Part II

Defining a Language

Chapter 3

Concrete Syntax

The *concrete syntax* of a language consists of the rules for representing expressions as strings, linear sequences of characters (or symbols) 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 (grounded in the theory of formal languages) for eliminating ambiguity, improving readability is, of course, a matter of taste about which reasonable people may disagree. Techniques for eliminating ambiguity include precedence conventions for binary operators and various forms of parentheses for grouping sub-expressions. Techniques for enhancing readability include the use of suggestive key words and phrases, and establishment of punctuation and layout conventions.

3.1 Context-Free Grammars

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

1. An alphabet Σ of *terminals*, or *letters*.

2. A finite set $\mathcal{N}$ of *non-terminals* that stand for the syntactic categories.

3. A set $\mathcal{P}$ of *productions* of the form $A ::= \alpha$, where A is a non-terminal and α is a string of terminals and non-terminals.

Whenever there is a set of productions

$$\begin{array}{rcl} A & ::= & \alpha_1 \\ & \vdots & \\ A & ::= & \alpha_n. \end{array}$$

all with the same left-hand side, we often abbreviate it as follows:

$$A ::= \alpha_1 \mid \cdots \mid \alpha_n.$$

A context-free grammar is essentially a simultaneous inductive definition of its syntactic categories. Specifically, we may associate a rule set R with a grammar according to the following procedure. First, we treat each non-terminal as a label of its syntactic category. Second, for each production

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

of the grammar, where $A_1, \ldots, A_n$ are all of the non-terminals on the right-hand side of that production, and $s_1, \ldots, s_n$ are strings of terminals, add a rule

$$\frac{t_1\, A_1 \quad \ldots \quad t_n\, A_n}{s_1\, t_1\, s_2\, \ldots\, s_{n-1}\, t_n\, s_n\, A}$$

to the rule set R. For each non-terminal A, we say that *s is a string of syntactic category A*, written $s \in L(A)$, iff $s \in I(R)_A$ (*i.e.*, $s\ A \in I(R)$).

An example will make these ideas clear. Let us give a grammar defining the syntax of a simple language of arithmetic expressions extended with a variable-binding construct.

$$\begin{array}{lrcl} \textit{Digits} & d & ::= & 0 \mid 1 \mid \cdots \mid 9 \\ \textit{Numbers} & n & ::= & d \mid n\, d \\ \textit{Expressions} & e & ::= & n \mid e\texttt{+}e \mid e\texttt{*}e \end{array}$$

A number n is a non-empty sequence of decimal digits. An expression e is either a number n, or the sum or product of two expressions.

Here is this grammar presented as a simultaneous inductive definition:

$$\frac{}{0\ \mathsf{digit}} \quad \cdots \quad \frac{}{9\ \mathsf{digit}} \tag{3.1}$$

$$\frac{d\ \mathsf{digit}}{d\ \mathsf{number}} \qquad \frac{n\ \mathsf{number} \quad d\ \mathsf{digit}}{n\, d\ \mathsf{number}} \tag{3.2}$$

$$\frac{n\ \mathsf{number}}{n\ \mathsf{expr}} \tag{3.3}$$

$$\frac{e_1\ \mathsf{expr} \qquad e_2\ \mathsf{expr}}{e_1\mathsf{+}e_2\ \mathsf{expr}} \tag{3.4}$$

$$\frac{e_1\ \mathsf{expr} \qquad e_2\ \mathsf{expr}}{e_1\mathsf{*}e_2\ \mathsf{expr}} \tag{3.5}$$

Let R be the above set of rules, and let $I = I(R)$. The syntactic categories of the grammar are the sections of I by the non-terminal standing for that category. For example, the set of expressions is I_{expr}, and so forth.

3.2 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 strings may be thought of as arising in several different ways. For example, the string 1+2*3 may be thought of as arising by applying the rule for multiplication first, then the rule for addition, or *vice versa*. The former interpretation corresponds to the expression (1+2)*3; the latter corresponds to the expression 1+(2*3).

The trouble is that we cannot simply tell from the generated string which reading is intended. This causes numerous problems. For example, suppose that we wish to define a function $eval$ that assigns to each arithmetic expression e its value $n \in N$. A natural approach is to use rule induction on the rules determined by the grammar of expressions.

We will define three functions simultaneously, as follows:

$$\begin{aligned}
eval_{\mathsf{dig}}(0) &= 0 \\
&\vdots \\
eval_{\mathsf{dig}}(9) &= 9 \\
\\
eval_{\mathsf{num}}(d) &= eval_{\mathsf{dig}}(d) \\
eval_{\mathsf{num}}(n\,d) &= 10 \times eval_{\mathsf{num}}(n) + eval_{\mathsf{dig}}(d) \\
\\
eval_{\mathsf{exp}}(n) &= eval_{\mathsf{num}}(n) \\
eval_{\mathsf{exp}}(e_1\mathsf{+}e_2) &= eval_{\mathsf{exp}}(e_1) + eval_{\mathsf{exp}}(e_2) \\
eval_{\mathsf{exp}}(e_1\mathsf{*}e_2) &= eval_{\mathsf{exp}}(e_1) \times eval_{\mathsf{exp}}(e_2)
\end{aligned}$$

The all-important question is: *are these functions well-defined*? The answer is *no*! The reason is that a string such as 1+2*3 arises in two different ways, using either the rule for addition expressions (thereby reading it as 1+(2*3)) or the rule for multiplication (thereby reading it as (1+2)*3). Since these have different values, it is impossible to prove that there exists a unique value for every string of the appropriate grammatical class. (It is true for digits and numbers, but not for expressions.)

What do we do about ambiguity? The two most common methods to eliminate this kind of ambiguity are these:

1. Introduce parenthesization into the grammar so that the person writing the expression can choose the intended intepretation.

2. Introduce precedence relationships that resolve ambiguities between distinct operations (*e.g.*, by stipulating that multiplication takes precedence over multiplication).

3. Introduce associativity conventions that determine how to resolve ambiguities between operators of the same precedence (*e.g.*, by stipulating that addtion is right-associative).

