Fundamental Concepts and Formal Semantics of Programming Languages

Peter D. Mosses

Department of Computer Science
Swansea University, UK

E-mail: p.d.mosses@swan.ac.uk
Web: https://pdmosses.github.io

Last updated: 7 September 2019

Preface

These lecture notes are about the conceptual analysis and formal semantic description of programming languages. They explain how to analyse programming constructs in terms of fundamental concepts, and how to formulate precise descriptions of their intended interpretation using a structural approach to operational semantics. The approach is illustrated mainly by analysing and describing constructs taken from the functional programming language Standard ML; familiarity with Standard ML (or a related language) is assumed. The analysis and description of constructs from other languages are explored in the exercises.

An important novel aspect of the material is the use of Modular SOS (MSOS), a variant of the well-known Structural Operational Semantics (SOS). As the name suggests, this variant has the advantage of providing an exceptionally high degree of modularity: individual language constructs can be described independently, and freely combined to obtain descriptions of complete languages. The descriptions of the individual constructs remain unchanged when further constructs are added to the described language. Conventional SOS descriptions do not enjoy such modularity. For instance, an SOS description of constructs from a pure functional language require extensive reformulation when expressions are extended to allow side-effects or concurrent processes.

Descriptions in MSOS can be automatically translated into programs in the logic programming language Prolog, using the software accompanying these notes. This provides the basis for prototype implementations of languages involving the described constructs: programs can be run according to their specified semantics. The main purpose of such prototyping is to check that the specified semantics conforms to expectations; inspecting the course of program executions in the prototype implementation also helps to understand the techniques used in MSOS.

The notes also consider (albeit very briefly) alternative approaches to formal semantics, including action semantics, axiomatic semantics, denotational semantics, and other variants of operational semantics. The aim is merely to provide students with a general impression of previous approaches to formal semantics; references to the literature are provided for those who wish to continue with more advanced studies in this area.

About these lecture notes: The material in these notes is intended to be read in the order in which it appears. It is strongly recommended to try most of the exercises listed at the end of each chapter: this checks that the presented material has been properly grasped, and provides further illustration of various points. A reference to an exercise in the margin indicates that the material required for that exercise has now been covered. Exercises marked with one or more asterisks (*) are expected to be relatively demanding (regarding time and/or understanding).

▷ All the main points in these notes are displayed like this...

Each point is followed by further explanations and/or illustrations. The display of the main points should help to keep an overview of the main line during a first reading (when the intervening material may even be skipped, if desired) as well as facilitating revision.

About the course based on these notes: The material in these lecture notes was used for an undergraduate course on *Semantics of Programming Languages* at the Department of Computer Science, University of Aarhus, from 2001 until the author moved to Swansea in 2005.[1] The course took seven weeks, with three 45-minute lectures and three 45-minute exercise classes each week. The examination for the course required the students to complete a small project over a couple of days.

Most of the students took the course in their 5th semester; they were assumed to have previously completed an introductory Programming course (the one at Aarhus included programming in Java), a course on Models and Logic, and a course on Programming Languages (the one at Aarhus covered functional programming in ML and, to a lesser extent, logic programming in Prolog).

Changes in version 0.4d:

- URLs have been updated.
- The author's coordinates have been updated.
- Code is formatted in a different font, with upright quotes.
- The accompanying Prolog files now have extension `.pro` (instead of `.pl`).
- Minor clarifications have been made in the text.

[1] It has also been used for a course on *Programming Concepts and Formal Semantics* at Monash University, Australia.

Contents

Chapter 1

Introduction

This chapter introduces and motivates the study of the fundamental concepts and formal semantics of programming languages.

1.1 Fundamental Concepts

Let us start by considering fundamental computational concepts, relating them to familiar constructs of programming languages and to various programming paradigms.

▷ Programming languages include various kinds of constructs: expressions, declarations, commands, etc.

ML, for instance, distinguishes between expressions, declarations, types, patterns, and matches; Pascal has expressions, declarations, types, formal parameters, and commands.[1] Such distinctions are not merely syntactic: they reflect essential differences in the computational meaning of the various kinds of construct.

▷ Each construct may involve both flow of control and flow of information.

An execution of a program consists of a computation on some machine. In general, the computation may be regarded as a combination of sub-computations for the constructs that occur in the program.[2] Control flows into a construct when its sub-computation starts, and flows out again when it finishes. Information processing during the computation corresponds to flow of information between constructs. Constructs can differ greatly in how they let control and information flow to and from their sub-constructs.

[1] Commands are often referred to as 'statements', quite inappropriately.

[2] Let us ignore optimizations, which may obscure the relationship between the original program and the computation.

1.1.1 Flow of Control

▷ Fundamental control-flow concepts include sequencing, interleaving, choice between alternatives, exception raising and handling, iteration, procedural abstraction and activation, concurrent processes, and synchronization.

The main features of the above control-flow concepts are as follows:

- The familiar concept of *sequencing* involves letting control flow into sub-constructs from left to right, so long as each terminates normally.
- *Interleaving* lets control move back and forth between two or more sub-constructs.
- *Choice* between several alternatives usually lets control flow into only one of the alternatives—although when the chosen alternative *fails*, this may cause *back-tracking*, whereby another alternative is tried.
- Raising or throwing an *exception* generally interrupts normal control flow (sequencing, etc.) and skips the rest of enclosing constructs until control reaches some enclosing construct that can handle or catch the exception, thereby resuming normal control flow.
- *Iteration* lets control flow into the same sub-construct repeatedly.
- Activation of a *procedural abstraction* lets control flow into a construct from different parts of the program, resuming from the point of activation when the activation terminates.
- *Concurrent processes* split the flow of control into separate *threads*, which may subsequently be subject to *synchronization*.

1.1.2 Flow of Information

▷ Fundamental information-flow concepts include computation of values, use of computed values, effects on storage, scopes of bindings, and message-passing.

Obviously, expressions are evaluated primarily to compute values. Somewhat less obviously, we may profitably regard other kinds of constructs as computing corresponding kinds of values. For instance, a declaration computes a so-called

environment (i.e., a set of bindings for identifiers); and a command, which conceptually computes no value at all, can be regarded as computing a fixed dummy value.

Computed values correspond to information that flows out of constructs; the values may subsequently flow into other constructs. Computing a value always involves termination. Raising an exception may be seen as a combination of exceptional control flow and the computation of a value that identifies the exception.

Effects on storage are stable: when a new cell has been added to the storage, or a value stored in a cell (overwriting any previous value), the cell remains in the same state until some new value is stored in it, or the cell is removed (explicitly, or by garbage collection).

The bindings computed by one sub-construct may flow (often together with the current bindings) into another sub-construct, which is then the scope of the computed bindings. Scopes of bindings generally extend into sub-constructs, except where overridden by new bindings.

Message-passing by a construct corresponds to information flow both ways: the construct itself gets the information that its context accepts the message, and the context gets the information in the message itself. Ex.1.1

1.1.3 Programming Paradigms

▷ Different programming paradigms emphasize different fundamental concepts.

Programming languages can be classified according to which programming paradigm they support best. Sometimes, a language intended for one paradigm can be used for other paradigms as well, with some extra effort.

▷ *Imperative programming* emphasizes sequencing, iteration, procedural abstraction and activation, and effects on storage.

Imperative programs tend to involve numerous changes of stored values, whereas the computations of the values themselves are usually rather simple. For instance, the only value computations needed in a program for sorting might be numerical comparisons and index arithmetic. Flow of control is expressed primarily by sequential, conditional, and iterative commands.

Languages supporting the imperative paradigm include Algol60, Pascal and C.[3]

[3] Business and scientific programming languages such as COBOL and FORTRAN support the imperative paradigm too.

▷ *Functional programming* emphasizes computation of values, scopes of bindings, and procedural abstraction and activation.

In marked contrast to imperative programs, functional programs tend to involve large and complex value computations, but few (if any) assignments. Flow of control is usually expressed by activation of functions (i.e., procedural abstractions that compute values from argument values) giving them other functions as arguments. The scopes of bindings in declarations are often recursive.

Functional programming languages include Lisp, Scheme, Miranda, Standard ML, CAML, and Haskell.[4]

▷ *Concurrent programming* emphasizes concurrent processes, choice between alternatives, and message-passing.

Each process generally does a very limited amount of computation in between sending or receiving messages. Typically, choices between alternatives are decided by the kind of message next received, or by synchronization between two processes.

Languages supporting concurrent programming include PL/I, Algol68, Modula, Ada, occam, and Concurrent ML. (Let us here ignore languages primarily used for specification, such as CCS and LOTOS.)

▷ *Object-oriented programming* may be regarded as imperative programming with a particular discipline for scopes of bindings.

The object-oriented programming paradigm is based on the use of bindings between objects and methods. Objects are essentially identifiable fragments of storage, created either by instantiating classes or by cloning existing objects. In terms of our fundamental concepts, however, object correspond closely to record variables in imperative languages. Methods are procedural abstractions that usually have direct effects on the objects to which they are bound.

Object-oriented programming languages include Simula67, SmallTalk, Beta, Self, C++, and Java.

▷ *Logic programming* emphasizes choice between alternatives with back-tracking, and procedural abstraction.

[4]The scientific language APL supports the functional programming paradigm too.

The logic programming paradigm is quite different from the others (and relatively complicated to explain in terms of the usual fundamental concepts). The basic idea is that a logic program consists of a set of alternative procedure abstractions for each identifier. The choice between them is made by so-called *unification* of patterns between the parameters of the abstractions and the arguments of the activations. This unification generally binds identifiers to terms, and the bindings revert when a failure causes back-tracking.

The primary logic programming language is Prolog. Ex.1.2

Ex.1.3

1.1.4 Variants of Constructs

▷ Analysis of constructs in terms of fundamental concepts helps to understand the various constructs and their relationship.

Sometimes, exactly the same mixture of control and information flow occurs in constructs of many different programming languages, e.g., sequencing of commands. It is useful to regard such a mixture as a basic, language-independent construct, and to analyze particular programming languages in terms of these basic constructs. Moreover, naming the basic constructs establishes a vocabulary for use also in informal description and discussion of programming languages.

▷ Constructs often have *variants*.

Each basic construct is often accompanied by a (usually small) number of *variants*. For instance, the condition of a conditional command is normally a boolean expression, but in languages that don't have boolean values (e.g., C), the condition is a numerical expression, whose value is tested for equality with zero. A conditional command may also have either just a single branch, or alternative branches.

▷ Some variants of constructs can be derived in terms of other constructs.

It can be quite straightforward to derive variants of a construct as combinations of other constructs. In the case of conditional commands, all we need to convert the numerical expression in the condition to a boolean expression is to insert an explicit test for zero; and the single-branch conditional command can obviously be derived by taking a 'skip' command as the alternative branch. Use of derivations allows us to avoid an excessive proliferation of variants of constructs, and also clarifies the relationship between different constructs.

Constructs of a language that can be derived from other constructs of the same language are sometimes referred to as 'syntactic sugar', and serve only to increase the conciseness and perspicuity of programs. Some languages allow programmers to define their own syntactic sugar as so-called *macros*.

▷ Other variants have to be treated separately.

In other cases, derivation of a variant may be complicated, or even impossible. A typical example is application of binary arithmetic operations in expressions: the main concept is that both operands have to be evaluated before the operation can be applied, but languages differ regarding whether the order of operand evaluation is sequential (left to right) or unrestricted. (The former choice, adopted by ML, makes it easier for the programmer to follow the computation; the latter choice discourages the use of side-effects in expressions, and facilitates optimization.) Unfortunately, it is not possible to derive the sequential variant from the unrestricted variants, nor vice versa. Many other constructs have both sequential and unrestricted variants. The proliferation can to some extent be reduced by treating lists of operands or arguments as separate constructs.

Ex.1.4

1.2 Formal Language Descriptions

The fundamental concepts introduced in the preceding section, and the suggested analysis of programming constructs in terms of them, were somewhat vague and informal. We shall next consider how to express the computational meaning of constructs precisely, using formal semantics.

1.2.1 Syntax, Semantics, and Formality

▷ A *semantics* for a programming language models the computational meaning of each program.

The computational meaning of a program—i.e., what actually happens when a program is executed on some real computer—is far too complex to be described in its entirety. Instead, a semantics for a programming language provides abstract entities that represent just the relevant features of all possible executions, and ignores details that have no relevance to the correctness of implementations. Usually, the only relevant features are the relationship between input and output, and whether the execution terminates or not. Features of implementations such as the actual machine addresses where the values of variables are stored, the possibility of 'garbage collection', and the exact running time of the program, may all be regarded as irrelevant, and ignored in the semantics.

▷ The form and structure of programs are determined by their *syntax*.

Before we can even start to give a semantics to a programming language, we need a precise definition of the form and structure of the programs allowed by the language. The syntax of a language determines not only whether a program is legal or not, but also its internal grouping structure.

▷ Descriptions are called *formal* when written in a notation that already has a precise meaning.

Reference manuals and standards for programming languages generally provide some formal descriptions of program syntax, written in some variant of BNF.[5] Regarding the computational meaning of the language, however, the description in the reference manual is generally completely informal, being expressed only in natural language which, even when used very pedantically, is inherently imprecise and open to misinterpretation.

▷ Various frameworks allow formal description of semantics.

There are several main approaches to semantics:

- *operational semantics*, where computations are modelled explicitly;
- *denotational semantics*, where only the contribution of each construct to the computational meaning of the enclosing program is modelled; and
- *axiomatic semantics*, which (in effect) models the relationship between pre- and post-conditions on program variables.

The operational framework known as *Structural Operational Semantics (SOS)* [1] is a good compromise between simplicity and practical applicability, and it has been widely taught at the undergraduate level [2, 3, 4, 5]. The modular variant of SOS used in the present notes, called MSOS [6], has some significant pragmatic advantages, but otherwise remains conceptually very close to the original SOS framework. The other approaches mentioned above, together with a hybrid framework called Action Semantics, will be reviewed in more detail in Chapter 11.

Several kinds of syntax can be distinguished: concrete, abstract, regular, context-free, and context-sensitive. Before proceeding to semantics, let us briefly review and illustrate the differences between them.

[5] BNF is a notation for context-free grammars where terminal and nonterminal symbols are distinguished by the way they are written, and where alternatives for the same nonterminal can be combined into a single rule.

1.2.2 Concrete Syntax

▷ *Concrete* syntax deals with text and parsing.

Concrete syntax determines which text strings are accepted as programs, and provides a parse tree for each accepted program, indicating how the text is supposed to be grouped. Concrete syntax is typically specified by formal grammars, with productions giving sets of alternatives for each nonterminal symbol. A grammar for concrete syntax should be unambiguous, so that each accepted program has a unique parse tree; an alternative approach is to let the grammar remain ambiguous, and provide precedence rules to select a unique parse tree for each program text.

▷ Reference manuals use grammars to specify concrete syntax.

To give a full concrete syntax for a complete programming language is not at all easy, and in fact the BNF-like grammars provided in reference manuals are often qualified by informally-expressed restrictions. The details of lexical symbols such as identifiers and strings are seldom specified formally at all, their informal descriptions generally being regarded as sufficiently clear. (The published *Definition of Standard ML* [7] provides illustrations of both deficiencies, as does the *Java Language Specification* [8].)

▷ Definite Clause Grammars have some advantages over BNF.

An introduction to the variety of techniques available for describing concrete syntax is out of the scope of these notes. Here, we shall adopt one particular technique that generalizes BNF, called *Definite Clause Grammars* (DCGs) [9, Section 7]. These grammars were developed for use in describing natural languages, but they turn out to be rather well-suited for describing programming languages as well. Prolog supports parsing for *arbitrary* DCGs. We shall consider how to write DCGs in Chapter 3, in connection with prototyping, but for now it suffices to be able to read them.

Table 1.1 shows a DCG for a small subset of the language called *bc* [10]. (The full POSIX bc language incorporates precedence rules, real numbers, arrays, combinations of assignment with various operations, and function definitions.) It is straightforward to understand each rule of the form '`·····>····.`' as a production of a context-free grammar: the lowercase words are nonterminal symbols, the characters in double quotation marks form terminal symbols, and the commas simply form sequences of symbols. The semicolons in the rule for `infix` indicate alternative productions for the same nonterminal. (The rules for the nonterminals

Table 1.1: Concrete syntax for a subset of bc

```
prog --> optnewlines, stmts, optnewlines.

stmts --> stmt, i, sep, optnewlines, stmts.
stmts --> stmt.

sep --> ";".
sep --> newline.

stmt --> assign_expr.
stmt --> normal_expr.
stmt --> "{", optnewlines, stmts, optnewlines, "}".
stmt --> "{", optnewlines, "}".
stmt --> "if", i, "(", i, expr, i, ")", i, stmt.
stmt --> "while", i, "(", i, expr, i, ")", i, stmt.

expr --> assign_expr.
expr --> normal_expr.

assign_expr --> var, i, "=", i, expr.

normal_expr --> grouped_expr, i, infix, i, grouped_expr.
normal_expr --> grouped_expr.

grouped_expr --> var.
grouped_expr --> num.
grouped_expr --> "(", i, expr, i, ")".

infix --> "+" ; "-"  ; "*" ; "/"  ; "%"  ;
          "<" ; "<=" ; ">" ; ">=" ; "==" ; "!=".

var --> lower(_).
num --> digits(_).

i --> optwhites.
i --> "/*", up_to("*/").
newlines --> i, newline, optnewlines.
optnewlines --> newlines ; i.
```

`lower(_)`, `digits(_)`, `optwhites`, `newline`, and `up_to("···")` will be given in Chapter 3.)

The DCG in Table 1.1 is completely formal so, provided that it is unambiguous, it can serve as a formal description of the concrete syntax of a programming language. Let us now proceed to compare and contrast concrete syntax with abstract syntax.

Ex.1.5

1.2.3 Abstract Syntax

▷ *Abstract* syntax deals only with structure.

Whereas concrete syntax deals with the actual character strings used to write programs, abstract syntax is concerned only with the 'deep structure' of programs, which is generally represented by trees. In abstract syntax trees, each node is created by a particular *constructor*, and has a branch for each argument of its constructor. Leaves can be created by constant constructors with no arguments. We shall also allow leaves of abstract syntax trees to be abstract mathematical entities: truth-values, numbers, and even operations such as addition.

The constructors of abstract syntax can be specified in various ways, e.g., using datatype definitions (as found in ML), or simply by indicating the argument and result sets of each constructor. Here, we adopt a notation similar to that found in CASL, the Common Algebraic Specification Language [11, 12], which generalizes ML datatype definitions by allowing subset inclusions.

Table 1.2: Abstract syntax for a subset of bc

```
1  C:   Cmd ::= skip | seq(Cmd,Cmd) | cond-nz(Exp,Cmd)
2             | while-nz(Exp,Cmd) | effect(Exp) | print(Exp)
3
4  E:   Exp ::= Boolean | Integer | Var
5             | assign-seq(Var,Exp) | app(Op,Arg)
6
7  VAR: Var ::= Id
8
9  O:   Op  ::=  +  |  -  |  *  | div | mod | ord
10            |  <  |  =< |  >  |  >= |  =  |  \=
11
12 ARG: Arg ::= Exp | tup-seq(Exp,Exp)
13
14 I:   Id  ::= id(Atom)
```

Table 1.2 specifies an abstract syntax which represents the deep structure of the expressions whose concrete syntax was given in Table 1.1. Each mixed-case word starting with a capital letter denotes a set, whereas lowercase words identify constructors. An uppercase letter or word is a variable, restricted to range over a particular set; adding digits and/or primes to it provides further variables over the same set.

The sets `Boolean` and `Integer` are the usual mathematical sets of truth-values and integers, and the operations of the set `Op` are taken to have 'their usual interpretation' as functions on these sets.[6] Notice that `Boolean`, `Integer` and `Var` are included directly as subsets of `Exp`, without an explicit constructor. The set `Id` is a given set of values that represent identifiers.

▷ We shall use symbols *consistently* in the abstract syntax of different languages.

Various choices have to be made when specifying abstract syntax, e.g.: which symbols to use as names for sets and constructors; when to group similar constructs into distinguished sets; and whether to leave lexical constructs represented by sequences of characters, or replace them by the corresponding mathematical entities. Usually, these choices don't significantly affect the process of describing the semantics of a single language. However, considering descriptions of different languages, it is clearly best to use symbols consistently, i.e., adopt a particular (open-ended) nomenclature and stick to it. For example, `Exp` should always be the set of expressions, `skip` should always be the null command/statement, and the target variable of an assignment command should always be the first argument of the corresponding constructor.[7]

Before proceeding, try exercise 1.6.

Ex.1.6

Ex.1.7

1.2.4 Mapping Concrete to Abstract Syntax

▷ *Complete* language descriptions specify both concrete *and* abstract syntax, and the relationship between them.

As should be clear by comparing Table 1.1 with Table 1.2, abstract syntax can be significantly simpler than concrete syntax. In particular, constructs can be grouped into sets of abstract syntax trees purely according to their intended *interpretation*,

[6] Formal definitions of the intended interpretation of operations could be given, e.g., in CASL [11, 12].

[7] N.B. The nomenclature used in the current version of these notes was *experimental*, and has been superseded by CBS.

without regard to parsing issues such as grouping. This makes abstract syntax much more appropriate than concrete syntax as the basis for semantic descriptions. To obtain the semantics of a concrete program, however, we then need to map it (unambiguously) to the corresponding abstract syntax tree. Such a mapping can be specified in various ways, e.g., as a function from concrete derivation trees (produced by parsing the program text) to abstract syntax trees.

▷ DCGs can be used to map concrete to abstract syntax.

DCGs allow the mapping to abstract syntax trees to be specified along with the concrete syntax, which is particularly convenient. Table 1.3 shows a DCG which augments the concrete syntax from Table 1.1 with a mapping to the abstract syntax of Table 1.2.[8] We shall consider how to write such DCGs in Chapter 3, in connection with prototyping.

The main idea is that a nonterminal symbol such as `stmt(C)` is accepted only when `C` is the abstract syntax tree constructed by parsing `stmt`. Each rule generally specifies a constructor to the left of the `-->` symbol; when the constructor is missing, as in the rules for `expr(E)`, the tree constructed by the left side of the rule is the same as that constructed by the right side. The 'symbol' in braces `{···}` in the rule for `num` maps sequences of decimal digits to the corresponding integers (without affecting the concrete syntax).

Ex.1.8

1.2.5 Context-Sensitive Syntax

▷ Context-*free* syntax assumes fixed grouping rules.

The rules for grouping in programming languages are usually fixed, so that grouping analysis can be part of context-free parsing, and specified by context-free grammars. However, some languages (including ML) allow the precedence of infix and prefix operators to be specified in programs, and the grouping of expressions in such languages is clearly beyond the power of context-free grammars: context-*sensitive* parsing is needed.

In practice (e.g., when using parser-generators such as `yacc`) variables can be set and read during context-free parsing to deal with declarations of precedence in programs. An alternative technique is to parse the program first without regard to the precedence declarations, and afterwards rearrange the resulting tree to reflect the intended parse.

[8] The concrete syntax grammar had to be slightly modified first.

