numeric literals).

## 40.3 Vector Parallelism

A more general form of parallelism arises from considering a type of *vectors*, which are finite sequences of values of a specified type. Unlike tuples, the number of components of a vector is not determined until execution time. The primitive operations on vectors are chosen so that they may be executed in parallel on a *shared memory multiprocessor*, or *SMP*, in constant depth for an arbitrary vector.

Support for parallel computation with vectors consists of the following constructs:

| | | | |
|---|---|---|---|
| Type | $\tau$ | ::= | $\texttt{vec}(\tau)$ |
| Expr | $e$ | ::= | $\texttt{vec}(e_0,\ldots,e_{n-1}) \mid \texttt{velt}(e_1,e_2) \mid \texttt{vsiz}(e) \mid \texttt{vidx}(e) \mid$ |
| | | | $\texttt{vscan}(e) \mid \texttt{vmap}(e_1,e_2) \mid \texttt{vupd}(e_1,e_2)$ |

These expressions may be informally described as follows. The expression $\texttt{vec}(e_0,\ldots,e_{n-1})$ evaluates to an $n$-vector whose elements are given by the expressions $e_i$, $0 \leq i < n$. The operation $\texttt{velt}(e_1,e_2)$ retrieves the element of the vector given by $e_1$ at the index given by $e_2$. The operation

`vsiz(`$e$`)` returns the number of elements in the vector given by $e$. The operation `vidx(`$e$`)` creates a vector of length $n$ (given by $e$) whose elements are $0, \ldots, n-1$. The operation `vscan(`$e$`)` takes a vector of natural numbers and returns a vector of the same length obtained by computing the partial sums obtained by a left-to-right scan of the input vector. The operation `vmap(`$e_1, e_2$`)` applies the function given by $e_1$ to every element of $e_2$ in parallel. Finally, the operation `vupd(`$e_1, e_2$`)` yields a new vector of the same size, $n$, as the vector $v_1$ given by $e_1$, but whose elements are updated according to the vector $v_2$ given by $e_2$. The elements of $e_2$ are pairs specifying an index $0 \leq i < n$ and a value $v$, indicating that the $i$th element of the output should be $v$, all other elements remaining untouched. If $v_2$ specifies an index $i$ more than once, then the result is determined by the rightmost such occurrence.

The static semantics of these primitives is given by the following typing rules:

$$\frac{\Gamma \vdash e_1 : \tau \quad \ldots \quad \Gamma \vdash e_n : \tau}{\Gamma \vdash \texttt{vec}(e_0, \ldots, e_{n-1}) : \texttt{vec}(\tau)} \tag{40.14a}$$

$$\frac{\Gamma \vdash e_1 : \texttt{vec}(\tau) \quad \Gamma \vdash e_2 : \texttt{nat}}{\Gamma \vdash \texttt{velt}(e_1, e_2) : \tau} \tag{40.14b}$$

$$\frac{\Gamma \vdash e : \texttt{vec}(\tau)}{\Gamma \vdash \texttt{vsiz}(e) : \texttt{nat}} \tag{40.14c}$$

$$\frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{vidx}(e) : \texttt{vec}(\texttt{nat})} \tag{40.14d}$$

$$\frac{\Gamma \vdash e : \texttt{vec}(\texttt{nat})}{\Gamma \vdash \texttt{vscan}(e) : \texttt{vec}(\texttt{nat})} \tag{40.14e}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \texttt{vec}(\tau)}{\Gamma \vdash \texttt{vmap}(e_1, e_2) : \texttt{vec}(\tau')} \tag{40.14f}$$

$$\frac{\Gamma \vdash e_1 : \texttt{vec}(\tau) \quad \Gamma \vdash e_2 : \texttt{vec}(\texttt{prod}(\texttt{nat}, \tau))}{\Gamma \vdash \texttt{vupd}(e_1, e_2) : \texttt{vec}(\tau)} \tag{40.14g}$$

The parallel dynamic semantics is given by the following rules.

$$\frac{\varnothing \neq I \subseteq \{0, \ldots, n-1\} \quad \forall i \in I\ (e_i \mapsto_{par} e_i') \quad \forall i \notin I\ (e_i' = e_i \mathbin{\&} e_i\ \mathsf{val})}{\texttt{vec}(e_0, \ldots, e_{n-1}) \mapsto_{par} \texttt{vec}(e_0', \ldots, e_{n-1}')} \tag{40.15a}$$

$$\frac{e_0\ \mathsf{val} \quad \ldots \quad e_{n-1}\ \mathsf{val}}{\mathtt{velt(vec}(e_0,\ldots,e_{n-1}),i) \mapsto_{par} e_i} \tag{40.15b}$$

$$\frac{e_0\ \mathsf{val} \quad \ldots \quad e_{n-1}\ \mathsf{val}}{\mathtt{vsiz(vec}(e_0,\ldots,e_{n-1})) \mapsto_{par} n} \tag{40.15c}$$

$$\frac{}{\mathtt{vidx}(n) \mapsto_{par} \mathtt{vec}(0,\ldots,n-1)} \tag{40.15d}$$

$$\frac{\forall\ 0 \leq i < n\ m'_i = \sum_{j=0}^{i-1} m_j}{\mathtt{vscan(vec}(m_0,\ldots,m_n)) \mapsto_{par} \mathtt{vec}(m'_0,\ldots,m'_n)} \tag{40.15e}$$

$$\frac{e\ \mathsf{val} \quad e_0\ \mathsf{val} \quad \ldots \quad e_{n-1}\ \mathsf{val}}{\mathtt{vmap}(e,\mathtt{vec}(e_0,\ldots,e_{n-1})) \mapsto_{par} \mathtt{vec}(e(e_0),\ldots,e(e_{n-1}))} \tag{40.15f}$$

$$\frac{e_0\ \mathsf{val} \quad \ldots \quad e_n\ \mathsf{val} \quad e'_0\ \mathsf{val} \quad \ldots \quad e'_{n'-1}\ \mathsf{val}}{\mathtt{vupd(vec}(e_0,\ldots,e_{n-1}),\mathtt{vec}(e'_0,\ldots,e'_{n'-1})) \mapsto_{par} \mathtt{vec}(e''_0,\ldots,e''_{n-1})} \tag{40.15g}$$

where for each $0 \leq i < n$,

$$e''_i = \begin{cases} d_i & \text{if } 0 \leq k < n' \text{ is largest s.t. } e'_k = \mathtt{pair}(i,d_i) \\ e_i & \text{otherwise} \end{cases}$$

The sequential dynamic semantics of vectors is defined similarly to the parallel semantics. The only difference is that vector expressions are evaluated in left-to-right order, rather than in parallel. This is expressed by the following rule:

$$\frac{e_0\ \mathsf{val} \quad \ldots \quad e_{i-1}\ \mathsf{val} \quad e_i \mapsto_{seq} e'_i}{\mathtt{vec}(e_0,\ldots,e_{i-1},e_i,e_{i+1},\ldots,e_{n-1}) \mapsto_{seq} \mathtt{vec}(e_0,\ldots,e_{i-1},e'_i,e_{i+1},\ldots,e_{n-1})} \tag{40.16}$$

With these two basic semantics in mind, we may also derive a cost semantics for vectors, where the work corresponds to the number of steps required in the sequential semantics, and the depth corresponds to the number of steps required in the parallel semantics.

Vector expressions are evaluated in parallel.

$$\frac{\forall\, 0 \leq i < n\ e_i \Downarrow^{w_i,d_i} v_i}{\texttt{vec}(e_0,\ldots,e_{n-1}) \Downarrow^{w,d} \texttt{vec}(v_0,\ldots,v_{n-1})} \tag{40.17a}$$

where $w = \sum_{i=0}^{n-1} w_i$ and $d = \max_{i=0}^{n-1} d_i$.

Retrieving an element of a vector takes constant work and depth.

$$\frac{e_1 \Downarrow^{w_1,d_1} \texttt{vec}(v_0,\ldots,v_{n-1}) \quad e_2 \Downarrow^{w_2,d_2} i \quad (0 \leq i < n)}{\texttt{velt}(e_1,e_2) \Downarrow^{w_1+w_2+1,d_1+d_2+1} v_i} \tag{40.17b}$$

Retrieving the size of a vector takes constant work and depth.

$$\frac{e \Downarrow^{w,d} \texttt{vec}(v_0,\ldots,v_{n-1})}{\texttt{vsiz}(e) \Downarrow^{w+1,d+1} n} \tag{40.17c}$$

Creating an index vector takes linear work and constant depth.

$$\frac{e \Downarrow^{w,d} n}{\texttt{vidx}(e) \Downarrow^{w+n,d+1} \texttt{vec}(0,\ldots,n-1)} \tag{40.17d}$$

Scanning takes linear work and constant depth.

$$\frac{e \Downarrow^{w,d} \texttt{vec}(m_0,\ldots,m_{n-1}) \quad m'_i = \sum_{j=0}^{i-1} m_j}{\texttt{vscan}(e) \Downarrow^{w+n,d+1} \texttt{vec}(m'_0,\ldots,m'_{n-1})} \tag{40.17e}$$

Mapping a function across a vector takes constant work and depth beyond the cost of the function applications.

$$\frac{\begin{array}{c} e_1 \Downarrow^{w_1,d_1} v \\ e_2 \Downarrow^{w_2,d_2} \texttt{vec}(v_0,\ldots,v_{n-1}) \\ \texttt{vec}(v(v_0),\ldots,v(v_{n-1})) \Downarrow^{w,d} \texttt{vec}(v'_0,\ldots,v'_{n-1}) \end{array}}{\texttt{vmap}(e_1,e_2) \Downarrow^{w_1+w_2+w+1,d_1+d_2+d+1} \texttt{vec}(v'_0,\ldots,v'_{n-1})} \tag{40.17f}$$

Updating a vector takes linear work and constant depth.

$$\frac{\begin{array}{c} e_1 \Downarrow^{w_1,d_1} \texttt{vec}(v_0,\ldots,v_{n-1}) \\ e_2 \Downarrow^{w_2,d_2} \texttt{vec}(v'_0,\ldots,v'_{n'-1}) \end{array}}{\texttt{vupd}(e_1,e_2) \Downarrow^{w_1+w_2+k+n,d_1+d_2+1} \texttt{vec}(v''_0,\ldots,v''_{n-1})} \tag{40.17g}$$

where for each $0 \leq i < n$,

$$v''_i = \begin{cases} w_i & \text{if } 0 \leq k < n' \text{ is largest s.t. } v'_k = \texttt{pair}(i, w_i) \\ v_i & \text{otherwise} \end{cases}$$

**Theorem 40.5.** *$e \Downarrow^{w,d} v$ iff $e \mapsto^{d}_{par} v$ and $e \mapsto^{w}_{seq} v$.*

## 40.4 Provably Efficient Implementations

The semantics of parallelism given above is based on an idealized parallel computer with an unlimited number of processors. In practice this idealization must be simulated using some fixed number, $p$, of physical processors. In practice $p$ is on the order of 10's of processors, but may even rise (at the time of this writing) into the 100's. In any case $p$ does not vary with input size, but is rather a fixed parameter of the implementation platform. The important question is how efficiently can one simulate unbounded parallelism using only $p$ processors? That is, how realistic are the costs assigned to the language by our semantics? Can we make accurate predictions about the running time of a program on a real parallel computer based on the idealized cost assigned to it by our semantics?

The answer is *yes*, through the notion of a *provably efficient implementation*. Although a full treatment is beyond the scope of this book, it is worthwhile to summarize the main ideas.

**Theorem 40.6** (Provable Implementation)**.** *If $e \Downarrow^{w,d} v$, then $e$ can be evaluated on a shared memory multi-processor with $p$-processors in time $O(w/p + d \lg p)$.*

Observe that for $p = 1$, the stated bound simplifies to $O(w)$, as would be expected. The class of shared-memory multiprocessors is broad, and includes most commercial multiprocessors.

To understand the significance of Theorem 40.6, observe that the definition of work and depth yields a lower bound of $\Omega(\max(w/p, d))$ on the execution time on $p$ processors. We can never complete execution in fewer

than $d$ steps, and can, at best, divide the total work evenly among the $p$ processors. The theorem tells us that we can come within a constant factor of this lower bound. The constant factor, $\lg p$, represents the overhead of scheduling parallel computations on $p$ processors.

The goal of parallel programming is to maximize the use of parallelism so as to minimize the execution time. By Theorem 40.6 on the preceding page this will occur if the term $w/p$ dominates, which occurs if the ratio $w/d$ of work to depth is at least $p \lg p$. This ratio is sometimes called the *parallelizability* of the program. For highly sequential programs, $d$ is directly proportional to $w$, yielding a low parallelizability — increasing the number of processors will not speed up the computation. For highly parallel programs, $d$ might be constant or proportional to $\lg w$, resulting in a large parallelizability, and good utilization of the available computing resources. It is important to keep in mind that *it is not known* whether there are inherently sequential problems (for which no parallelizable solution is possible), or whether, instead, all problems can benefit from parallelism. The best that we can say at the time of this writing is that there are problems for which no parallelizable solution is known.