Using these techniques, we arrive at the following revised grammar for arithmetic expressions.

$$\begin{array}{lllll} \textit{Digits} & d & ::= & 0 \mid 1 \mid \cdots \mid 9 \\ \textit{Numbers} & n & ::= & d \mid n\,d \\ \textit{Expressions} & e & ::= & t \mid t\texttt{+}e \\ \textit{Terms} & t & ::= & f \mid f\texttt{*}t \\ \textit{Factors} & f & ::= & n \mid \texttt{(}e\texttt{)} \end{array}$$

We have made two significant changes. The grammar has been "layered" to express the precedence of multiplication over addition and to express right-associativity of each, and an additional form of expression, parenthesization, has been introduced.

It is a straightforward exercise to translate this grammar into an inductive definition. Having done so, it is also straightforward to revise the definition of the evaluation functions so that are well-defined. The revised definitions are given by rule induction; they require additional clauses for

the new syntactic categories.

$$\begin{array}{rcl}
eval_{\mathsf{dig}}(0) & = & 0 \\
 & \vdots & \\
eval_{\mathsf{dig}}(9) & = & 9 \\
 & & \\
eval_{\mathsf{num}}(d) & = & eval_{\mathsf{dig}}(d) \\
eval_{\mathsf{num}}(n\,d) & = & 10 \times eval_{\mathsf{num}}(n) + eval_{\mathsf{dig}}(d) \\
 & & \\
eval_{\mathsf{exp}}(t) & = & eval_{\mathsf{trm}}(t) \\
eval_{\mathsf{exp}}(t\texttt{+}e) & = & eval_{\mathsf{trm}}(t) + eval_{\mathsf{exp}}(e) \\
 & & \\
eval_{\mathsf{trm}}(f) & = & eval_{\mathsf{trm}}(f) \\
eval_{\mathsf{trm}}(f\texttt{*}t) & = & eval_{\mathsf{trm}}(f) \times eval_{\mathsf{trm}}(t) \\
 & & \\
eval_{\mathsf{trm}}(n) & = & eval_{\mathsf{num}}(n) \\
eval_{\mathsf{trm}}(\texttt{(}e\texttt{)}) & = & eval_{\mathsf{exp}}(e)
\end{array}$$

A straightforward proof by rule induction shows that these functions are well-defined.

3.3 Exercises

1. Give context-free grammars for various languages.
2. Ensure that a grammar is parseable using the techniques described here.

Chapter 4

First-Order Abstract Syntax

The concrete syntax of a language is an inductively-defined set of strings over a given alphabet. Its *first-order abstract syntax* is an inductively-defined set of first-order terms, or ast's, over a set of operators. Abstract syntax avoids the ambiguities of concrete syntax by employing operators that determine the outermost form of any given expression, rather than relying on parsing conventions to disambiguate strings. The reason to call this representation "first-order" will become apparent in Chapter 5, where we introduce mechanisms to account for the binding and scope of variables.

4.1 Abstract Syntax Trees

To specify the first-order abstract syntax of a language, it is necessary to specify in advance the set of operators, and their arities, used to build ast's. For example, to specify the abstract syntax of a language of arithmetic expressions, we may specify the following operators:

Operator	*Arity*
`num`(n)	0
`+`	2
`*`	2

Here n ranges over the natural numbers; the operator `num`(n) is the nth numeral, which takes no arguments. The operators `+` and `*` take two arguments each, as might be expected.

Having specified the operators, we then give an inductive definition of the various syntactic categories of the language. For example, in the

case of arithmetic expressions there is only one syntactic category, that of expressions, which may be specified as follows:

$$\frac{}{\texttt{num}(n)\ \textsf{expr}} \tag{4.1}$$

$$\frac{e_1\ \textsf{expr} \quad e_2\ \textsf{expr}}{\texttt{plus}(e_1, e_2)\ \textsf{expr}} \tag{4.2}$$

$$\frac{e_1\ \textsf{expr} \quad e_2\ \textsf{expr}}{\texttt{times}(e_1, e_2)\ \textsf{expr}} \tag{4.3}$$

Notice that the conclusion of each rule is an ast whose outermost constructor uniquely identifies the rule used to construct it.

As an alternative to rules, we often use a notation similar to context-free grammars to specify the abstract syntax. The difference compared to similar specifications of concrete syntax lies in how we interpret the grammar. In the case of concrete syntax we interpret the grammar as a simultaneous inductive definition of sets of strings, whereas in the case of (first-order) abstract syntax, we interpret it as a simultaneous inductive definition of sets of ast's. For example, the abstract syntax of the language of arithmetic expressions introduced in Chapter 3 may be defined by the following grammar:

$$\textit{Expressions} \quad e \quad ::= \quad \texttt{num}(n) \mid \texttt{plus}(e_1, e_2) \mid \texttt{times}(e_1, e_2)$$

This grammar, understood as a specification of abstract syntax, has the same meaning as the rules just given for the same language.

In practice we do not explicitly declare the operators and their arities in advance of giving an inductive definition of the abstract syntax of a language. Instead we leave it to the reader to infer the set of operators and their arities required for the definition to make sense.

4.2 Structural Induction

When applied to the rules defining the abstract syntax of a language, the principle of rule induction is called *structural induction*. We say that a proposition is proved "by induction on the structure of ..." or "by structural induction on ..." to indicate that we are applying the general principle of rule induction to the rules defining the abstract syntax of some expression.

In the case of the abstract syntax of arithmetic expressions just given, the principle of structural induction is as follows. To prove that a property P holds of every expression e of the abstract syntax, it is enough to show that P is closed under the rules defining the abstract syntax. Specifically,

1. Show that P holds of $\texttt{num}(n)$ for any number n.
2. Assuming that P holds of e_1 and e_2, show that P holds of $\texttt{plus}(e_1, e_2)$.
3. Assuming that P holds of e_1 and e_2, show that P holds of $\texttt{times}(e_1, e_2)$.

For example, we may prove that the equations

$$\begin{array}{rcl} \mathit{eval}(\texttt{num}(n)) & = & n \\ \mathit{eval}(\texttt{plus}(e_1, e_2)) & = & \mathit{eval}(e_1) + \mathit{eval}(e_2) \\ \mathit{eval}(\texttt{times}(e_1, e_2)) & = & \mathit{eval}(e_1) \times \mathit{eval}(e_2) \end{array}$$

determine a function eval from the abstract syntax of expressions to numbers. That is, we may show by induction on the structure of e that there is a unique n such that $\mathit{eval}(e) = n$.