Table 1.3: Mapping concrete to abstract syntax for a subset of bc

```
prog(C:'Cmd') --> optnewlines, stmts(C), optnewlines.

stmts(seq(C1,C2)) --> stmt(C1), i, sep, optnewlines, stmts(C2).
stmts(C) --> stmt(C).
sep --> ";".
sep --> newline.
stmt(effect(E)) --> assign_expr(E).
stmt(print(E))  --> normal_expr(E).
stmt(C)    --> "{", optnewlines, stmts(C), optnewlines, "}".
stmt(skip) --> "{", optnewlines, "}".
stmt(cond_nz(E,C)) -->
        "if", i, "(", i, expr(E), i, ")", i, stmt(C).
stmt(while_nz(E,C)) -->
        "while", i, "(", i, expr(E), i, ")", i, stmt(C).

expr(E) --> assign_expr(E).
expr(E) --> normal_expr(E).
assign_expr(assign_seq(VAR,E)) -->
        var(VAR), i, "=", i, expr(E).
normal_expr(app(O,tup_seq(E1,E2))) -->
        grouped_expr(E1), i, infix_op(O), i, grouped_expr(E2).
normal_expr(app(ord, app(O,tup_seq(E1,E2)))) -->
        grouped_expr(E1), i, infix_rel(O), i, grouped_expr(E2).
normal_expr(E) --> grouped_expr(E).
grouped_expr(VAR) --> var(VAR).
grouped_expr(N) --> num(N).
grouped_expr(E) --> "(", i, expr(E), i, ")".

infix_op(+)   --> "+".  infix_op(-)   --> "-".
infix_op(*)   --> "*".  infix_op(div) --> "/".
infix_op(mod) --> "%".
infix_rel(<)  --> "<".  infix_rel(=<) --> "<=".
infix_rel(>)  --> ">".  infix_rel(>=) --> ">=".
infix_rel(=)  --> "==". infix_rel(\=) --> "!=".

var(id(L)) --> lower(L).

num(N) --> digits(Ds), { number_chars(N, Ds) }.

i --> optwhites ; "/*", up_to("*/").
newlines --> i, newline, optnewlines.
optnewlines --> newlines ; i.
```

▷ Context-*sensitive* syntax can also deal with constraints.

Well-formedness properties, such as declaration-before-use and type-checking constraints, are *inherently* context-sensitive, and cannot be specified by context-free grammars. The abstract syntax of programs satisfying these well-formedness conditions can however be regarded as a subset of an abstract syntax where the conditions are ignored. Equivalently, we may regard checking constraints as part of semantics, as explained in the next section.

DCGs are sufficiently powerful to deal with context-sensitive parsing; in fact they were originally developed for use in connection with parsing natural languages, which involves not only context-sensitivity but also ambiguity. So-called *attribute grammars* provide an alternative framework for context-sensitive parsing.

So much for syntax. Let us now consider the various kinds of semantics: static semantics, dynamic semantics, and semantic equivalences.

1.2.6 Static Semantics

▷ *Static* semantics models compile-time checks.

When syntax is restricted to be context-free, checking whether programs satisfy well-formedness constraints necessarily becomes part of semantics. This kind of semantics is called static, since it concerns only those checks that are to be performed before running the program, e.g., checking that all parts of the program are type-correct.[9] The only relevant feature of the static semantics of a program is whether the program has passed the checks or not (although error reports issued by compilers could be modelled when these are implementation-independent).

▷ We use MSOS to specify static semantics.

Tables 1.4–1.6 give an impression of how static semantics can be specified in MSOS. The notation used will be explained in detail in the next two chapters, and its use illustrated throughout the rest of these notes. For now, it should suffice to observe that `E ===> VT` asserts that expression `E` has values of type `VT` (similarly for commands, etc.), and that a horizontal line indicates that the assertion below the line holds whenever all the assertions above the line hold. Assertions are assumed to be relative to a so-called environment, `ENV`, which provides a type for each initially-bound identifier. Notice that `ENV` is mentioned only in the rule where it is used, and in fact the initial type environment is implicitly propagated to all sub-constructs in this sublanguage of bc.

Ex.1.9

[9] Static *analysis* goes further than static semantics by checking properties that do not affect the well-formedness of programs.

Table 1.4: Static semantics for a subset of bc

```
State ::= Cmd  | Exp | Var | Op | Arg

Final ::= void | ValueType | CellType | FuncType | PassableType

VT : ValueType    ::= bool | int

CLT: CellType     ::= ref(StorableType)

FNT: FuncType     ::= func(PassableType,ValueType)

PVT: PassableType ::= ValueType | tup(ValueType,ValueType)

BV : Bindable     ::= CellType

SVT: StorableType ::= int

/* Labels: */

Label = {env:Env, ...}

/* Command typechecking rules: */

skip:Cmd ===> void

C1 ===> void, C2 ===> void
-------------------------
 seq(C1,C2):Cmd ===> void

  E ===> int, C ===> void
-------------------------
cond-nz(E,C):Cmd ===> void

  E ===> int, C ===> void
-------------------------
while-nz(E,C):Cmd ===> void

       E ===> VT
----------------------
effect(E):Cmd ===> void

       E ===> VT
----------------------
print(E):Cmd ===> void
```

Continued in Table 1.5

Table 1.5: Static semantics for a subset of bc, ctd.

```
/* Expression typechecking rules: */

B:Exp ===> bool

N:Exp ===> int

VAR ===> ref(VT)
----------------
VAR:Exp ===> VT

 VAR ===> ref(VT), E ===> VT
-----------------------------
assign-seq(VAR,E):Exp ===> VT

O ===> func(PVT,VT), ARG ===> PVT
---------------------------------
     app(O,ARG):Exp ===> VT

/* Operation typechecking rules: */

  (+):Op ===> func(tup(int,int),int).
  (-):Op ===> func(tup(int,int),int).
  (*):Op ===> func(tup(int,int),int).
(div):Op ===> func(tup(int,int),int).
(mod):Op ===> func(tup(int,int),int).
(ord):Op ===> func(bool,int).
  (<):Op ===> func(tup(int,int),bool).
 (=<):Op ===> func(tup(int,int),bool).
  (>):Op ===> func(tup(int,int),bool).
 (>=):Op ===> func(tup(int,int),bool).
  (=):Op ===> func(tup(int,int),bool).
 (\=):Op ===> func(tup(int,int),bool).

/* Variable typechecking rules: */

    lookup(I,ENV) = ref(SVT)
--------------------------------
I:Var ={env=ENV,---}=> ref(SVT)
```

Continued in Table 1.6

Table 1.6: Static semantics for a subset of bc, continued

```
84  /* Argument typechecking rules: */
85
86    E ===> VT
87  -------------
88  E:Arg ===> VT
89
90        E1 ===> VT1, E2 ===> VT2
91  ------------------------------------
92  tup-seq(E1,E2):Arg ===> tup(VT1,VT2)
```

1.2.7 Dynamic Semantics

▷ *Dynamic* semantics models run-time behaviour.

Dynamic semantics concerns the observable behaviour when programs are run. Here, we assume that well-formedness of the programs has already been checked by static semantics: we do not need to bother with the dynamic semantics of ill-formed programs.

▷ We use MSOS to specify dynamic semantics too.

Tables 1.7–1.9 give an impression of how dynamic semantics is specified in MSOS. The notation used is compatible with that used for describing static semantics in Tables 1.4–1.6, but we use a different notation for transitions, to avoid confusion between static and dynamic semantics: `E ---> V` asserts that `E` evaluates to value `V`. However, evaluation of expressions (and execution of most other constructs in programming languages) is essentially a gradual process, often taking many small steps. This is reflected by considering intermediate states for evaluation: `E ---> E'` indicates that the next step of evaluating `E` leads to the expression `E'`, which need not be the value computed by `E`.

More generally, `E --{...}-> E'` allows a step to have *side-effects*, such as assigning to variables (although in fact this extra generality isn't needed for the selection of expression constructs shown in Table 1.2). We refer to `{...}` as the *label* of the step. The exploitation of labels to represent side-effects and other kinds of information processing is one of the main characteristics of MSOS. It will be explained in detail in Chapter 2. Ex.1.10

Table 1.7: Dynamic semantics for a subset of bc

```
State ::= Cmd  | Exp | Var | Arg

Final ::= skip | Value | Cell | Passable

Var   ::= Cell

Arg   ::= Passable

/* Data: */

V :  Value    ::= Boolean | Integer

BV : Bindable ::= Cell

PV : Passable ::= Value | tup(Value,Value)

SV : Storable ::= Integer

/* Labels: */

Label = {env:Env,store,store':Store,out':Value*,...}


/* Command transition rules: */

            C1 --{...}-> C1'
-------------------------------------
seq(C1,C2):Cmd --{...}-> seq(C1',C2)

seq(skip,C2):Cmd ---> C2


               E --{...}-> E'
-----------------------------------------
cond-nz(E,C):Cmd --{...}-> cond-nz(E',C)

cond-nz(0,C):Cmd ---> skip

        N \= 0
-----------------------
cond-nz(N,C):Cmd ---> C
```

Continued in Table 1.8

Table 1.8: Dynamic semantics for a subset of bc, ctd.

```
while-nz(E,C):Cmd --->
 cond-nz(E,seq(C,while-nz(E,C)))


            E --{...}-> E'
---------------------------------
effect(E):Cmd --{...}-> effect(E')

effect(V):Cmd ---> skip


           E --{...}-> E'
-------------------------------
print(E):Cmd --{...}-> print(E')

print(V):Cmd --{out'=V,---}-> skip


/* Expression transition rules: */

    VAR --{...}-> VAR'
----------------------
VAR:Exp --{...}-> VAR'

        lookup(CL,S) = V
---------------------------------
CL:Exp --{store=S,store'=S,---}-> V


                  VAR --{...}-> VAR'
-------------------------------------------------
assign-seq(VAR,E):Exp --{...}-> assign-seq(VAR',E)

                   E --{...}-> E'
-----------------------------------------------
assign-seq(CL,E):Exp --{...}-> assign-seq(CL,E')

                 (CL|->SV)/S = S'
-----------------------------------------------------
assign-seq(CL,SV):Exp --{store=S,store'=S',---}-> SV
```

Continued in Table 1.9

Table 1.9: Dynamic semantics for a subset of bc, ctd.

```
           ARG --{...}-> ARG'
-----------------------------------
app(O,ARG):Exp --{...}-> app(O,ARG')

        O(V1,V2) = V'
-----------------------------
app(O,tup(V1,V2)):Exp ---> V'

      O(V) = V'
--------------------
app(O,V):Exp ---> V'


/* Variable transition rules: */

    lookup(I,ENV) = CL
-------------------------
I:Var --{env=ENV,---}-> CL


/* Argument transition rules: */

    E --{...}-> E'
------------------
E:Arg --{...}-> E'

                E1 --{...}-> E1'
-----------------------------------------
tup-seq(E1,E2):Arg --{...}-> tup-seq(E1',E2)

                E2 --{...}-> E2'
-----------------------------------------
tup-seq(V1,E2):Arg --{...}-> tup-seq(V1,E2')

tup-seq(V1,V2):Arg ---> tup(V1,V2)
```

1.2.8 Semantic Equivalence

▷ *Equivalences* between programs abstract from details of models.

A formal semantics should give, for each program, an abstract model that represents just the relevant features of all possible executions of that program. Then two programs are regarded as semantically equivalent when their models are the same (up to isomorphism). An alternative (and more practical) approach is to give models which are less abstract, and define semantic equivalence uniformly for all such models. We shall consider this topic in Chapter 4, in the context of the models provided by MSOS specifications.

1.2.9 Complete Descriptions and Modularization

▷ Complete descriptions include static semantics, dynamic semantics, and semantic equivalence.

Given a syntactically-correct program, its static semantics is needed in order to determine whether the program is well-formed, and thus executable. The dynamic semantics then provides a model of program executions. The semantic equivalence relation abstracts from those features that are irrelevant to implementation correctness. All together this provides the *complete* semantics of the given program.

▷ Modularization makes reuse explicit and ensures consistency.

Thanks to the inherent modularity of MSOS specifications, it is possible to reuse the description of a particular construct from one language when the same construct occurs in another language. For example, we should be able to reuse the description of the `seq(Cmd,Cmd)` construct in any language that includes (binary) sequencing of commands.[10] However, if we were merely to 'copy and paste' the rules for `seq(Cmd,Cmd)` from one language description to another, the only link between the two descriptions of the same construct would be their use of the same constructor symbol—and there would be no inherent protection against making changes in one of them, without making the same changes in the other.

By dividing an MSOS into named modules, and reusing parts of descriptions by simply referring to the name of a module (without making a copy) the possibility of inconsistent changes is eliminated: any change to the named module affects

[10] It is also possible to describe sequences of arbitrary numbers of commands in MSOS, with binary sequences as a special case.

also all references to that name.[11] As an added advantage, it becomes immediately apparent when two languages have some constructs in common.

▷ MSOS modules describe individual abstract constructs.

To maximize reusability, each module should specify the semantics of just a *single construct* (or of a very small group of strongly associated constructs). Auxiliary modules may however be introduced as well (e.g., for common datatypes such as integers or lists, needed in the description of different constructs).

▷ Modules are divided into abstract syntax, static semantics, and dynamic semantics.

The descriptions of the static and dynamic semantics of a construct are both based on its abstract syntax, but they are otherwise quite independent. Moreover, the abstract syntax of a construct is of independent interest. We shall therefore divide each module that describes an abstract construct into separate descriptions of abstract syntax, static semantics, and dynamic semantics, where the semantic descriptions (implicitly) refer to the abstract syntax description.

▷ The semantic description of a complete language is a list of references to the descriptions of individual constructs.

Given a collection of named modules that cover all the abstract constructs involved in a particular programming language, the semantic description of the language merely consists of a list of references to the required modules (possibly augmented by some further language-specific requirements).

To get an impression of how this works in practice, consider the descriptions of the abstract syntax and semantics (static and dynamic) given earlier in this chapter. We shall use the symbols that name sets of abstract syntax trees and their constructors to form module names. For example, `Cmd/seq/ABS` names the module that specifies the abstract syntax of the constructor `seq(Cmd,Cmd)` (shown in Table 1.10), `Cmd/seq/CHK` its static semantics (Table 1.11), and `Cmd/seq/RUN` its dynamic semantics (Table 1.12). The auxiliary module `Cmd/ABS` merely declares `C` as a variable ranging over the set `Cmd` (Table 1.13), whereas `Cmd/CHK` (Table 1.14) and `Cmd/RUN` (Table 1.15) specify common notation required in the static, resp. dynamic semantics of arbitrary commands. Modules for all the other sets of constructs are named analogously. The modules for datatypes have names such as `Data/Integer/ABS`.

[11] Version control could be added too, if desired.

Table 1.10: Cmd/seq/ABS: Abstract syntax of sequential commands

```
Cmd ::= seq(Cmd,Cmd)
```

Table 1.11: Cmd/seq/CHK: Static semantics of sequential commands

```
C1 ===> void, C2 ===> void
--------------------------
 seq(C1,C2):Cmd ===> void
```

Table 1.12: Cmd/seq/RUN: Dynamic semantics of sequential commands

```
            C1 --{...}-> C1'
-----------------------------------
seq(C1,C2):Cmd --{...}-> seq(C1',C2)

seq(skip,C2):Cmd ---> C2
```

Table 1.13: Cmd/ABS: Abstract syntax of arbitrary commands

```
C: Cmd
```

Table 1.14: Cmd/CHK: Static semantics of arbitrary commands

```
State ::= Cmd

Final ::= void
```

Table 1.15: Cmd/RUN: Dynamic semantics of arbitrary commands

```
State ::= Cmd

Final ::= skip

Cmd ::= skip
```

▷ References to other modules can usually be left implicit.

It would be quite tedious if every module had to specify explicitly the names of all the modules that introduce the notation it uses. For instance, the module `Cmd/assign-seq` would then need to refer to the modules `Cmd`, `Var`, and `Exp`, which declare the corresponding sets. We shall allow such 'evident' module dependencies to be omitted. This is possible since we are using the same symbols for sets of constructs and for forming module names: from the rule `Cmd ::= assign-seq(Var,Exp)` we see immediately that modules declaring the sets `Cmd`, `Var`, and `Exp` are required.

Table 1.16: Modular abstract syntax for a subset of bc

```
3  see     Cmd/skip,            Cmd/seq,             Cmd/cond-nz,
4          Cmd/while-nz,        Cmd/effect,          Cmd/print
5
6  see     Exp/Boolean,         Exp/Integer,         Exp/Var,
7          Exp/assign-seq,      Exp/app-Op
8
9  see     Var/Id,   Id/id
10
11 see     Op/plus,  Op/minus,  Op/times,  Op/div,  Op/mod,  Op/ord,
12         Op/lt,    Op/leq,    Op/gt,     Op/geq,  Op/eq,   Op/neq
13
14 see     Arg/Exp, Arg/tup-seq
```

Ex.1.11

▷ We obtain a complete language description by combining concrete and abstract syntax with static and dynamic semantics.

Given abstract syntax modules for all the abstract constructs listed in Table 1.2, the complete abstract syntax for our subset of bc can now be specified as in Table 1.16. When we are specifying the abstract syntax part of an MSOS, names of other modules refer implicitly to their abstract syntax parts too; similarly for the static and dynamic semantics parts. When these collected parts are combined with the mapping from concrete to abstract syntax that was specified in Table 1.3, we obtain a *complete* description[12] of our little language, with the semantics based entirely on highly reusable modules. (Our descriptions of concrete syntax and its mapping to abstract syntax could be modularized too, but here we shall not bother with this, except in connection with pure extensions of languages.)

[12] Strictly speaking, to complete the language description we should also specify which semantic equivalence to use.

1.2.10 Practical Uses

The design and implementation of programming languages is a topic of major importance in Computer Science. Formal descriptions of program syntax (regular, context-free, and context-sensitive grammars) have become accepted as practically useful for documentation in reference manuals and standards, as well as for generating efficient (and automatically correct) parsers for use in compilers. What practical uses might there be for formal descriptions of *semantics*? Here are some possibilities:

▷ Consideration of formal semantics can lead to better language design.

The designers of some programming languages have a particularly good awareness of formal semantics, which appears to have a positive influence on the coherence of their designs. In particular, ML was designed by expert semanticists, and is generally regarded as being quite well designed, despite some infelicities of the concrete syntax.

Modularization of semantic descriptions might encourage language designers to reuse familiar constructs from previous languages on a greater scale. However, random selections of constructs are unlikely to lead to good designs.

▷ Semantic descriptions should be useful for recording language design decisions.

Language designers use grammars for recording their decisions regarding the syntax of programs; semantic descriptions should be useful for recording decisions about order of evaluation, scopes of bindings, etc. The published definition of ML [7] covers concrete and abstract syntax, static semantics, and dynamic semantics, and captures the design decisions remarkably concisely (although not always perspicuously, and at the expense of introducing some semi-formal 'conventions').

Modularization of semantic descriptions could significantly lower the amount of effort required to use them, and make them more attractive for documentation of evolving designs than hitherto.

▷ Development of semantic descriptions tends to reveal flaws in completed designs.

Writing a complete semantic description of a language entails studying every detail of the language, and in the process, weaknesses of the documentation and/or the implementation may come to light. For instance, the authors of a recent description of (most of) Java reported several previously-unnoticed questions that

had inadvertently been left open by the Java Language Specification. However, it should be kept in mind that formal semantic descriptions can be flawed too; validation by prototyping and theorem-proving are essential to ensure that a description indeed captures the *intended* semantics.

▷ Semantic descriptions could be used to generate compilers and interpreters.

Although some interesting prototype systems have been implemented, compilers and interpreters generated from semantic descriptions are generally not efficient enough to be practically useful—except for allowing empirical validation of the semantic description itself.

Another barrier to use of semantics-based compiler generation is that developing a complete semantic description from scratch can be as much work as writing a (prototype) compiler directly. Modularization removes this barrier. Compiler generation based on MSOS has not yet been explored. However, it appears to be feasible for Action Semantics (see Chapter 11), which is just as modular as MSOS (and which is itself defined using MSOS).

▷ Semantic descriptions might become widely accepted for use in standards.

Occasionally, a formal semantics has actually been included in a standard or reference manual, as a supplement to the usual informal description. However, a non-modular semantic description of a complete language would often be a very lengthy document, and its size alone would discourage its insertion in the reference manual.

Given a public repository of semantic descriptions of all the required individual abstract constructs, all that would be needed in the standards document would be the description of the mapping from concrete to abstract syntax, which is much more concise.

▷ Formal semantics provides a basis for sound reasoning about program behaviour or equivalence.

Formal proofs about the specified semantics of programs are feasible—but quite laborious—for small-scale languages; the complexity of the semantics of practical, large-scale languages makes it difficult to prove useful facts about interesting programs in them. Use of semi-automatic theorem provers such as PVS can help.

Development of modular proof techniques for MSOS might allow properties of particular constructs to be established independently of particular languages, leading to higher-level proofs about properties of programs based on those constructs.

▷ Proofs about semantic descriptions could establish general properties of languages.

Again, this is fine in principle, and has been demonstrated in several textbooks on semantics, but there are few examples for large-scale languages. Modularization might be an enabling factor here too.

▷ Formal semantics may support the teaching of concepts of programming languages.

The author is hoping that the present lecture notes will demonstrate this...

▷ *Modularization of formal semantic descriptions* is of crucial importance for the realization of their potential usefulness.

As we shall see in Chapter 11, few other semantic frameworks come close to the degree of modularization provided by MSOS. This pragmatic feature of MSOS descriptions, together with proper tool support for their development and prototyping, should greatly facilitate description and reasoning about full-scale programming languages.

Ex.1.12

Ex.1.13

Summary

In this chapter:

- We considered fundamental concepts of information and control flow, and their degrees of involvement in the main programming paradigms.
- We also advocated the use of formal semantics based on context-free abstract syntax; complete language descriptions involve concrete syntax and its mapping to abstract syntax, both static and dynamic semantics, and appropriate definitions of semantic equivalence.
- We illustrated how complete language descriptions can be formulated and modularized.
- Finally, we surveyed the potential usefulness of formal semantic descriptions in connection with language design, implementation, and standardization, as well as for reasoning about programs and supporting the teaching of their fundamental concepts. Modularization of semantic descriptions was claimed to be of crucial significance for realization of this potential.

Exercises

Ex. 1.1 List the main kinds of constructs in Java. For each kind of construct, indicate some characteristic features of the flow of information (e.g., scopes of bindings, the possibility of imperative effects) and control (e.g., regarding sequencing, and the possibility of exceptions).

Ex. 1.2 List all the programming languages that you have ever used, indicating which programming paradigm(s) they support best.

Ex. 1.3 Discuss how one might (systematically) encode:

(a) imperative programming in a pure functional language (e.g, in Standard ML without the use of references);

(b) functional programming in an imperative language (e.g., higher-order functions in C).

Ex. 1.4 Many languages incorporate 'short-cut' evaluation of conjunction and disjunction of boolean expressions (e.g., the infix constructs && and || in Java, and `andalso` and `orelse` in ML).

(a) Explain the control and information flow of 'short-cut' evaluation.

(b) Give an expression (e.g., in Java or ML) whose result depends on whether the second operands of conjunctions are evaluated or not.

(c) Indicate how this variant of operation application can be derived from other (more fundamental) expression constructs.

(d) For which numerical operations would it make sense to allow 'short-cut' evaluation in purely numerical expressions? What are the characteristic properties of such operations?