To get a sense of what is involved in the proof of Theorem 40.6 on the facing page, let us consider the assumption that the index operation on vectors given above has constant depth. The theorem implies that index is implementable on an SMP in time $O(n/p + \lg p)$. To do this, we allocate, but do not initialize, a region of memory of size $n$ in constant time. The idea is to assign responsibility for a segment of $n/p$ elements to each of the $p$ processors, which then simultaneously initialize their segment. To determine the starting point, $n_i$, for processor $i$, we construct in $O(\lg p)$ steps the vector consisting of the numbers $0, 1, \ldots, p-1$, then multiply every element of this vector by $n/p$ to obtain the required sequence of starting points. We then ask in parallel for the $i$th processor to initialize its segment with the numbers $n_i$ through $n_i + (n/p) - 1$, requiring $O(n/p)$ steps in parallel. The total time required is then $O(n/p + \lg p)$, as required.

Another source of the $O(\lg p)$ overhead is the need to re-schedule the processors after each round of execution. At the end of a round execution on each processor either terminates, continues from its present point, or creates two or more new tasks for parallel execution. The scheduler must poll the processors to determine their status, enqueueing any new or continuing work, then re-assigning work from the queue to the processors. The polling process amounts to a computation of partial sums, where each processor is assigned 0, if it has completed, 1, if it is continuing, and 2 or more, if it creates new tasks. The partial sums are then used to enter the tasks into

a queue for scheduling on the next round. Computation of the partial sums requires $O(\lg p)$ steps, which is the overhead of scheduling on each round.

The overall bound of $O(w/p + d \lg p)$ quoted in Theorem 40.6 on page 324 arises from two considerations. If the depth, $d$, is the dominant factor, then the per-round overhead results in the $d \lg p$ component of the overall cost. Otherwise the scheduling cost can be hidden in the workload by activating $\lg p$ items of work per processor, for a total of $p \lg p$ units of work on each round. Note that the partial sums computation required for load balancing still takes $O(\lg p)$ time for this slightly larger workload. This implies that the scheduling overhead is hidden, as witnessed by the simple calculation $(w \lg p)/(p \lg p) = w/p$.

# Part XV

# Concurrency

# Chapter 41

# Process Calculus

So far we have mainly studied the static and dynamic semantics of programs in isolation, without regard to their interaction with the world. But to extend this analysis to even the most rudimentary forms of input and output requires that we consider external agents that interact with the program. After all, the whole purpose of a computer is to interact with a person!

To extend our investigations to interactive systems, we begin with the study of *process calculi*, which are abstract formalisms that capture the essence of interaction among independent agents. There are many forms of process calculi, differing in technical details and in emphasis. We will consider the best-known formalism, which is called the $\pi$-*calculus*. The development will proceed in stages, starting with simple action models, then extending to interacting concurrent processes, and finally to the $\pi$-calculus itself.

## 41.1 Actions and Events

Our treatment of concurrent interaction is based on the notion of an *event*, which specifies the set of *actions* that a process is prepared to undertake in concert with another process. Two processes interact by undertaking two complementary actions, which may be thought of as a *read* and a *write* on a common channel. The processes synchronize on these complementary actions, after which they may proceed independently to interact with other processes.

To begin with we will focus on sequential processes, which simply await

the arrival of one of several possible actions, known as an event.

$$\begin{array}{lrcl} \textit{Process} & P & ::= & \texttt{await}(E) \\ \textit{Event} & E & ::= & \mathbf{0} \mid E_1 + E_2 \mid \alpha \,.\, P \\ \textit{Action} & \alpha & ::= & \texttt{?}a \mid \texttt{!}a \end{array}$$

The variables $a$, $b$, and $c$ range over *channels*, which serve as conduits for synchronization. The actions $\texttt{?}a$ and $\texttt{!}a$ are said to be *complementary*. As a mnemonic device, it may help to think of the first as a read action, and the second as a write action, on a communication channel (but one could equally well take the opposite point of view).

We will handle events modulo *structural congruence*, written $P_1 \equiv P_2$ and $E_1 \equiv E_2$, respectively, which is the strongest equivalence relation closed under the following rules:

$$\frac{E \equiv E'}{\texttt{await}(E) \equiv \texttt{await}(E')} \tag{41.1a}$$

$$\frac{E_1 \equiv E_1' \quad E_2 \equiv E_2'}{E_1 + E_2 \equiv E_1' + E_2'} \tag{41.1b}$$

$$\frac{P \equiv P'}{\alpha \,.\, P \equiv \alpha \,.\, P'} \tag{41.1c}$$

$$\overline{E + \mathbf{0} \equiv E} \tag{41.1d}$$

$$\overline{E_1 + E_2 \equiv E_2 + E_1} \tag{41.1e}$$

$$\overline{E_1 + (E_2 + E_3) \equiv (E_1 + E_2) + E_3} \tag{41.1f}$$

The importance of imposing structural congruence on sequential processes is that it enables us to think of an event as having the form

$$\alpha_1 \,.\, P_1 + \ldots + \alpha_n \,.\, P_n$$

for some $n \geq 0$, with the understanding that when $n = 0$ this notation stands for the null event, $\mathbf{0}$. The derived process expression $\mathbf{1}$ stands for the "inert" process $\texttt{await}(\mathbf{0})$, which awaits the event that will never occur.

An illustrative example of Robin Milner's is a simple vending machine that may take in a 2p coin, then optionally either permit selection of a cup of tea, or take another 2p coin, then permit selection of a cup of coffee.

$$V = \texttt{await(?2p.await(!tea.}V + \texttt{?2p.await(!coffee.}V\texttt{)))}$$

As the example indicates, we tacitly permit recursive definitions of processes, with the understanding that a defined identifier may always be replaced with its definition wherever it occurs.

Because we have suppressed the internal computation occurring within a process, sequential processes have no dynamic semantics on their own—their dynamics arises only as a result of interaction with another process. For the vending machine to operate there must be another process (you!) who initiates the events expected by the machine, causing both your state (the coins in your pocket) and its state (as just described) to change as a result.

## 41.2 Concurrent Interaction

We enrich the language of processes with concurrent composition.

$$\text{Process} \qquad P \quad ::= \quad \texttt{await}(E) \mid P_1 \parallel P_2$$

Structural congruence for processes is enriched by the following rules:

$$\overline{P \parallel \mathbf{1} \equiv P} \tag{41.2a}$$

$$\overline{P_1 \parallel P_2 \equiv P_2 \parallel P_1} \tag{41.2b}$$

$$\overline{P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3} \tag{41.2c}$$

$$\frac{P_1 \equiv P_1' \quad P_2 \equiv P_2'}{P_1 \parallel P_2 \equiv P_1' \parallel P_2'} \tag{41.2d}$$

This permits us to regard a process as having the form

$$\texttt{await}(E_1) \parallel \ldots \parallel \texttt{await}(E_n)$$

where $n \geq 0$, it being understood that when $n = 0$ this is the process $\mathbf{1}$.

We may now define the dynamic semantics of concurrent processes by a judgement of the form $P_1 \longrightarrow P_2$, where $P_1$ and $P_2$ are processes. The dynamic semantics consists of interactions among two processes offering to undertake complementary actions.

$$\overline{\texttt{await}(E_1 + \texttt{!}a\texttt{.}P_1) \parallel \texttt{await}(E_2 + \texttt{?}a\texttt{.}P_2) \longrightarrow P_1 \parallel P_2} \tag{41.3a}$$

The interaction may be understood as the synchronized occurrence of a read and a write on the same channel by two processes.

In addition to synchronizing, concurrent processes may also proceed to execute independently of one another.

$$\frac{P_1 \longrightarrow P_1'}{P_1 \parallel P_2 \longrightarrow P_1' \parallel P_2} \tag{41.3b}$$

$$\frac{P_2 \longrightarrow P_2'}{P_1 \parallel P_2 \longrightarrow P_1 \parallel P_2'} \tag{41.3c}$$

As an example, let us consider the interaction of the vending machine, $V$, with the user process, $U$, defined as follows:

$$U = \texttt{await(!2p.await(!2p.await(?coffee.}\mathbf{1}\texttt{)))}.$$

Here is a trace of the interaction between $V$ and $U$, suppressing uses of structural congruence:

$$\begin{aligned} V \parallel U &\longrightarrow \texttt{await(!tea.}V + \texttt{?2p.await(!coffee.}V\texttt{))} \parallel \texttt{await(!2p.await(?coffee.}\mathbf{1}\texttt{))} \\ &\longrightarrow \texttt{await(!coffee.}V\texttt{)} \parallel \texttt{await(?coffee.}\mathbf{1}\texttt{)} \\ &\longrightarrow V \end{aligned}$$

## 41.3 Replication

Some presentations of process calculus forego reliance on defining equations for processes in favor of a *replication* construct, which we write $*P$. This process stands for as many concurrently executing copies of $P$ as one may require, which may be modeled by the structural congruence

$$*P \equiv P \parallel *P.$$

This hides the overhead of process creation, and gives no hint as to how often it can or should be applied. One could alternatively build replication into the reaction rules to model replication behavior more closely.

Replication may be used to model recursive definitions by introducing an "activator" process that is contacted to effect the recursion. Consider the recursive definition $A = P(A)$, where $P$ is a process expression involving occurrences of the defined process $A$. This may be simulated by defining

$$A = *\texttt{await(?}a\texttt{.}P(\texttt{await(!}a\texttt{.}\mathbf{1}\texttt{))}),$$

in which we have replaced occurrences of $A$ within $P$ by the process that signals the event $a$. To initiate $A$ we must put it in parallel with the process `await(!`$a$`.1)`, which initiates process.

As an example, let us consider Milner's vending machine written using replication, rather than using recursive process definition:

$$V_1 = \texttt{*await(?}v\texttt{.}V_2\texttt{)} \tag{41.4}$$

$$V_2 = \texttt{await(?2p.await(!tea.}V_0 + \texttt{?2p.await(!coffee.}V_0\texttt{)))} \tag{41.5}$$

$$V_0 = \texttt{await(!}v\texttt{.}\mathbf{1}\texttt{)} \tag{41.6}$$

The process $V_1$ is a replicated server that awaits a signal on channel $v$ to create another instance of the vending machine. The recursive calls are replaced by signals along $v$ to re-start the machine. The original machine, $V$, is simulated by the parallel composition $V_0 \parallel V_1$.

## 41.4 Private Channels

It is often desirable to isolate interactions among a group of concurrent processes from those among another group of processes. This can be achieved by creating a private channel that is shared among those in the group, and which is inaccessible from all other processes. This may be modeled by enriching the language of processes with a construct for creating a new channel:

$$\text{Process} \qquad P \quad ::= \quad \nu(a\,.\,P)$$

As the syntax suggests, this is a binding operator in which the channel $a$ is bound within $P$.

Structural congruence is extended with the following rules:

$$\frac{P =_\alpha P'}{P \equiv P'} \tag{41.7a}$$

$$\frac{P \equiv P'}{\nu(a\,.\,P) \equiv \nu(a\,.\,P')} \tag{41.7b}$$

$$\frac{a \# P_2}{\nu(a\,.\,P_1) \parallel P_2 \equiv \nu(a\,.\,P_1 \parallel P_2)} \tag{41.7c}$$

The last rule, called *scope extrusion*, will be important for the treatment of communication in the next section.

Reaction is extended with one additional rule permitting reactions to take place within the scope of a binder.

$$\frac{P \longrightarrow P'}{\nu(a\,.\,P) \longrightarrow \nu(a\,.\,P')} \tag{41.8}$$

No process may interact with $\nu(a\,.\,P)$ along the newly-allocated channel, for to do so would require knowledge of the private channel, $a$, which is chosen, by the magic of $\alpha$-equivalence, to be distinct from all other channels in the system.

As an example, let us consider again the non-recursive definition of the vending machine. The channel, $v$, used to initialize the machine should be considered private to the machine itself, and not be made available to a user process. This is naturally expressed by the process expression $\nu(v\,.\,V_0 \parallel V_1)$, where $V_0$ and $V_1$ are as defined above using the designated channel, $v$. This process correctly simulates the original machine, $V$, because it precludes interaction with a user process on channel $V$. If $U$ is a user process, the interaction begins as follows:

$$\nu(v\,.\,V_0 \parallel V_1) \parallel U \longrightarrow \nu(v\,.\,V_2) \parallel U \equiv \nu(v\,.\,V_2 \parallel U)$$

The interaction continues as before, albeit within the scope of the binder, provided that $v$ has been chosen (by structural congruence) to be apart from $U$, ensuring that it is private to the internal workings of the machine.

## 41.5 Synchronous Communication

The concurrent process calculus presented in the preceding section models synchronization based on the willingness of two processes to undertake complementary actions. A natural extension of this model is to permit data to be passed from one process to another as part of synchronization. Since we are abstracting away from the computation occurring within a process, it would not make much sense to consider, say, passing an integer during synchronization. A more interesting possibility is to permit passing *channels*, so that new patterns of connectivity can be established as a consequence of inter-process synchronization. This is the core idea of the $\pi$-calculus.