In practice we often (somewhat sloppily) define both the concrete and abstract syntax of a language by a single grammar. The idea is that the same grammar can be read as a (possibly ambiguous) specification of the concrete syntax, and as an (unambiguous) specification of the abstract syntax. Since the ambiguities in the concrete syntax can, presumably, be resolved using standard methods, we do not bother to specify them, but rather rely on the reader's experience to fill in the details. It takes a little experience to get used to this approach, but it greatly simplifies the presentation of languages for which we are not concerned to design a "pretty" concrete syntax.

4.3 Parsing

The process of translation from concrete to abstract syntax is called *parsing*. If C is the concrete syntax of a language (an inductively-defined set of strings), and A is its abstract syntax (an inductively-defined set of ast's), then a parser is a function $\mathit{parse} : C \to A$ mapping strings to ast's. Since C is inductively defined, it is natural to formulate the definition of parse by induction on the rules defining the concrete syntax.

For example, consider the language of arithmetic expressions discussed in Chapter 3. Since we wish to define a function on the concrete syntax, it

should be clear from the discussion in Section 3.2 that we should work with the disambiguated grammar that makes explicit the precedence and associativity of addition and multiplication. With the rules of this grammar in mind, we may define simultaneously a family of parsing functions for each syntactic category by the following equations:

$$\begin{array}{rcl}
parse_{\mathsf{dig}}(0) & = & 0 \\
 & \vdots & \\
parse_{\mathsf{dig}}(9) & = & 9 \\
 & & \\
parse_{\mathsf{num}}(d) & = & \mathtt{num}(parse_{\mathsf{dig}}(d)) \\
parse_{\mathsf{num}}(n\,d) & = & \mathtt{num}(10 \times k + parse_{\mathsf{dig}}\,d),\ \mathit{where}\ \ parse_{\mathsf{num}}\,n = \mathtt{num}(k) \\
 & & \\
parse_{\mathsf{exp}}(t) & = & parse_{\mathsf{trm}}(t) \\
parse_{\mathsf{exp}}(t\mathtt{+}e) & = & \mathtt{plus}(parse_{\mathsf{trm}}(t), parse_{\mathsf{exp}}(e)) \\
 & & \\
parse_{\mathsf{trm}}(f) & = & parse_{\mathsf{fct}}(f) \\
parse_{\mathsf{trm}}(f\mathtt{*}t) & = & \mathtt{times}(parse_{\mathsf{fct}}(f), parse_{\mathsf{trm}}(t)) \\
 & & \\
parse_{\mathsf{fct}}(n) & = & parse_{\mathsf{num}}(n) \\
parse_{\mathsf{fct}}(\mathtt{(}e\mathtt{)}) & = & parse_{\mathsf{exp}}(e)
\end{array}$$

It is a simple matter to prove by rule induction that these functions are all well-defined.

There is one remaining issue about this specification of the parsing function that requires further remedy. Look closely at the definition of the function $parse_{\mathsf{num}}$. It relies on a decomposition of the input string into two parts: a string, which is parsed as a number, followed a character, which is parsed as a digit. This is quite unrealistic, at least if we expect to process the input "on the fly", since it requires us to work from the *end* of the input, rather than the *beginning*. To remedy this, we modify the grammatical clauses for numbers to be *right recursive*, rather than *left recursive*, as follows:

$$\mathit{Numbers} \quad n \quad ::= \quad d \mid d\,n$$

This re-formulation ensures that we may process the input from left-to-right, one character at a time. It is a simple matter to re-define the parser to reflect this change in the grammar, and to check that it is well-defined.

An implementation of a parser that obeys this left-to-right discipline and is defined by induction on the rules of the grammar is called a *recursive*

descent parser. This is the method of choice for hand-coded parsers. Parser generators, which automatically create parsers from grammars, make use of a different technique that is more efficient, but much harder to implement by hand.

4.4 Exercises

1. Give a concrete and (first-order) abstract for a language.
2. Write a parser for that language.

Chapter 5

Higher-Order Abstract Syntax

First-order abstract syntax captures the "deep structure" of an expression in the sense that it makes explicit the hierarchical relationships among the components of an expression. For example, in the case of arithmetic expressions the rules of abstract syntax make clear whether a given expression is an addition, one of whose arguments is a multiplication, or vice-versa.

Higher-order abstract syntax takes this process one step further to take account of the binding and scope of variables. The binding of a variable is the point at which it is introduced; its scope is its range of significance. The names of bound variables are not significant; this is captured by the notion of α-*conversion*. Variables may be replaced by other terms by a process called *substitution*. Both of these notions are captured through the mechanism of *higher-order terms*.

5.1 Variables, Binding, and Scope

Variables are place-holders. They may be replaced by other terms (possibly involving other variables) to obtain a new term. For example, if we extend the syntax of arithmetic expressions to include variables, we obtain (integer) polynomials.

Variables	x	$::=$	*any identifier*
Expressions	e	$::=$	$\mathtt{var}(x) \mid \mathtt{num}(n) \mid \mathtt{plus}(e_1, e_2) \mid \mathtt{times}(e_1, e_2)$

We will not be specific about what counts as an identifier; typically we admit any string over some specified alphabet.

For example, $\mathtt{plus}(\mathtt{times}(3, \mathtt{var}(x)), 1)$ is an expression involving the variable $\mathtt{var}(x)$, written in abstract syntax notation. Using standard con-

crete syntax conventions, this expression would be written 3`*`x+1, relying on precedence to disambiguate. We may replace $\texttt{var}(x)$ by another expression, say $\texttt{plus}(2, y)$ to obtain the expression $\texttt{plus}(\texttt{times}(3, \texttt{plus}(2, y)), 1)$. Written in terms of concrete syntax, this is $\texttt{plus}(\texttt{times}(3, (\texttt{plus}(2, y))), 1)$; parentheses are required to disambiguate.