Ex. 1.5 This exercise involves running Prolog (as do many later exercises). An efficient implementation of Prolog called SWI-Prolog can be downloaded free for various operating systems from **`http://www.swi-prolog.org`**.[13] It is assumed that you are already familiar with loading and running Prolog programs, but no experience of programming in Prolog is required for this exercise.

All the formal specifications shown in these notes are available (together with some free software) at **`http://www.cs.swan.ac.uk/~cspdm/MSOS/`**, and for

[13] The author's Prolog code has been tested mainly on SWI-Prolog version 5.0, but should run also on later versions of SWI-Prolog. Please report any incompatibilities with other versions, including sufficient details to indicate what needs to be fixed.

downloading as a single file at **http://www.cs.swan.ac.uk/~cspdm/MSOS.zip**. A copy may also be installed on your local filesystem. The DCG of Table 1.1 is located at `Test/bc/CFG.pro`, and the omitted rules are available in `Test/bc/LEX.pro`.

(a) Start Prolog from the `MSOS` directory.

(b) Load the DCGs in `Test/bc/LEX.pro` and `Test/bc/CFG.pro` (in that order).

(c) Using the query `phrase(prog,"···")` with various second arguments, check that some of the bc statements in the files `Test/bc/*.bc` are accepted by the loaded DCGs.

(d) Read carefully through DCG of Table 1.1, to understand exactly which restrictions have been made to obtain a simple sublanguage of bc. Check that your understanding is correct by formulating and running some further (positive and negative) tests—noting any surprises that arise.

*(e) Specify the same subset of bc in any other framework that supports grammar-based parsing, and check that your test statements give equivalent results.

(Further exercises on concrete syntax will be set in Chapter 3.)

***Ex. 1.6** This exercise requires some familiarity with programming in Prolog and ML. Consider the specification of abstract syntax given in Table 1.2.

(a) Write a Prolog term that involves all the specified constructors for the sets `Cmd`, `Exp`, `Var`, and `Arg` (replacing hyphens by underscores) as well as the boolean values `true` and `false`, a couple of integers, a few elements of `Op`, and at least one element of `Id`.[14]

(b) For each specified set of abstract syntax trees, give a Prolog predicate which holds iff its argument is in that set. Check that your predicates give the expected answers on a variety of terms, including one that involves most of the specified constructors.

(c) Give an ML datatype definition corresponding (as closely as possible) to the abstract syntax given in Table 1.2, and an ML term that involves most of the specified constructors. Check that your term has the expected ML type.

[14] Elements of `Id` are written `id(a)`, ..., `id(z)`.

***Ex. 1.7** This exercise assumes that you have *not* yet studied Section 1.2.4.

(a) Propose a mapping from the concrete syntax given in Table 1.1 to the abstract syntax given in Table 1.2, assuming that the intended interpretation of the abstract syntax constructors is consistent with that suggested by the symbols themselves (where `cond` abbreviates 'conditional', and `seq` abbreviates 'sequential').

(b) Give a concrete bc program which is mapped to a term involving at least all the abstract syntax constructors for `Cmd` and `Exp`.

(c) Study Section 1.2.4 and compare your mapping with that specified in Table 1.3.

Ex. 1.8

(a) Consider the DCG shown in Table 1.3. What differences are there between it and the DCG shown in Table 1.1, apart from the addition of arguments to the nonterminal symbols?

(b) Start Prolog from the `MSOS` directory. Load the DCGs in `Test/bc/LEX.pro` and `Test/bc/SYN.pro` (in that order). By using the query `phrase(prog(C),"···")` with various second arguments, check that some of the bc statements in the directory `Test/bc/` are accepted by the loaded DCGs, observing the terms that Prolog reports as the value of `C`.

(c) (Assuming that you have answered exercise 1.6:) Check that all the observed terms represent elements of the set `Cmd`. Hint: Combine the check on `C` with the above query.

*(d) Specify the same mapping from concrete to abstract syntax in any other framework that supports grammar-based parsing, and check that your test statements give equivalent results.

Ex. 1.9 Prolog code generated from the specification of static semantics in Tables 1.4–1.6 is located at `Test/bc/CHK.pro`. This exercise involves running that code and testing which bc programs are well-formed according to the static semantics.

(a) Start Prolog from the `MSOS` directory. Load the files `Tool/load.pro`, `Test/bc/SYN.pro`, `Test/bc/ABS.pro`, `Test/bc/CHK.pro`, and `Test/bc/CHK-init.pro`. By using the query `prog_chk(_,"···")` with various second arguments, check that some of the bc statements in the directory `Test/bc/` are accepted by the loaded DCGs.

(b) Study Tables 1.4–1.6, to understand exactly which combinations of abstract constructs are well-formed (i.e., are accepted by the static semantics). By using the query `chk(T)` with various abstract syntax trees `T`, check that your understanding is correct by formulating and running some further (positive and negative) tests—noting any surprises that arise.

*(c) Specify the same static semantics of bc in any other framework that supports grammar-based parsing and type-checking, and check that your test statements give equivalent results.

Ex. 1.10 Repeat the previous exercise for dynamic semantics, replacing all occurrences of `CHK` above by `RUN`, and using queries of the form `prog_run(_,"···")` and `run(T)`.

***Ex. 1.11** All available reusable (language-independent) modules for sets of constructs and individual constructs are located in `MSOS/Cons/`. The original MSOS specifications have the extension `msdf`[15] and the Prolog files generated from them have the extension `pl`.

(a) Find the `ABS` modules listed in Table 1.16, and compare them with the monolithic specification of abstract syntax given in Table 1.2.

(b) Find the corresponding `CHK` modules, and compare them with the monolithic specification of static semantic given in Tables 1.4–1.6.

(c) Find the corresponding `RUN` modules, and compare them with the monolithic specification of static semantic given in Tables 1.7–1.9.

Ex. 1.12 For each of the potential practical uses of formal semantics, discuss whether modularity and reusability of descriptions is likely to be important, desirable, or irrelevant.

***Ex. 1.13** Take a look at the official Java Language Specification, available at `https://docs.oracle.com/javase/specs/`.

(a) Compare the grammar for `Expression` in Section 18.1 with that given in Section 15. Discuss the pros and cons of having both grammars. Which would you prefer to use as the basis for an abstract syntax for Java?

(b) Assess the extent to which the Java Language Specification provides a completely precise and unambiguous self-contained description of Java constructs.

[15] MSDF stands for Modular SOS Definition Formalism.

Chapter 2

Modular Structural Operational Semantics

Section 2.1 introduces *labelled transition systems*, which are used as models in the structural approach to operational semantics (SOS). It generalizes the treatment of labels on transitions to provide the foundations for MSOS (our modular variant of SOS), and explains the basic principles for modelling flow of control and flow of information.

Section 2.2 introduces a Modular SOS Definition Formalism, MSDF, for defining models in MSOS, and provides its mathematical interpretation. Chapter 3 will illustrate how MSDF should be written so as to allow its translation to Prolog in connection with modular prototyping.

▷ The Structural Operational Semantics (SOS) framework gives simple, intuitive models of computations.

Plotkin, in his original 1981 Aarhus lecture notes on SOS [1], wrote:

> *It is the purpose of these notes to develop a simple and direct method for specifying the semantics of programming languages. Very little is required in the way of mathematical background [...]. Apart from a simple kind of mathematics the method is intended to produce concise comprehensible semantic definitions. Indeed the method is even intended as a direct formalization of (many aspects of) the usual informal descriptions [...]. It is an operational method of specifying semantics based on syntactic transformations of programs and simple operations on discrete data.*

▷ SOS descriptions of programming languages are not modular.

Plotkin achieved his aims with SOS, and the framework has subsequently become widely known, However, he was well aware that there are problems regarding modularity in SOS descriptions of programming languages (op.cit.):

> *As regards modularity we just hope that if we get the other things in a reasonable shape, then current ideas for imposing modularity on specifications will prove useful.*

Plotkin's hopes were not fulfilled. For instance, when extending the expressions of a pure functional language to include references and/or concurrency primitives, a conventional SOS description of all the functional constructs has to be completely reformulated. The way that each construct is described in SOS depends crucially on what other constructs are included in the language being described. This dependence precludes giving a single reusable description of each common programming construct.

▷ MSOS solves the modularity problems of SOS.

The MSOS variant of SOS [6, 13, 14] is quite simple, both conceptually and notationally. It allows a remarkably high degree of modularity. In contrast to what Plotkin envisaged, MSOS does not simply "impose" modularity on SOS: the foundations have to be adjusted, and some new notation introduced, to allow reusable descriptions of individual constructs. Imposing the explicit naming of modules on MSOS descriptions of programming languages is then a relatively easy matter, and only serves to facilitate reuse of descriptions that are already *inherently* reusable.

The present notes provide the first full-scale pedagogical presentation of MSOS.

2.1 Operational Models

▷ Operational models of programming languages are *transition systems*.

Various kinds of abstract machines have been defined in studies of computability and algorithms: finite automata, push-down automata, Turing machines, etc. One feature they share is that a machine has only finitely-many control states. In the structural approach to operational semantics (both SOS and MSOS), however, the abstract syntax of the program is used to control the flow of computations, which obviously requires machines with infinitely-many control states. Such models are provided by *transition systems*. Program executions are represented by (possibly-infinite) sequences of transitions in these systems.

Ex.2.1

2.1.1 Transition Systems

▷ A transition system has a set of states and a transition relation.

The basic definition is independent of the details of states:

Definition 2.1 (transition system, computation) *A transition system TS consists of a set State of states S, together with a binary relation $\longrightarrow$ on State, i.e., $\longrightarrow \subseteq (State \times State)$. We write $S \longrightarrow S'$ to indicate the existence of a transition from S to S', i.e. $(S,S') \in \longrightarrow$.*

A computation *starting from some particular state S_1 is a possibly-empty, possibly-infinite, maximal sequence of transitions, written $S_1 \longrightarrow S_2 \longrightarrow \cdots$ (or simply S_1, when there are no transitions from S_1).*

Maximality of a sequence means that it cannot be extended with further transitions (either because it is already infinitely long, or because it is finite and there are no transitions from its last state).

▷ We can restrict the sets of *initial* and *final* states of computations.

The definition of computations can be modified to allow only those sequences where the initial state is in a given set Init. Then states of the transition system that do not occur in any such computation are called *unreachable*.

Similarly, the definition can be changed to allow only those sequences where the final state is in a given set Final. This doesn't restrict infinite computations, since they have no final states. In contrast to automata theory, we insist that there are to be no further transitions from states in Final; other states from which there are no further transitions are called *stuck*.

Reachable states that are neither initial nor final are called *intermediate*. Notice that the intersection of Init and Final might be non-empty, and that computations from states in this set have no transitions. Ex.2.2

Example 2.2 Let the elements of State represent stages in the execution of bc statements. Consider computations whose initial state represents `if(n==42){n}`, and let Final include an element that represents the empty statement `{}`, indicating the successful termination of the execution of a statement. Some of these computations are depicted in Figure 2.1.

The second state of a computation from the initial state depends on the value of `n`: suppose this is 27, then the first transition leads to an intermediate state representing `if(27==42){n}`. The next state represents `if(0){n}`, and the only transition from this state leads to a final state.

There are similar computations for all possible (integer) values of `n`, each having a different second state. The computation where `n` initially has the value 42 also has a different third state, representing the remaining statement `if(1){n}`, leading to the state representing `{n}`. Provided that `n` still has its initial value 42, the next state should represent `{42}`, and one more transition is needed before reaching the final state that represents termination of the statement. If the value of `n` might have changed during the previous transitions, however, there is also the possibility of transitions from `{n}` to other intermediate states before reaching the final state.

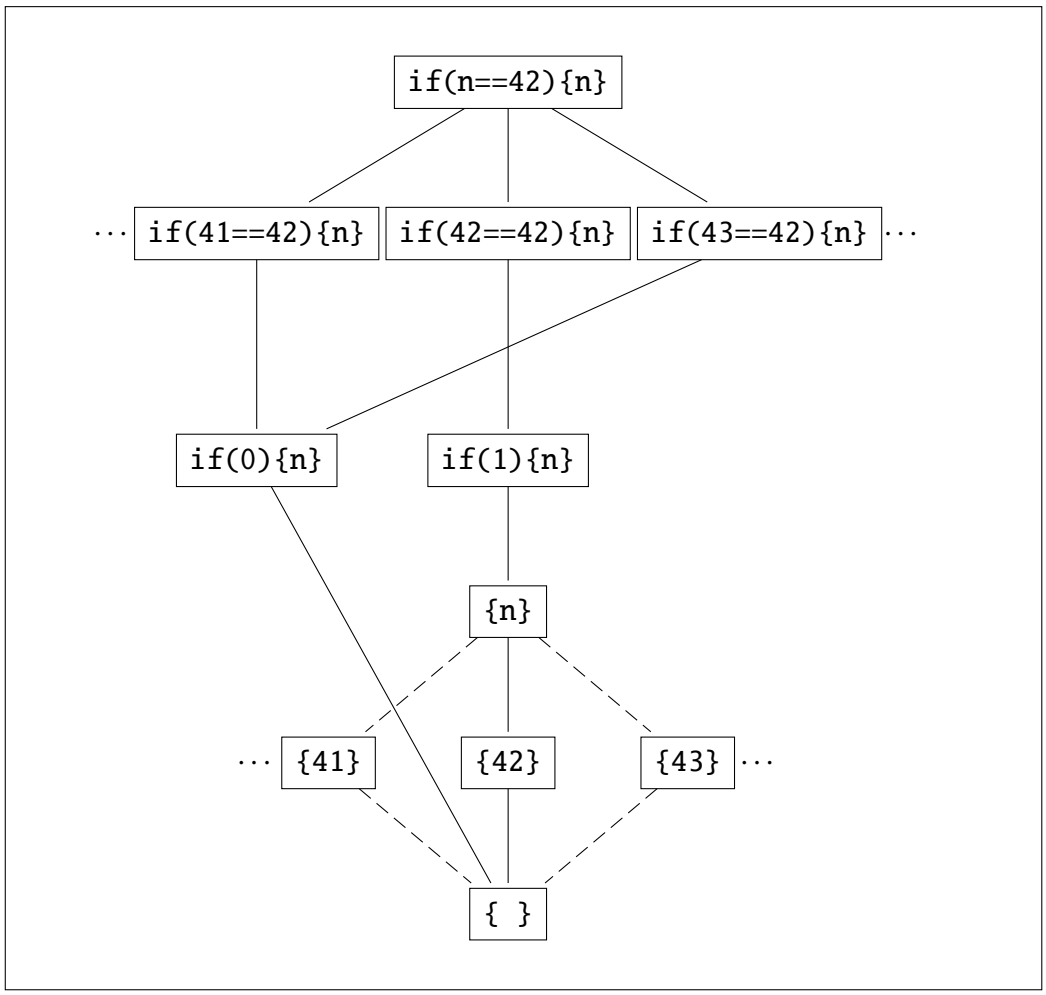

Figure 2.1: Part of a transition system for bc statement execution

It might also be that n initially has no value.[1] Then there is no transition from the initial state—and hence no computation, since this initial state should clearly not be a final state when n does have a value. Notice that the initial state is not hereby regarded as stuck, since we are considering all possibilities in the same transition system. Ex.2.3

▷ *Labels* on transitions can model *why* a computation is going from one state to another.

Unlabelled transitions are quite uninformative, merely indicating that it is possible to go (directly) from one state to another during a computation. By allowing transitions to be *labelled*, we can model the information that determines choices between transitions (e.g., the current values of identifiers) and any changes to the current information during each transition (e.g., updating the value of an identifier, or printing a value).

Definition 2.3 (labelled transition system, computation, trace) *A labelled transition system LTS consists of a set State of states S, a set Label of labels L for transitions, and a ternary relation* $\longrightarrow \subseteq (State \times Label \times State)$. *We write* $S \xrightarrow{L} S'$ *to indicate the existence of a transition labelled L from S to* S'.

A computation *starting from some particular state* S_1 *in a labelled transition system is a possibly-empty, possibly-infinite, maximal sequence of transitions, written* $S_1 \xrightarrow{L_1} S_2 \xrightarrow{L_2} \cdots$. *The* trace *of a computation is the sequence of labels on its transitions.*

Sets of final and initial states can be given for labelled transition systems exactly as for ordinary transition systems.

Example 2.4 Let us label all the transitions considered in the previous example by the current value of n, as depicted in Figure 2.2. Computations are now of the form $S_1 \xrightarrow{n_1} S_2 \xrightarrow{n_2} \ldots$. The labels on the transitions exhibit the information that determines the choice between the various transitions from the initial state. When the initial value of n is 42, the transition from the state representing the remaining statement {n} to that representing {42} is labelled by 42 too, and transitions for the case that the value of n has somehow changed are labelled differently.[2] Ex.2.4

[1] For the sake of the example, let us assume that bc variables might be uninitialized.

[2] For more complex expressions that depend on the values of several identifiers, we could label transitions with an environment.

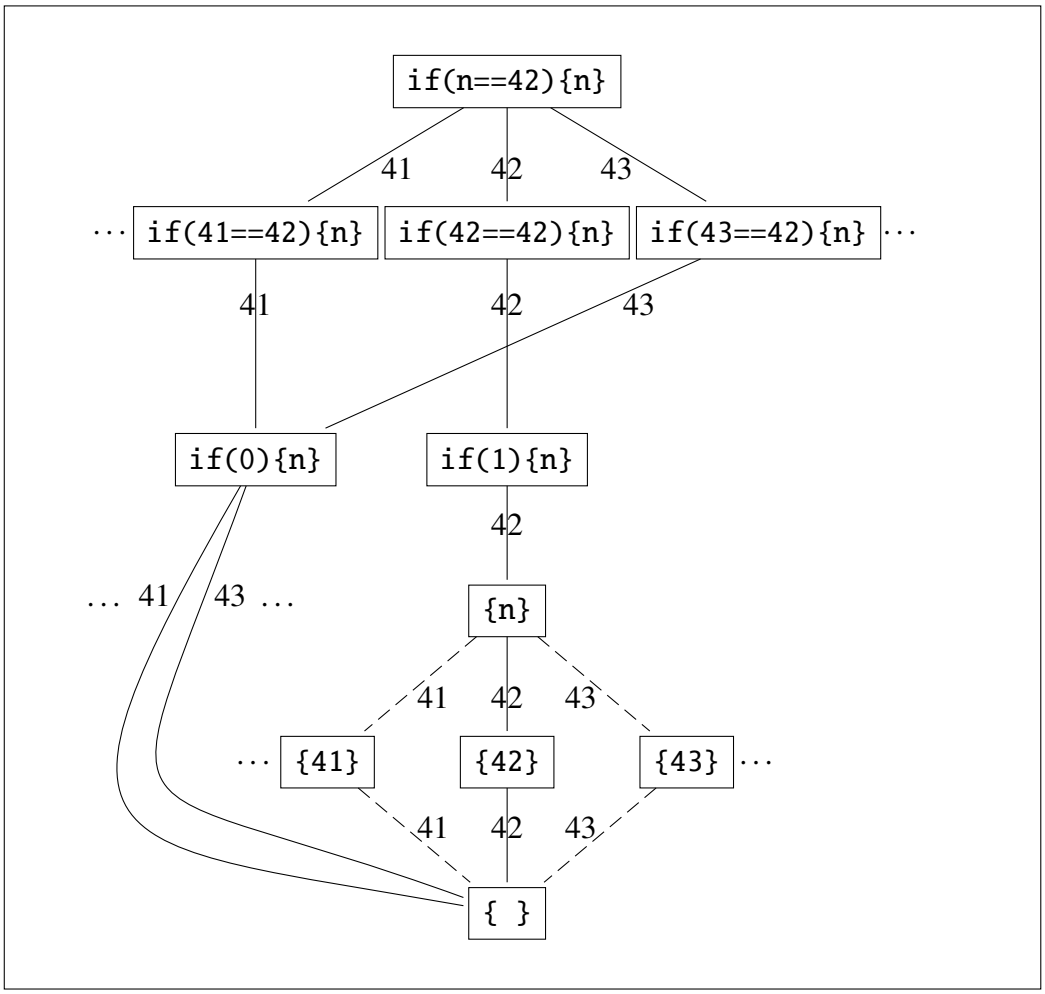

Figure 2.2: Part of a labelled transition system for bc statement execution

▷ Unrestricted labels do not adequately reflect the different ways that information can *flow* during program executions.

Suppose that an identifier is a constant, so its value remains fixed during a computation. Sequences of transitions where the labels on successive transitions indicate that the identifier has changed its value should then be disallowed as computations. On the other hand, if the identifier is a variable, and can be updated (e.g., by assignment statements, as in bc) such a sequence should be allowed. How can we resolve this dilemma?

The solution in the conventional SOS framework is to represent information such as the current values of identifiers in the *states* of transition systems, rather than in the labels on transitions. (Labels are used in conventional SOS only to rep-

resent information processing typical of concurrent programs: the transmission of messages and synchronization signals.) The state-based technique is feasible, and certainly 'works' when describing the semantics of any particular language; but explicitly indicating the extra components of states severely undermines modularity of SOS descriptions.

▷ An *info-labelled* transition system requires labels on adjacent transitions in computations to be *composable*.

In MSOS, we shall exploit labels to model *all* aspects of information processing, and avoid adding extra components to states. This requires sufficient structure in labels to distinguish between different kinds of information flow, e.g., between identifiers having fixed or changeable values. The crucial feature of the extra structure (which will be explained in detail in Section 2.1.2) is that it determines when labels are *composable*. Modifying the definition of computations in labelled transition systems to require that labels on adjacent transitions are always composable gives us appropriate models for MSOS specifications.

Definition 2.5 (info-labelled transition system, composability) *An info-labelled transition system ILTS is an LTS together with a binary relation Composable on Label. We write Composable*(L_1, L_2) *to indicate that* L_1, L_2 *are composable, i.e.,* $(L_1, L_2) \in$ *Composable.*

A computation *in an info-labelled transition system is a computation of the underlying LTS such that for all adjacent transitions* $S_i \xrightarrow{L_i} S_{i+1} \xrightarrow{L_{i+1}} S_{i+2}$ *occurring in the sequence, Composable*(L_i, L_{i+1}) *holds.*

Notice that taking Composable to be the universal relation Label $\times$ Label gives an ILTS whose computations are defined just as in an ordinary LTS. The possibility of restricting Composable is however crucial when labels are used to model arbitrary kinds of information processing, as illustrated by the following examples:

Example 2.6 Continuing Example 2.4, suppose that labels are composable only when they are identical. Then in any particular computation, all the labels on the transitions are the same. This ensures that different occurrences of the same identifier in a computation always have the same value, i.e., it represents a constant.

Example 2.7 To model imperative programs where the values of identifiers generally do change during each computation, let each label give both the current and the (possibly) new values of identifiers. Let any two labels L_1, L_2 be composable only when the new values of identifiers according to L_1 are the same as the current values according to L_2. This ensures that each identifier keeps its value except when some transition requires it to change, i.e., it represents a variable.

Example 2.8 To model reactive programs that may emit values as messages[3] during the course of a computation, let each label be a possibly-empty sequence of values, with all labels being composable (since the value(s) emitted during one transition do not *a priori* restrict the values that may be emitted in the following transitions). The values emitted by an entire computation can be obtained by concatenating the sequences labelling the individual transitions; this may even give an infinite sequence of values when the computation doesn't terminate.