Traditionally, the $\pi$-calculus is viewed as an extension of the concurrent calculus given in the preceding section in which actions are generalized to express the asymmmetric communication from one process to another at

the point of synchronization.

$$\text{Action} \qquad \alpha \;::=\; \texttt{?}a\texttt{(}x\texttt{)} \mid \texttt{!}a\texttt{<}b\texttt{>}$$

The action $\texttt{?}a\texttt{(}x\texttt{)}$ represents a read, or receive, along channel, $a$, of another channel that will be bound to the variable $x$ when received. The action $\texttt{!}a\texttt{<}b\texttt{>}$ represents a write, or send, of a channel, $b$, along a channel, $a$.

This notation is, however, rather confusing, because it separates a binder from the scope of its binding. An alternative, which we adopt here, is to change the syntax of events to account for the two forms of actions directly.

$$\text{Event} \qquad E \;::=\; \mathbf{0} \mid E_1 + E_2 \mid \texttt{?}a\texttt{(}x\texttt{)}\,\texttt{.}\,P \mid \texttt{!}a\texttt{<}b\texttt{>}\,\texttt{.}\,P$$

The event $\texttt{?}a\texttt{(}x\texttt{)}\,\texttt{.}\,P$ binds the variable $x$ within the process expression $P$. The syntax for processes remains as in the case of the simple concurrent calculus, *albeit* with an enriched concept of event.

Interaction in the $\pi$-calculus consists of synchronization on the concurrent availability of complementary actions on a channel, passing a channel from the sender to the receiver.

$$\overline{\texttt{await(}E_1 + \texttt{!}a\texttt{<}b\texttt{>}\,\texttt{.}\,P_1\texttt{)} \parallel \texttt{await(}E_2 + \texttt{?}a\texttt{(}x\texttt{)}\,\texttt{.}\,P_2\texttt{)} \longrightarrow P_1 \parallel [b/x]P_2} \tag{41.9}$$

In contrast to pure synchronization the message-passing form of interaction is fundamentally asymmetric — the receiver continues with the channel passed by the sender substituted for the bound variable of the action.

## 41.6 Mutable Cells as Processes

Let us consider a reference cell server that, when given an initial value, creates a cell that listens on two dedicated channels, one to get the current value of the cell, the other to set it to a new designated value. This may be defined using recursion equations as follows:

$$C(x,g,s) = \texttt{await(}S(g,s) + G(x,g,s)\texttt{)} \tag{41.10}$$
$$S(g,s) = \texttt{?}s\texttt{(}y\texttt{)}\,\texttt{.}\,C(y,g,s) \tag{41.11}$$
$$G(x,g,s) = \texttt{!}g\texttt{<}x\texttt{>}\,\texttt{.}\,C(x,g,s) \tag{41.12}$$

The cell is parameterized by its current value and two channels on which to contact it to get and set its value. Each message causes a new cell to be created, reflecting any update to its value.

To avoid the recursion implicit in the equations we may instead define a server that creates fresh cells whenever contacted on a specified channel, $c$, specifying an initial value, $x$, for that cell and two channels, $g$ and $s$, on which to contact it to get and set its value.

$$R(c) = \texttt{*await(?}c\texttt{(}x,g,s\texttt{)}\texttt{.}C'(c,x,g,s)\texttt{)} \tag{41.13}$$
$$C'(c,x,g,s) = \texttt{await(}S'(c,g,s) + G'(c,x,g,s)\texttt{)} \tag{41.14}$$
$$S'(c,g,s) = \texttt{?}s\texttt{(}y\texttt{)}\texttt{.await(!}c\texttt{<}y,g,s\texttt{>}\texttt{.}\mathbf{1}\texttt{)} \tag{41.15}$$
$$G'(c,x,g,s) = \texttt{!}g\texttt{<}x\texttt{>}\texttt{.await(!}c\texttt{<}x,g,s\texttt{>}\texttt{.}\mathbf{1}\texttt{)} \tag{41.16}$$

The reference cell server repeatedly awaits receipt of a creation message on channel $r$, and creates a new cell with the specified initial value and channels on which to contact it. The cell awaits contact, then behaves appropriately, but this time contacting the server to create a new cell with updated value after each message.

To use reference cells in a process $P$, we put $P$ in parallel with an instance, $R(c)$, of the cell server, which is contacted via channel $c$. For example, the process

$$\nu\texttt{(}c\texttt{.}R(c) \parallel \nu\texttt{(}g\texttt{.}\nu\texttt{(}s\texttt{.await(!}c\texttt{<}0,g,s\texttt{>.await(!}s\texttt{<}1\texttt{>.await(?}g\texttt{(}x\texttt{)....))))))}$$

allocates a channel for communication with the reference cell server, then allocates two channels for a new cell, initializes it to `0`, sets it to `1`, then retrieves its value, and so forth.

This example illustrates the importance of scope extrusion in the $\pi$-calculus. Initially, the process $R(c)$ is run concurrently with a process that allocates two new channels, $g$ and $s$, and then sends these channels, along with the initial value, 0, along $c$. Tracing out the reactions, this results in a process offering to send along $g$ and to receive along $s$, which represents the new reference cell, running in parallel with the subsequent process that manipulates this newly allocated cell. For this to make sense, the scope of $g$ and $s$ must be enlarged to encompass the body of $R(c)$ after receipt of 0, $g$, and $s$ along $c$. Structural congruence ensures that we may "lift" the allocation of $g$ and $s$ to encompass $R(c)$, since $g$ and $s$ may be chosen, by $\alpha$-equivalence, to be distinct from any channels already occurring in $R(c)$. This enables communication of the cell server with the cell client along the channels $g$ and $s$.

## 41.7 Asynchronous Communication

This form of interaction is called *synchronous*, because both the sender and the receiver are blocked from further interaction until synchronization has occurred. On the receiving side this is inevitable, because the receiver cannot continue execution until the channel which it receives has been determined, much as the body of a function cannot be executed until its argument has been provided. On the sending side, however, there is no fundamental reason why notification is required; the sender could simply send the message along a channel without specifying how to continue once that message has been received. This "fire and forget" semantics is called *asynchronous* communication, in constrast to the *synchronous* form just described.

The *asynchronous $\pi$-calculus* is obtained by removing the synchronous send event, $\texttt{!}a\texttt{<}b\texttt{>.}P$, and adding a new form of process, the asynchronous send, written $\texttt{!}a\texttt{<}b\texttt{>}$. The syntax of the asynchronous $\pi$-calculus is therefore given by the following grammar:

$$\begin{array}{lrcl} \text{Process} & P & ::= & \texttt{!}a\texttt{<}b\texttt{>} \mid \texttt{await(}E\texttt{)} \mid P_1 \parallel P_2 \mid \nu\texttt{(}a\texttt{.}P\texttt{)} \\ \text{Event} & E & ::= & \mathbf{0} \mid \texttt{?}a\texttt{(}x\texttt{).}P \mid E_1 + E_2 \end{array}$$

Up to structural congruence, an event is just a choice of zero or more reads along any number of channels.

The basic reaction rule of communication is re-phrased for the asynchronous case as follows:

$$\overline{\texttt{await(}E + \texttt{?}a\texttt{(}x\texttt{).}P\texttt{)} \parallel \texttt{!}a\texttt{<}b\texttt{>} \longrightarrow [b/x]P} \tag{41.17}$$

Should there be more than one read on the channel $a$ within the awaited event, the choice of which to select for communication is not specified — either could be chosen. One may regard the pending asynchronous write as a kind o buffer in which the message is held until a receiver is chosen.

In a sense the synchronous $\pi$-calculus is more fundamental than the asynchronous variant, because we may always mimic the asynchronous send by a process of the form $\texttt{await(!}a\texttt{<}b\texttt{>.}\mathbf{1}\texttt{)}$, which performs the send, and then becomes the inert process $\mathbf{1}$. In another sense, however, the asynchronous $\pi$-calculus is more fundamental, because we may encode a synchronous send by introducing a notification channel on which the receiver sends a message to notify the sender of the successful receipt of its message. This exposes the implicit communication required to implement synchronous send, and avoids it in cases where it is not needed (in particular, when the resumed process is just the inert process, as just illustrated).

## 41.8 Exercises

1. Explore Church-like encodings of data structures in the $\pi$-calculus.
2. Show how to avoid implicit replication.
3. Implement synchronous send in the asynchronous $\pi$-calculus.
4. Formulate the polyadic $\pi$-calculus. Can it be encoded in the monadic $\pi$-calculus?
5. Show that in the asynchronous $\pi$-calculus, events are definable as processes.

# Chapter 42

# Concurrent ML

Concurrent ML, or CML, is an extension of Standard ML with concurrency. The basic mechanisms of CML can be seen as analogues of those that make up the $\pi$-calculus, but there are, in addition, a number of mechanisms, such as negative acknowledgements, that go beyond what is found in abstract process calculi. We will study the basic mechanisms of CML, and show how to give them a semantics using techniques similar to those of the $\pi$-calculus.

## 42.1 Channels and Events

CML is based on two fundamental abstract types, channels and events. Channels carry values of a specified type, including, potentially, other channels. Events are also typed, with the type of the event indicating the type of value arising from synchronizing with it. We will take some liberties in defining the primitives of CML so as to simplify the presentation. The actual signatures are substantial enrichments of these, and make use of slightly different naming conventions.

The signature of channels is given by the following declaration:

```
signature CHANNEL = sig
  type 'a chan
  val channel : unit -> 'a chan
end
```

The expression `channel()` allocates a fresh channel whose type is determined from context.

The signature of events is given by the following declaration:

```
signature EVENT = sig
  type 'a event
  val recv : 'a chan -> 'a event
  val send : 'a chan * 'a -> unit event
  val never : 'a event
  val always : 'a -> 'a event
  val choose : 'a event * 'a event -> 'a event
  val wrap : 'a event * ('a -> 'b) -> 'b event
end
```

The basic forms of event are the send and receive events. The event `recv` $a$ is enabled when data is available to be received on channel $a$; the value of the event is the received value. The event `send (`$a$`,`$v$`)` is enabled when transmission of the value $v$ on the channel $a$ is possible; the value of this event is always `()`. The event `never` is never enabled. The event `always` $v$ is always enabled, and has value $v$. The event `choose(`$e_1$`,`$e_2$`)` is enabled when either $e_1$ or $e_2$ is enabled, and its value is the value of the enabled event, with ties broken arbitrarily. Finally, the event `choose(`$e$`,`$f$`)` is enabled whenever $e$ is enabled; if the value of $e$ is $v$, then the value of this event is the value of the application $f(v)$.

Processes are constructed from the following primitives:

```
signature PROCESS = sig
  val spawn : (unit -> 'a) -> 'a
  val exit : unit -> 'a
  val sync : 'a event -> 'a
end
```

The expression `spawn` $f$ creates a new process executing $f$`()`, and returns `()`. The expression `exit()` terminates the current process. The expression `sync` $e$ waits for the event $e$ to be enabled, synchronizes on it, and returns its value.

## 42.2 Dynamic Semantics

The dynamic semantics of CML may be given in terms of an implementation of the `EVENT` signature. Events are represented as values of an ML datatype representing events in a normalized form in which `wrap` events occur only in juxtaposition with `send` and `recv` events, and in which tree-structured choices are flattened into lists. We assume given a structure `Ch :> CHANNEL`.

```
structure Event :> EVENT = struct

  datatype 'a comm_event =
    Recv of 'a Ch.chan | Send of 'a Ch.chan * 'a
  datatype 'a atomic_event =
    Never | Always of 'a | Wrap of 'a comm_event * ('a -> 'b)
  type 'a event = 'a atomic_event list

  val never = [Never]
  fun always v = [Always v]
  fun recv a = [Wrap (Recv a, fn x => x)]
  fun send (a, v) = [Wrap (Send (a, v), fn x => x)]
  fun choose (es1, es2) = es1 @ es2
  fun wrap1 (Never, f) = Never
    | wrap1 (Always v, f) = Always (f v)
    | wrap1 (Wrap (c, f), g) = Wrap (c, g o f)
  fun wrap (es, f) = map (wrap1 f) es

end
```

The dynamic semantics of CML may be given in terms of this representation of events by erecting the infrastructure of the $\pi$-calculus around it. Processes are defined by the following grammar:

| | | | |
|---|---|---|---|
| *CML Processes* | $P$ | $::=$ | $\{e\} \mid P_1 \parallel P_2 \mid \nu(a.P)$ |
| *CML Atomic Events* | $A$ | $::=$ | $\texttt{Never} \mid \texttt{Always}(v) \mid \texttt{Wrap}(\texttt{Recv}(a), f) \mid \texttt{Wrap}(\texttt{Send}(a, v), f)$ |
| *CML Events* | $E$ | $::=$ | $[A_1, \ldots, A_n] \quad (n \geq 0)$ |

We define structural congruence of CML processes analogously to that for $\pi$-calculus processes (see Chapter 41). In particular the order of events in a list of atomic events does not matter, and any occurrences of `Never` in an event are eliminated.

The reaction relation is defined on structural congruence classes of processes by the following rules, in addition to the general rules governing the process calculus combining forms given in Chapter 41.

The fundamental synchronization rule is as follows:

$$\frac{}{\{\texttt{sync}(\texttt{Wrap}(\texttt{Recv}(a), f) :: \_)\} \parallel \{\texttt{sync}(\texttt{Wrap}(\texttt{Send}(a, v), g) :: \_)\} \longrightarrow \{f(v)\} \parallel \{g()\}} \tag{42.1a}$$

The receiving process yields the value of $f(v)$, where $v$ is obtained from the sending process. The sending process yields the value of $g()$.