As far as first-order abstract syntax is concerned there is nothing else to be said. Variables are just a form of abstract syntax, no different from any other piece of abstract syntax in the language. But something interesting happens when we introduce operators that introduce, or bind, variables. For example, we might extend the language of arithmetic expressions to include a "let" statement that introduces a variable and gives it a definition. The syntax is extended as follows:

$$\begin{array}{llll} \textit{Variables} & x & ::= & \textit{any identifier} \\ \textit{Expressions} & e & ::= & \texttt{var}(x) \mid \texttt{num}(n) \mid \texttt{plus}(e_1, e_2) \mid \texttt{times}(e_1, e_2) \mid \\ & & & \texttt{let}(x, e_1, e_2) \end{array}$$

The ast $\texttt{let}(x, e_1, e_2)$ might be written in concrete syntax as $\texttt{let}\, x\, \texttt{be}\, e_1\, \texttt{in}\, e_2$ to make it a little easier to read.

What is important about the `let` expression is that the variable x is *introduced*, or *bound*, for use within its *scope*, the expression e_2. Ordinarily we would interpret this as defining x to be the expression e_1 for use within e_2, but that interpretation is not especially important for the present purposes. All that matters is that the `let` binds a variable within a specified scope.

All occurrences of a variable within the scope of a binding are treated as references to its *binding site*, the point at which the variable is bound. Whenever we see a variable, we consider it to be a reference to the *nearest enclosing* binding occurrence of that variable. Thus the occurrences of x in the expression $\texttt{let}\, x\, \texttt{be}\, 2\, \texttt{in}\, x$+$x$ refer to the binding introduced by the `let`. Similarly, in the expression

$$\texttt{let}\, x\, \texttt{be}\, 2\, \texttt{in}\, \texttt{let}\, y\, \texttt{be}\, 3\, \texttt{in}\, x\texttt{+}y,$$

the occurrence of x refers to the outermost `let`, and the occurence of y refers to the innermost. Finally, in the expression

$$\texttt{let}\, x\, \texttt{be}\, 2\, \texttt{in}\, \texttt{let}\, x\, \texttt{be}\, 3\, \texttt{in}\, x\texttt{+}x,$$

both occurrences of x in the addition refer to the *innermost* binding occurrence, since it is the nearest enclosing binding for the variable x. There is no way to refer to the outermost `let`; the inner binding for x is said to *shadow* the outer.

Determining the binding occurrence corresponding to a use of a variable is called *scope resolution*. The convention of treating a variable occurrence as a reference to the nearest enclosing binding of that variable is called *lexical*, or *static*, scope. The adjectives "lexical" and "static" indicate that scope resolution is determined by the *program text*, rather than its execution behavior. In Chapter 11 we will consider an alternative, called *dynamic* scope, in which bindings are left unresolved until execution time.

Not all variables in an expression refer to a binding. For example, consider the expression

$$\texttt{let}\ x\ \texttt{be}\ 2\ \texttt{in}\ x\texttt{+}y.$$

The variable x is bound by the `let`, according to the rules just given; it is said to be a *bound* variable of the expression. On the other hand the variable y is not bound anywhere in this expression; it is said to be a *free* variable of the expression. An expression containing free variables is said to be *open*, whereas one that does not contain any free variables is said to be *closed*. Note that expressions with no variables at all are closed, as are expressions all of whose variables are bound.

Since bound variables are used only to refer to their binding site, the choice of names of bound variable does not matter. Thus the expression

$$\texttt{let}\ x\ \texttt{be}\ 2\ \texttt{in}\ x\texttt{+}x$$

is not materially different from the expression

$$\texttt{let}\ y\ \texttt{be}\ 2\ \texttt{in}\ y\texttt{+}y,$$

since their binding structure is the same. That is, the variable x is used to refer to the outermost binding in the first expression, whereas the variable y is used for the same purpose in the second. Of course we typically choose mnemonic identifiers, but for the purposes of scope resolution the choice does not matter.

Two expressions that differ only in the choice of bound variable names are said (for historical reasons) to be *α-equivalent*. We write $e_1 \equiv e_2$ to indicate that e_1 and e_2 are α-equivalent. This is clearly an equivalence relation. Moreover, it is a *congruence*, which means that if we replace any sub-expression by an α-equivalent sub-expression, the result is α-equivalent to the original.

The fundamental principle of higher-order abstract syntax is that we *identify α-equivalent expressions*. Put in other terms, higher-order abstract

syntax is the *quotient* of first-order abstract syntax by α-equivalence. Elements of the quotient are equivalence classes of first-order ast's under α-equivalence.

The main consequence of working with higher-order abstract syntax is that we do not distinguish between ast's that differ only in the names of their bound variables, because we are really working with equivalence classes. However, to write down an equivalence class requires that we choose a representative. That is, we must make an arbitrary choice of names for the bound variables. The beauty of higher-order abstract syntax is that we may always choose the bound variable name to be different from any given finite set of variable names. Such a choice is said to be *fresh*, or *new*, relative to that set of names. Thus, when we write `let` x `be` 3 `in` x`+`x, we implicitly choose x to be a "new" variable, different from all others currently in use. This completely avoids the problem of shadowing, since we may always choose another representative that avoids re-use of variable names. Provided that we make such a choice, the variable name uniquely determines its binding occurrence.

We will often wish to replace all occurrences of a free variable x in an expression e' by another expression e. This process is called *substitution*, and is written $\{e/x\}e'$. While substitution may, at first glance, seem like a simple process of replacement (changing all x's into e'), there is a subtle difficulty that must be avoided, called *capture*. Since the variable x might occur within the scope of a binding occurrence of some variable y within e', if y also occurs free in e, then simple replacement would incur capture, thereby changing the meaning of the expression.

For example, suppose that e' is the expression `let` y `be` 7 `in` x`+`y, and let e be the expression y`*`2. The result of simply replacing x by y`*`2 yields the expression

$$\texttt{let}\ y\ \texttt{be}\ 7\ \texttt{in}\ y\texttt{*}2\texttt{+}y.$$

The binding for y in e' is said to *capture* the free occurrence of y in e. Capture is to be avoided is because it is inappropriately sensitive to the choice of bound variable names. If we had chosen a different representative for e', say

$$\texttt{let}\ z\ \texttt{be}\ 7\ \texttt{in}\ x\texttt{+}z,$$

then the result of substitution would be

$$\texttt{let}\ z\ \texttt{be}\ 7\ \texttt{in}\ y\texttt{*}2\texttt{+}z,$$

which is a *different* expression!