▷ The states of info-labelled transition systems model flow of *control*, whereas the labels model flow of *information*.

We shall consider these two ingredients of our operational models in more detail in the next two sections. In particular, we shall introduce a particular structure for labels such that we can see immediately whether they are composable or not.

2.1.2 Modelling Flow of Control

▷ The abstract syntax of a program gives the initial control state for its computations

In MSOS, as suggested by the examples considered in the preceding section, we use the syntax of programming constructs to keep track of the flow of control through a program during the course of its computations. However, we use *abstract* syntax, rather than concrete syntax, since it is usually much simpler, and free of questions concerning grouping and precedence. The initial state of a computation is simply the abstract syntax tree of the entire program; subsequent states represent what remains to be computed, together with any previously-computed values that may still be needed.

▷ Nodes of abstract syntax trees often get replaced by simpler trees, and ultimately by their computed values, during computations.

The transitions of a computation generally correspond to small local changes in the abstract syntax tree: a constant or variable in an expression gets replaced by its current value; an application of an operation to two computed values gets replaced by the result of the application; after the condition of a one-armed conditional command has been replaced by its value, the entire construct gets replaced either by its sub-command or by the skip command.

[3] Printing a value is a special case of emitting a message.

A computation may require more than one of the sub-trees of a node to be replaced by computed values before the node itself can be replaced. An simple assignment command is a good example of this: after the variable has computed the cell where the value of the expression is to be stored, the cell is kept at the node while the value of the expression is computed. The simplicity of this way of combining control flow with recording computed values is one of the many strengths of the structural approach to operational semantics.

▷ Nodes may also get replaced by larger trees.

Replacing a node by a computed value or by a sub-tree always reduces the size (i.e. number of nodes) of the tree. If those were the only possibilities, all computations would have to terminate, since the initial abstract syntax tree has only a finite number of nodes. To model iteration and recursion, it is necessary to allow nodes to be replaced by larger trees.

For instance, a transition might replace a node for a while-command `while-nz(E,C)` by `cond-nz(E,seq(C,while-nz(E,C)))`. Notice that the sub-trees `E` and `C` hereby get duplicated. The immediate duplication of `E` is clearly necessary, since computing the value of `E` replaces that sub-tree by a value, and it is that value which determines whether the original `E` will need re-evaluating. In contrast, the duplication of `C` could be delayed until after the evaluation of `E`, if desired. Ex.2.5

▷ The type State contains all the control states, whereas Final includes just the computed values.

Transition systems for so-called *big-step* MSOS have no intermediate states: initial states are abstract syntax trees, final states are computed values, and computations have at most one transition. Such transition systems are appropriate for static semantics, and for the dynamic semantics of constructs that *inherently* never involve infinite computations. Ex.2.6

Transition systems for unrestricted MSOS have intermediate states that generally include *mixtures* of the original abstract syntax trees with computed values. These mixtures are formed by (gradually) replacing one or more sub-trees of an abstract syntax tree by the values that they have computed. However, recall from Chapter 1 that in MSOS we already allow the initial abstract syntax trees to include abstract mathematical values such as integers and arithmetic operations; those trees which arise as intermediate states may involve different kinds of values, but they are of exactly the same nature as the initial ones.[4]

[4] If an abstract syntax tree is used as a computed value and inserted in another tree, the resulting tree is the same as if the first tree occurred originally as a sub-tree of the second tree.

So much for the nature of the abstract syntax trees and computed values to be used as the states of transition systems. Let us next consider the nature of the labels on transitions.

2.1.3 Modelling Flow of Information

▷ The label on a transition between two control states records *all the information* processed by that step of the computation.

The information recorded in the labels is classified according to how it flows through sequences of transitions, as follows:

- information that remains *fixed*;
- information that can be *changed*, but otherwise remains stable; and
- new information that may be *produced*.

This classification turns out to be particularly appropriate. By distinguishing between the different ways that the information actually flows, our modelling will be more accurate.

Ex.2.7

▷ Components of the processed information are referred to by *symbolic indices*.

The processed information may have several independent components, referred to using symbolic indices i. These indices allow us to refer to one component of a label without mentioning its other components at all, which is crucial in connection with modularity. The main alternative to using symbolic indices for referring to the components would be to rely on positional notation, which requires at least mentioning how many other components precede the intended component, and is inherently non-modular.

▷ *Primes* on indices are significant.

For changeable information, we need separate components for the information current at the start and end of a transition, and we use *primes* on indices to distinguish between them: the unprimed index refers to the start of the transition, and the primed index refers to the end. (This use of primes is quite conventional, and exploited in various frameworks for reasoning about transitions.) Since new information produced by a transition is not available at the start the transition, we use only primed indices for referring to such components.

Thus if L is a label on a transition, and i is the index of a component:

- $L.i$ refers to component i of the fixed or changeable information at the *start* of the transition, and
- $L.i'$ refers to component i of the changeable or produced information at the *end* of the transition.

▷ The notation used for referring to a component determines how that part of the processed information may flow.

A component that is only referred to by $L.i$ is fixed information; if it is only referred to by $L.i'$, it is produced information; and if it is referred to both by $L.i$ and $L.i'$, it is changeable information.

▷ Labels are analogous to records in ML.

An entire label L is determined by giving the values of all its components. For a changeable component indexed by i, the values of both $L.i$ and $L.i'$ have to be given. Thus labels correspond closely to the record values provided in various programming languages,[5] with fields of the form $i{=}v$ and/or $i'{=}v$. In fact the notation used for specifying labels in MSOS (as already illustrated in Chapter 1) is very closely related to ML syntax for record types and values.

▷ Components of produced information are assumed to be *sequences*.

In general, we make no assumptions about the abstract entities used to represent components of information (although in practice, fixed information will usually be represented by environments giving bindings for identifiers, and changeable information by stores giving the contents of cells). However, we may occasionally need to *compose* the labels of transitions. For fixed and changeable information, the component(s) of the composition of L_1 and L_2 are simply *selected* from the components of L_1 and L_2; but for produced information, we need some way to *combine* two components to form a new component.

Here we shall simply assume that each component of produced information is a *sequence*, and use concatenation to combine sequences. Note that a single value is regarded as a sequence of length 1. The absence of new produced information is conveniently represented by the empty sequence, $()$. Ex.2.8

[5] Our indices are called labels in ML.

▷ The flow of the information recorded in labels determines their *composability*.

Definition 2.9 (composability) *Composable*(L_1, L_2) *holds iff* L_1, L_2 *have the same set of indices, and for each index* i*:*

- *if* i *indexes* fixed *information, then* $L_1.i = L_2.i$*, and*
- *if* i *indexes* changeable *information, then* $L_1.i' = L_2.i$*.*

(Produced information doesn't affect composability of labels.)

Ex.2.9

▷ Labels are regarded as *unobservable* when changeable information doesn't change, and no new information is produced.

The distinction between observable and unobservable labels will allow us to specify that a transition doesn't change or produce information, merely by requiring it to be unobservable, *without* referring explicitly to its components. This is just as crucial as the use of symbolic indices for achieving modularity in MSOS.

▷ The flow of the information recorded in labels determines their observability.

Let Unobs be the set of unobservable labels. It is defined as follows:

Definition 2.10 (unobservability) $L \in$ *Unobs iff for each index* i*:*

- *if* i *indexes* changeable *information, then* $L.i = L.i'$*; and*
- *if* i *indexes* produced *information, then* $L.i' = (\,)$*, the empty sequence.*

(Components of fixed *information don't affect observability of labels.)*

That concludes the explanation of the general structure of our operational models. Let us now proceed to introduce an efficient formal notation for defining particular models.

2.2 Operational Specifications

▷ Specification of an operational model of a programming language in MSOS involves defining sets of *states*, the *components of labels*, and an info-labelled *transition relation*.

As explained in the foregoing section, an operational model in MSOS is an info-labelled transition system. Let us now introduce *MSDF* (Modular SOS Definition Formalism), the notation that we shall use to define such models. Here, we shall focus on its general form and mathematical interpretation; guidance in the practical use of MSDF will be provided in Chapter 3. The illustrations given in this section will generally be fragments of the specifications given in Chapter 1.

▷ Sets of states are defined by declaring their *constructors* and *subsets*.

The states of the transition system are the abstract syntax trees of programs (and of their various sub-constructs) and computed values. To specify them, we need to declare the sets and constructors of the abstract syntax and computed values, and include the relevant sets in State and Final.

▷ The set of labels is defined by specifying sets of *extensible* records.

The labels on transitions are always records. Here, for each index, we simply have to define the set of values for the corresponding component(s). The priming of the indices determines the kind of information flow involved, and defines the Composable relation (which is needed to obtain an ILTS) and the subset Unobs of unobservable labels (which is needed in connection with specifying the transition relation). We shall ensure that labels can always be extended with new components merely by adding further definitions, leaving the original definitions unchanged.

▷ The transition relation is defined by a set of rules for *deriving* transitions.

Our main task will be to specify the transition relation. We have to specify transitions not only for complete programs, but also for all the sub-constructs from which they are formed. Transitions for primitive constructs are specified directly; transitions for compound constructs generally depend on transitions for one (sometimes more) of their components, and are specified using conditional rules (which may also be regarded as rules of inference in a formal proof system). The set of rules specifying all the transitions is a so-called *inductive definition* of the transition relation.

▷ The form and intended interpretation of MSDF should be mostly quite obvious.

The reader is advised to skim the rest of this chapter, so as to get a broad impression of MSDF. The less obvious bits of notation (e.g., for labels) will be explained when they are used in the examples in subsequent chapters.

2.2.1 Specification of Sets and Constructors

These specifications resemble datatype definitions in ML, and so-called free specifications of datatypes in CASL. Let *set* range over terms in MSDF that denote sets of values, and *setid* over set-identifiers.

▷ MSDF includes notation for some familiar sets and set-constructors.

Several identifiers have a *fixed* interpretation as sets and set-constructors. They can be freely used when defining other sets, without explicit declaration. These sets are accompanied by pre-defined notation for their elements, but let us defer the details until we consider terms more generally, later in this chapter.

Boolean is the set consisting of the usual boolean truth-values.

Integer is the set of all integers.

set+ is the set of non-empty, finite, non-nested sequences of elements in *set*. A sequence of length 1 isn't distinguished from its only element, so *set*+ includes *set* as a subset,

*set** is the set of possibly-empty, finite, non-nested sequences of elements in *set*. It includes both *set*+ and *set*? as subsets.

set? is the set consisting of the empty sequence and the elements of *set* (i.e., sequences of length 0 or 1).

(*set*)List is the set of possibly-empty, finite lists of elements in *set*. A list of length 1 is distinguished from its only element, so (*set*)List does *not* include *set* as a subset.

(*set*)Set is the set of (values representing) possibly-empty, finite sets of elements in *set*. (To keep a clear distinction between the two levels of sets involved here, we could refer to a set such as (Integer)Set as a meta-set. However, our meta-sets will generally have infinitely-many elements, so confusion seems unlikely to arise.)

(*set1*,*set2*)Map is the set of finite mappings from *set1* to *set2*.

▷ MSDF reserves identifiers for the set of environments and for other commonly-needed sets.

Some sets play a fixed rôle in MSOS, and MSDF provides identifiers for them. Sets that are left open can be defined by including other sets in them; others are defined in a fixed way in terms of such open sets, and cannot themselves be extended by subset inclusion.

`Id` is an open set of identifiers.

`Bindable` is an open set of values that can be bound to identifiers.

`Env` is the set `(Id,Bindable)Map`.

`Cell` is an open set of storage cells (also known as 'locations' or 'addresses').

`Storable` is an open set of values that can be stored in (single) cells..

`Store` is the set `(Cell,Storable)Map`.

`State`, **`Final`**, **`Label`**, and **`Unobs`** are reserved as identifiers for the corresponding sets that define an info-labelled transition system.

▷ Unreserved set identifiers can be declared, and defined by declaring their constructors and subsets, or by equating them to other sets.

MSDF allows identifiers for new sets to be declared. The elements can be defined by declaring constructors and/or including other sets in them. They can also be defined as abbreviations for existing sets. Let *`setid`* range over all set identifiers.

`setid` declares a set identifier. Such identifiers start with a capital letter, and may contain hyphens as well as letters and digits.

Example: `Cmd`.

`setid``::=`*`opid`* declares *`opid`* as a constructor constant of *`setid`*. Constructor identifiers start with a lowercase letter, and may contain hyphens as well as letters.

Example: `Cmd::=skip`.

`setid``::=`*`opid`*`(`*`set1`*`,···,`*`setn`*`)` declares *`opid`* as an n-ary constructor function for *`setid`* values with arguments in *`set1`*, ..., *`setn`*.

Example: `Cmd::=cond-nz(Exp,Cmd)`.

setid`::=`*set* declares *setid* to include *set* as a subset.

Example: `Exp::=Boolean`.

setid`=`*set* declares *setid* and defines it as an abbreviation for *set*.

Example: `Env=(Id,Bindable)Map`.

`Label={`*opid1*`:`*set1*`,···,`*opidn*`:`*setn*`,...}** declares each *opid* (which may be followed by a prime `'`) as a label index for a component of information with values in the corresponding *set*. The set doesn't need repeating when specifying the same index both primed and unprimed. N.B. The ordinary dots '`...`' are a part of the notation. They indicate that further components can still be added.

Example: `Label = {env:Env,store,store':S,out':Value*,...}`.

The declarations of a set identifier, its constructors, and its subsets can be specified together (as in CASL).

Example: `Exp::=null|Boolean|app(Op,Arg)`.

▷ MSDF enforces consistent use of variable identifiers.

Variable identifiers (used when specifying rules for transitions) have to be declared *globally* as ranging over particular sets. Let *varid* range over variable identifiers.

varid`:`*set* declares that *varid*, and all variables formed from *varid* by adding digits and/or primes (in that order), are restricted to range over *set*. A *varid* is itself formed from uppercase letters only. The variable declarations `X:Label` and `U:Unobs` are implicit in MSDF.

Example: `E:Exp` declares the variables `E`, `E'`, `E23''`, etc.

The declaration of a variable identifier can be combined with declarations for the set over which it ranges.

Example: `C: Cmd`.

Example: `E: Exp::=null|Boolean|app(Op,Arg)`.

Example: `S: Store=(Cell,Storable)Map`.

▷ The combined declarations in a complete MSDF specification define all the elements of all the declared sets.

The elements of a declared set consist of all values that can be constructed using its declared constructors (possibly with values of the sets with fixed interpretations as arguments) and values of all sets that are declared to be included in it. Sets that have no constructors or subsets are thus defined to be empty.

The order of declarations in MSDF is insignificant, and repetitions of declarations are ignored.

2.2.2 Specification of Transitions

▷ Transitions are specified by simple and conditional *rules*.

Let *term* range over terms that denote individual values, and let *form* range over formulae (see below).

▷ Transitions are written with the label in the middle of the arrow.

term1`--`*term2*`-->`*term3* is an unconditional rule for transitions from state *term1* with label *term2* to state *term3*; similarly for transitions written *term1*`==`*term2*`=>`*term3*.

Example: `print(V):Cmd --{out'=V,---}-> skip`.

form1`,···,`*formn* `----` *term1*`--`*term2*`-->`*term3* is a transition rule with conditions *form1*, ..., *formn*; similarly for transitions written *term1*`==`*term2*`=>`*term3*. (MSDF requires at least 4 dashes to separate the conditions from the conclusion of the rule.)

Example: `E --{...}-> E' ---- print(E):Cmd --{...}-> print(E')`.

Usually (to enhance readability) the '`----`' separating the condition(s) from the conclusion of the rule is written on a line by itself, and extended with further dashes:

```
            E --{...}-> E'
  --------------------------------
  print(E):Cmd --{...}-> print(E')
```

All variables occurring in rules that specify transitions must be declared globally to range over particular sets. Transitions can be derived by assigning values from the specified sets to all their variables, provided that the values of the terms denoting their states and labels are defined, and all their conditions hold.

Definition 2.11 (structural rules) *A rule is called* structural *when every term that occurs on the left of a transition in a condition is a subterm of the term on the left of the transition in the conclusion of the rule.*

In practice, rules in MSOS are almost always structural, and the terms on the left of transitions in the conditions are usually just variables.

▷ Conditions of rules can be transitions or side-conditions.

Formulae other than transitions are known as *side-conditions*, since they do not involve the relation that is being defined inductively. For notational convenience, however, we shall write all conditions in a single (comma-separated) list.

Formulae

Formulae *form* used as conditions in rules are interpreted as follows:

term1`--`*term2*`-->`*term3* holds when the indicated transition can be derived; similarly for transitions written *term1*`==`*term2*`=>`*term3*. In MSOS, we shall generally reserve the latter notation for use in static semantics.

Example: `E --{...}-> E'`.

term1`--->`*term3* abbreviates *term1*`--U->`*term3*, i.e., the label is unobservable, and identified by the variable `U`. Similarly, *term1*`===>`*term3* abbreviates *term1*`==U=>`*term3*.

Example: `E ===> VT`.[6]

term1`=`*term2* holds when the values of *term1* and *term2* are defined and equal.

Example: `lookup(CL,S) = V`.

`def` *term* holds when the value of *term* is defined.

Example: `def lookup(CL,S)`.

`not` *form* holds when *form* does not hold, and vice versa.

Example: `not def lookup(CL,S)`.

term as a formula abbreviates *term*`=true` (when *term* is boolean-valued).

Example: `N \= 0` (where '`\=`' is defined as a boolean-valued operation for all sets, and '`0`' is the zero integer).

[6] Unobservable (big-step) transitions are typical of static semantics, and do not often occur as conditions in dynamic semantics.

Terms

▷ Apart from some special notation for labels, the notation for terms is quite conventional.

var gives the value assigned to *var*.

Example: `E1'`.

opid gives the value of a constructor or of a fixed constant.

Example: `skip`.

opid`(`*term1*`,···,`*termn*`)` gives the value (if defined) of applying a constructor or fixed operation to the values of *term1*, ..., *termn*. If the value of *any* of the *term*s is undefined, so is the value of the application.

Example: `cond-nz(E,C)`.

opid may be also a variable ranging over a specified set of operations. Example: `O(V1,V2)`.

`{`*opid1*`=`*term1*`,···,`*opidn*`=`*termn*`,`*term*`}` gives the label where the components indexed by the *opid*s have the values of the corresponding *term*s, and all other components are determined by the label given by the value of the last *term*. Each *opid*s may be followed by a prime.

Example: `{store=S,store'=S',U}`.

The last *term* may also be '`...`',[7] which is equivalent to the variable `X`, or '`---`', which is equivalent to the variable `U`; in these cases, *n* may be 0.

Example: `{...}`.

Example: `{store=S,store'=S',---}`.

term`.`*opid* gives the *opid* component of the label given by the *term*. The *opid* may be followed by a prime.

Example: `X.store'`.

term`:`*set* restricts the values of *term* to be elements of *set*.

Example: `N:Exp`.

[7] The '`...`' notation used in label terms is similar to that used in record patterns in ML, but note that in MSDF, different occurrences of '`...`' in the same rule always stand for the *same* label components.

Fixed Notation

▷ Fixed notation is provided for expressing values in the sets that have fixed interpretations.

The fixed notation for values and operations on them is mostly borrowed from ML and Prolog. Here is a brief overview of it:

Boolean Values: **true**, **false**. Unary: **not**. Binary: \/, /\, =, \= (equality and inequality are defined on all sets).

Integer Values: decimal numerals. Unary: -. Binary: +, -, *, **div**, **mod**, **rem**, <, =<, >, >=.

set+, *set**, *set*? Values: (*term1*,⋯,*termn*). Binary: ^ (concatenate).

(*set*)**List** Values: [*term1*,⋯,*termn*]. Unary: **first**, **rest**, **front**, **last**, **length**. Binary: + (concatenate), **nth**, **take**, **drop** (list, integer).

(*set*)**Set** Values: {*term1*,⋯,*termn*} ($n \geq 1$), **empty**. Unary: **size**. Binary: + (ordinary union), - (ordinary difference), * (intersection), **in** (element, set), <, =<, >, >=.

(*set1*,*set2*)**Map** Values: {*term11*|->*term12*,⋯, *termn1*|->*termn2*} ($n \geq 1$), **void**. Unary: **dom** (domain), **ran** (range). Binary: + (disjoint domains), - (domain restriction), / (first map overrides second), **lookup** (element, map).

▷ Fixed *precedence rules* allow omission of some parentheses.

Grouping in MSDF terms is independent of whether the operation identifiers involved have been declared as unary or binary, and applications of all binary operations can be written both prefix and infix.

- Iterated prefix application requires parentheses.

  Example: `a b c` is grouped as `(a)b(c)`; omitting parentheses in `a(b c)` changes the grouping.

- Multiple infixes *always* require explicit grouping parentheses.

  Example: The parentheses cannot be omitted in `(3-2)-1` or `3-(2-1)`.

- Mixtures of prefixes and infixes require explicit grouping parentheses.

  Example: The parentheses cannot be omitted in `a b (c d)` or `(a b) c d`.

- Infixes can be used in ordinary applications.

  Example: `+(N1,N2)`, `lookup(I,ENV)`.

2.2.3 Modular Specifications

▷ MSDF specifications can easily be split into modules.

msdf1`. ... .` *msdfn* forms a module from MSDF declarations, rules, and references to other modules. The items have to be separated by blank lines and/or periods. All notation used in the module has to be declared. Declarations that involve sets give implicit references to the modules that declare those sets.

Example: `Exp::=app(Op,Arg)` gives implicit references to the modules `Exp`, `Op`, and `Arg`, which declare the corresponding set identifiers.

see *path* refers to the module identified by *path*, relative to the local module repository. A module is regarding as including everything that the referenced module includes.

Example: `see Cmd/cond-nz`.

A collection of modules is interpreted as the *least* interpretation of the declared notation that satisfies all the declarations included in and by the modules, together with the *least* info-labelled transition relation that is closed under all the specified rules.

Ex.2.10

Ex.2.11

Ex.2.12

Summary

In this chapter:

- We introduced operational models for MSOS in the form of info-labelled transition systems. The modelling of flow of control (by states consisting of abstract syntax trees) is kept separate from the modelling of the flow of information (by labels, which are records with independent components).
- We explained the mathematical interpretation of MSDF, a formal notation for defining sets of states, components of labels, and info-labelled transition relations.

In the next chapter, we shall explain how MSDF should be written so as to allow its translation to Prolog in connection with modular prototyping. Numerous examples of operational specifications in MSDF will be given in the following chapters.

Exercises

Ex. 2.1 Programming languages generally have infinitely-many programs.

(a) What properties of the syntax of a programming language are required for it to have infinitely-many programs?

(b) Do you know of any (useful) programming language that has only a finite number of programs?

(c) Might having only a finite number of programs make a language easier to describe, either regarding syntax or semantics?

Ex. 2.2 Consider the sets of initial, final, intermediate, unreachable, and stuck states. Draw a diagram to show which of these sets can overlap (i.e., have states in common).

Ex. 2.3 Consider the part of a transition system for bc statement execution shown in Figure 2.1. Classify the depicted states as initial, final, or intermediate.

Ex. 2.4 Consider the part of a labelled transition system for bc statement execution shown in Figure 2.2.

(a) Identify a pair of states such that there is more than one transition between them.

(b) Is it at all useful to distinguish between the different transitions here? If so, explain why; if not, propose an example where it would be useful.