The always-enabled process is always enabled:

$$\overline{\{\texttt{sync(Always(}v\texttt{)::\_)}\} \longrightarrow \{v\}} \tag{42.1b}$$

Allocating a new channel is easily expressed by creating a new channel and returning it:

$$\overline{\{\texttt{channel()}\} \longrightarrow \nu(a.\{a\})} \tag{42.1c}$$

## 42.3 Exercises

# Chapter 43

# Monadic Input/Output

# Part XVI

# Dependent Types

# Chapter 44

# Indexed Families of Types

# Chapter 45

# Dependent Types

# Part XVII

# Modularity

# Chapter 46

# Separate Compilation and Linking

# Chapter 47

# Basic Modules

# Chapter 48

# Parameterized Modules

# Part XVIII

# Equivalence

# Chapter 49

# Equational Reasoning for Functional Programs

Equations are the heart and soul of mathematics. We derive equations such as

$$(x+1)^2 = x^2 + 2\,x + 1$$

to express the equivalence of two functions of the variable $x \in \mathbb{R}$. We solve equations such as

$$z^2 + 1 = 0$$

for $z \in \mathbb{C}$ for the complex number $i = \sqrt{-1}$. In elementary geometry congruence and similarity are forms of equality between geometric objects. The laws of physics are expressed as differential equations governing the structure and evolution of a physical system.

The beauty of functional programming is that equality of expressions in a functional language corresponds very closely to familiar patterns of mathematical reasoning. For example, in the language $\mathcal{L}\{\mathtt{nat} \to\}$ of Chapter 14 in which we can express addition as the function `plus`, the expressions

$$\lambda(x{:}\mathtt{nat}.\,\lambda(y{:}\mathtt{nat}.\,\mathtt{plus}(e_1)(e_2)))$$

and

$$\lambda(x{:}\mathtt{nat}.\,\lambda(y{:}\mathtt{nat}.\,\mathtt{plus}(e_2)(e_1)))$$

are equal, regardless of what $e_1$ and $e_2$, so long as they are of type `nat`. In other words, the addition function *as programmed* in $\mathcal{L}\{\mathtt{nat} \to\}$ is commutative. This may seem obvious, in the sense that you may have expected it to be true, but *why*, precisely, is it so? More importantly, what do we even

*mean* when we say that two expressions of a programming language are equal? In this chapter we will develop answers to these questions for the language $\mathcal{L}\{\texttt{nat} \to\}$ introduced in Chapter 14.

## 49.1 Observational Equivalence

When are two expressions equal? Whenever we cannot tell them apart! This may seem tautological, but it is not, because it depends on what we consider to be a means of telling expressions apart. What "experiment" are we permitted to perform on expressions in order to distinguish them? What counts as an observation that, if different for two expressions, is a sure sign that they are different?

If we permit ourselves to consider the syntactic details of the expressions, then very few expressions could be considered equal. For example, if it is deemed significant that an expression contains, say, more than one function application, or that it has an occurrence of $\lambda$-abstraction, then very few expressions would come out as equivalent. But such considerations seem silly, because they conflict with the intuition that the significance of an expression lies in its contribution to the *outcome* of a computation, and not to the process of obtaining that outcome. In short, if two expressions make the same contribution to the outcome of a complete program, then they ought to be regarded as equal.

We must fix what we mean by a complete program. Two considerations inform the definition. First, the dynamic semantics of $\mathcal{L}\{\texttt{nat} \to\}$ is given only for expressions without free variables, so a complete program should clearly be a *closed* expression. Second, the outcome of a computation should be *observable*, so that it is evident whether the outcome of two computations differs or not. We define a *complete program* to be a closed expression of type `nat`.

An *experiment*, or *observation*, about an expression is any means of using that expression within a complete program. We define an *expression context* to be an expression with a "hole" in it serving as a placeholder for another expression. The hole is permitted to occur anywhere, including within the scope of a binder. The bound variables within whose scope the hole lies are said to be *exposed (to capture)* by the expression context. These variables may be assumed, without loss of generality, to be distinct from one another. A *program context* is a closed expression context of type `nat`—that is, it is a complete program with a hole in it. The meta-variable $\mathcal{C}$ stands for any expression context.

*Replacement* is the process of filling a hole in an expression context, $\mathcal{C}$, with an expression, $e$, which is written $\mathcal{C}\{e\}$. Importantly, the free variables of $e$ that are exposed by $\mathcal{C}$ are *captured* by replacement (which is why replacement is not a form of substitution, which is defined so as to avoid capture). If $\mathcal{C}$ is a program context, then $\mathcal{C}\{e\}$ is a complete program iff all free variables of $e$ are captured by the replacement. For example, if $\mathcal{C} = \lambda(x{:}\mathtt{nat}.\circ)$, and $e = x{+}x$, then

$$\mathcal{C}\{e\} = \lambda(x{:}\mathtt{nat}.\, x{+}x).$$

The free occurrences of $x$ in $e$ are captured by the $\lambda$-abstraction as a result of the replacement of the hole in $\mathcal{C}$ by $e$.

We sometimes write $\mathcal{C}\{\circ\}$ to emphasize the occurrence of the hole in $\mathcal{C}$. Expression contexts are closed under *composition* in that if $\mathcal{C}_1$ and $\mathcal{C}_2$ are expression contexts, then so is

$$\mathcal{C}\{\circ\} := \mathcal{C}_1\{\mathcal{C}_2\{\circ\}\},$$

and we have $\mathcal{C}\{e\} = \mathcal{C}_1\{\mathcal{C}_2\{e\}\}$. The *trivial*, or *identity*, expression context is the "bare hole", written $\circ$, for which $\circ\{e\} = e$.

The static semantics of expressions of $\mathcal{L}\{\mathtt{nat}\to\}$ is extended to expression contexts by defining the typing judgement

$$\mathcal{C} : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')$$

so that if $\Gamma \vdash e : \tau$, then $\Gamma' \vdash \mathcal{C}\{e\} : \tau'$. This judgement may be inductively defined by a collection of rules derived from the static semantics of $\mathcal{L}\{\mathtt{nat}\to\}$ (for which see Rules (14.1)). Some representative rules are as follows:

$$\frac{}{\circ : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma \vdash \tau)} \tag{49.1a}$$

$$\frac{\mathcal{C} : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')}{\mathtt{s}(\mathcal{C}) : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')} \tag{49.1b}$$

$$\frac{\mathcal{C}_2 : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma', x : \tau_1 \vdash \tau_2)}{\lambda(x{:}\tau_1.\mathcal{C}_2) : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau_1 \to \tau_2)} \tag{49.1c}$$

$$\frac{\mathcal{C}_1 : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau_2 \to \tau') \quad \Gamma' \vdash e_2 : \tau_2}{\mathcal{C}_1(e_2) : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')} \tag{49.1d}$$

$$\frac{\Gamma' \vdash e_1 : \tau_2 \to \tau' \quad \mathcal{C}_2 : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau_2)}{e_1(\mathcal{C}_2) : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')} \tag{49.1e}$$

The remaining rules follow a similar pattern.

**Lemma 49.1.** *If* $\mathcal{C} : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')$, *then* $\Gamma' \subseteq \Gamma$, *and if* $\Gamma \vdash e : \tau$, *then* $\Gamma' \vdash \mathcal{C}\{e\} : \tau'$.

Observe that the trivial context consisting only of a "hole" acts as the identity under replacement. Moreover, contexts are closed under composition in the following sense.

**Lemma 49.2.** *If* $\mathcal{C} : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')$, *and* $\mathcal{C}' : (\Gamma' \vdash \tau') \rightsquigarrow (\Gamma'' \vdash \tau'')$, *then* $\mathcal{C}'\{\mathcal{C}\{\circ\}\} : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma'' \vdash \tau'')$.

*Kleene equivalence* determines when two experiments have the same outcome. If $e$ and $e'$ are complete programs, then $e$ is *Kleene equivalent* to $e'$, written $e \simeq e'$, iff $e \mapsto^* \overline{n}$ iff $e' \mapsto^* \overline{n}$. This condition does not demand that both sides terminate, but it can be shown that all well-typed expressions in $\mathcal{L}\{\texttt{nat} \rightarrow\}$ terminate, and so this need not be explicitly required.

**Definition 49.1.** *Suppose that* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau$ *are two expressions of the same type. We say that* $e$ *and* $e'$ *are* observationally equivalent, *written* $\Gamma \vdash e \cong e' : \tau$, *iff* $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ *for every program context* $\mathcal{C}$.

In other words, for all possible experiments, the outcome of an experiment on $e$ is the same as the outcome on $e'$. This is obviously an equivalence relation.

A type-indexed family of equivalence relations $\Gamma \vdash e_1 \equiv e_2 : \tau$ is a *congruence* iff it is preserved by all contexts. That is,

$$\text{if } \Gamma \vdash e \equiv e' : \tau, \text{ then } \Gamma' \vdash \mathcal{C}\{e\} \equiv \mathcal{C}\{e'\} : \tau'$$

for every expression context $\mathcal{C} : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')$. Such a family of relations is *consistent* iff it coincides with Kleene equivalence at the type `nat`.

**Theorem 49.3.** *Observational equivalence is the coarsest consistent congruence on expressions.*

*Proof.* Consistency follows directly from the definition by noting that the trivial context is a program context. Observational equivalence is obviously an equivalence relation. To show that it is a congruence, we need only observe that type-correct composition of a program contex with an arbitrary expression context is again a program context. Finally, it is the coarsest such equivalence relation, for if $\Gamma \not\vdash e \cong e' : \tau$, then there is a program context $\mathcal{C}$ such that $\mathcal{C}\{e\} \not\simeq \mathcal{C}\{e'\}$, so that extending observational equivalence with this pair would be inconsistent. □

Theorem 49.3 on the facing page licenses the principle of *proof by coinduction* to show that two expressions are observational equivalence: to show that $\Gamma \vdash e \cong e' : \tau$, it is enough to exhibit a consistent congruence such that $\Gamma \vdash e \equiv e' : \tau$. It can be difficult, however, to construct such a relation. In the next section we will provide a general method for doing so that will prove useful in many situations.

## 49.2 Logical Equivalence

The key to simplifying reasoning about observational equivalence is to exploit types. Informally, we may classify the uses of values of a type into two broad categories, the *passive* and the *active* uses. The passive uses are those that merely manipulate values without actually inspecting them. For example, we may pass a value of type $\tau$ to a function that merely returns it. The active uses are those that operate on the value itself; these are the elimination forms associated with the type of that value. For the purposes of distinguishing two expressions, it is only the active uses that matter; the passive uses merely manipulate values at arm's length, affording no opportunities to distinguish one from another. This leads to the definition of *typed*, or *logical*, *equivalence*.

**Definition 49.2.** Logical equivalence *is a type-indexed family of relations $e \sim e' : \tau$ between closed expressions of type $\tau$. It is defined along with the relation $e \approx e' : \tau$ of logical equivalence between closed values by induction on the structure of $\tau$ as follows:*

$$
\begin{array}{lll}
e \sim e' : \tau & \textit{iff} & \textit{if } e \mapsto^* e_1 \ \mathsf{val} \textit{ then } e' \mapsto^* e_1' \ \mathsf{val} \textit{ and } e_1 \approx e_1' : \tau \textit{ and} \\
& & \textit{if } e' \mapsto^* e_1' \ \mathsf{val} \textit{ then } e \mapsto^* e_1 \ \mathsf{val} \textit{ and } e_1 \approx e_1' : \tau \\
\\
e \approx e' : \mathtt{nat} & \textit{iff} & e = e' = \mathtt{z}, \textit{ or} \\
& & e = \mathtt{s}(e_1) \textit{ and } e' = \mathtt{s}(e_1') \textit{ and } e_1 \approx e_1' : \mathtt{nat} \\
\\
e \approx e' : \tau_1 \to \tau_2 & \textit{iff} & e = \lambda(x{:}\tau_1.\, e_2),\ e' = \lambda(x{:}\tau_1.\, e_2'), \textit{ and} \\
& & e_1 \approx e_1' : \tau_1 \textit{ implies } [e_1/x]e_2 \sim [e_1'/x]e_2' : \tau_2
\end{array}
$$

Observe that if $e \approx e' : \tau$, then $e$ val and $e'$ val, and that if $e \sim e' : \tau$, then if $d \mapsto e$, then $d \sim e' : \tau$, and if $d' \mapsto e'$, then $e \sim d' : \tau$. Moreover, if $e$ val and $e'$ val, then $e \sim e' : \tau$ iff $e \approx e' : \tau$. Finally, note that $e \approx e' : \mathtt{nat}$ iff $e = e' = \overline{n}$ for some $n \geq 0$.

The general form of logical equivalence is extended to open terms by considering logically equivalent substitution instances. An *expression assignment*, $\gamma$, for a context $\Gamma$ is an assignment of a closed expression $\gamma(x) : \Gamma(x)$ to each variable $x \in dom(\Gamma)$. The relation $\gamma \approx \gamma' : \Gamma$ holds iff $dom(\gamma) = dom(\gamma') = dom(\Gamma)$, and $\gamma(x) \approx \gamma'(x) : \Gamma(x)$ for every variable, $x$, in their common domain. We then define $\Gamma \vdash e \sim e' : \tau$ to mean that $\hat{\gamma}(e) \sim \widehat{\gamma'} e' : \tau$ whenever $\gamma \approx \gamma' : \Gamma$.