Thus, substitution is well-defined on α-equivalence classes only if we *avoid capture*. This can always be achieved by simply choosing all bound variable names in the target of the the substitution to be different from those that occur free in the substituting expression. Since there are only finitely many such variables, this requirement can always be met by a suitable choice of representatives. In the above example we would choose the bound variable in e' to be z, rather than y, since y occurs free in e.

5.2 Higher-Order Terms

To make all of this more precise and systematic, we will introduce higher-order (more precisely, second-order) terms as a generalization of first-order terms. Recall that a first-order term has the form $o(t_1, \ldots, t_n)$ where o is an operator of *arity* n, meaning that it takes n arguments, and t_1, ..., t_n are themselves first-order terms. To make the step to higher-order, we will generalize the notion of arity and simultaneously extend the notion of term.

In the higher-order case an arity is a list of natural numbers, written $[n_1, \ldots, n_k]$. An operator of this arity takes k arguments; we may form a *simple* term by applying o to k higher-order terms, written $o(t_1, \ldots, t_k)$ as before. However, in contrast to the first-order case, the arguments may either be simple terms (including variables) or *abstractions* of the form

$$x_1, \ldots, x_n.t,$$

where $n \geq 0$ and t is a simple term.[1] Such an abstraction the variables $x_1, \ldots, x_n$ in the simple term t, and is said to have *degree* n. If o is an operator of arity $[n_1, \ldots, n_k]$, then $o(t_1, \ldots, t_k)$ is well-formed only if t_i has degree n_i for each $1 \leq i \leq k$. Notice that first-order terms arise as a special case; a first-order operator of (first-order) arity k is just a higher-order operator of (higher-order) arity $[0, \ldots, 0]$, where 0 occurs k times, once for each sub-term.

Thus, in the higher-order case, the operator `+` may be regarded as an operator of arity $[0, 0]$, and `let` may be regarded as an operator of arity $[0, 1]$. This indicates that no variables are bound in the first position, but one argument is bound in the second. Thus we would write $\texttt{let}(e_1, x.e_2)$ in the higher-order case, rather than $\texttt{let}(x, e_1, e_2)$ as we did in the first-order case. This notation, while a bit unfamiliar, makes clear which variables are bound where in the expression, whereas in the first-order case we must state this information separately.

[1] We do not distinguish between $.t$ and t in the case that $n = 0$.

Higher-order abstract syntax, then, will be the quotient of higher-order terms by α-equivalence (renaming of bound variables). In particular the abstraction $x.\mathtt{plus}(x,x)$ is α-equivalent to the abstraction $y.\mathtt{plus}(y,y)$. Consequently, the simple terms

$$\mathtt{let}(7, x.\mathtt{plus}(x,x))$$

and

$$\mathtt{let}(7, y.\mathtt{plus}(y,y))$$

are also α-equivalent.

It is important to observe that α-equivalence preserves the structure of a term. That is, a term of the form $o(t_1,\ldots,t_n)$ is α-equivalent only to terms of the same form. This means that structural induction can be extended from first-order to higher-order abstract syntax without special mention. Note, however, that a proof by structural induction on higher-order abstract syntax must respect α-conversion in the sense that the validity of the proof cannot depend upon the exact choice of bound variable names. In practice this is never an issue.

5.3 Renaming and Substitution

In this section we will give a more rigorous account of variable renaming and substitution for higher-order terms. First, we will define variable renaming as a function on *raw* higher-order terms. Second, we will use this to define α-equivalence. Third, we will define capture-avoiding substitution on α-equivalence classes of higher-order terms.

The set of *free variables*, $\mathrm{FV}(t)$, in a raw higher-order term t is inductively defined by the following equations:

$$\begin{array}{rcl}
\mathrm{FV}(\mathtt{var}(x)) & = & \{\, x \,\} \\
\mathrm{FV}(o(t_1,\ldots,t_n)) & = & \bigcup_{1\le i\le n} \mathrm{FV}(t_i) \\
\mathrm{FV}(x_1,\ldots,x_n.t) & = & \mathrm{FV}(t) \setminus \{\, x_1,\ldots,x_n \,\}
\end{array}$$

Let $\vec{x} = x_1,\ldots,x_k$ and $\vec{x'} = x'_1,\ldots,x'_k$. The simultaneous renaming of variables $\vec{x}$ to $\vec{x'}$ in a higher-order term t, written $\{\vec{x'}/\vec{x}\}t$, defined by induction on the structure of t as follows:

$$\begin{array}{rcl}
\{\vec{x'}/\vec{x}\}y & = & \left\{ \begin{array}{ll} x'_i & \textit{if } y = x_i \\ y & \textit{ow} \end{array} \right. \\
\{\vec{x'}/\vec{x}\}o(t_1,\ldots,t_n) & = & o(\{\vec{x'}/\vec{x}\}t_1,\ldots,\{\vec{x'}/\vec{x}\}t_n) \\
\{\vec{x'}/\vec{x}\}y_1,\ldots,y_n.t & = & y_1,\ldots,y_n.\{\vec{x'}/\vec{x}\}t
\end{array}$$

The last clause is defined only in case that no y_i $(1 \leq i \leq n)$ is among the $\vec{x'}$ (so as to preclude capture) or among the $\vec{x}$ (so as to avoid confusion).[2]

We may now define α-equivalence as the least congruence containing all instances of the axiom

$$x_1, \ldots, x_n.t \equiv x'_1, \ldots, x'_n.\{x'_1, \ldots, x'_n/x_1, \ldots, x_n\}t$$

whenever the substitution on the right-hand side is defined. By the "least congruence", we mean that $\equiv$ is the least equivalence relation closed under the following congruence principles:

$$\frac{t_1 \equiv t'_1 \quad \cdots \quad t_n \equiv t'_n}{o(t_1, \ldots, t_n) \equiv o(t'_1, \ldots, t'_n)} \qquad \frac{t \equiv t'}{x_1, \ldots, x_n.t \equiv x_1, \ldots, x_n.t'}$$

Thus we may replace any sub-term of a term by an α-equivalent one, and obtain a term that is α-equivalent to the one we started with. Moreover, α-equivalence preserves the structure of a higher-order term. In particular, an abstraction of degree n is α-equivalent only to other abstractions of the same degree.