Ex. 2.5 As explained on page 41, a transition might replace a node for a (boolean) while-command `while-nz(E,C)` by `cond-nz(E,seq(C,while-nz(E,C)))`. Consider a construct `c(E,C)` such that the transition could instead replace `while-nz(E,C)` by `cond-nz(E,c(E,C))`, and subsequently replace `c(E,C)` by `seq(C,while-nz(E,C))`, if needed.

(a) To which concrete programming construct does `c(E,C)` correspond?

(b) What are the pros and cons of introducing `c(E,C)` in connection with the semantics of `while-nz(E,C)`

Ex. 2.6 Why might it be necessary for a big-step MSOS to allow computations that have no transitions? Hint: consider the same question for small-step MSOS.

Ex. 2.7 Give some examples of programming constructs whose execution involves more than one kind of information flow.

Ex. 2.8 When produced information is a sequence of elements, it is combined by concatenation. Under what restrictions on the set of elements and the length of sequences could we replace concatenation by selection of one of its arguments?

Ex. 2.9

(a) Define a (partial) binary composition operation on labels, in a style similar to that of Definition 2.9.

(b) Check whether composition is associative (when defined).

(c) Check whether composition has left and right units, and whether these are unobservable.

***Ex. 2.10** Some of the tables given in Chapter 1 are in MSDF. Read through each of them, observing the use of MSDF, and considering its mathematical interpretation. Make a note of any notation used in the tables whose interpretation seems unclear.

***Ex. 2.11** MSDF provides fixed notation for various familiar sets of values: booleans, integers, lists, etc. For each of these sets, give some equations that you expect to hold universally (i.e., for all values of the variables that occur in them). In connection with partial operations, consider also whether each side of the equation is always defined (conditions may be required to ensure definedness).

***Ex. 2.12** For those who are familiar with CASL:

(a) List the main differences between MSDF specifications and basic specifications in CASL, both regarding notation and its formal interpretation.

(b) Reformulate each of the MSDF tables from Chapter 1 in CASL, and discuss the pros and cons of MSDF and CASL in the context of these examples.

Chapter 3

Modular Prototyping

The previous chapter explained the nature of the labelled transition systems that we use as operational models in MSOS, and the mathematical interpretation of MSDF, a formal notation for specifying the such models. This chapter gives guidance on how to write specifications in MSDF so that they can be translated to Prolog, using the software accompanying these notes.

▷ Prototyping here is mainly for *validation* of language descriptions.

The Prolog generated from a collection of modules forming a complete MSOS of a programming language, together with a DCG for parsing programs and mapping them to abstract syntax, constitutes a prototype implementation of the described language, and can be used to run programs. The main purpose of the prototype implementation is to *validate* the MSOS from which it was generated. Validation involves checking not only that the MSOS is complete, but also that it specifies the *intended* semantics for programs.

Typically—thanks to the use of the structural style of operational semantics—rather few tests are needed to discover any mistakes that have been made in an MSOS or the accompanying DCG. Moreover, once an MSOS module has been properly validated in connection with the semantics of one programming language, it is unlikely to be a source of mistakes when it is reused in connection with the description of another language.

A non-modular version of the abstract syntax, static semantics, and dynamic semantics of a subset of bc was shown in Tables 1.2 and 1.4–1.9. The basic idea of modularization was explained in Section 1.2.9, and a few reusable MSOS modules were illustrated in Tables 1.10–1.15. This chapter explains the way that MSOS modules are organized so as to maximize their reusability, with reference to a completely modularized version of the MSOS given in Chapter 1. All the relevant files are available on the Internet at `http://www.cs.swan.ac.uk/~cspdm/MSOS`,

and can be downloaded as a single zip file from `http://www.cs.swan.ac.uk/~cspdm/MSOS.zip`. *All references to file names in the rest of these notes are relative to this MSOS directory.*

Ex.3.1

3.1 Individual Constructs

▷ Abstract syntax is specified separately from semantics.

As explained in Section 1.2.9, the MSOS of each individual language construct is divided into three parts, describing its abstract syntax, static semantics, and dynamic semantics. The same goes for the auxiliary modules for the various sets of constructs. The static and dynamic semantics modules (implicitly) refer to the abstract syntax modules, and the module for an individual construct (implicitly) refers to the modules for all the sets of constructs involved in it.

Let us now consider each part in turn, explaining and illustrating its general form. Subsequent chapters motivate the choice of individual constructs, and discuss details of their semantic descriptions.

3.1.1 Abstract Syntax

▷ Sets of constructs are distinguished only with regard to fundamental conceptual and semantic differences.

The abstract syntax part of the MSOS of `Cmd` is stored at `Cons/Cmd/ABS`.[1] It consists simply of the declaration `C:Cmd`. Similarly for all the other sets of constructs.

The number of sets of constructs is kept low by avoiding inessential (and potentially context-sensitive) distinctions. For instance, we shall *not* distinguish sets of boolean and numerical expressions, but rather consider all their constructs together in a more general set of expressions; this avoids having to worry about how to classify an identifier whose value might be either boolean or numerical, depending on context. In contrast, the distinction between commands and expressions is quite clear conceptually, and evident in context-free syntax.

Ex.3.2

▷ Basic constructs and variants are named systematically.

The abstract syntax part of the MSOS of the construct `cond-nz` is stored at `Cons/Cmd/cond-nz/ABS`. It consists of the single constructor declaration `Cmd::= cond-nz(Exp,Cmd)`. Similarly for all the other constructs.

[1] All MSOS files are written in MSDF, and have the extension '`msdf`'.

The names of variants of a construct are typically formed by adding a suffix to a fixed root, with the name of the basic form of the construct having no suffix. For instance, we shall regard a conditional command with a boolean-valued condition as the primary form, naming its constructor simply 'cond'.

Variants corresponding to alternative numbers or sets of arguments keep the name of the primary form. For instance, we shall use the name 'cond' also for binary conditional choice between commands, and 'app' for application when the operation is replaced by an identifier, or, more generally, by an arbitrary expression. (The module names then have to be disambiguated by suffixes.)

The arguments of a constructor (i.e., the components of a construct) can themselves be either constructs or elements of data, such as the familiar abstract mathematical integers, or even abstract operations like addition.

▷ Subset inclusion corresponds to a construct.

An important special case of a basic construct is when one set of constructs (or a set of data elements) is simply included as a subset of another. For instance, the set Var of variables is included in the set Exp of expressions, as is the set Integer. (We may also regard subset inclusion as an 'invisible constructor'.) In this case we use the name of the included set when forming the module name, e.g., the abstract syntax part of the MSOS of integers used as expressions is stored at Cons/Exp/Integer/ABS.

▷ Variants should be avoided when they can be derived as simple combinations of other constructs.

In fact the conditional command with a single subcommand cond-nz(E,C) can be derived as cond-nz(E,C,skip), and the only reason for introducing it in Chapter 1 was to make our initial examples of MSOS as simple as possible. When we consider conditional commands with boolean conditions in a later chapter, we shall introduce only the construct that chooses between alternative subcommands, and avoid the single-branch variant. Ex.3.3

3.1.2 Static Semantics

▷ The static semantics of a set of constructs specifies the relevant sets of states and of final states.

The static semantics part of the MSOS of Cmd is stored in Cons/Cmd/CHK. It specifies the inclusion of Cmd in State, and the inclusion of the void type in Final. Static semantics involves only big-step computations: no intermediate states are needed. Similarly for the other sets of constructs, except that they generally involve more interesting sets of types.

▷ Each set of computed values in dynamic semantics has a corresponding set of types for use in static semantics.

The type computed by a construct in static semantics represents the set of possible values that might be computed by the same construct in dynamic semantics. The type `void` represents the empty set of values, since we regard commands as computing a fixed value, which provides no computed information at all.

▷ The static semantics of an individual construct represents its well-formedness.

The static semantics part of the MSOS of the construct `cond-nz` is stored at `Cons/Cmd/cond-nz/CHK`. A construct is well-formed when all its components are well-formed, and their types consistent. Thus the static semantics of a construct involves structural transition rules with a condition for each component. Checking consistency can usually be expressed by use of patterns (i.e., terms formed from constructors and variables) as the target states of transitions. For `cond-nz(E,C)`, the type of `E` is required to be `int`, representing the set of integers. For a more interesting example, consider the construct `app(O,ARG)`: the types of the components are consistent when the type of `O` is of the form `func(PVT,VT)`, and the type of `ARG` is `PVT`.

▷ The order of the conditions may need adjusting to reflect the intended flow of information between the components of a construct.

Mathematically, the order of the conditions in a transition rule makes no difference at all. When the rule is translated to Prolog, however, the conditions are checked from left to right. For the simple semantics of bc, there should be no problems with that; but we shall see in Chapter 10 that the order of the conditions in the rule for `app` requires more careful consideration when the first component can be an arbitrary expression, rather than just an operation.

▷ The static semantics of an individual construct declares all its own requirements.

When several similar constructs have identical requirements, excessive duplication can sometimes be avoided by specifying those requirements in a separate auxiliary module. Requirements on components of labels are however best stated explicitly each time, for perspicuity.

▷ A separate transition relation is specified for each set of constructs.

In a rule, the set of constructs to which the source state belongs is self-evident when the source term is simply a variable. In all other cases, it has to be specified explicitly, by using a term restricted to a set, e.g., `seq(C1,C2):Cmd`. This requirement avoids a possible confusion that might arise when specifying rules for subset inclusions such as `Exp::=Var`.

3.1.3 Dynamic Semantics

▷ The dynamic semantics of a set of constructs specifies the relevant sets of states and of final states.

The dynamic semantics part of the MSOS of `Cmd` is stored in `Cons/Cmd/RUN`, and similarly for the other sets of constructs. It specifies the inclusion of `Cmd` in `State`, and the inclusion of its set of computed values in `Final`. Dynamic semantics generally involves small-step computations: intermediate states are needed. These intermediate states may involve abstract syntax trees where some branches have been replaced by their computed values.

When the computed values for a particular set of constructs were not originally included in the specification of abstract syntax, they need to be added. For example, `Cmd` already includes `skip`, and `Exp` includes (the elements of) `Value`, so the required intermediate states for these constructs are already included in the abstract syntax. But `Var` doesn't initially include `Cell`, so `Var::=Cell` has to be added, and similarly for `Arg` and `Passable`.

▷ The dynamic semantics of an individual construct represents its run-time computation of values.

The dynamic semantics part of the MSOS of the construct `cond-nz` is stored at `Cons/Cmd/cond-nz/RUN`. In contrast to static semantics, the dynamic semantics of a construct doesn't always involve that of all its components. A particular component may simply get ignored, depending perhaps on the current information, or on the values computed by one (or more) of the other components. This is quite obvious for `cond-nz(E,C)`, where `C` is ignored when the value computed by `E` is zero. Less obviously, in `seq(C1,C2)`, `C2` may get ignored, depending on the semantics of `C1`. Can you see how that can happen? Ex.3.4

▷ Dynamic semantics involves several rules for each construct.

The small-step nature of dynamic semantics generally requires specifying several rules for each construct. This is in marked contrast to the big-step semantics used for static semantics, where there is just one rule for each construct, with a condition for each component. Each rule for dynamic semantics usually has a transition for only one of its components as a condition. This reflects the basic idea that each step in a computation is due to progress made at just one place in the abstract syntax tree that represents control flow. Such a step gives rise to a corresponding step for each construct which encloses that place—and ultimately to a corresponding step for the entire abstract syntax tree representing what remains to be executed of the original program.

Ex.3.5

▷ The rules for a construct may be non-overlapping.

The rules for a construct are said to be non-overlapping when their use to derive transitions is mutually exclusive. For instance, consider the two rules for `seq(C1,C2)` given in Table 1.12: the first rule can be used only where `C1` has some transition, whereas the second rule can be used only when `C1` is `skip`, and since `skip` is a final state, there can be no transition at all from it. Similarly, the first rule for `cond-nz(E,C)` doesn't overlap with either of the other two, and the latter obviously don't overlap with each other due to the explicit inequality test.

▷ Overlapping rules give rise to *nondeterminism* in computations.

For example, suppose that we were to replace `V1` by `E1` in the second rule for `tup-seq` in Table 1.9; then evaluation of `tup-seq(E1,E2)` could start not only with steps of `E1`, but also with steps of `E2`—and even switch back and forth at irregular intervals. Such *interleaving* of the computations of components is actually allowed by some languages (e.g., Pascal), although ML, Java, and many other languages insist on sequential evaluation for constructs that involve more than one subexpression.

When expressions have no side-effects (nor throw exceptions), the nondeterminism that arises in computations due to interleaving their evaluations doesn't affect their computed values, and all the interleavings are essentially rearrangements of a single computation. The possibility of 'interfering' side-effects can however give rise to computations that lead to different final states (as well as different stores), so then the nondeterminism is more apparent. For a more obvious example of nontrivial nondeterminism, consider what happens if we remove the test for `N` being nonzero in the third rule for `cond-nz(E,C)`: now the next state after `cond-nz(0,C)` can be either `skip` or `C`.

▷ The order of overlapping rules can affect prototyping.

Each MSOS rule gets translated to a single Prolog rule. The prototype implementation based on the generated rules tries using them in the order in which they are given. This gives computations with a particular sequential order of evaluation: the first of a set of overlapping rules is always used, and the others are simply ignored.

Once Prolog has completed a computation, however, the prototype implementation can be forced to look for further computations in which rules are applied in different orders. When overlapping rules allow interleaving, the number of alternative computations grows exponentially in the length of the interleaved computations, and it is usually infeasible to explore more than a tiny proportion of them when running a program. When overlapping rules lead to direct nondeterministic choice, however, it should be possible to explore all the possible computations in the prototype implementation.

▷ The dynamic semantics of an individual construct declares all its own requirements.

As with static semantics, duplication can be avoided by specifying frequently-occurring requirements in a separate auxiliary module. Requirements on components of labels are however best stated explicitly for each construct that refers to these components, for perspicuity.

▷ A separate transition relation is specified for each set of constructs.

As with static semantics, the set to which the source of a transition belongs has to be apparent. This avoids confusion in connection with subset inclusions in abstract syntax. For example, consider the rules for `Exp::=Var` in Table 1.8: in the first rule, the transition in the condition is implicitly for `VAR:Var`. In the conclusion of the same rule, `VAR:Exp` cannot be simplified to just `VAR`, since it would then be treated as `VAR:Var`, and it is nonsensical to have a rule where the only condition is identical to the conclusion.[2]

[2]Such rules lead to the prototype implementation getting into an infinite loop.

3.2 Complete Languages

▷ Complete language descriptions are based on modular descriptions of the abstract syntax and semantics of individual constructs.

Reuse of modules describing individual constructs has several major advantages:

- No time is wasted on repeating descriptions of familiar constructs.
- Attention can be focussed on the analysis of languages in terms of abstract constructs, and on the description of novel constructs.
- Previously-used modules should not need further validation.
- When two or more language descriptions refer to the same module, it is immediately apparent that they have a construct in common.
- Readers of the description of a new language may already be familiar with the abstract syntax and semantics of the constructs involved, in which case study of the mapping from concrete syntax to abstract syntax suffices to understand the semantics of the new language.

▷ Only the description of concrete syntax and its mapping to abstract syntax is non-modular.

A description of the concrete syntax of a programming language is relatively small compared to its semantic description. Large-scale reuse of parts of grammars for concrete syntax might be possible, but the advantages appear to be relatively minor. One exception is when describing an extension of a language: then it is natural to refer to the location of the original grammar, instead of including it verbatim—provided that no productions have to be changed or removed when adding the new ones.

Ex.3.6 The following sections consider each part of the description of a complete language, following the same order as in Chapter 1. The files containing the complete description of our illustrative subset of bc are stored at `Lang/bc/1`.

3.2.1 Concrete Syntax

In theory, programs can be created as abstract syntax trees, and we needn't bother with concrete syntax. In practice, however, it is much easier to both write and read programs as text strings in the usual style. Also when we are studying the semantics of a language, to have a concrete syntax for the language facilitates writing test programs for use in the validation of its semantic description.

▷ Describing concrete syntax is hard work!

One should not underestimate the difficulty of writing a complete formal grammar that exactly captures the concrete syntax of a large programming language. Even using one of the most powerful formalisms available for specifying context-free syntax, SDF [15], developing and validating grammars for languages like C++ and Java is a formidable task [15, SdfGrammars]. One might expect to find usable grammars in programming language reference manuals, but they often omit details concerning the relationship between lexical and context-free analysis, and give only informal explanations of delicate issues such as precedence and disambiguation.

▷ It is usually quite easy to write a DCG that accepts simple test programs.

Fortunately, it turns out to be quite straightforward to capture the main features of the concrete syntax of typical programming languages. When writing test programs, we do not need to exploit precedence between operations, and can simply insert parentheses where required. In a similar spirit, we can make do with a simple syntax for identifiers, and ignore the full richness that might be exploited when writing large application programs. Table 1.1 illustrated how such a simplified concrete syntax (for a very small language) can be described quite straightforwardly using a DCG.

▷ Left-recursive nonterminals have to be avoided.

DCGs are remarkably powerful, but their usual implementation by Prolog has one weak point: so-called left-recursion leads to nontermination of parsing. (A production of the form `A-->A,...` is directly left-recursive, but in general, left-recursion can be indirect, and involve more than one production.) This is particularly annoying, since the natural grouping of arithmetic expressions such as `3-2-1` is as `(3-2)-1`, which requires left-recursion for a straightforward description. The same problem arises in connection with LL-grammars and the associated top-down parsing algorithms.

Here, we simply insert parentheses when writing expressions such as `3-2-1` in test programs.

▷ Lexical symbols are specified together with context-free phrase structure.

Table 1.1 illustrated how lexical analysis can be specified directly in a DCG. The traditional treatment of lexical analysis as a separate phase of parsing (i.e., scanning) can be used with DCGs, but it tends to complicate the grammars, so we

avoid it. This implies that we have to specify explicitly in our DCGs exactly where optional layout characters (and comments) are allowed to occur; this is usually between separate terminal symbols, and it is straightforward to insert a nonterminal symbol such as '`i`' at the corresponding places in the DCG.

▷ Validation would involve systematically testing all rules of a DCG.

It is quite tedious to write a comprehensive suite of programs to check whether or not a DCG accepts just the intended ones. Any significant deficiencies of a DCG are usually revealed in connection with validation of a mapping from concrete to abstract syntax, and of the static and dynamic semantics of programs.

The software accompanying these notes assumes that the start symbol of each DCG for a programming language is '`prog`'. The Prolog query `phrase(prog,S)` tries to recognize the string `S` as `prog`, reporting simply `Yes` or `No`. In the latter case, the easiest way of locating which part of `S` caused the string not to be recognized may be to try parsing substrings of `S` (with the corresponding nonterminal symbols instead of `prog`).[3]

Ex.3.8

3.2.2 Abstract Syntax

▷ Selection of just the right abstract constructs can be difficult.

Each concrete syntactic construct of a complete programming language has to be mapped to abstract syntax. Identifying the range of constructs to be used when defining this mapping is a non-trivial matter. The illustrative subset of bc shown in Chapter 1 might be a bit misleading in this respect, since most of its concrete constructs correspond directly to abstract constructs. The treatment of infix operations hints at the kind of complications that can arise when there is a mismatch: the abstract relations for '<', etc., are all boolean-valued, so an application of the operation `ord` (which maps false to zero and true to one) has to be inserted to obtain the expected integer.

Many examples of this kind of analysis will be given in Chapters 5–10.

▷ New constructs and variants can be added as needed.

When it is difficult to derive an intended abstract construct by combining other available constructs, a new abstract construct (or a variant of an old one) can be added to the repository of abstract constructs. This may sound like an easy option, but it carries with it the responsibility to provide also the MSOS of the new construct.

[3] It would be better if Prolog could indicate the exact location of the mismatch between the DCG and the string `S`, but this appears to be impossible...

▷ The collected abstract syntax is simply a list of references to modules.

Module names are essentially paths in the MSOS repository. For example, the full path for referring to the abstract syntax of the `seq` command is '`Cons/Cmd/seq/ABS`'. For perspicuity, such a path can be abbreviated to '`Cmd/seq`', and a path referring to a set of constructs such as `Cmd` can be abbreviated to the name of the set. Table 1.16 lists the references to the modules needed for the abstract syntax of the illustrative subset of bc.

▷ Validation would involve testing that abstract syntax trees are in the expected sets.

The software accompanying these notes provides a Prolog command to convert MSDF modules for abstract syntax to Prolog; loading the Prolog code generated from a list of references to modules loads all the code for those modules. The Prolog query `check(T:'Cmd')` tests whether the abstract syntax tree represented by the Prolog term `T` is in the set named `Cmd`. N.B. Hyphens in MSDF are replaced by underscores in Prolog, e.g., `cond-nz` has to be written `cond_nz`. Thus the query `check(cond_nz(1,print(42)):'Cmd')` should give the response `Yes` (assuming that appropriate generated Prolog code has been loaded) whereas `check(cond-nz(1,print(42)):'Cmd')` will give `No`. Ex.3.9

As with concrete syntax, systematic validation of abstract syntax specifications is somewhat tedious. In practice, it is sufficient to check that the names of sets of constructs have not been mis-spelled.

3.2.3 Mapping Concrete to Abstract Syntax

▷ Specifying a mapping in a DCG is easy.

Given a DCG for concrete syntax and an appropriate collection of abstract constructs, it is generally quite a simple matter to edit the DCG so that recognized programs are mapped to their abstract syntax trees. For instance, the DCG given in Table 1.3 was easily obtained from that given in Table 1.1.

▷ The structure of the underlying DCG for concrete syntax may need to be adjusted.

Adjustments are required when constructs which look the same from the point of view of concrete syntax have to be mapped to different abstract constructs. This was the case for applications of infix operations in our subset of bc: an application of the operation `ord` was needed for relational expressions, but not for numerical expressions, so these two constructs had to be separated in the concrete syntax, before adding the mapping to abstract syntax.

▷ Mapping decimal notation to integers requires use of a standard Prolog predicate.

It is possible to insert arbitrary Prolog code in a DCG. This allows us to map decimal numerals directly to abstract integers, removing the need to introduce abstract syntax constructors for decimal numerals, and to define their semantics in MSOS. A similar predicate can be used to map a list of characters to a Prolog atom that conveniently represents an abstract identifier.

Ex.3.10

▷ An entire program should be mapped to a tree with an explicit type.

The set of abstract constructs to which entire programs are mapped is needed in connection with validation of the mapping, and also for running programs. It is specified by a term such as `(C:'Cmd')`.

▷ Validation involves testing all rules of the DCG, and checking that the resulting trees are in the expected sets.

In contrast to the situation with the descriptions of concrete and abstract syntax, it is advisable to check carefully that trees constructed by the mapping from concrete syntax are in fact well-formed according to the abstract syntax. This can be achieved using compound Prolog queries of the form `phrase(prog(T),S), check(T)`. (A Prolog predicate provided by the software accompanying these notes lets the query be formulated a bit more concisely.)

Ex.3.11

3.2.4 Static Semantics

▷ The collected static semantics is the same list of references to modules as for abstract syntax.

An abbreviated module name (omitting the final `ABS`, `CHK`, or `RUN`) refers to the same part of an MSOS as where it is used. Thus the list of module names that collected together the abstract syntax constructs in Table 1.16 can be reused to collect the static semantics of the same constructs.