## 49.3 Logical and Observational Equivalence Coincide

In this section we prove the coincidence of observational and logical equivalence.

**Lemma 49.4** (Substitution and Functionality). *If* $\Gamma, x : \sigma \vdash e \cong e' : \tau$, *and* $d : \sigma$, *then* $\Gamma \vdash [d/x]e \cong [d/x]e' : \tau$. *Furthermore, if* $d \cong d' : \sigma$, *then* $\Gamma \vdash [d/x]e \cong [d'/x]e' : \tau$.

*Proof.* Suppose that $\mathcal{C} : (\Gamma \vdash \tau) \rightsquigarrow (\vdash \mathtt{nat})$ is a program context. We are to show that $\mathcal{C}\{[d/x]e\} \simeq \mathcal{C}\{[d/x]e'\}$. Since $d$ and $d'$ are closed, and since $\mathcal{C}$ is a program context, this is equivalent to showing that $[d/x]\mathcal{C}\{e\} \simeq [d/x]\mathcal{C}\{e'\}$. Let $\mathcal{D}$ be the context $(\lambda(x{:}\sigma.\mathcal{C}\{\circ\}))(d)$, and note that $\mathcal{D} : (\Gamma, x : \sigma \vdash \tau) \rightsquigarrow (\vdash \mathtt{nat})$. It follows from the assumption that $\mathcal{D}\{e\} \simeq \mathcal{D}\{e'\}$. But by construction $\mathcal{D}\{e\} \simeq [d/x]\mathcal{C}\{e\}$, and $\mathcal{D}\{e'\} \simeq [d/x]\mathcal{C}\{e'\}$. Let $\mathcal{D}'$ be the context $(\lambda(x{:}\sigma.\mathcal{C}\{\circ\}))(d')$, and note that it, too, is a program context. Now if $d \cong d' : \sigma$, by congruence of observational equivalence, $\mathcal{D}\{e\} \cong \mathcal{D}'\{e\} : \mathtt{nat}$, and similarly $\mathcal{D}\{e'\} \cong \mathcal{D}'\{e'\} : \mathtt{nat}$. By consistency of observational equivalence these are valid Kleene equivalences, from which the result follows. □

**Lemma 49.5** (Closure Under Application). *Suppose that* $e \approx e' : \tau_1 \to \tau_2$. *If* $e_1 \approx e_1' : \tau_1$, *then* $e(e_1) \sim e'(e_1') : \tau_2$.

*Proof.* By assumption and Definition 49.2 on the previous page we know that $e = \lambda(x{:}\tau_1.e_2)$ and $e' = \lambda(x{:}\tau_1.e_2')$, and hence that $[e_1/x]e_2 \sim [e_1'/x]e_2' : \tau_2$. But since $e(e_1) \mapsto [e_1/x]e_2$ and $e'(e_1') \mapsto [e_1'/x]e_2'$, the result follows immediately from the definition of logical equivalence of expressions. □

**Lemma 49.6** (Consistency). $e \sim e' : \mathtt{nat}$ *iff* $e \simeq e'$.

*Proof.* Immediate, from Definition 49.2 on the preceding page. □

**Lemma 49.7** (Reflexivity). *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash e \sim e : \tau$.

*Proof.* We are to show that if $\Gamma \vdash e : \tau$ and $\gamma \approx \gamma' : \Gamma$, then $\hat{\gamma}(e) \sim \widehat{\gamma'}(e') : \tau$. The proof proceeds by induction on typing derivations. For example, consider the case of Rule (13.2b), in which $\tau = \tau_1 \rightharpoonup \tau_2$, $e = \lambda(x{:}\tau_1.\, e_2)$ and $e' = \lambda(x{:}\tau_1.\, e_2')$. Since $\hat{\gamma}(e)$ and $\widehat{\gamma'}(e')$ are values, it is enough to show that

$$\lambda(x{:}\tau_1.\, \hat{\gamma}(e_2)) \approx \lambda(x{:}\tau_1.\, \widehat{\gamma'}(e_2')) : \tau_1 \rightharpoonup \tau_2.$$

Assume that $e_1 \approx e_1' : \tau_1$; we are to show that $[e_1/x]\hat{\gamma}(e_2) \sim [e_1'/x]\widehat{\gamma'}(e_2') : \tau_2$. Let $\gamma_2 = \gamma[x \mapsto e_1]$ and $\gamma_2' = \gamma'[x \mapsto e_1']$, and observe that $\gamma_2 \approx \gamma_2' : \Gamma, x : \tau_1$. Therefore, by induction we have $\hat{\gamma_2}(e_2) \sim \hat{\gamma_2'}(e_2') : \tau_2$, from which the result follows directly. □

Symmetry and transitivity of logical equivalence are easily established by induction on types, noting that Kleene equivalence is symmetric and transitive. Logical equivalence is therefore an equivalence relation.

**Lemma 49.8** (Congruence)**.** *If* $\mathcal{C}_0 : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma_0 \vdash \tau_0)$, *and* $\Gamma \vdash e \sim e' : \tau$, *then* $\Gamma_0 \vdash \mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_0$.

*Proof.* By induction on the derivation of the typing of $\mathcal{C}_0$. In the case of Rule (49.1a), the result is immediate. Suppose that the context typing is derived by Rule (49.1c), so that $\mathcal{C}_0 = \lambda(x{:}\tau_1.\, \mathcal{C}_2)$ and $\tau_0 = \tau_1 \to \tau_2$ for some type $\tau_2$. We are to show that

$$\Gamma_0 \vdash \mathcal{C}_0\{e\} \sim \mathcal{C}_0\{e'\} : \tau_0,$$

which is to say that

$$\Gamma_0 \vdash \lambda(x{:}\tau_1.\, \mathcal{C}_2\{e\}) \sim \lambda(x{:}\tau_1.\, \mathcal{C}_2\{e'\}) : \tau_1 \to \tau_2$$

To this end, suppose that $\gamma_0 \approx \gamma_0' : \Gamma_0$, and that $e_1 \approx e_1' : \tau_1$. Let $\gamma_1 = \gamma_0[x \mapsto e_1]$, $\gamma_1' = \gamma_0'[x \mapsto e_1']$, and observe that $\gamma_1 \approx \gamma_1' : \Gamma, x : \tau_1$. By Definition 49.2 on page 365 it is enough to show that

$$\hat{\gamma_1}(\mathcal{C}_2\{e\}) \sim \hat{\gamma_1'}(\mathcal{C}_2\{e'\}) : \tau_2.$$

But this follows immediately from the inductive hypothesis. □

**Theorem 49.9.** *If* $\Gamma \vdash e \sim e' : \tau$, *then* $\Gamma \vdash e \cong e' : \tau$.

*Proof.* By Lemmas 49.6 on the facing page and 49.8, and Theorem 49.3 on page 364. □

**Lemma 49.10.** *If* $e \cong e' : \tau$, *then* $e \sim e' : \tau$.

*Proof.* By induction on the structure of $\tau$. If $\tau = \mathtt{nat}$, then the result is immediate, since the trivial expression context is a program context. If $\tau = \tau_1 \to \tau_2$, then suppose that $e \mapsto^* d$ val and $e' \mapsto^* d'$ val. Since $d$ and $d'$ are closed values of function type, $d = \lambda(x{:}\tau_1.\, e_2)$ and $d' = \lambda(x{:}\tau_1.\, e_2')$ for some $e_2$ and $e_2'$. We are to show that $d \approx d' : \tau_1 \to \tau_2$. So suppose further that $d_1 \approx d_1' : \tau_1$, and show that $[d_1/x]e_2 \sim [d_1'/x]e_2' : \tau_2$. By Theorem 49.9 on the previous page $d_1 \cong d_1' : \tau_1$, and hence by Lemma 49.4 on page 366 $[d_1/x]e_2 \cong [d_1'/x]e_2' : \tau_2$, from which the result follows by induction. □

**Theorem 49.11.** *If* $\Gamma \vdash e \cong e' : \tau$, *then* $\Gamma \vdash e \sim e' : \tau$.

*Proof.* Assume that $\Gamma \vdash e \cong e' : \tau$. Suppose that $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ for some $n \geq 0$, and that $e_1 \approx e_1' : \tau_1$, ..., $e_n \approx e_n' : \tau_n$. By Theorem 49.9 on the previous page we have that $e_1 \cong e_1' : \tau_1, \ldots, e_n \cong e_n' : \tau_n$, and hence by Lemma 49.4 on page 366 we have

$$[e_1, \ldots, e_n/x_1, \ldots, x_n]e \cong [e_1', \ldots, e_n'/x_1, \ldots, x_n]e' : \tau.$$

Therefore by Lemma 49.10 on the previous page we have

$$[e_1, \ldots, e_n/x_1, \ldots, x_n]e \sim [e_1', \ldots, e_n'/x_1, \ldots, x_n]e' : \tau,$$

as required. □

**Corollary 49.12.** $\Gamma \vdash e \cong e' : \tau$ *iff* $\Gamma \vdash e \sim e' : \tau$.

## 49.4 Some Laws of Equivalence

In this section we summarize some useful principles of observational equivalence for $\mathcal{L}\{\mathtt{nat} \to\}$. For the most part these may be proved as laws of logical equivalence, and then appealing to Corollary 49.12.

### 49.4.1 General Laws

Logical equivalence is indeed an equivalence relation: it is reflexive, symmetric, and transitive.

$$\frac{}{\Gamma \vdash e \cong e : \tau} \tag{49.2a}$$

$$\frac{\Gamma \vdash e' \cong e : \tau}{\Gamma \vdash e \cong e' : \tau} \tag{49.2b}$$

$$\frac{\Gamma \vdash e \cong e' : \tau \quad \Gamma \vdash e' \cong e'' : \tau}{\Gamma \vdash e \cong e'' : \tau} \tag{49.2c}$$

Observational equivalence is a congruence: we may replace equals by equals anywhere in an expression.

$$\frac{\Gamma \vdash e \cong e' : \tau \quad \mathcal{C} : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')}{\Gamma' \vdash \mathcal{C}\{e\} \cong \mathcal{C}\{e'\} : \tau'} \tag{49.3a}$$

Equivalence is stable under substitution of values for free variables, and substituting equivalent values in an expression gives equivalent results.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash [e/x]e_2 \cong [e/x]e_2' : \tau'} \tag{49.4a}$$

$$\frac{\Gamma \vdash e_1 \cong e_1' : \tau \quad \Gamma, x : \tau \vdash e_2 \cong e_2' : \tau'}{\Gamma \vdash [e_1/x]e_2 \cong [e_1'/x]e_2' : \tau'} \tag{49.4b}$$

### 49.4.2 Symbolic Evaluation Laws

All of the instruction steps of an operational semantics are valid laws of equivalence. These are called *symbolic evaluation* laws, because they are extensions of the operational semantics to expressions with free variables that may occur anywhere within a program.

$$\overline{\Gamma \vdash \mathtt{rec}\,\mathtt{z}\,\{\mathtt{z} \Rightarrow e_0 \,|\, \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\} \cong e_0 : \tau} \tag{49.5a}$$

$$\frac{e = \mathtt{rec}\,\mathtt{s}(e')\ \{\mathtt{z} \Rightarrow e_0 \,|\, \mathtt{s}(x)\ \mathtt{with}\ y \Rightarrow e_1\}}{\Gamma \vdash e \cong [e', e/x, y]e_1 : \tau} \tag{49.5b}$$

$$\overline{\Gamma \vdash (\lambda(x{:}\tau_1.\, e_2))(e_1) \cong [e_1/x]e_2 : \tau_2} \tag{49.5c}$$

### 49.4.3 Extensionality Laws

Two functions are equivalent if they are equivalent on all arguments.

$$\frac{\Gamma, x : \tau_1 \vdash e(x) \cong e'(x) : \tau_2}{\Gamma \vdash e \cong e' : \tau_1 \to \tau_2} \tag{49.6}$$

Consequently, every expression of function type is equivalent to a $\lambda$-abstraction:

$$\frac{}{\Gamma \vdash e \cong \lambda(x{:}\tau_1.\, e(x)) : \tau_1 \to \tau_2} \tag{49.7}$$

### 49.4.4 Induction Law

An equation involving a free variable, $x$, of type `nat` can be proved by induction on $x$.

$$\frac{\Gamma \vdash [\overline{n}/x]e \cong [\overline{n}/x]e' : \tau \text{ (for every } n \in \mathbb{N})}{\Gamma, x : \mathtt{nat} \vdash e \cong e' : \tau} \tag{49.8a}$$

To apply the induction rule, we proceed by mathematical induction on $n \in \mathbb{N}$, which reduces to showing:

1. $\Gamma \vdash [\mathtt{z}/x]e \cong [\mathtt{z}/x]e' : \tau$, and
2. $\Gamma \vdash [\mathtt{s}(\overline{n})/x]e \cong [\mathtt{s}(\overline{n})/x]e' : \tau$, if $\Gamma \vdash [\overline{n}/x]e \cong [\overline{n}/x]e' : \tau$.