Finally, we may define simultaneous substitution of a sequence $\vec{t} = t_1, \ldots, t_k$ of terms for a sequence $\vec{x} = x_1, \ldots, x_k$ in a term t, written $\{\vec{t}/\vec{x}\}t$. We will define substitution only up to α-equivalence, relying on implicit renaming of bound variables to ensure that capture is avoided. Simultaneous substitution is defined by induction on the structure of t as follows:

$$\begin{array}{rcll} \{\vec{t}/\vec{x}\}y & = & \left\{ \begin{array}{ll} t_i & \textit{if } y = x_i \\ y & \textit{ow} \end{array} \right. & \\ \{\vec{t}/\vec{x}\}o(t'_1, \ldots, t'_n) & = & o(\{\vec{t}/\vec{x}\}t'_1, \ldots, \{\vec{t}/\vec{x}\}t'_n) & \\ \{\vec{t}/\vec{x}\}y_1, \ldots, y_n.t' & = & y_1, \ldots, y_n.\{\vec{t}/\vec{x}\}t' & \textit{if } \forall 1 \leq i \leq n\ y_i \notin \mathrm{FV}(\vec{t}) \end{array}$$

Notice that by α-conversion the restriction on the last clause may always be met, and hence substitution is always defined on α-equivalence classes. Consequently, the result of substitution is only defined up to α-equivalence.

5.4 de Bruijn Indices

Most of the difficulties in formalizing the notion of higher-order abstract syntax stem from the use of names to refer to binding occurrences of variables. An alternative is to replace variables by "direct references" to their

[2] These restrictions are stronger than strictly necessary, but are weak enough for present purposes.

binding occurrences, dispensing with names entirely. Such a representation is said to be *name-free*, since it avoids the need to choose names for bound variables at all. While such a representation is technically convenient (since it avoids the messiness of working with equivalence classes), it is practically impossible to read.

A reasonable compromise is to reserve named form for the concrete syntax, and to use name-free form for the abstract syntax. One popular method, invented by the Dutch mathematicians N. G. deBruijn, is to use an indexing scheme, now called *de Bruijn indices*, to refer to the binding occurrence of a variable. Specifically, a bound variable is represented by a natural number representing the "distance" from the occurrence to the binder for that variable. The index 1 refers to the nearest enclosing binder, 2 to the second nearest, and so on. We write deBruijn indices in the form $\mathtt{idx}(i)$, where $i \geq 1$. Note that this convention works well only in the case that all binders have degree 1. The method can be generalized to binders of higher-degree, but we do not consider this possibility here.

Using deBruijn's notation the expression

$$\mathtt{let}\, x\, \mathtt{be}\, 7\, \mathtt{in}\, x\mathtt{+}1$$

has the name-free representation

$$\mathtt{let}(\mathtt{num}(7), \mathtt{plus}(\mathtt{idx}(1), \mathtt{num}(1))).$$

Similarly, the expression

$$\mathtt{let}\, x\, \mathtt{be}\, 1\, \mathtt{in}\, \mathtt{let}\, y\, \mathtt{be}\, 2\, \mathtt{in}\, x\mathtt{+}y$$

has the name-free representation

$$\mathtt{let}(\mathtt{num}(1), \mathtt{let}(\mathtt{num}(2), \mathtt{plus}(\mathtt{idx}(2), \mathtt{idx}(1)))).$$

A peculiarity of the deBruijn represenation is that different occurences of the same variable may be represented by different indices. The reason is that it may be necessary to "hop over" intervening binders in some positions, but not in others. For example, the expression

```
let x=10 in (let y=11 in y+x)+x
```

is represented in name-free form as

$$\mathtt{let}(\mathtt{num}(10), \mathtt{plus}(\mathtt{let}(\mathtt{num}(11), \mathtt{plus}(\mathtt{idx}(1), \mathtt{idx}(2))), \mathtt{idx}(1)))$$

The two uses variable that we called `x` in the named form are represented by the occurrence of the deBruijn index `idx`(2) and the second occurrence of the deBruijn index `idx`(1). The first occurrence of `idx`(1) refers not to `x`, but to `y`.

5.5 Exercises

1. Give the higher-order abstract syntax for an interesting language.
2. Implement named and name-free representations of hoas.
3. Develop the theory of deBruijn indices, including substitution.
4. Prove some properties of susbtitution and renaming.

Chapter 6

Static Semantics

The *static semantics* of a language isolates a subset of the abstract syntax of the language that is deemed *well-formed*. In simple cases every ast is well-formed, but in most cases additional constraints must be imposed. Chief among these are *type constraints* that ensure that the constructors of the language are combined in a sensible manner. An inductive definition of the type constraints governing a language is called a *(static) type system*, or *static semantics*.

6.1 Well-Formed Arithmetic Expressions

Since it is difficult to give a fully general account of static semantics, we will instead illustrate the main ideas by example. We will give a formal definition of well-formedness of arithmetic expressions that ensures that there are no unbound variables in a complete expression. Of course we could simply define e to be well-formed in this sense iff $\mathrm{FV}(e) = \emptyset$, but we will instead give a direct axiomatization of well-formedness.

A *well-formedness judgement*, or *well-formedness assertion*, has the form $\Gamma \vdash e\ \mathsf{ok}$, where Γ is a finite set of variables. The intended meaning of this assertion is that e is an arithmetic expression all of whose free variables are in the set Γ. In particular, if $\emptyset \vdash e\ \mathsf{ok}$ (often abbreviated to $e\ \mathsf{ok}$), then e has no unbound (free) variables, and is therefore suitable for evaluation to an integer.