▷ Subset inclusions may be added.

In general, some sets used in MSOS might be left under-specified by the collection of constructs included in a complete language, and can be extended with further sets of elements. For instance, none of the individual constructs listed in Table 1.16 (nor sets of constructs) requires the type `int` to be included in the set `StorableType`; this has to be specified explicitly. Notice that `bool` does not need to be included in `StorableType`, since bc programs are mapped to combinations of abstract constructs that do not allow boolean values to be assigned to variables.

▷ Validation involves testing that the static semantics of each construct has a type iff the types of its components are consistent.

When a program is well-formed, its static semantics computes a type in a single step, and the prototype implementation of the language shows the type of the program. Otherwise, the static semantics has no computations at all, and the prototype implementation simply reports '`No`'. Thanks to the structural nature of the semantic rules, the number of tests required to properly validate the semantics is directly proportional to the number of constructs included in the complete language. Ex.3.12

3.2.5 Dynamic Semantics

In most respects, the dynamic semantics of a complete language is specified and validated in the same way as the static semantics.

▷ The collected dynamic semantics is the same list of references to modules as for abstract syntax.

This reuse of the list of references works in exactly the same way as explained above for static semantics.

▷ Some subset inclusions may need to be added.

For the dynamic semantics of our subset of bc, the inclusion `Storable::=Integer` has to be added.

▷ Validation involves testing that all constructs compute the expected values when used in complete programs.

Dynamic semantics inherently involves more rules and alternatives than static semantics, so proper validation requires substantially more tests. However, as with static semantics, the structural nature of MSOS transition rules helps to limit the number of tests required to reveal any mistakes in the rules. Ex.3.13

▷ Validation also involves following the progress of computations by inspecting intermediate states.

The prototype implementation generated from the MSOS of a complete language allows inspection of intermediate states of computations. This is particularly useful for locating the cause of failure to reach a final state: the last intermediate state shown usually reveals the reason why no rule could be applied to produce a next state. Ex.3.14

Summary

In this chapter:

- We indicated how monolithic MSOS descriptions of languages are divided into reusable modules, each concerned with an individual abstract construct. Auxiliary modules declare common notation associated with particular sets of constructs.
- We considered various features of MSOS modules for abstract syntax, static semantics, and dynamic semantics, referring to a repository of reusable modules available on the Internet.
- We recalled how DCGs can be used to specify the concrete syntax of complete languages, and enhanced to map concrete constructs to abstract constructs.
- We explained how the same list of module names can be used to assemble the abstract syntax, static semantics, and dynamic semantics for a complete language.

Exercises

The exercises for this chapter involve use of the Prolog code accompanying these notes. The aim is to generate a prototype implementation for the subset of bc described in Chapter 1, based on the reusable MSOS modules and on the (non-modular) DCG provided on the Internet. You are also asked to validate the modules by running bc programs using the prototype implementation.

Familiarity with the process of generating Prolog from MSOS descriptions of individual constructs, and of running test programs, is essential preparation for writing and validating your own MSOS modules and language descriptions in connection with Chapters 5–10.

Ex. 3.1

(a) Download and unzip the file `http://www.cs.swan.ac.uk/~cspdm/MSOS.zip`. (The original version of `MSOS` included two files named '`CON.pro`'. On Windows, the name '`CON`' is apparently reserved for system use, so these files have now been renamed to '`CFG.pro`'.)

(b) Start Prolog from the `MSOS` directory.

(c) Load the file `Tool/load.pro`. This can normally be achieved using the following Prolog query:

```
['Tool/load'].
```

The quotation marks are needed to make Prolog treat `Tool/load` as an atom—recall that words beginning with capital letters are treated as variables in Prolog, and '`/`' is the infix operator for division.

The Prolog response should indicate that `Tool/load` has been loaded (and compiled) successfully; any warnings or error reports indicate that something is wrong, possibly due to inadvertent use of implementation-dependent features in the code.[4]

(d) View the contents of `Tool/load.pro`, reading the usage hints regarding the various commands near the top of the file.

Note that some of the following exercises might have side-effects on the Prolog system. Usually, these shouldn't cause any problems. However, if something in a later exercise doesn't work as expected, it might help to quit and restart Prolog (reloading `Tool/load`) and try again.

Ex. 3.2

(a) Discuss whether or not it is appropriate to distinguish the set of assignable variables `Var` from the set of expressions `Exp` in the abstract syntax of bc. Can you find a syntactic construct (in any language) involving an identifier whose classification as a variable or an expression is context-dependent?

(b) Would it be useful to distinguish a set `Const` of constant expressions as a subset of `Exp`? If so, should constant expressions include constant identifiers? How can we avoid deciding whether an occurrence of an identifier in the abstract syntax tree of an expression is a variable or a constant identifier?

[4] Please include details of operating system and Prolog version when reporting errors to the author.

Ex. 3.3

(a) Derive `cond-nz(E,C)` using `cond(E,C1,C2)`, assuming the condition of `cond-nz` to be an integer-valued expression, and that of `cond` to be a boolean-valued expression.

(b) Write down the derived tree for:

```
cond-nz(app(ord,app(<,tup-seq(E1,E2))),C) .
```

(c) Write down a simpler tree that you would expect to be semantically equivalent to the tree you derived above.

*(d) What restrictions on bc would allow your simpler tree to be produced directly when mapping concrete to abstract syntax? Propose changes to the DCG given in Table 1.3 to implement the mapping to the simpler trees.

Ex. 3.4 Consider the dynamic semantics of `seq(C1,C2)`. What features of `C1` might prevent the subsequent execution of `C2`?

***Ex. 3.5**

(a) Convert some of the big-step rules for the static semantics of bc, given in Tables 1.4–1.6, to small-step rules. Discuss any pros or cons of making such changes.

(b) Convert some of the small-step rules for the dynamic semantics of bc, given in Tables 1.7–1.9, to big-step rules. Discuss any pros or cons of making such changes.

Ex. 3.6 Consider how the DCG for the concrete syntax of our sublanguage of bc, given in Table 1.1, might be divided into modules. To what extent would you expect such modules to be reusable in the concrete syntax of related languages, such as C?

Ex. 3.7

(a) Reformulate the DCG given in Table 1.1 so that expressions such as `3-2-1` are accepted.

*(b) Reformulate the DCG given in Table 1.3 so that `3-2-1` is accepted, and mapped to the same abstract syntax tree as `(3-2)-1`.

Ex. 3.8 The following text is provided in the file `Test/bc/bad.bc`:

```
a=3;a;while(a<40){a=(a+3};a
```

(a) Load the file `Test/bc/LEX`, then `Test/bc/CFG`, which contains the DCG with the concrete syntax shown in Table 1.1.

(b) Check that `phrase(prog,"a=3;a;while(a<40){a=(a+3};a")` fails.

(c) Using the Prolog query `phrase(prog,S)` with various substrings of `"a=3;a;while(a<40){a=(a+3};a"` for `S`, locate the cause of the parsing failure, correct the string, and check that `phrase(prog,S)` succeeds with the corrected string for `S`.

(d) Propose an efficient general strategy for locating syntax errors in larger strings.

(e) Write some longer programs in our sublanguage of bc, and check whether they contain any syntax errors using the above DCG.

Ex. 3.9 Prolog regards `check(cond-nz(1,print(42)):'Cmd')` as a syntactically well-formed query. How does Prolog group the argument of `check`?

Ex. 3.10

(a) Load the file `Test/bc/LEX`, then `Test/bc/SYN`, which contains the DCG with the mapping from concrete to abstract syntax shown in Table 1.3.

(b) The Prolog predicate `atom_chars(A, CL)` can be used to split an atom `A` into a list of one-character atoms `CL`, or combine a list of one-character atoms `CL` into a single atom `A`. Locate the description of this predicate in the SWI-Prolog Reference Manual.[5]

(c) Making use of nonterminal symbols defined in `Test/bc/LEX`, extend the syntax of variable identifiers in `Test/bc/SYN` to allow all sequences of lowercase letters and digits that start with a letter, mapping them to terms of the form `id(A)`, where `A` is generally a multi-character atom.

(d) Validate the changes that you have made.

[5]The SWI-Prolog Reference Manual is available from `http://www.swi-prolog.org`, and might also be installed locally.

Ex. 3.11

(a) Start Prolog in the `MSOS` directory and load `Tool/load`.

(b) Try the following query, which not only maps concrete to abstract syntax, but also checks that the resulting tree is a well-formed element of the set `Cmd`:

```
parse('Test/bc', "a=42;a").
```

The response should be roughly as follows:

```
% Test/bc/ABS compiled 0.01 sec, 3,668 bytes
% Test/bc/SYN compiled 0.00 sec, 7,224 bytes

    seq
    ( effect
      ( assign_seq(id(a), 42) ),
      print(id(a)) )
    :Cmd

Parsed OK

Yes
```

(If the files `Test/bc/SYN` and `Test/bc/ABS` have already been loaded, they won't be reloaded.)

(c) Try the following query, which reads the string to be parsed from the file `Test/bc/4.bc`:

```
parse('Test/bc', 'Test/bc/4.bc').
```

It should again indicate that the parsing was successful

*(d) Write a single bc program whose parsing involves every production of the DCG given in Table 1.3, then use a query like the one above to parse the program and check the structure of the resulting abstract syntax tree. (If the Prolog response to your query is '`No`', this may be due to a syntax error in your program. Locate the source of the error, e.g., by removing parts of the program. In the case that the program is syntactically correct, but the resulting tree does not conform to the abstract syntax, please send a bug report to the author.)

Ex. 3.12

(a) Start Prolog in the `MSOS` directory and load `Tool/load`.

(b) Try the following query, which not only maps concrete to abstract syntax and checks that the resulting tree is a well-formed element of the set `Cmd`, but also runs the Prolog code generated from the static semantics shown in Tables 1.4–1.6.

```
check('Test/bc', "a=42;a").
```

The response should be roughly as follows:

```
% Test/bc/CHK compiled 0.00 sec, 10,196 bytes
%  ABS compiled 0.00 sec, 2,952 bytes
%  SYN compiled 0.01 sec, 7,184 bytes
% Test/bc/CHK-init compiled 0.01 sec, 10,700 bytes

    seq
    ( effect
      ( assign_seq(id(a), 42) ),
      print(id(a)) )
    :Cmd

Parsed OK

== { env =
        map
        ( [ id(a) |-> ref(int),
            id(b) |-> ref(int) ] ) }
=>
    void

Yes
```

(Files that are already loaded won't be reloaded.)

(c) Try the following query, which reads the string to be checked from the file `Test/bc/4.bc`:

```
check('Test/bc', 'Test/bc/4.bc').
```

Note that the printing of the label can be switched off and on using the query `hide_label`, resp. `show_label`.

*(d) Write a suite of bc programs which can be used to validate the static semantics specified in Tables 1.4–1.6.

Ex. 3.13

(a) Start Prolog in the `MSOS` directory and load `Tool/load`.

(b) Try the following query, which not only maps concrete to abstract syntax and checks that the resulting tree is a well-formed element of the set `Cmd`, but also runs the Prolog code generated from the dynamic semantics shown in Tables 1.7–1.9.

```
run('Test/bc', "a=42;a").
```

The response should be roughly as follows:

```
% Test/bc/RUN compiled 0.01 sec, 9,756 bytes
%  ABS compiled 0.00 sec, 2,980 bytes
%  SYN compiled 0.01 sec, 7,184 bytes
% Test/bc/RUN-init compiled 0.01 sec, 10,900 bytes

    seq
    ( effect
      ( assign_seq(id(a), 42) ),
      print(id(a)) )
    :Cmd

Parsed OK

-- { env =
        map
        ( [ id(a) |-> cell(1),
            id(b) |-> cell(2) ] ),
      store =
        map
        ( [ cell(1) |-> 0,
            cell(2) |-> 0 ] ),
      store' =
        map
```

```
        ( [ cell(1) |-> 42,
            cell(2) |-> 0 ] ),
      out' = 42 }
->*
    skip

Yes
```

(Files that are already loaded won't be reloaded.)

(c) Try the following query, which reads the string to be checked from the file `Test/bc/4.bc`:

```
run('Test/bc', 'Test/bc/4.bc').
```

(That one may take a bit longer than the previous queries.)

*(d) Write a suite of bc programs which can be used to validate the dynamic semantics specified in Tables 1.7–1.9.

Ex. 3.14

(a) Start Prolog in the `MSOS` directory and load `Tool/load`.

(b) Try the following query, which not only maps concrete to abstract syntax and checks that the resulting tree is a well-formed element of the set `Cmd`, and runs the Prolog code generated from the dynamic semantics shown in Tables 1.7–1.9, but also shows the intermediate states of the computation:

```
run('Test/bc', "a=42;a", 6).
```

The response should be roughly as follows:

```
-- { env =
        map
        ( [ id(a) |-> cell(1),
            id(b) |-> cell(2) ] ),
      store =
        map
        ( [ cell(1) |-> 0,
            cell(2) |-> 0 ] ),
      store' =
```

```
        map
        ( [ cell(1) |-> 42,
            cell(2) |-> 0 ] ),
      out' = null }
-6>*
    print(42)

Yes
```

(If the number specified is greater than the length of the computation, the response is the same as if the number had been omitted, except that it ends with `No`.)

(c) Try the following query, which reads the string to be checked from the file `Test/bc/4.bc`, and starts printing intermediate states only after the first three transitions:

```
hide_label.
run('Test/bc', "a=42;a", 99, 3).
```

The response should be roughly as follows:

```
---3>*
    seq(skip, print(id(a)))
---4>*
    print(id(a))
---5>*
    print(cell(1))
---6>*
    print(42)
---7>*
    skip

No
```

Notice that the last step shows the length of the computation.

Chapter 4

Modular Proof

Section 4.1 explains and illustrates *structural induction*, which is a proof technique for establishing general properties of abstract syntax trees. It shows how to use structural induction to prove properties of the transition relation specified by an MSOS, and how such proofs can be decomposed following the modular structure of MSOS specifications.

Section 4.2 defines several *equivalences* on computations, and considers which of them might correspond to semantic equivalence. It then introduces the notion of *bisimulation* between states of labelled transition systems, and explains and illustrates the basic proof technique for bisimulation. Proof of bisimulation between two states of an MSOS involves only the rules for the constructs involved in those states, so the technique is inherently modular.

4.1 Induction

Let us first consider the general idea of structural induction, before looking at its application to MSOS.

4.1.1 Structural Induction

▷ Proofs by structural induction are based on *constructors*.

Suppose that all the elements of a set can be constructed using a fixed collection of functions and constants. Typically, the set is simply defined to be the *least* set which both contains a particular collection of constants, and is *closed* under a particular collection of functions, so it automatically has the required property. Then we can use structural induction to prove general properties of the elements of that set.

Let us refer to the indicated functions and constants as the *constructors* of the set. Usually, we shall assume that the functions are injective (mapping different argument lists to different results) with disjoint ranges, and that the constants are distinct values, but these properties are actually not required for structural induction to be a sound proof technique.

▷ To prove that a property holds for all elements of a set, we merely have to show that it is *preserved* by each constructor for the set.

Let $P(x)$ be a property of elements x in some set A. We say that P is *preserved* by an n-ary constructor function f for A if, whenever $P(x_1),\ldots,P(x_n)$ all hold, so does $P(f(x_1,\ldots,x_n))$; and P is preserved by a constant constructor c for A if $P(c)$ holds.

Definition 4.1 (structural induction principle) *If P is preserved by all the constructors of a set A, then $P(x)$ holds for all x in A.*

Thus if we can prove that a property is preserved by each constructor for a set, we can conclude by the Structural Induction Principle (SI) that all the elements of the set have that property.

The following simple illustration shows how a proof by SI is expressed.

Example 4.2 Consider the following declarations in MSDF:

```
BinTree ::= zero | one | bin(BinTree,BinTree) .
```

Let A be the specified set of abstract syntax trees, i.e., the least set containing the constants `zero` and `one`, and closed under formation of pairs. For any t in A, let $z(t)$ denote the number of occurrences of `zero`, $o(t)$ the number of occurrences of `one`, and $b(t)$ the number of occurrences of `bin` pairs in t. For instance, if t is constructed by `bin(one,bin(bin(zero,one),one))` we have $z(t)=1$, $o(t)=3$, and $b(t)=3$.

Let $P(t)$ be the property $z(t)+o(t)=b(t)+1$. We prove that $P(t)$ holds for all t in A by showing that P is preserved by each constructor for A, as follows.

`zero`: Let $t=$ `zero`. Then $z(t)=1, o(t)=0, b(t)=0$, so $P(t)$ is $1+0=0+1$, which is true, so P is preserved by `zero`.

`one`: Let $t=$ `one`. Then $z(t)=0, o(t)=1, b(t)=0$, so $P(t)$ is $0+1\leq 0+1$, which is true, so P is preserved by `one`.[1]

[1] A case that is entirely analogous to a previous case is usually abbreviated to "Similarly".

bin: Let $t = \texttt{bin}(t_1, t_2)$. Assuming that $P(t_1)$ and $P(t_2)$ both hold, we need to show that $P(t)$ holds. Clearly, $z(t) = z(t_1) + z(t_2), o(t) = o(t_1) + o(t_2)$, and $b(t) = b(t_1) + b(t_2) + 1$. So $z(t) + o(t) = (z(t_1) + z(t_2)) + (o(t_1) + o(t_2)) = (b(t_1) + 1) + (b(t_2) + 1) = b(t) + 1$, hence $P(t)$, so P is preserved by `bin`.

We have shown that P is preserved by all the constructors for A. By the Structural Induction Principle, we conclude that $P(t)$ holds for all t in A.

Ex.4.1

Before we proceed to consider proofs of properties of transition relations by SI, let us extend the technique a bit so that it covers proofs about properties of all sets that can be specified in MSDF:

▷ Properties can be proved by SI also when the constructors of several sets are declared together, possibly with subset inclusions.

Example 4.3 Consider the following declarations in MSDF:

```
BitTree ::= Bit | bin(BitTree,BitTree)

Bit ::= zero | one .
```

Then the property P defined in the preceding example can be proved by SI in exactly the same way.

Ex.4.2

Ex.4.3

▷ Arguments from given sets don't require extra cases in proofs.

Example 4.4 Consider the following declarations in MSDF:

```
IntList ::= nil | cons(Integer,IntList) .
```

Let A be the specified set `IntList`, which corresponds to the set of finite, possibly-empty lists of integers. For any t in A, let $l(t)$ denote the number of components of t, $s(t)$ the absolute value of the sum of the components of t (0 when t is the empty list), and $m(t)$ the maximum of the absolute values of the integers contained in t (0 when t is the empty list). For instance, if t is constructed by `cons(27,cons(-42,cons(1,nil)))` we have $l(t) = 3$, $s(t) = 15$, and $m(t) = 42$.

Let $P(t)$ be the property $s(t) \leq l(t) \times m(t)$. To illustrate the treatment of the given set of integers, we prove that $P(t)$ holds for all t in A, as follows.

`nil`: Let $t = \texttt{nil}$. Then $l(t) = 0, s(t) = 0, m(t) = 0$, so $P(t)$ is $0 \leq 0 \times 0$, which is true, so P is preserved by `nil`.

`cons`: Let $t = \texttt{cons}(n, t')$. Assuming that $P(t')$ holds (notice that we do not make any inductive assumption about n) we need to show that $P(t)$ holds. We have $s(t) \leq abs(n) + s(t') \leq abs(n) + (l(t') \times m(t')) \leq m(t) + (l(t') \times m(t)) = l(t) \times m(t)$, hence $P(t)$, so P is preserved by `cons`.

We have shown that P is preserved by all the constructors for A. By the Structural Induction Principle, we conclude that $P(t)$ holds for all t in A.

Note, however, that when a given set is included as a subset of a set being specified, the elements of the given set are essentially treated as if they were constant constructors, and their preservation of the property being proved has to be shown.

▷ Numerical induction can be regarded as a special case of structural induction.

Example 4.5 Consider the following declarations in MSDF:

```
Nat ::= z | s(Nat) .
```

The elements of the specified set `Nat` correspond to natural numbers in the obvious way, and Structural Induction on `Nat` corresponds exactly to ordinary mathematical induction.

Ex.4.4

4.1.2 Proofs About Transitions

In MSOS, we specify abstract syntax by declaring a constructor function or constant for each syntactic construct in a separate module. When we combine these modules for use in the MSOS of a complete language, the specified sets of abstract syntax trees are taken to be the least sets which contain the declared constructor constants, and are closed under the declared constructor functions. Thus the Structural Induction Principle (SI) is always available for proving properties of abstract syntax trees.

▷ Using structural induction, one may be able to prove general properties of a transition relation, provided that all transition rules are *structural*.

Recall that a transition rule with conclusion $S \xrightarrow{L} S'$ is called structural when for each of its conditions of the form $S_i \xrightarrow{L} S_i'$ (if any), S_i is a component of S. In these notes, we shall stick to structural transition rules.[2]

The kind of property that is straightforwardly provable by SI is illustrated in the following example:

Example 4.6 Let A be the (disjoint) union of the sets of abstract syntax trees for a subset of the language bc, as specified in Table 1.2. Let $S \xrightarrow{L} S'$ be the transition relation for the dynamic semantic of bc, as specified in Tables 1.7–1.9.

The *determinism* of the transition relation at a state S is expressed by the property $P(S)$ which holds iff, whenever there exist transitions $S \xrightarrow{L} S'$ and $S \xrightarrow{L} S''$, we always have $S' = S''$.

Here is the proof that the constructor `seq` (of the set `Cmd` of commands) preserves the above property P:

`seq`: Let $S = \texttt{seq}(C_1, C_2)$, and suppose that for some S', S'' we have $S \xrightarrow{L} S'$ and $S \xrightarrow{L} S''$. These transitions have to be derivable by some rules (since the specified transition relation is the least relation that is closed under the rules). There are only two specified rules that allow derivation of transitions $S \xrightarrow{L} S'$ when S is of the form $\texttt{seq}(C_1, C_2)$; let us consider them separately. We need to show $S' = S''$ in each case.

- ```
  C1 --{...}-> C1'
  ----------------------------------
  seq(C1,C2):Cmd --{...}-> seq(C1',C2)
  ```

  In this case, there must be a transition $C_1 \xrightarrow{L} C_1'$, so we have $C_1 \neq \texttt{skip}$ and $S' = \texttt{seq}(C_1', C_2)$. Since $C_1 \neq \texttt{skip}$, the only possibility for deriving $S \xrightarrow{L} S''$ is by the same rule, so there must be a transition $C_1 \xrightarrow{L} C_1''$, and $S'' = \texttt{seq}(C_1'', C_2)$. However, we may assume that $P(C_1)$ holds, which gives $C_1' = C_1''$, hence $S' = S''$ as required.

- ```
  seq(skip,C2):Cmd ---> C2
  ```

  In this case, we have $C_1 = \texttt{skip}$ and $S' = C_2$. Since there is no transition from `skip`, the only possibility for deriving $S \xrightarrow{L} S''$ is by the same rule, giving $S'' = C_2$, hence $S' = S''$ as required.

[2] In fact it seems that non-structural rules are not needed at all in MSOS.

This shows that P is preserved by `seq`.

The proofs that P is preserved by all the other constructors of the abstract syntax for bc are completely *independent* of the above proof for `seq`:

▷ A property holds for the transition relation specified by a collection of MSOS modules when it is preserved by the constructor of each module separately.