## 49.5 Exercises

# Chapter 50

# Parametricity

The motivation for introducing polymorphism was to enable more programs to be written — those that are "generic" in one or more types, such as the composition function given in Chapter 23. Then if a program *does not* depend on the choice of types, we can code it using polymorphism. Moreover, if we wish to insist that a program *can not* depend on a choice of types, we demand that it be polymorphic. Thus polymorphism can be used both to expand the class of programs we may write, and also to limit the class of programs that are permissible in a given context.

The restrictions imposed by polymorphic typing give rise to the experience that in a polymorphic functional language, if the types are correct, then the program is correct. Roughly speaking, if a function has a polymorphic type, then the strictures of type genericity vastly cut down the set of programs with that type. Thus if you have written a program with this type, it is quite likely to be the one you intended!

The technical foundation for these remarks is called *parametricity*. The goal of this chapter is to give an account of parametricity for $\mathcal{L}\{\mathbf{2}{\to}\forall\}$. *For the sake of technical simplicity, we consider a call-by-name dynamic semantics for* $\mathcal{L}\{\mathbf{2}{\to}\forall\}$. The results described herein can be extended to the call-by-value case as well, at the expense of some additional complications.

## 50.1 Overview

We will begin with an informal discussion of parametricity based on a "seat of the pants" understanding of the set of well-formed programs of a type.

Suppose that a function value $f$ has the type $\forall(t.\,t \to t)$. What function could it be? When instantiated at a type $\tau$ it should evaluate to a function

$g$ of type $\tau \to \tau$ that, when further applied to a value $v$ of type $\tau$ returns a value $v'$ of type $\tau$. Since $f$ is polymorphic, $g$ cannot depend on $v$, so $v'$ must be $v$. In other words, $g$ must be the identity function at type $\tau$, and $f$ must therefore be the *polymorphic identity*.

Suppose that $f$ is a function of type $\forall(t.t)$. What function could it be? A moment's thought reveals that it cannot exist at all! For it must, when instantiated at a type $\tau$, return a value of that type. But not every type has a value (including this one), so this is an impossible assignment. The only conclusion is that $\forall(t.t)$ is an *empty* type.

Let $N$ be the type of polymorphic Church numerals introduced in Chapter 23, namely $\forall(t.t \to (t \to t) \to t)$. What are the values of this type? Given any type $\tau$, and values $z : \tau$ and $s : \tau \to \tau$, the expression

$$f[\tau](z)(s)$$

must yield a value of type $\tau$. Moreover, it must behave uniformly with respect to the choice of $\tau$. What values could it yield? The only way to build a value of type $\tau$ is by using the element $z$ and the function $s$ passed to it. A moment's thought reveals that the application must amount to the $n$-fold composition

$$s(s(\ldots s(z)\ldots)).$$

That is, the elements of $N$ are in one-to-one correspondence with the natural numbers.

## 50.2 Observational Equivalence

In this section we give a precise formulation of observational equivalence for $\mathcal{L}\{\mathbf{2}{\to}\forall\}$, the extension of $\mathcal{L}\{{\to}\forall\}$ with the base type, **2**, containing two distinct values, **tt** and **ff**.

An expression context is, as in Chapter 49, an expression with a single occurrence of a "hole" that may be filled by an open expression. The typing judgement for expression contexts,

$$\mathcal{C} : (\Delta; \Gamma \vdash \tau) \rightsquigarrow (\Delta'; \Gamma' \vdash \tau'),$$

is defined as in Chapter 49 to mean that $\mathcal{C}$ exposes the variables $\Gamma$, and is such that $\Delta'; \Gamma' \vdash \mathcal{C}\{e\} : \tau'$ whenever $\Delta; \Gamma \vdash e : \tau$.

As in Chapter 49, we define a program to be a closed expression of type **2**, and define a program context to be a closed expression context of this type. Kleene equivalence is defined for programs is defined by $e \simeq e'$ iff

(1) $e \mapsto^* \mathbf{tt}$ iff $e' \mapsto^* \mathbf{tt}$, and (2) $e \mapsto^* \mathbf{ff}$ iff $e' \mapsto^* \mathbf{ff}$. This is obviously an equivalence relation. We say that a type-indexed family of relations between closed expressions of the same type is *consistent* iff it coincides with Kleene equivalence at the type **2**.

**Definition 50.1.** *Two expressions of the same type are* observationally equivalent, *written* $\Delta;\Gamma \vdash e \cong e' : \tau$, *iff* $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$ *whenever* $\mathcal{C} : (\Delta;\Gamma \vdash \tau) \rightsquigarrow (\vdash \mathbf{2})$.

**Lemma 50.1.** *Observational equivalence is the coarsest consistent congruence.*

*Proof.* The composition of a program context with another context is itself a program context. It is consistent by virtue of the empty context being a program context. □

**Lemma 50.2** (Closed Substitution and Functionality)**.**

1. *If* $\Delta, t;\Gamma \vdash e \cong e' : \tau$ *and* $\rho$ type, *then* $\Delta;[\rho/t]\Gamma \vdash [\rho/t]e \cong [\rho/t]e' : [\rho/t]\tau$.
2. *If* $\Gamma, x : \sigma \vdash e \cong e' : \tau$ *and* $d : \sigma$, *then* $\Gamma \vdash [d/x]e \cong [d/x]e' : \tau$. *Moreover, if* $d \cong d' : \sigma$, *then* $\Gamma \vdash [d/x]e \cong [d'/x]e : \tau$, *and similarly for* $e'$.

*Proof.* 1. Let $\mathcal{C} : (\Delta;[\rho/t]\Gamma \vdash [\rho/t]\tau) \rightsquigarrow (\vdash \mathbf{2})$ be a program context. We are to show that

$$\mathcal{C}\{[\rho/t]e\} \simeq \mathcal{C}\{[\rho/t]e'\}.$$

Since $\mathcal{C}$ and $\rho$ are closed, this is equivalent to

$$[\rho/t]\mathcal{C}\{e\} \simeq [\rho/t]\mathcal{C}\{e'\}.$$

Let $\mathcal{C}'$ be the context $\Lambda(t.\mathcal{C}\{\circ\})\,[\rho]$, and observe that

$$\mathcal{C}' : (\Delta, t;\Gamma \vdash \tau) \rightsquigarrow (\vdash \mathbf{2}).$$

Therefore, from the assumption, it follows that

$$\mathcal{C}'\{e\} \simeq \mathcal{C}'\{e'\}.$$

But $\mathcal{C}'\{e\} \simeq [\rho/t]\mathcal{C}\{e\}$, and $\mathcal{C}'\{e'\} \simeq [\rho/t]\mathcal{C}\{e'\}$, from which the result follows.

2. By an argument essentially similar to that for Lemma 49.4 on page 366. □

## 50.3 Logical Equivalence

In this section we introduce a form of logical equivalence that captures the informal concept of parametricity, and also provides a characterization of observational equivalence. This will permit us to derive properties of observational equivalence of polymorphic programs of the kind suggested earlier.

The definition of logical equivalence for $\mathcal{L}\{\mathbf{2}{\to}\forall\}$ is somewhat more complex than for $\mathcal{L}\{\mathbf{2}\mathtt{nat}{\to}\}$. The main idea is to define logical equivalence for a polymorphic type, $\forall(t.\tau)$ to satisfy a very strong condition that captures the essence of parametricity. As a first approximation, we might say that two expressions, $e$ and $e'$, of this type should be logically equivalent if they are logically equivalent for "all possible" interpretations of the type $t$. More precisely, we might require that $e[\rho]$ be related to $e'[\rho]$ at type $[\rho/t]\tau$, for any choice of type $\rho$. But this runs into two problems, one technical, the other conceptual. The same device will be used to solve both problems.

The technical problem stems from impredicativity. In Chapter 49 logical equivalence is defined by induction on the structure of types. But when polymorphism is impredicative, the type $[\rho/t]\tau$ might well be larger than $\forall(t.\tau)$! At the very least we would have to justify the definition of logical equivalence on some other grounds, but no criterion appears to be available. The conceptual problem is that, even if we could make sense of the definition of logical equivalence, it would be too restrictive. For such a definition amounts to saying that the unknown type $t$ is to be interpreted as logical equivalence at whatever type it turns out to be when instantiated. To obtain useful parametricity results, we shall ask for much more than this. What we shall do is to consider *separately* instances of $e$ and $e'$ by types $\rho$ and $\rho'$, and treat the type variable $t$ as standing for *any relation* (of a suitable class) between $\rho$ and $\rho'$. One may suspect that this is asking too much: perhaps logical equivalence is the *empty* relation! Surprisingly, this is not the case, and indeed it is this very feature of the definition that we shall exploit to derive parametricity results about the language.

To manage both of these problems we will consider a generalization of logical equivalence that is parameterized by a relational interpretation of the free type variables of its classifier. The parameters determine a separate binding for each free type variable in the classifier for each side of the equation, with the discrepancy being mediated by a specified relation between them. This permits us to consider a notion of "equivalence" between two expressions of different type—they are equivalent, *modulo* a relation

between the interpretations of their free type variables.

We will restrict attention to a certain class of "admissible" binary relations between closed expressions. The conditions are imposed to ensure that logical equivalence and observational equivalence coincide.

**Definition 50.2** (Admissibility)**.** *A relation $R$ between expressions of types $\rho$ and $\rho'$ is* admissible*, written $R : \rho \leftrightarrow \rho'$, iff it satisfies two requirements:*

1. *Respect for observational equivalence: if $R(e, e')$ and $d \cong e : \rho$ and $d' \cong e' : \rho'$, then $R(d, d')$.*
2. *Closure under converse evaluation: if $R(e, e')$, then if $d \mapsto e$, then $R(d, e')$ and if $d' \mapsto e'$, then $R(e, d')$.*

The second of these conditions will turn out to be a consequence of the first, but we are not yet in a position to establish this fact.

The judgement $\delta : \Delta$ states that $\delta$ is a *type assignment* that assigns a closed type to each type variable $t \in \Delta$. A type assignment, $\delta$, induces a substitution function, $\hat{\delta}$, on types given by the equation

$$\hat{\delta}(\tau) = [\delta(t_1), \ldots, \delta(t_n)/t_1, \ldots, t_n]\tau,$$

and similarly for expressions. Substitution is extended to contexts pointwise by defining $\hat{\delta}(\Gamma)(x) = \hat{\delta}(\Gamma(x))$ for each $x \in dom(\Gamma)$.

Let $\delta$ and $\delta'$ be two type assignments of closed types to the type variables in $\Delta$. A *relation assignment*, $\eta$, between $\delta$ and $\delta'$ is an assignment of an admissible relation $\eta(t) : \delta(t) \leftrightarrow \delta'(t)$ for each $t \in \Delta$. The judgement $\eta : \delta \leftrightarrow \delta'$ states that $\eta$ is a relation assignment between $\delta$ and $\delta'$.

Logical equivalence is defined in terms of its generalization, called *parameterized logical equivalence*, written $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$, is defined as follows.

**Definition 50.3** (Parameterized Logical Equivalence)**.** *The relation $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$ is defined by induction on the structure of $\tau$ by the following conditions:*

| | | |
|---|---|---|
| $e \sim e' : \mathbf{2}\ [\eta : \delta \leftrightarrow \delta']$ | *iff* | $e \simeq e'$ |
| $e \sim e' : t\ [\eta : \delta \leftrightarrow \delta']$ | *iff* | $\eta(t)(e, e')$ |
| $e \sim e' : \tau_1 \to \tau_2\ [\eta : \delta \leftrightarrow \delta']$ | *iff* | $e_1 \sim e_1' : \tau_1\ [\eta : \delta \leftrightarrow \delta']$ *implies* $e\,(e_1) \sim e'\,(e_1') : \tau_2\ [\eta : \delta \leftrightarrow \delta']$ |
| $e \sim e' : \forall\,(t.\tau)\ [\eta : \delta \leftrightarrow \delta']$ | *iff* | *for every $\rho$, $\rho'$, and every $R : \rho \leftrightarrow \rho'$,* $e\,[\rho] \sim e'\,[\rho']\ : \tau\ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']]$ |

Logical equivalence is defined in terms of parameterized logical equivalence by considering all possible interpretations of its free type- and expression variables. An *expression assignment*, $\gamma$, for a context $\Gamma$, written $\gamma : \Gamma$, is an assignment of a closed expression $\gamma(x) : \Gamma(x)$ to each variable $x \in dom(\Gamma)$. An expression assignment, $\gamma : \Gamma$, induces a substitution function, $\hat{\gamma}$, defined by the equation

$$\hat{\gamma}(e) = [\gamma(x_1), \ldots, \gamma(x_n)/x_1, \ldots, x_n]e,$$

where the domain of $\Gamma$ consists of the variables $x_1, \ldots, x_n$.

The relation $\gamma \sim \gamma' : \Gamma\ [\eta : \delta \leftrightarrow \delta']$ is defined to hold iff $dom(\gamma) = dom(\gamma') = dom(\Gamma)$, and $\gamma(x) \sim \gamma'(x) : \Gamma(x)\ [\eta : \delta \leftrightarrow \delta']$ for every variable, $x$, in their common domain.