Formally, well-formedness is inductively defined by the following rules:

$$\frac{(x \in \Gamma)}{\Gamma \vdash \mathtt{var}(x)\ \mathsf{ok}} \qquad \frac{(n \geq 0)}{\Gamma \vdash \mathtt{num}(n)\ \mathsf{ok}}$$

$$\frac{\Gamma \vdash e_1\ \mathsf{ok} \quad \Gamma \vdash e_2\ \mathsf{ok}}{\Gamma \vdash \mathtt{plus}(e_1, e_2)\ \mathsf{ok}} \qquad \frac{\Gamma \vdash e_1\ \mathsf{ok} \quad \Gamma \vdash e_2\ \mathsf{ok}}{\Gamma \vdash \mathtt{times}(e_1, e_2)\ \mathsf{ok}}$$

$$\frac{\Gamma \cup \{\, x \,\} \vdash e_2\ \mathsf{ok} \quad (x \notin \Gamma)}{\Gamma \vdash \mathtt{let}(e_1, x.e_2)\ \mathsf{ok}}$$

Frequently well-formedness rules are stated using concrete syntax for the sake of readability, but it is understood that we are really referring to the abstract syntax of the language.

There are a few things to notice about these rules. First, a variable is well-formed iff it is in Γ. This is consistent with the informal reading of the judgement. Second, a `let` expression adds a *new* variable to Γ for use within e_2. The "newness" of the variable is captured by the requirement that $x \notin \Gamma$. By the conventions of higher-order abstract syntax, this condition can always be met by a suitable renaming prior to application of the rule. Third, the rules are *syntax-directed* in the sense that there is one rule for each form of expression; as we will see later, this is not necessarily the case.

6.2 Exercises

1. Show that $\Gamma \vdash e\ \mathsf{ok}$ iff $\mathrm{FV}(e) \subseteq \Gamma$. From left to right, proceed by rule induction. From right to left, proceed by induction on the structure of e.

2. Integers and floats. Add types to variable declarations.

Chapter 7

Dynamic Semantics

The *dynamic semantics* of a language specifies how programs are to be executed. There are two popular methods for specifying dynamic semantics. One method, called *structured operational semantics (SOS)*, or *transition semantics*, presents the dynamic semantics of a language as a transition system specifying the step-by-step execution of programs. Another, called *evaluation semantics*, or *ES*, presents the dynamic semantics as a binary relation specifying the result of a complete execution of a program.

7.1 Structured Operational Semantics

A structured operational semantics for a language consists of a transition system whose states are programs and whose transition relation is defined by induction over the structure of programs. We will illustrate SOS for the simple language of arithmetic expressions (including `let` expressions) presented in Chapter 5.

The set of states is the set of well-formed arithmetic expressions:

$$S = \{\, e \mid \exists \Gamma\ \Gamma \vdash e\ \mathsf{ok} \,\}.$$

The set of initial states, $I \subseteq S$, is the set of closed expressions:

$$I = \{\, e \mid \emptyset \vdash e\ \mathsf{ok} \,\}.$$

The set of final states, $F \subseteq S$, is just the set of numerals for natural numbers:

$$F = \{\, \mathsf{num}(n) \mid n \geq 0 \,\}.$$

The transition relation $\mapsto \subseteq S \times S$ is inductively defined by the following rules:

$$\frac{(p = m + n)}{\mathtt{plus}(\mathtt{num}(m), \mathtt{num}(n)) \mapsto \mathtt{num}(p)} \qquad \frac{(p = m \times n)}{\mathtt{times}(\mathtt{num}(m), \mathtt{num}(n)) \mapsto \mathtt{num}(p)}$$

$$\frac{}{\mathtt{let}(\mathtt{num}(n), x.e) \mapsto \{\mathtt{num}(n)/\mathtt{var}(x)\}e}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{plus}(e_1, e_2) \mapsto \mathtt{plus}(e_1', e_2)} \qquad \frac{e_2 \mapsto e_2'}{\mathtt{plus}(\mathtt{num}(n_1), e_2) \mapsto \mathtt{plus}(\mathtt{num}(n_1), e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{times}(e_1, e_2) \mapsto \mathtt{times}(e_1', e_2)} \qquad \frac{e_2 \mapsto e_2'}{\mathtt{times}(\mathtt{num}(n_1), e_2) \mapsto \mathtt{times}(\mathtt{num}(n_1), e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{let}(e_1, x.e_2) \mapsto \mathtt{let}(e_1', x.e_2)}$$

Observe that variables are stuck states, but they are not final. Free variables have no binding, and hence cannot be evaluated to a number.

To enhance readability we often write SOS rules using concrete syntax, as follows:

$$\frac{(p = m + n)}{m\mathtt{+}n \mapsto p} \qquad \frac{(p = m \times n)}{m\mathtt{*}n \mapsto p}$$

$$\frac{}{\mathtt{let}\, x\, \mathtt{be}\, n\, \mathtt{in}\, e \mapsto \{n/x\}e}$$

$$\frac{e_1 \mapsto e_1'}{e_1\mathtt{+}e_2 \mapsto e_1'\mathtt{+}e_2} \qquad \frac{e_2 \mapsto e_2'}{n_1\mathtt{+}e_2 \mapsto n_1\mathtt{+}e_2'}$$

$$\frac{e_1 \mapsto e_1'}{e_1\mathtt{*}e_2 \mapsto e_1'\mathtt{*}e_2} \qquad \frac{e_2 \mapsto e_2'}{n_1\mathtt{*}e_2 \mapsto n_1\mathtt{*}e_2'}$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{let}\, x\, \mathtt{be}\, e_1\, \mathtt{in}\, e_2 \mapsto \mathtt{let}\, x\, \mathtt{be}\, e_1'\, \mathtt{in}\, e_2}$$

The intended meaning is the same, the only difference is the presentation.

The first three rules defining the transition relation are somtimes called *instructions*, since they correspond to the primitive execution steps of the machine. Addition and multiplication are evaluated by adding and multiplying; `let` bindings are evaluated by substituting the definition for the

variable in the body. In all three cases the *principal arguments* of the constructor are required to be numbers. Both arguments of an addition or multiplication are principal, but only the binding of the variable in a *let* expression is principal. We say that these primitives are evaluated *by value*, because the instructions apply only when the principal arguments have been fully evaluated.