This is assuming that no two modules in the collection specify transitions for the same constructor (and that no module specifies transitions for states with arbitrary constructors, of course).

Thus we see that proofs of properties of transitions by structural induction can be just as modular as the MSOS specification of the transition relation itself.

Ex.4.5 However, if even one rule in an MSOS were to be non-structural, it would prevent completion of all proofs by structural induction. As mentioned before, such rules are not expected to be needed in practical applications of MSOS, so we shall not worry unduly about this point, but merely note here that a more general (and equally modular) proof technique is available:

Ex.4.6

▷ So-called *rule induction* is more general, and does not require rules to be structural.

Properties proved by Rule Induction are of the form $P(S, L, S')$. To prove that P holds for all derivable transitions $S \xrightarrow{L} S'$, one proves for each rule that whenever it holds for all the transitions in the conditions (if any), it also holds for the transition in the conclusion. (When the property does not depend on L, one can write $P(S, S')$ instead of $P(S, L, S')$.)

4.2 Semantic Equivalences

Let us now consider when programs could be regarded as *semantically equivalent* on the basis of their computations. Recall that an equivalence relation is transitive, reflexive, and symmetric.

More generally, we shall consider semantic equivalence not only on complete programs but also on parts of programs, such as commands, expressions, and declarations. An equivalence relation is called a *congruence* when it is preserved by all constructs. Then, for example, when two expressions are equivalent, replacing one of them by the other in any complete program always gives equivalent programs. Here, we shall only be interested in equivalence relations that turn out to be congruences. Formally:

Definition 4.7 (congruence) *Let $S \equiv S'$ be an equivalence relation on a set State, which is generated by a collection of constructors. The relation $S \equiv S'$ is called a* congruence *if for each n-ary constructor function f for State, $f(S_1,\ldots,S_n) \equiv f(S'_1,\ldots,S'_n)$ whenever $S_i \equiv S'_i$ for each $i = 1,\ldots,n$.*

It is straightforward to generalize equivalence relations and congruences to collections of sets with constructors and subset inclusions between them, as allowed for abstract syntax trees in MSOS. Elements of given sets (booleans, integers, lists, etc.) are regarded as equivalent only when they are identical.

For the illustrations in the rest of this chapter, we assume that states and transitions are as specified for the dynamic semantics of a sublanguage of bc in Chapter 1, see Tables 1.2 and 1.7–1.9.

▷ Computations for different programs are always *different*.

A computation for a program always starts with the abstract syntax of that program as the initial state. Thus to regard two programs as equivalent when they have the same sets of computations is not useful: it makes (distinct) programs equivalent only when they have no computations at all (i.e., they are both stuck states, not being in Final nor having any transitions).

The simplest way of defining useful semantic equivalences is to introduce equivalence relations on computations. For instance, trace equivalence is obtained by saying that computations are equivalent whenever they have the same sequence of labels, disregarding the states of the computations altogether. For actually proving equivalence, however, it is easier to define a so-called bisimulation relation directly on states, and show that it is preserved by the transition relation.

Here, we shall explain both techniques for defining equivalence, but illustrate proof of equivalence only in connection with bisimulation. This latter technique also has the advantage of inherent modularity.

4.2.1 Computation Equivalences

▷ We define computation equivalences in terms of *observations* on computations.

In general, there are (infinitely-)many computations starting from the same initial state, due to the possibility of choosing different labels for the first transition. Even when the initial label is fixed, nondeterministic constructs may give rise to further choices between transitions from intermediate states. Thus for each state, we have to consider not just the observation for a single computation, but the observations for *all* possible computations that start from it.

Computational equivalence between two states requires that for each observation on a computation starting from one of them, there is some computation starting from the other with the same observation. Thus programs are equivalent when they have the same sets of possible observations; the observation of a particular computation for one of them may well be different from that for a particular computation for the other, but it should be possible to get the same observations for some pair of computations.

Definition 4.8 (computation equivalence) $S \equiv S'$ *iff the set of* observations *of all computations starting from S is the same as that for all computations starting from* S'.

▷ Observations can include the *values* computed by finite computations.

Let C be any computation. When C is finite, final(C) is defined to be the last state of C, and otherwise final(C) is the element $\perp$, representing the lack of a final state.

Obviously, taking observations to be simply final states and regarding constructs as equivalent whenever they have the same sets of final states goes much too far. For example, the commands `print(0)` and `print(42)` both have the final state `skip`, but we surely do not want to regard them them as being equivalent.

▷ Observations can include the sequences of *labels* in computations.

Let C be any computation. Regardless of whether C is finite or infinite, trace(C) is defined to be the sequence of labels on the transitions in C.

Taking observations to be traces avoids making `print(0)` and `print(42)` equivalent. For constructs (such as expressions) that can compute different values, however, we would again be going too far. For instance, the expressions `0` and `42` have only empty computations, so the only trace in each case is the empty sequence of labels, making them trace equivalent. But if equivalence is to be a congruence, that would again require `print(0)` and `print(42)` to be equivalent.

So let us try the combination of computed values with traces:

▷ Observations can include *both* the values computed by finite computations *and* the sequences of labels in computations.

This amounts to taking the observation of an arbitrary computation C to be the pair $(\text{final}(C), \text{trace}(C))$. It avoids the obvious problems with command and expression equivalence that we noticed in connection with the previous candidates for a useful definition of equivalence. In fact it seems likely to give a congruence.

But just how useful is it? Can we generally expect to be able to justify interchangeability of two constructs by showing that their sets of observations are identical? Unfortunately not, because traces reflect *all* the steps of computations. For example, all computations starting from the expression `app(+,tup(2,2))` have a single step, whereas the only computation starting from the expression `4` is the empty sequence, having an empty trace. A semantic equivalence that cannot even justify replacing `2+2` by `4` is unlikely to be of much use in practice...

Recall, however, that in MSOS, we have distinguished a subset Unobs of labels for unobservable transitions. The label on the single step of computations starting from the expression `app(+,tup(2,2))` is always unobservable (reflecting the absence of side-effects). If we remove unobservable labels from traces, we make `app(+,tup(2,2))` equivalent to `4`.

▷ Observations can be computed values together with just the *observable* components of traces.

Let us now take the observation of an arbitrary computation C to be the pair $(\text{final}(C), \text{trace}(C) \setminus \text{Unobs})$, where for any finite or infinite sequence Q and set N, $Q \setminus N$ is the subsequence of Q obtained by discarding all elements of N.

Notice that $\text{final}(C) = \bot$ iff C is a nonterminating computation. The possibility of discarding an infinite subsequence of unobservable labels from (the end of) a computation may lead to nonterminating programs having the same observable traces as terminating ones, but their observations are still distinguished.

This is as far as we shall go with our exploration of semantic equivalence based on observations on computations. The combination of computed values with observable traces should give a reasonably useful congruence relation for many languages, including our subset of bc. (One might wish to replace each finite observable trace by the composition of its labels, so as to make `seq(assign-seq(CL1,1), assign-seq(CL2,2))` equivalent to the command obtained by reversing the order of the assignments when `CL1` is different from `CL2`; but in the presence of interleaving, this would prevent semantic equivalence from being a congruence.)

4.2.2 Bisimulation Equivalences

In the preceding section, we defined various semantic equivalences on entire computations. Let us now consider an alternative technique for defining semantic equivalences, based on so-called *bisimulation* relations between the individual states of computations. This technique comes equipped with a particularly straightforward way of proving that programs are equivalent.

▷ Various kinds of bisimulation can be defined.

When defining bisimulation between states, we may choose to ignore sequences of unobservable transitions—much as we did when defining observations on computations. The version which ignores unobservable transitions is called *weak* bisimulation, although it is actually much more useful in practice than the standard ('strong') version. Many other variations on the theme of bisimulation are possible: rooted, branching, barbed, ... However, the differences between them appear to be significant only in connection with the semantics of concurrent processes, and we shall ignore such variations in these notes.

▷ *Strong bisimulation* requires the labels on transitions from related states to match *exactly*.

A so-called *strong bisimulation* between two programs is a relation between the states of their respective computations such that whenever one of them has a transition with some particular label, so has the other—and the resulting states are then always in the bisimulation relation as well. When one of the programs can terminate in some particular state, the other has to be able to terminate in that same state too.

Definition 4.9 (strong bisimulation) *A binary relation $R \subseteq State \times State$ is a* strong bisimulation *if whenever $S_1\ R\ S_2$:*

1. *if $S_1 \xrightarrow{L} S_1'$ for some label L, then $S_2 \xrightarrow{L} S_2'$ for some S_2' with $S_1'\ R\ S_2'$;*
2. *if $S_2 \xrightarrow{L} S_2'$ for some label L, then $S_1 \xrightarrow{L} S_1'$ for some S_1' with $S_1'\ R\ S_2'$;*
3. *if $S_1 \in Final$ or $S_2 \in Final$ then $S_1 = S_2$.*

Definition 4.10 (strongly bisimilar) *S_1 and S_2 are* strongly bisimilar*, written $S_1 \sim S_2$, if there exists a strong bisimulation R with $S_1\ R\ S_2$.*

Proposition 4.11 *$\sim = \bigcup\{R \mid R$ is a strong bisimulation$\}$, is itself a strong bisimulation, the largest strong bisimulation, and an equivalence relation.*

Bisimulation comes together with a simple proof technique. Suppose that S_1, S_2 are two states. To prove $S_1 \sim S_2$, we proceed as follows:

- define *any* relation R that includes $S_1\ R\ S_2$;
- prove that R is a bisimulation (extending R, if needed, to include further pairs of states for the proof to go through).

Since $(S_1, S_2) \in R$ and $R \subseteq \sim$, we can then immediately conclude that $S_1 \sim S_2$ holds.

To illustrate the above technique, let us prove that the commands `seq(C1,seq(C2,C3))` and `seq(seq(C1,C2),C3)` are strongly bisimilar, i.e., that sequencing of commands is associative, up to strong bisimilarity.

Proposition 4.12 $\mathit{seq}(C_1, \mathit{seq}(C_2, C_3)) \sim \mathit{seq}(\mathit{seq}(C_1, C_2), C_3)$.

Proof: Define R to be the least reflexive relation on commands with $\mathtt{seq}(C_1, \mathtt{seq}(C_2, C_3))\ R\ \mathtt{seq}(\mathtt{seq}(C_1, C_2), C_3)$ for all commands C_1, C_2, C_3. Thus we have to consider two cases for $C\ R\ C'$:

1. $C = C'$, or
2. $C = \mathtt{seq}(C_1, \mathtt{seq}(C_2, C_3))$ and $C' = \mathtt{seq}(\mathtt{seq}(C_1, C_2), C_3)$.

In each case, we have to show that when there is a transition $C \xrightarrow{L} C_0$, there is a corresponding transition $C' \xrightarrow{L} C'_0$ with $C_0\ R\ C'_0$, and that when $C \in \text{Final}$, we have $C = C'$.

1. $C = C'$: The required properties are trivial.
2. $C = \mathtt{seq}(C_1, \mathtt{seq}(C_2, C_3))$ and $C' = \mathtt{seq}(\mathtt{seq}(C_1, C_2), C_3)$: C is clearly not in Final. Suppose that there is a transition $C \xrightarrow{L} C_0$. The only rules allowing derivation of such a transition are the two rules for `seq` specified in Table 1.12. So we have a case for each rule:
   - The first rule for `seq` can only be used when $C_1 \xrightarrow{L} C'_1$, and then we have $C_0 = \mathtt{seq}(C'_1, \mathtt{seq}(C_2, C_3))$. But we can also use the first rule to derive $\mathtt{seq}(C_1, C_2) \xrightarrow{L} \mathtt{seq}(C'_1, C_2)$, and again to derive $\mathtt{seq}(\mathtt{seq}(C_1, C_2), C_3) \xrightarrow{L} \mathtt{seq}(\mathtt{seq}(C'_1, C_2), C_3)$. Taking C'_0 to be $\mathtt{seq}(\mathtt{seq}(C'_1, C_2), C_3)$, we see that we have indeed the required transition $C' \xrightarrow{L} C'_0$ with $C_0\ R\ C'_0$.
   - The second rule for `seq` can only be used when $C_1 = \mathtt{skip}$, and then we have $C_0 = \mathtt{seq}(C_2, C_3)$. But we can also use the second rule to derive $\mathtt{seq}(C_1, C_2) \xrightarrow{L} C_2$, and the first rule to derive $\mathtt{seq}(\mathtt{seq}(C_1, C_2), C_3) \xrightarrow{L} \mathtt{seq}(C_2, C_3)$. Taking C'_0 to be $\mathtt{seq}(C_2, C_3)$, we see that we have indeed the required transition $C' \xrightarrow{L} C'_0$ with $C_0\ R\ C'_0$.

That finishes the case analysis, showing that R is indeed a bisimulation, so that we can conclude the desired result: $\mathtt{seq}(C_1, \mathtt{seq}(C_2, C_3)) \sim \mathtt{seq}(\mathtt{seq}(C_1, C_2), C_3)$.

▷ Unit laws generally require unobservable transitions to be ignored.

We might hope to be able to prove that `skip` is a left and right unit for `seq` up to strong bisimilarity, i.e., $\texttt{seq}(\texttt{skip}, C) \sim C$ and $\texttt{seq}(C, \texttt{skip}) \sim C$. To show that this is impossible, simply consider $\texttt{seq}(\texttt{skip}, \texttt{skip}) \sim \texttt{skip}$, which is a special case of both unit laws. There is clearly no strong bisimulation R such that $\texttt{seq}(\texttt{skip}, \texttt{skip})\ R\ \texttt{skip}$, since the left side is an intermediate state, and the right side is a final state. We need to be able to ignore or insert unobservable transitions when relating states, which requires *weak bisimulation*.

The following notation for unobservable transitions, and for sequences of these, facilitates the definition of weak bisimulation:

Definition 4.13 (unobservable transitions)

- $S \longrightarrow S'$ *holds iff* $S \xrightarrow{L} S'$ *for some* $L \in Unobs$*;*
- $\longrightarrow^+$ *is the transitive closure of* $\longrightarrow$*; and*
- $\longrightarrow^*$ *is the reflexive transitive closure of* $\longrightarrow$*.*

Definition 4.14 (weak bisimulation) *A binary relation* $R \subseteq State \times State$ *is a* weak bisimulation *if whenever* $S_1\ R\ S_2$*:*

1. *if* $S_1 \longrightarrow^+ S_1'$*, then* $S_2 \longrightarrow^* S_2'$ *for some* S_2' *with* $S_1'\ R\ S_2'$*;*
2. *if* $S_1 \longrightarrow^* \xrightarrow{L} \longrightarrow^* S_1'$ *for some observable label* L*, then* $S_2 \longrightarrow^* \xrightarrow{L} \longrightarrow^* S_2'$ *for some* S_2' *with* $S_1'\ R\ S_2'$*;*
3. *if* $S_1 \in Final$ *then* $S_2 \longrightarrow^* S_1$*; and*
4. *the symmetric versions of the above three cases.*

Definition 4.15 (weakly bisimilar) S_1 *and* S_2 *are* weakly bisimilar*, written* $S_1 \approx S_2$*, if there exists a weak bisimulation* R *with* $S_1\ R\ S_2$*.*

Proposition 4.16 $\approx = \bigcup\{R \mid R \text{ is a weak bisimulation}\}$*, is itself a weak bisimulation, the largest weak bisimulation, and an equivalence relation.*

The following facts about weak bisimulation are sometimes sufficient for establishing simple equivalences:

Proposition 4.17 *If* $S \longrightarrow S'$ *then* $S \approx S'$*.*

Proposition 4.18 *If* $S \sim S'$ *and* $S' \approx S''$ *then* $S \approx S''$*.*

You are invited to check that the unit laws for `seq` can now be shown to hold up to weak bisimilarity. Most other laws for the abstract constructs for commands and expressions hold only up to weak bisimilarity too, since they usually involve some unobservable transitions.

Summary

In this chapter:

- We introduced *structural induction*, which requires all the specified rules to be structural, and illustrated its use to prove that that the dynamic semantics of our sublanguage of bc is deterministic;
- We mentioned the more general principle of *rule induction*, but left it without illustration, since it has no significant advantage over structural induction for the rules to be given in these lecture notes;
- We considered various *computation equivalences* between constructs, based on their computations; and
- We gave definitions of *strong* and *weak bisimulation*, and illustrated the careful but straightforward case analysis required in proofs of laws up to bisimilarity.

Exercises

Ex. 4.1 Consider the definition of the set A by the constructors for `BinTree` in Example 4.2. Let $h(t)$ denote the height of t (the leaves `zero` and `one` both have height 0). Let $P(t)$ be the property $z(t)+o(t) \leq 2^{h(t)}$. Use Structural Induction to prove that $P(t)$ holds for all t in A.

Ex. 4.2 Go through the proof given in Example 4.2, indicating what changes would be needed to transform it into a proof of the same property for the two sets and a subset inclusion specified in Example 4.3.

Ex. 4.3

(a) Specify sets `EvenTree` and `OddTree` such that the elements of `EvenTree` correspond to elements of `BinTree` having even height, and elements of `OddTree` correspond to elements of `BinTree` having odd height.

(b) Are any elements of `BinTree` missing in the union of `EvenTree` and `OddTree`? Justify your answer.

(c) Prove by SI that trees in `EvenTree` always have even height, and trees in `OddTree` always have odd height.

Ex. 4.4 Consider the representation of natural numbers given in Example 4.5. Let $n(t)$ be the natural number corresponding to t: $n(z) = 0, n(s(z)) = 1$, etc. Let $P(t)$ be the property $0 + \cdots + n(t) = n(t) \times (n(t) + 1)/2$. Prove by SI that $P(t)$ holds for all t in the set `Nat`.

Ex. 4.5 Consider the property $P(S)$ concerning the determinism of the transition relation from Example 4.6. Prove that three other constructors for commands preserve P.

Ex. 4.6 Let A and $\longrightarrow$ be as in Example 4.6.

(a) Formulate a property that expresses the lack of stuck states.

(b) Prove that `seq` preserves this property.

(c) Do all constructors of the specified language preserve this property? Justify your answer.

Ex. 4.7

(a) Prove Proposition 4.17.

(b) Prove Proposition 4.18.

Ex. 4.8 Prove the following bisimilarities for the dynamic semantics of bc constructs given in Chapter 1:

(a) $\texttt{cond-nz}(1, C) \sim \texttt{seq}(\texttt{skip}, C)$

(b) $\texttt{cond-nz}(0, C) \sim \texttt{skip}$

(c) $\texttt{seq}(\texttt{skip}, C) \approx C$

(d) $\texttt{seq}(C, \texttt{skip}) \approx C$

(e) $\texttt{cond-nz}(1, C) \approx C$

Chapter 5

Expressions

Section 5.1 considers the fundamental concepts of expressions, including constants, operations, and identifiers. It introduces various abstract syntactic constructs involving expressions, and relates them to the concrete syntax of ML. Most of the constructs can be just as easily related to other languages; several of them were included in the illustrative sublanguage of bc in Chapter 1.

Section 5.2 gives an MSOS module for each of the abstract constructs. It also provides a DCG mapping a simple sublanguage of ML to these abstract constructs. The combination of the MSOS modules defines an MSOS for the specified sublanguage of ML. Familiarity with ML should help understanding the intended interpretation of the abstract constructs; conversely, after becoming familiar with the abstract constructs, their relationship to ML constructs may give a deeper understanding of the latter.

The Prolog files generated from the specified MSOS modules, together with the DCG, allow ML expressions to be parsed and run according to their MSOS, as explained in Chapter 3. Although this prototype implementation of ML expressions is (very) inefficient, it does allow validation of the MSOS and the DCG.

All the MSOS modules given in this chapter are intended to be directly reusable in descriptions of many languages. In general, each language involves a different selection of abstract constructs. The subsequent chapters provide further constructs involving expressions; others are left as exercises.

The files specific to this chapter are available online in the MSOS directory at `Lang/ML/Exp`.

5.1 Fundamental Concepts

▷ Expressions are constructs which normally compute values.

```
E:   Exp
```

We shall always use the symbol `Exp` for the set of expressions, and `E` as a meta-variable ranging over it.

Expressions are found in most (if not all) high-level programming languages. They are usually distinguished from commands and declarations on the basis of the kind of values which they may compute: commands don't compute any values at all, and declarations compute bindings.

In fact ML does not distinguish syntactically between expressions and commands: the latter are represented as expressions which compute the fixed null value of the unit type. However, the interpretation of some of its constructs (e.g., the while loop) are characteristic for commands. In Java, assignments can compute the same values as expressions, and are best regarded as such. Many other languages make a strict syntactic distinction between expressions and commands.

▷ Expressible values may be *simple* or *structured*.

The values that can be computed by expressions, called the *expressible values*, usually include numbers (often of various types: integer, long integer, fixed-point, floating point, etc.) and the boolean truth-values. They may also include structured values, e.g., array and record values, and lists. In ML, they include functions too.

▷ Subsets of expressions compute particular kinds of values.

It is tempting to distinguish a subset of expressions for each kind of computed values: `BoolExp` for boolean-valued expressions, `IntExp` for integer-valued expressions, etc. However, this temptation should be firmly resisted, for two main reasons:

- it leads to an excessively large and open-ended set of subsets; and
- sometimes, the kind of value computed by an expression is not determined by its form, but depends on the context.

▷ Expressions often resemble mathematical terms.

Expressions usually allow constants, identifiers which can be bound to values,[1] and applications of arithmetic operations and predicates to the values of subexpressions. However, evaluation of a mathematical term involves evaluation of all its subterms, and the order in which the subterms are evaluated is of no significance. Evaluation of an expression in a programming language does not generally enjoy such properties:

▷ Expression evaluation need not involve evaluation of *all* subexpressions.

A prime example of this feature is conditional expressions: evaluation has to start with the condition, and continue with the selected alternative (thus always omitting the evaluation of one of the subexpressions). Similarly, evaluation of a conditional conjunction (expressed by '`andalso`' in ML) omits evaluation of the second argument when the first argument already determines the result.

▷ Expression evaluation may terminate *abnormally*, or *not terminate* at all.

Division by zero, or trying to access an unavailable component of a structured value, generally causes the evaluation of the enclosing expression to terminate abnormally, e.g., throwing an exception and skipping the remaining evaluations of subexpressions. Evaluation of an expression involving application of a user-defined function may fail to terminate at all.

▷ Expressions may have *side-effects*.

The assignment expressions of ML and Java are obviously intended to have effects on stored information. Other possible side-effects include input and output, and synchronization and communication with concurrent processes.

▷ Expression evaluation is *sequential* in ML.

In contrast to mathematical terms, ML insists on sequential, left-to-right evaluation of subexpressions (subject to skipping of subexpressions due to conditionals or abnormal termination). Whether this design feature is beneficial to ML programmers and/or implementers is debatable, but of no concern here.

[1] Identifiers that can be bound to values are usually referred to as 'variables' in mathematics, but we shall reserve this terminology for variables which can be assigned different values during computations.

5.1.1 Constant Values

▷ A constant is an expression which *always* has the same value.

For instance, a particular numeral in a decimal notation always evaluates to the same integer, regardless of where it occurs in the program; similarly for hexadecimal numerals, characters, strings, etc. To avoid having to bother with pedantic details of how to interpret such low-level notations, we shall assume that all constants are replaced by their values in the mapping from concrete to abstract syntax. We include the corresponding sets of values directly in `Exp`.