**Definition 50.4** (Logical Equivalence)**.** *The expressions $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma \vdash e' : \tau$ are* logically equivalent*, written $\Delta; \Gamma \vdash e \sim e' : \tau$ iff for every assigment $\delta$ and $\delta'$ of closed types to type variables in $\Delta$, and every relation assignment $\eta : \delta \leftrightarrow \delta'$, if $\gamma \sim \gamma' : \Gamma\ [\eta : \delta \leftrightarrow \delta']$, then $\hat{\gamma}(\hat{\delta}(e)) \sim \widehat{\gamma'}(\widehat{\delta'}(e')) : \tau\ [\eta : \delta \leftrightarrow \delta']$.*

When $e$, $e'$, and $\tau$ are closed, then this definition states that $e \sim e' : \tau$ iff $e \sim e' : \tau\ [\emptyset : \emptyset \leftrightarrow \emptyset]$, so that logical equivalence is indeed a special case of its generalization.

**Lemma 50.3** (Closure under Converse Evaluation)**.** *Suppose that $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$. If $d \mapsto e$, then $d \sim e' : \tau$, and if $d' \mapsto e'$, then $e \sim d' : \tau$.*

*Proof.* By induction on the structure of $\tau$. When $\tau = \mathbf{2}$, the result holds by definition of Kleene equivalence. When $\tau = t$, the result holds because all relations under consideration are closed under converse evaluation. Otherwise the result follows by induction, making use of the definition of the transition relation for applications and type applications. □

**Lemma 50.4** (Respect for Observational Equivalence)**.** *Suppose that $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$. If $d \cong e : \hat{\delta}(\tau)$ and $d' \cong e' : \widehat{\delta'}(\tau)$, then $d \sim d' : \tau\ [\eta : \delta \leftrightarrow \delta']$.*

*Proof.* By induction on the structure of $\tau$, relying on the definition of admissibility, and the congruence property of observational equivalence. For example, if $\tau = \forall(t.\sigma)$, then we are to show that for every $R : \rho \leftrightarrow \rho'$,

$$d\,[\rho] \sim d'\,[\rho'] : \sigma\ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']].$$

Since observational equivalence is a congruence, $d\,[\rho] \cong e\,[\rho] : [\rho/t]\hat{\delta}(\sigma)$, $d'\,[\rho] \cong e'\,[\rho] : [\rho'/t]\widehat{\delta'}(\sigma)$. From the assumption it follows that

$$e\,[\rho] \sim e'\,[\rho'] : \sigma\ [\eta[t \mapsto R] : \delta[t \mapsto \rho] \leftrightarrow \delta'[t \mapsto \rho']],$$

from which the result follows by induction. □

**Corollary 50.5.** *The relation $e \sim e' : \tau\ [\eta : \delta \leftrightarrow \delta']$ is an admissible relation between closed types $\hat{\delta}(\tau)$ and $\widehat{\delta'}(\tau)$.*

*Proof.* By Lemmas 50.3 on the facing page and 50.4 on the preceding page. □

Logical Equivalence respects observational equivalence.

**Corollary 50.6.** *If $\Delta; \Gamma \vdash e \sim e' : \tau$, and $\Delta; \Gamma \vdash d \cong e : \tau$ and $\Delta; \Gamma \vdash d' \cong e' : \tau$, then $\Delta; \Gamma \vdash d \sim d' : \tau$.*

*Proof.* By Lemma 50.2 on page 373 and Corollary 50.5. □

**Lemma 50.7** (Compositionality)**.** *Suppose that*

$$e \sim e' : \tau\ [\eta[t \mapsto R] : \delta[t \mapsto \hat{\delta}(\rho)] \leftrightarrow \delta'[t \mapsto \widehat{\delta'}(\rho)]],$$

*where $R : \hat{\delta}(\rho) \leftrightarrow \widehat{\delta'}(\rho)$ is such that $R(d, d')$ holds iff $d \sim d' : \rho\ [\eta : \delta \leftrightarrow \delta']$. Then $e \sim e' : [\rho/t]\tau\ [\eta : \delta \leftrightarrow \delta']$.*

*Proof.* By induction on the structure of $\tau$. When $\tau = t$, the result is immediate from the definition of the relation $R$. When $\tau = t' \neq t$, or $\tau = \mathbf{2}$, the result holds vacuously. When $\tau = \tau_1 \to \tau_2$ or $\tau = \forall(u.\tau)$, where without loss of generality $u \neq t$ and $u \# \rho$, the result follows by induction. □

Despite the strong conditions on polymorphic types, logical equivalence is not vacuous—in fact, expression satisfies its constraints.

**Theorem 50.8** (Reynolds)**.** *If $e : \tau$ is a closed expression, then $e \sim e : \tau$.*

*Proof.* By induction on derivations of the typing judgement for $\mathcal{L}\{\mathbf{2}{\to}\forall\}$. We consider two representative cases here.

**Rule** (23.2a) By induction we have that for all $\delta : \Delta$, $\delta' : \Delta$, $\eta : \delta \leftrightarrow \delta'$, and all $\rho$, $\rho'$, and $R : \rho \leftrightarrow \rho'$,

$$[\rho/t]\hat{\gamma}(\hat{\delta}(e)) \sim [\rho'/t]\widehat{\gamma'}(\widehat{\delta'}(e)) : \tau\ [\eta_* : \delta_* \leftrightarrow \delta'_*],$$

where $\eta_* = \eta[t \mapsto R]$, $\delta_* = \delta[t \mapsto \rho]$, and $\delta'_* = \delta'[t \mapsto \rho']$. Since

$$\Lambda(t.\hat{\gamma}(\hat{\delta}(e)))\,[\rho] \mapsto^* [\rho/t]\hat{\gamma}(\hat{\delta}(e))$$

and

$$\Lambda(t.\widehat{\gamma'}(\widehat{\delta'}(e)))\,[\rho'] \mapsto^* [\rho'/t]\widehat{\gamma'}(\widehat{\delta'}(e)),$$

the result follows by Lemma 50.3 on the facing page.

**Rule** (23.2b) By induction we have that for all $\delta : \Delta$, $\delta' : \Delta$, $\eta : \delta \leftrightarrow \delta'$,

$$\hat{\gamma}(\hat{\delta}(e)) \sim \widehat{\gamma'}(\widehat{\delta'}(e)) : \forall(t.\tau)\ [\eta : \delta \leftrightarrow \delta']$$

Let $\hat{\rho} = \hat{\delta}(\rho)$ and $\hat{\rho'} = \widehat{\delta'}(\rho)$. Define the relation $R : \hat{\rho} \leftrightarrow \hat{\rho'}$ by $R(d, d')$ iff $d \sim d' : \rho\ [\eta : \delta \leftrightarrow \delta']$. By Corollary 50.5 on the previous page, this relation is admissible.

By the definition of logical equivalence at polymorphic types, we obtain

$$\hat{\gamma}(\hat{\delta}(e))\,[\hat{\rho}] \sim \widehat{\gamma'}(\widehat{\delta'}(e))\,[\hat{\rho'}] : \tau\ [\eta[t \mapsto R] : \delta[t \mapsto \hat{\rho}] \leftrightarrow \delta'[t \mapsto \hat{\rho'}]].$$

By Lemma 50.7 on the preceding page

$$\hat{\gamma}(\hat{\delta}(e))\,[\hat{\rho}] \sim \widehat{\gamma'}(\widehat{\delta'}(e))\,[\hat{\rho'}] : [\rho/t]\tau\ [\eta : \delta \leftrightarrow \delta']$$

But

$$\begin{aligned} \hat{\gamma}(\hat{\delta}(e))\,[\hat{\rho}] &= \hat{\gamma}(\hat{\delta}(e))\,[\hat{\delta}(\rho)] && (50.1)\\ &= \hat{\gamma}(\hat{\delta}(e\,[\rho])), && (50.2) \end{aligned}$$

and similarly

$$\begin{aligned} \widehat{\gamma'}(\widehat{\delta'}(e))\,[\hat{\rho'}] &= \widehat{\gamma'}(\widehat{\delta'}(e))\,[\widehat{\delta'}(\rho)] && (50.3)\\ &= \widehat{\gamma'}(\widehat{\delta'}(e\,[\rho])), && (50.4) \end{aligned}$$

from which the result follows.

□

**Corollary 50.9.** *If* $\Delta; \Gamma \vdash e \cong e' : \tau$, *then* $\Delta; \Gamma \vdash e \sim e' : \tau$.

*Proof.* By Theorem 50.8 on the previous page $\Delta; \Gamma \vdash e \sim e : \tau$, and hence by Corollary 50.6 on the preceding page, $\Delta; \Gamma \vdash e \sim e' : \tau$. □

**Lemma 50.10** (Congruence)**.** *If* $\Delta', \Delta; \Gamma', \Gamma \vdash e \sim e' : \tau$ *and* $\mathcal{C} : (\Delta; \Gamma \vdash \tau) \rightsquigarrow (\Delta'; \Gamma' \vdash \tau')$, *then* $\Delta'; \Gamma' \vdash \mathcal{C}\{e\} \sim \mathcal{C}\{e'\} : \tau$.

*Proof.* By induction on the structure of $\mathcal{C}$. □

**Corollary 50.11.** *If* $\Delta; \Gamma \vdash e \sim e' : \tau$, *then* $\Delta; \Gamma \vdash e \cong e' : \tau$.

*Proof.* Logical equivalence is obviously consistent, and by Lemma 50.10, it is a congruence, and hence is contained in observational equivalence. □

**Corollary 50.12.** *Logical and observational equivalence coincide.*

*Proof.* By Corollaries 50.9 on the facing page and 50.11 on the preceding page. □

If $d : \tau$ and $d \mapsto e$, then $d \sim e : \tau$, and hence by Corollary 50.11 on the facing page, $d \cong e : \tau$. Therefore if a relation respects observational equivalence, it must also be closed under converse evaluation. This shows that the second condition on admissibility is redundant, though it cannot be omitted at such an early stage.

## 50.4 Relational Parametricity

Using the Parametricity Theorem we may prove results about the inhabitants of polymorphic types. For example, if $e : \forall(t.t \to t)$, then we may show that if $\rho$ is any type, and $d : \rho$, then $e[\rho](d) \mapsto^* d$. Let $R$ be such that $R(d, d')$ iff $d \mapsto^* d$ and $d' \mapsto^* d$. Observe that $R : \rho \leftrightarrow \rho$, and that $R(d, d)$. It follows by Theorem 50.8 on page 377 that $R(e[\rho](d), e[\rho](d))$, which is to say that $e[\rho](d) \mapsto^* d$, as required.

*(More examples to follow....)*

## 50.5 Exercises

# Chapter 51

# Representation Independence

Parametricity is the essence of representation independence. The typing rules for `open` given in 24.1 on page 190 ensure that the client of an abstract type is polymorphic in the representation type. According to our informal understanding of parametricity this means that the client behavior of the client is independent of the choice of representation.

To say that no client can distinguish between two implementations of the same existential type is just to say that these two implementations are observationally equivalent as expressions of the existential type. Therefore representation independence for abstract types boils down to observational equivalence. But, as we have argued in Chapters 49 and 50, it can be quite difficult to reason directly about observational equivalence. A useful sufficient condition is derived from the concept of logical equivalence defined in Chapter 50 for polymorphic languages. This condition is called *bisimilarity*.

## 51.1 Bisimilarity of Packages

For two packages

$$e_1' = \texttt{pack}\,\rho_1\,\texttt{with}\,e_1\,\texttt{as}\,\exists(t.\tau)$$

and

$$e_2' = \texttt{pack}\,\rho_2\,\texttt{with}\,e_2\,\texttt{as}\,\exists(t.\tau)$$

of the same existential type, $\exists(t.\tau)$, to be observationally equivalent, it is sufficient to exhibit a relation $R : \rho_1 \leftrightarrow \rho_2$ between closed expressions of types $\rho_1$ and $\rho_2$, respectively, such that

$$e_1 \sim e_2 : \tau\ [[t \mapsto R] : [t \mapsto \rho_1] \leftrightarrow [t \mapsto \rho_2]].$$

This means that $e_1$ and $e_2$ are to be logically related as elements of type $\tau$, under the assumption that elements of type $t$ (which may occur free in $\tau$) are related by the specified relation $R$. When this is the case, we say that $R$ is a *bisimulation* between the two packages, and that the packages are thereby *bisimilar*.

Recall from Chapter 24 that the client, $e_c$, of the abstract type $\exists(t.\tau)$ is such that $t$ type, $x : \tau \vdash e_c : \tau_c$ for some type $\tau_c$ such that $t \# \tau_c$. It follows from Theorem 50.8 on page 377 that

$$[e_1/x]e_c \sim [e_2/x]e_c : \tau_c \; [[t \mapsto R] : [t \mapsto \rho_1] \leftrightarrow [t \mapsto \rho_2]]$$

whenever

$$e_1 \sim e_2 : \tau \; [[t \mapsto R] : [t \mapsto \rho_1] \leftrightarrow [t \mapsto \rho_2]].$$

It follows that

$$\texttt{open}\, e_1' \,\texttt{as}\, t \,\texttt{with}\, x\texttt{:}\tau \,\texttt{in}\, e_c \sim \texttt{open}\, e_2' \,\texttt{as}\, t \,\texttt{with}\, x\texttt{:}\tau \,\texttt{in}\, e_c : \tau_c.$$

That is, the two implementations are indistinguishable by any client of the abstraction. This crucial property is called *representation independence* for abstract types. It is crucial that $t \# \tau_c$ to ensure that the equivalence of the client under change of representation is independent of the relation $R$, which governs only the "private" parts of the abstraction.