What if the principal arguments have not (yet) been fully evaluated? Then we must evaluate them! In the case of arithmetic expressions we arbitrarily choose a left-to-right evaluation order. First we evaluate the first argument, then the second. Once both have been evaluated, the instruction rule applies. In the case of `let` expressions we first evaluate the binding, after which the instruction step applies. Note that evaluation of an argument can take multiple steps. The transition relation is defined so that one step of evaluation is made at a time, reconstructing the entire expression as necessary.

For example, consider the following evaluation sequence:

$$\begin{aligned}
\texttt{let}\ x\ \texttt{be}\ 1\texttt{+}2\ \texttt{in}\ (x\texttt{+}3)\texttt{*}4 &\mapsto \texttt{let}\ x\ \texttt{be}\ 3\ \texttt{in}\ (x\texttt{+}3)\texttt{*}4 \\
&\mapsto (3\texttt{+}3)\texttt{*}4 \\
&\mapsto 6\texttt{*}4 \\
&\mapsto 24
\end{aligned}$$

Each step is justified by a rule defining the transition relation. Instruction rules are axioms, and hence have no premises, but all other rules are justified by a subsidiary deduction of another transition. For example, the first transition is justified by a subsidiary deduction of $1\texttt{+}2 \mapsto 3$, which is justified by the first instruction rule defininig the transition relation. Each of the subsequent steps is justified similarly.

Since the transition relation in SOS 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 relation. For example, it is a simple matter to show by rule induction that the transition relation for evaluation of arithmetic expressions is deterministic: if $e \mapsto e'$ and $e \mapsto e''$, then $e' = e''$. This may be proved by simultaneous rule induction over the definition of the transition relation.

7.2 Evaluation Semantics

Another method for defining the dynamic semantics of a language, called *evaluation semantics*, consists of a direct inductive definition of the evaluation relation, written $e \Downarrow v$, specifying the value, v, of an expression, e. More precisely, an evaluation semantics consists of a set E of *evaluatable* expressions, a set V of *values*, and a binary relation $\Downarrow \subseteq E \times V$. In contrast to SOS the set of values need not be a subset of the set of expressions; we are free to choose values as we like. However, it is often advantageous to choose $V \subseteq E$.

We will give an evaluation semantics for arithmetic expressions as an example. The set of evaluatable expressions is defined by

$$E = \{\, e \mid \emptyset \vdash e \;\mathsf{ok} \,\}.$$

The set of values is defined by

$$V = \{\, \mathsf{num}(n) \mid n \geq 0 \,\}.$$

The evaluation relation for arithmetic expressions is inductively defined by the following rules:

$$\frac{}{\mathsf{num}(n) \Downarrow \mathsf{num}(n)}$$

$$\frac{e_1 \Downarrow \mathsf{num}(n_1) \quad e_2 \Downarrow \mathsf{num}(n_2) \quad (n = n_1 + n_2)}{\mathtt{plus}(e_1, e_2) \Downarrow \mathsf{num}(n)}$$

$$\frac{e_1 \Downarrow \mathsf{num}(n_1) \quad e_2 \Downarrow \mathsf{num}(n_2) \quad (n = n_1 \times n_2)}{\mathtt{times}(e_1, e_2) \Downarrow \mathsf{num}(n)}$$

$$\frac{e_1 \Downarrow \mathsf{num}(n_1) \quad \{\mathsf{num}(n_1)/\mathsf{var}(x)\}e_2 \Downarrow v}{\mathtt{let}(e_1, x.e_2) \Downarrow v}$$

Notice that the rules for evaluation semantics are *not* syntax-directed! 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 relation is inductively defined, it has associated with it a principle of proof by rule induction. Specifically, to show that $P(e, v)$ holds for some property $P \subseteq E \times V$, it is enough to show that P is closed under the rules given above. Specifically,

1. Show that $P(\mathtt{num}(n), \mathtt{num}(n))$.
2. Assume that $P(e_1, \mathtt{num}(n_1))$ and $P(e_2, \mathtt{num}(n_2))$. Show that $P(\mathtt{plus}(e_1, e_2), \mathtt{num}(n_1 + n_2))$ and that $P(\mathtt{times}(e_1, e_2), \mathtt{num}(n_1 \times n_2))$.
3. Assume that $P(e_1, v_1)$ and $P(\{v_1/\mathtt{var}(x)\}e_2, v_2)$. Show that $P(\mathtt{let}(e_1, x.e_2), v_2)$.

7.3 Relating Transition and Evaluation Semantics

We have given two different forms of dynamic semantics for the same language. 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 relation in the evaluation semantics.

Theorem 2
For all well-formed, closed arithmetic expressions e and all natural numbers n, $e \mapsto^! \mathtt{num}(n)$ iff $e \Downarrow \mathtt{num}(n)$.

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

Lemma 3
If $e \Downarrow \mathtt{num}(n)$, then $e \mapsto^! \mathtt{num}(n)$.

Proof: By induction on the definition of the evaluation relation. 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. ■

What about the converse? Recall from Chapter 2 that the complete evaluation relation, $\mapsto^!$, is the restriction of the multi-step evaluation relation,

$\mapsto^*$, to initial and final states (here closed expressions and numerals). Recall also that multi-step evaluation is inductively defined by two rules, reflexivity and closure under head expansion. By definition $\mathtt{num}(n) \Downarrow \mathtt{num}(n)$, so it suffices to show closure under head expansion.

Lemma 4
If $e \mapsto e' \Downarrow \mathtt{num}(n)$, then $e \Downarrow \mathtt{num}(n)$.

Proof: By induction on the definition of the transition relation. 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 $n = n_1 + n_2$. By induction $e_1 \Downarrow \mathtt{num}(n_1)$, and hence $\mathtt{plus}(e_1, e_2) \Downarrow n$, as required. ■

7.4 Exercises

1. Prove that if $e \mapsto e_1$ and $e \mapsto e_2$, then $e_1 \equiv e_2$.
2. Prove that if $e \in I$ and $e \mapsto e'$, then $e' \in I$. Proceed by induction on the definition of the transition relation.
3. Prove that if $e \in I \setminus F$, then there exists e' such that $e \mapsto e'$. Proceed by induction on the rules defining well-formedness given in Chapter 6.
4. Prove that if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 \equiv v_2$.
5. Complete the proof of equivalence of evaluation and transition semantics.