Representation independence the following technique for proving the correctness of an ADT implementation. Suppose that we have a "clever" implementation of an abstract type $\exists(t.\tau)$ whose correctness we wish to verify. Let us call this the *candidate* implementation. To prove correctness of the candidate, we exhibit a *reference* implementation that is taken to be manifestly correct (or proved correct by a separate argument), and show that the reference and candidate implementations are bisimilar. It follows that they are observationally equivalent, and hence interchangeable in all contexts. In other words the candidate is "as correct as" the reference implementation.

## 51.2 Two Representations of Queues

Returning to the queues example, let us take as a reference implementation the package determined by representing queues as lists. As a candidate implementation we take the package corresponding to the following ML code:

```
structure QFB :> QUEUE =
  struct
    type queue = int list * int list
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    fun remove (bs, nil) = remove (nil, rev bs)
      | remove (bs, f::fs) = (f, (bs, fs))
  end
```

We will show that `QL` and `QFB` are bisimilar, and therefore indistinguishable by any client.

Letting $\rho_{\mathsf{ls}} = \texttt{nat list}$ and $\rho_{\mathsf{fb}} = \texttt{nat list} \times \texttt{nat list}$, define the relation $R : \rho_{\mathsf{ls}} \leftrightarrow \rho_{\mathsf{fb}}$ as follows:

$$R = \{\, (l, \langle b, f\rangle)) \mid l \cong b \mathbin{@} \texttt{rev}(f) : \texttt{nat list} \,\}$$

We will show that $R$ is a bisimulation by showing that implementations of `empty`, `insert`, and `remove` determined by the structures `QL` and `QFB` are equivalent relative to $R$.

To do so, we will establish the following facts:

1. $\texttt{QL.empty}\ R\ \texttt{QFB.empty}$.
2. Assuming that $m \sim n : \texttt{nat}$ and $l\ R\ \langle b, f\rangle$, show that
$$\texttt{QL.insert}(\langle m, l\rangle)\ R\ \texttt{QFB.insert}(\langle n, \langle b, f\rangle\rangle).$$
3. Assuming that $l\ R\ \langle b, f\rangle$, show that
$$\texttt{QL.remove}(l) \sim \texttt{QFB.remove}(\langle b, f\rangle) : \texttt{nat} \times t\ [[t \mapsto R] : [t \mapsto \rho_{\mathsf{ls}}] \leftrightarrow [t \mapsto \rho_{\mathsf{fb}}]].$$

Observe that the latter two statements amount to the assertion that the operations *preserve* the relation $R$ — they map related input queues to related output queues.

The proofs of these facts are relatively straightforward, given some relatively obvious lemmas about expression equivalence.

1. To show that $\texttt{QL.empty}\ R\ \texttt{QFB.empty}$, it suffices to show that
$$\texttt{nil} \mathbin{@} \texttt{rev(nil)} \cong \texttt{nil} : \texttt{nat list},$$
which follows by symbolic execution, using the definitions of the operations involved.

2. For `insert`, we assume that $m \sim n : \mathtt{nat}$ and $l\ R\ \langle b, f\rangle$, and prove that

$$\mathtt{QL.insert}(m, l)\ R\ \mathtt{QFB.insert}(n, \langle b, f\rangle).$$

By the definition of `QL.insert`, the left-hand side is observationally equivalent to $m :: l$, and by the definition of `QR.insert`, the right-hand side is observationally equivalent to $\langle n :: b, f\rangle$. It suffices to show that

$$m :: l \cong (n :: b)\ \mathtt{@}\ \mathtt{rev}(f) : \mathtt{nat\ list}.$$

Calculating, we obtain

$$(n :: b)\ \mathtt{@}\ \mathtt{rev}(f) \cong n :: (b\ \mathtt{@}\ \mathtt{rev}(f)) : \mathtt{nat\ list}$$

and

$$n :: (b\ \mathtt{@}\ \mathtt{rev}(f)) \cong n :: l : \mathtt{nat\ list},$$

since $l \cong b\ \mathtt{@}\ \mathtt{rev}(f) : \mathtt{nat\ list}$. Since $m \sim n : \mathtt{nat}$, it follows that $m = n$, which completes the proof.

3. For `remove`, we assume that $l$ is related by $R$ to $\langle b, f\rangle$, which is to say that $l \cong b\ \mathtt{@}\ \mathtt{rev}(f) : \mathtt{nat\ list}$. We are to show

$$\mathtt{QL.remove}(l) \sim \mathtt{QFB.remove}(\langle b, f\rangle) : \mathtt{nat} \times t\ [[t \mapsto R] : [t \mapsto \rho_{\mathsf{ls}}] \leftrightarrow [t \mapsto \rho_{\mathsf{fb}}]].$$

Assuming that the queue is non-empty, so that removing an element is well-defined, it can be shown that $l \cong l'\ \mathtt{@}\ [m] : \mathtt{nat\ list}$ for some $l'$ and $m$. We proceed by cases according to whether or not $f$ is empty. If $f$ is non-empty, then it can be shown that $f \cong n :: f' : \mathtt{nat\ list}$ for some $n$ and $f'$. Then by the definition of `QFB.remove`,

$$\mathtt{QFB.remove}(\langle b, f\rangle) \cong \langle n, \langle b, f'\rangle\rangle : \mathtt{nat} \times t,$$

taking equality at type $t$ to be the relation $R$. We must show that

$$\langle m, l'\rangle \sim \langle n, \langle b, f'\rangle\rangle : \mathtt{nat} \times t,$$

with $t$ equality being $R$. This means that we must show that $m = n$ and $l' \cong b\ \mathtt{@}\ \mathtt{rev}(f') : \mathtt{nat\ list}$.

Calculating from our assumptions,

$$\begin{aligned} l &\cong l'\ \mathtt{@}\ [m] \\ &\cong b\ \mathtt{@}\ \mathtt{rev}(f) \\ &\cong b\ \mathtt{@}\ \mathtt{rev}(n :: f') \\ &\cong b\ \mathtt{@}\ (\mathtt{rev}(f')\ \mathtt{@}\ [n]) \\ &\cong (b\ \mathtt{@}\ \mathtt{rev}(f'))\ \mathtt{@}\ [n], \end{aligned}$$

from which the result follows. Finally, if $f$ is empty, then it can be shown that $b \cong b' \texttt{@} [n] : \texttt{nat list}$ for some $b'$ and $n$. But then

$$\texttt{rev}(b) \cong n :: \texttt{rev}(b') : \texttt{nat list},$$

which reduces to the case for $f$ non-empty.

This completes the proof — by representation independence the reference and candidate implementations are equivalent.

## 51.3 Exercises

# Chapter 52

# Process Equivalence

Leibniz's Principle, when applied to program fragments, states that two fragments are equal if and only if they cannot be separated by a specified class of experiments that may be performed on them. In the case of pure functional programs the experiments are program contexts, which are complete programs computing an answer of finitely observable type containing a "hole" where the fragments are to be placed. If the answers are the same whenever the one fragment is replaced by the other in all possible program contexts, then those two fragments are equal.

In a richer language, such as one with input and output, it makes sense to consider the effect of the fragments on the environment as well as, or instead of, considering their contribution to the final answer computed by a complete program. For example, one might consider observations to consist of everything that is output onto the screen during execution, and regard two fragments as equal iff they induce the same output in all program contexts.

More generally still, we may consider equivalence of processes based on the actions they take during execution, equating two process expressions iff they induce the same actions when placed in an arbitrary concurrent programming context. This includes input and output as special cases, for we may regard the screen as a concurrent process that interacts with other processes along specified channels (*e.g.*, `stdin` and `stdout`).

This leads us to consider a theory of process equivalence based on the availability of visible actions that may affect their synchronization behavior. We will confine ourselves to the pure synchronization calculus of Chapter 41 so as to avoid the complexities of taking into account the possibilities afforded by channel-passing.

## 52.1 Labelled Transition Systems

To define equivalence of processes it is necessary to first define the permissible actions of a process. This is specified by defining two families of transitions systems, one whose states are process expressions, and the other whose states are event expressions. In each case a label is a *generalized action*, either a synchronization action in the sense of Chapter 41, or the distinguished *silent action*, $\tau$. Informally, the judgement $P \stackrel{\alpha}{\longmapsto} P'$ means that process $P$ may perform action $\alpha$ and become process $P'$, and similarly for event expressions. The silent action represents an externally invisible step taken by a process, rather than an externally visible one with which another process may synchronize. Silent actions correspond to reactions as defined in Chapter 41.

The precise definitions of the labelled transition system for process calculus is given by the following rules.

$$\frac{E \stackrel{\alpha}{\longmapsto} E'}{\texttt{await}(E) \stackrel{\alpha}{\longmapsto} \texttt{await}(E')} \tag{52.1a}$$

$$\frac{P_1 \stackrel{\alpha}{\longmapsto} P_1'}{P_1 \parallel P_2 \stackrel{\alpha}{\longmapsto} P_1' \parallel P_2} \tag{52.1b}$$

$$\frac{P_2 \stackrel{\alpha}{\longmapsto} P_2'}{P_1 \parallel P_2 \stackrel{\alpha}{\longmapsto} P_1 \parallel P_2'} \tag{52.1c}$$

$$\frac{P_1 \stackrel{?a}{\longmapsto} P_1' \quad P_2 \stackrel{!a}{\longmapsto} P_2'}{P_1 \parallel P_2 \stackrel{\tau}{\longmapsto} P_1' \parallel P_2'} \tag{52.1d}$$

$$\frac{P \stackrel{\alpha}{\longmapsto} P' \quad a \# \alpha}{\nu(a.P) \stackrel{\alpha}{\longmapsto} \nu(a.P')} \tag{52.1e}$$

$$\frac{}{\alpha.P \stackrel{\alpha}{\longmapsto} P} \tag{52.1f}$$

$$\frac{E_1 \stackrel{\alpha}{\longmapsto} E_1'}{E_1 + E_2 \stackrel{\alpha}{\longmapsto} E_1'} \tag{52.1g}$$

$$\frac{E_2 \stackrel{\alpha}{\longmapsto} E_2'}{E_1 + E_2 \stackrel{\alpha}{\longmapsto} E_2'} \tag{52.1h}$$

## 52.2 Simulations, Strong and Weak

Labelled transition tells us what actions a process is capable of undertaking. This gives rise to the notion of the simulation of one process by another. We will say that a process $Q$ *simulates* a process $P$ iff there exists a *simulation relation* $\mathcal{R}$ on processes such that (a) $P\ \mathcal{R}\ Q$, and (b) whenever $P\ \mathcal{R}\ Q$ and $P \stackrel{\alpha}{\longmapsto} P'$, then there exists $Q'$ such that $P'\ \mathcal{R}\ Q'$ and $Q \stackrel{\alpha}{\longmapsto} Q'$. Such a relation $R$ is said to be a simulation relation between $P$ and $Q$. It is a *bisimulation* between $P$ and $Q$ iff it is also a simulation between $Q$ and $P$, which is to say that if $P\ \mathcal{R}\ Q$ and $Q \stackrel{\alpha}{\longmapsto} Q'$, then there exists $P'$ such that $P \stackrel{\alpha}{\longmapsto} P'$ and $P'\ \mathcal{R}\ Q'$. The processes $P$ and $Q$ are then said to be *bisimilar*.

Bisimilarity of processes with respect to labelled transition is a rather strong condition, because it requires not only that two processes exhibit equivalent behavior with respect to *observable* actions, but also with respect to the *unobservable*, or *silent*, action, $\tau$ — the unobservable action is treated as if it were observable!

To remedy this we introduce an auxiliary transition judgement derived from the original in which silent actions are disregarded. The $\tau$-*collapse* of a labelled transition system to mean that $P$ may make a transition to $P'$ exhibiting the observable action $\alpha$ by making any number of $\tau$-transitions prior to and subsequent to an $\alpha$-transition. More precisely, we define $P \stackrel{\alpha}{\longmapsto}_\tau P'$, where $\alpha \neq \tau$, to hold iff there are process expressions $Q$ and $Q'$ such that

$$P \stackrel{\tau}{\longmapsto}^* Q \stackrel{\alpha}{\longmapsto} Q' \stackrel{\tau}{\longmapsto}^* P'.$$

We then define a *weak simulation* to be a simulation with respect to the $\tau$-collapse of the original transition judgement. That is, a weak simulation is a relation $\mathcal{R}$ between process expressions such that if $P\ \mathcal{R}\ Q$ and $P \stackrel{\alpha}{\longmapsto}_\tau P'$, then there exists $Q'$ such that $P'\ \mathcal{R}\ Q'$ and $Q \stackrel{\alpha}{\longmapsto}_\tau Q'$. A weak bisimulation is a weak simulation for which the converse of this latter condition also holds. Simulation with respect to the original transition system is then called *strong simulation*. As the name suggests, strong similarity is sufficient for weak similarity, but the converse is by no means the case.

## 52.3 Exercises