▷ Constant values usually include numbers.

```
Exp ::= Integer
```

In fact ML allows various numeric types apart from integers, e.g., long integers and real numbers. Conventional decimal and hexadecimal notation can be used for integers—except that negative numbers are written with a tilde '~' instead of the usual minus sign.

▷ The boolean truth-values are naturally constants.

```
Exp ::= Boolean
```

In ML, '`true`' and '`false`' are actually treated as identifiers with fixed bindings—presumably for uniformity with user-defined types of values. In most other languages, '`true`' and '`false`' are reserved words.

▷ Other types of values can have constants too.

For instance, '`()`' in ML is the constant for the only value of the unit type, and '`[]`' can be regarded as a constant for the empty list (it can also be expressed by the identifier '`nil`', which has a fixed binding). The only remaining constants in ML are the string and character constants (the latter being written '`#STR`', where `STR` is a single-character string constant).

5.1.2 Operations

▷ An operation has a *fixed* interpretation as a function.

```
O:   Op
```

An operation is similar to a constant, in that its interpretation does not vary. However, it is only in higher-order functional programming languages such as ML that operations can themselves be used as expressions. We shall return to this issue in Chapter 8, when we consider function declarations and abstractions.

▷ An operation can be applied to values of argument expressions.

```
Exp ::= app(Op,Arg)
```

As in a mathematical term, the arguments are evaluated first, then the operation is applied to the values. Thus conditional choice is *not* regarded as an operation, and similarly for other 'lazy operations' such as '`andalso`' in ML. (In fact one could do it: by forming parameterless functions from the expressions whose evaluation may get delayed. This would however be more complicated than a straightforward description of conditionals.)

▷ Multi-argument operations can be regarded as unary operations on tuples.

ML is atypical in treating tuples of expressions as ordinary expressions. Nevertheless, we shall do something similar in our abstract syntax here. Thus when going from concrete to abstract syntax for an application of an operation, we will form a tuple from the argument expressions, and an application of the operation to that tuple.

Consequently, for ML, the set `Arg` will be simply `Exp`, and the tuples of expressions will themselves be expressions. For other languages (such as the sublanguage of bc in Chapter 1) we can let `Arg` include tuples of expressions, and avoid the introduction of tuples in `Exp`. Notice that whereas tuple expressions are naturally nestable, tuples of argument expressions need not involve any nesting at all. By distinguishing between `Exp` and `Arg` we keep our options open.

It is often easier to describe something which is *general* and *uniform*, compared to something which is more restricted and non-uniform. Generalization of abstract syntax constructs is a powerful technique for simplifying semantic descriptions. In the case of tuples, it is just as easy to specify nestable tuple expressions as non-nestable tuples of argument expressions.

▷ Predicates are regarded as boolean-valued operations.

We use `true` to represent that a predicate holds of its argument values, and `false` to represent that it doesn't.

The syntax of ML actually has very few operations and predicates: symbols such as '=' and '+' are identifiers, bound to the actual operations, and all but a few of them can be redeclared with new interpretations. In many other languages, all the usual arithmetical and relational operations are treated as such, and cannot be redefined.

▷ Operations on given sets can be included as abstract constructs, as can constructors and derived operations.

It would be tedious to have a separate module for including each operation in the abstract syntax of a language (although we did this for our simple sublanguage of bc in Chapter 1). We shall simply allow all the symbols for functions defined on our given sets (see Chapter 2) to be included in `Op`. The same goes for the constructors of specified sets.

Sometimes, it is possible to derive a required operation in terms of those that are already available. For instance, we already have a binary operation `nth(L,N)` on lists, and `list` maps tuples to lists, so we can obtain the effect of selecting the Nth component of a tuple `T` by the combination `nth(list(T),N)`.

5.1.3 Identifiers

▷ Identifiers may be already bound to values.

```
I: Id
```

```
Exp ::= Id
```

In contrast to constants and operations, the value of an identifier generally depends on its context. The values determined for identifiers at any particular point in a computation are called the *current bindings*, and map each bound identifier to its bound value. The map itself is called the *current environment*.

▷ Identifiers may have initial bindings.

In ML, the top-level environment provides a large number of initial bindings for identifiers. As mentioned above, symbols such as '`true`', '`=`' and '`+`', which would be constants and operations in most languages, are simply identifiers in ML, so we need to be able to form applications of identifiers to arguments:

```
Exp ::= app(Id,Arg)
```

Identifiers which cannot be redefined correspond essentially to constants and operations.

▷ Declarations can provide or override bindings for identifiers.

In the present chapter, the current bindings remain the same throughout evaluations of subexpressions. In the next chapter, we shall consider local declarations, which allow the current bindings to be changed.

5.1.4 Tuples

▷ A tuple expression collects together the values of its components.

```
Exp ::= tup(Exp*)
```

The null value of the unit type in ML can be regarded as the empty tuple. Tuples with a single component do not occur directly in ML, but we shall need them for our treatment of lists below. For tuple expressions with more than one subexpression, we need to be able to reflect sequential evaluation of the components:

▷ Evaluation of the component expressions may be sequential.

```
Exp ::= tup-seq(Exp*)
```

The above construct is the sequential variant of tuple expressions: the subexpressions are evaluated from left to right.

▷ Selection of a particular component is an operation in ML.

```
Op ::= nth(Integer)
```

Selection on tuples is one of the few operations that are not treated as identifiers in the syntax of ML. It is written '`#N`', where `N` is a positive integer.

▷ Lists in ML can be represented as flat trees.

```
Op ::= list
```

Regarding '`list`' as an additional operation in abstract syntax, the ML expression '`[E1,...,En]`' is treated as the application '`app(list,tup-seq(E1,...,En))`'. When the value of each `Ei` is `Vi`, the result of the application is the value `list(V1,...,Vn)`'. Thus conceptually, we represent lists as flat trees with unbounded numbers of components.

In ML, the empty list '`list()`' is the value bound to the identifier '`nil`'. The operation bound to the identifier '`::`' is treated as a constructor, whereas here, we shall interpret it as the operation '`cons`' on the given set of lists. It would be closer to implementations of ML if we were to introduce a new set with `nil` and `cons` as constructors, and map '`[E1,...,En]`' to `cons(E1,...,cons(En,nil)...)`. This would have some minor advantages in connection with the semantics of patterns, but in general, it seems better not to let implementation details influence abstract semantic descriptions.

5.1.5 Conditionals

▷ A conditional expression is a special case of a generic conditional construct.

```
Exp ::= cond(Exp,Exp,Exp)
```

The abstract syntax and semantics of conditional expressions are analogous to those of conditional commands with alternative branches. In both cases, the primary construct involves a boolean-valued expression as condition. It is possible to parametrize the MSOS specification of constructs to exhibit such genericity, but let us not bother with this in these introductory notes, since the effort of specifying conditional expressions and commands separately is negligible.

▷ A conditional construct may also be regarded as a choice between cases.

Although a conditional expression might seem to be a primitive construct, it can also be regarded as a case selection: each case is essentially a partial function which 'refuses' to be applied to arguments outside its domain. This is actually the way that conditional expressions are implemented in ML: they are simply abbreviations (i.e., 'syntactic sugar') for case-expressions, where the alternative 'rules' match true and false. However, it seems more straightforward and perspicuous to regard conditional expressions as primitive.

5.2 Formal Semantics

Let us now introduce the MSOS modules that define the dynamic semantics of the abstract constructs considered above. At the end of the chapter, we shall give a DCG that maps concrete ML expressions into combinations of these abstract constructs, thus providing the complete dynamic semantics of a sublanguage of ML.

In subsequent chapters, we shall extend the sublanguage to include declarations, commands, and function abstractions. We shall also extend it with a simple form of concurrent processes, and consider types and static semantics. Thanks to the modularity of MSOS, we shall not need to revise the modules given in the present chapter when making such extensions.

▷ Expressions get replaced by their computed values.

Module 5.1 Cons/Exp

```
1  State ::= Exp
2
3  Final ::= Value
4
5  Exp ::= Value
```

Recall that in MSOS, constructs occurring in states normally get replaced by their computed values during the course of a computation. `Value` is the set of expressible values:

```
V : Value
```

Although we normally include particular subsets of `Value` directly in `Exp` in connection with abstract syntax, we must ensure that all remaining expressible values are included in `Exp` too.

5.2.1 Constant Values

▷ Constants are assumed to be already evaluated.

For each set of constant values, we have to specify its inclusion in `Value`. For example:

Module 5.2 Cons/Exp/Integer

```
1  Value ::= Integer
```

This ensures that the set is also included in `Final`. The semantics of a constant is simply the empty computation that starts from it.

5.2.2 Operations

▷ The semantics of an operation is specified in connection with its application.

Module 5.3 Cons/Exp/app-Op

```
1  Passable ::= Value
2
3  Value ::= tup(Value*)
4
5             ARG --{...}-> ARG'
6  ------------------------------------
7  app(O,ARG):Exp --{...}-> app(O,ARG')
8
9          O(V*) = V'
10 --------------------------
11 app(O,tup(V*)):Exp ---> V'
12
13       O(V) = V'
14 --------------------
15 app(O,V):Exp ---> V'
16
17 /*HACK*/  O(V1,V2) = V'
18 ----------------------------
19 app(O,tup(V1,V2)):Exp ---> V'
```

Each element of `Op` should be a known operation on given sets (e.g., addition on the integers), a constructor of a specified set (not illustrated here), or an operation defined in terms of other operations (e.g., `nth`). An operation symbol may be *overloaded*, which corresponds to being a partial function on the union of all sets of values. The operands of an operation form a sequence of values: a single value is a sequence of length one, a pair of values is a sequence of length two. The result of applying an operation (when defined) is assumed to be a single value.

Sequences of values are quite similar in some respects to tuples of values. However, sequences cannot be components of other sequences, whereas tuples can be components of other tuples. The operation `tup` forms a tuple from the sequence of its components; the tuple is also a sequence of length one, but distinct from sequences of other lengths.

Thus in the semantics of applying an operation, we not only need to ensure that the arguments are evaluated to a tuple, but also to apply the operation to the sequence of components of the tuple. Fortunately, this can be specified for all operations uniformly, with a minimum of bother, as in Module 5.3 above.

5.2.3 Identifiers

▷ The semantics of an identifier requires that labels on transitions provide the current environment.

Module 5.4 Cons/Id

```
State ::= Id

Final ::= Bindable

Id ::= Bindable

Label = {env:Env,...}

   lookup(I,ENV) = BV
-------------------------
I:Id --{env=ENV,---}-> BV

  not def lookup(I,ENV),
     init-env = ENV',
  lookup(I, ENV') = BV
-------------------------
I:Id --{env=ENV,---}-> BV
```

An environment is simply a map representing a set of bindings:

```
ENV: Env = (Id,Bindable)Map
```

The operation `lookup(I,ENV)` has an undefined result when `ENV` does not include a binding for `I`.

The environment `init-env` is left open here. It could be the empty map, in which case the second rule above for evaluating an identifier to its bound value becomes inapplicable.

Module 5.5 Cons/Exp/Id

```
1  |    I ---> BV
2  |-------------
3  |I:Exp ---> BV
```

The evaluation of an identifier that occurs as an expression involves first obtaining the value bound to the identifier; if the bound value is an element of the set `Value` of expression values, this is already the computed value (although other cases are possible, e.g., reading the contents of a storage cell, or applying a parameterless function to no arguments).

Another context in which an identifier has to be evaluated is in an application:

Module 5.6 Cons/Exp/app-Id

```
1  |             I ---> I'
2  |------------------------------
3  |app(I,ARG):Exp ---> app(I',ARG)
4  |
5  |           ARG --{...}-> ARG'
6  |-----------------------------------
7  |app(I,ARG):Exp --{...}-> app(I,ARG')
```

This covers the case that the identifier is bound to an operation, since the rules given in Module 5.3 above for applying an operation can continue from the state `app(O,ARG)`.

5.2.4 Tuples

▷ The components of a tuple expression may be evaluated in any order.

Module 5.7 Cons/Exp/tup

```
1  Value ::= tup(Value*)
2
3   E+ = (E1*,E2+), E2+ = (E,E3*),
4              E --{...}-> E',
5  (E',E3*) = E2'+, (E1*,E2'+) = E'+
6  --------------------------------
7    tup(E+):Exp --{...}-> tup(E'+)
```

The conditions of the rule extract any component expression `E` from the sequence of expressions `E+` (since either or both the sequences `E1*` and `E2*` can be empty), let it make a transition, and construct the resulting sequence `E'+`.[2]

Notice the requirement that a tuple of expression values should be itself an expression value. Once all the components of the original tuple have been evaluated, we are done.

▷ The semantics of a sequential tuple expression involves an extra step.

Module 5.8 Cons/Exp/tup-seq

```
1  Value ::= tup(Value*)
2
3    E+ = (V1*,E2+), E2+ = (E,E3*),
4                E --{...}-> E',
5    (E',E3*) = E2'+, (V1*,E2'+) = E'+
6  ------------------------------------
7  tup-seq(E+):Exp --{...}-> tup-seq(E'+)
8
9  tup-seq(V*):Exp ---> tup(V*)
```

Clearly, we must ensure that evaluation of components doesn't start evaluating later components before all the earlier component expressions have been replaced by their values. Simply replacing `E1*` in the corresponding rule for `tup` by `V1*` is all that is needed.

[2] A future version of the software for translating MSDF to Prolog may allow the use of sequence patterns also in the states of rules.

5.2.5 Conditionals

▷ The semantics of conditional constructs is particularly straightforward to specify.

Module 5.9 Cons/Exp/cond

```
1  Value ::= Boolean
2
3                  E --{...}-> E'
4  ----------------------------------------
5  cond(E,E1,E2):Exp --{...}-> cond(E',E1,E2)
6
7  cond(true,E1,E2):Exp ---> E1
8
9  cond(false,E1,E2):Exp ---> E2
```

That concludes the illustrations of how to specify the MSOS of various abstract constructs involving expressions.

5.2.6 ML Expressions

▷ We might have made mistakes in our MSOS specifications.

How can we ensure that we haven't made any mistakes? We can check that our modules are valid MSDF specifications by parsing each of them and translating them to Prolog using the software accompanying these notes.[3] We might also check that some simple abstract syntax trees have the expected computations: either empirically, by tracing the Prolog computations, or mathematically, by proving facts about the transition relation defined by the collection of rules. Unfortunately, that could be quite tedious, and quite error-prone in itself.

▷ Concrete syntax from a familiar language can help.

An alternative approach to validation is to specify a precise mapping from concrete programs in a familiar programming language to the abstract constructs, as illustrated for a simple sublanguage of bc in Chapter 1. Provided that the concrete programs involve sufficiently general combinations of the abstract constructs, and

[3] Future versions of the translation should make more stringent checks on the MSDF, e.g., to warn about variables that might be used before they have values.

that running the corresponding abstract syntax trees using the Prolog code generated from the rules gives the expected results (i.e., computed values, in the case of expressions), we can establish a high degree of confidence in at least the completeness and correctness of our rules. At the same time, we can quickly detect and localize any problems with the specifications.

▷ We shall define some simple sublanguages of ML.

In these notes, we shall focus on simple sublanguages of Standard ML for concrete syntax. The abstract constructs used to illustrate MSOS are chosen accordingly. Our aim is *not* to cover all the trickier details of the actual semantics of Standard ML, but rather to focus on the main features. The reader should be sufficiently familiar with ML to read (small) test programs and immediately grasp the expected results; some experience with writing ML programs will be useful for many of the exercises.

▷ We use MSDF to specify the required abstract constructs.

Thanks to modularity, all we need to do is refer to the names of the modules to obtain the complete abstract syntax that forms the target of our mapping. For the dynamic semantics, we need to specify also which data operations are included in `Op`, and the initial environment.

Module 5.10 Lang/ML/Exp

```
1  see     Prog, Prog/Exp
2
3  see     Exp, Exp/Boolean, Exp/Integer, Exp/Character, Exp/String,
4          Exp/Id, Exp/cond, Exp/app-Op, Exp/app-Id,
5          Exp/tup, Exp/tup-seq
6
7  see     Arg, Arg/Exp
8
9  see     Op
10
11 see     Id
12
13 Value ::= (Value)List
14
15 Bindable ::= Value | Op
16
17 Op ::= cons | = | abs | + | - | * | div | mod | < | > | =< | >= | nth
18
19 Passable ::= Value
```

```
20
21  init-env =
22         {id(true)  |-> true,
23          id(false) |-> false,
24          id(nil)   |-> [ ],
25          id('::')  |-> cons,
26          id('=')   |-> (=),
27          id('~')   |-> (-),
28          id(abs)   |-> abs,
29          id('+')   |-> (+),
30          id('-')   |-> (-),
31          id('*')   |-> (*),
32          id(div)   |-> div,
33          id(mod)   |-> mod,
34          id('<')   |-> (<),
35          id('>')   |-> (>),
36          id('<=')  |-> (=<),
37          id('>=')  |-> (>=)}
```

▷ We use a DCG to specify concrete syntax and its mapping to abstract constructs.

Since it is completely straightforward to recover a context-free grammar from the DCG used to specify a mapping from concrete to abstract syntax, we shall specify only the latter.

Module 5.11 Lang/ML/Exp/SYN

```
9   :- ensure_loaded('Lang/Lex/LEX.pro').
10
11  % Programs
12
13  prog(E:'Prog') --> i, exp(E), i.
14
15  % Expressions
16
17  exp(E) -->
18          infexp(E).
19
20  exp(cond(E1,E2,E3)) -->
21          "if", i, exp(E1), i, "then", i,
22          exp(E2), i, "else", i, exp(E3).
23
24  exp(cond(E1,E2,false)) -->
25          atexp(E1), i, "andalso", i, exp(E2).
26
```

```
27 exp(cond(E1,true,E2)) -->
28         atexp(E1), i, "orelse", i, exp(E2).
29
30 % Infix expressions:
31
32 infexp(E) -->
33         appexp(E).
34
35 infexp(app(I,tup_seq([E1,E2]))) -->
36         atexp(E1), i, vid(I), i, atexp(E2).
37
38 % Application expressions:
39
40 appexp(E) -->
41         atexp(E).
42
43 appexp(app(I,E)) -->
44         vid(I), i, atexp(E).
45
46 appexp(app(nth,tup([app(list,E),N]))) -->
47         selop(nth(N)), i, atexp(E).
48
49 % Atomic expressions:
50
51 atexp(C) -->
52         scon(C).
53
54 atexp(I) -->
55         vid(I).
56
57 atexp(I) -->
58         "op", i, vid(I).
59
60 atexp(tup(null)) -->
61         "(", i, ")".
62
63 atexp(E1) -->
64         "(", i, exp(E1), i, ")".
65
66 atexp(tup_seq([E1|ES2])) -->
67         "(", i, exp(E1), ",", i, exps(ES2), i, ")".
68
69 atexp(list([])) -->
70         "[", i, "]".
71
72 atexp(app(list, E)) -->
73         "[", i, exp(E), i, "]".
74
75 atexp(app(list, tup_seq([E1|ES2]))) -->
```

```
76          "[", i, exp(E1), ",", i, exps(ES2), i, "]".
77
78  % Expression lists:
79
80  exps([E|ES]) -->
81          exp(E), i, ",", i, exps(ES).
82
83  exps([E]) -->
84          exp(E).
85
86  % Operations:
87
88  selop(nth(N)) -->
89          "#", integer(N).
90
91  % Constants:
92
93  scon(N) -->
94          integer(N).
95
96  scon(N1) -->
97          "~", integer(N), {N1 is -N}.
98
99  scon(list(L)) -->
100         string(L).
101
102 scon(C) -->
103         "#", string([C]).
104
105 % Identifiers:
106
107 reserved("andalso").
108 reserved("else").
109 reserved("if").
110 reserved("op").
111 reserved("orelse").
112 reserved("then").
113
114 vid(id(I)) --> alpha(C), optcsyms(L), lookahead(C1),
115   {\+char_type(C1, csym), \+reserved([C|L]), name(I, [C|L])}, !.
116
117 vid(id(I)) --> symbols(L), lookahead(C1),
118   {\+member(C1,"!%&$#+-/:<=>?@\\~`^|*"), \+reserved(L), name(I, L)}, !.
119
120 symbols([C|L]) --> symbol(C), optsymbols(L).
121
122 optsymbols(L)   --> symbols(L).
123 optsymbols([]) --> "".
124
```

```
125 symbol(C) --> [C], {member(C,"!%&$#+-/:<=>?@\\~`^|*")}.
126
127 % Integers:
128
129 integer(N) --> digits(L), !, {number_chars(N, L)}.
130
131 % Strings:
132
133 string(L) --> "\"", optchars(L), "\"".
134
135 optchars(L) --> chars(L).
136 optchars([]) --> "".
137
138 chars([C|L]) --> char(C), optchars(L).
139
140 char(C) --> graph(C), {C \= '"'}.
141 char(' ') --> " ".
142
143 % Layout and comments:
144
145 i --> optignores.
146
147 optignores --> ignore, !, optignores.
148 optignores --> "".
149
150 ignore --> whites.
151 ignore --> newline.
152 ignore --> "(*", up_to("*)").
153
```

You are encouraged to try all the exercises.

Summary

In this chapter:

- We distinguished between the fundamental concepts of expressions, constants, operations, and identifiers.
- We introduced a modest number of abstract constructs, sufficient to represent the abstract syntax of some simple expressions of ML.
- We also introduced some notation for auxiliary sets of values, including expressible and bindable values, bindings, and data constructors for tuples and lists.

- Finally, we specified the MSOS of all the abstract constructs, and gave a DCG mapping a simple expression sublanguage of ML to the abstract constructs, thereby defining its MSOS.

Exercises

***Ex. 5.1** Is it possible to specify the derivation of operations (such as selecting the Nth component of a tuple) in MSDF? If so, give an example; if not justify your answer.

Ex. 5.2

*(a) List all the identifiers which have fixed top-level bindings in Standard ML.

(b) Are there any operations in Standard ML, other than `#N` (and the corresponding selector for record components), which are not denoted by identifiers?

Ex. 5.3 Discuss the pros and cons of the illustrated representation of lists, comparing it with the representation used in implementations of ML.

Ex. 5.4

(a) Propose abstract constructs that represent case-expressions.

(b) Derive `cond(E,E1,E2)` from your abstract constructs for case expressions.

***Ex. 5.5** List all the sets of values that would have to be included in `Value` when giving an MSOS of all of Standard ML.

Ex. 5.6 Explain why the third rule in Module 5.3 is needed.

Ex. 5.7 Consider Module 5.4.

(a) Why is it necessary to include `Bindable` in `Id`?

(b) Why is the condition `not def lookup(I,ENV)` needed?

Ex. 5.8 Consider Module 5.7. Reformulate the rule for `tup(E+)` more concisely in MSDF. Investigate whether the revised module can be successfully translated to Prolog by the current version of the software accompanying these notes.

Ex. 5.9 Consider the definition of `init-env` in Module 5.11.

(a) List some further constant and function identifiers that should be included to facilitate writing test programs in our expression sublanguage of ML.

(b) Estimate how many identifiers would have to be added to provide all the top-level constants and functions of Standard ML.

Ex. 5.10 Using the Prolog code accompanying these notes, validate the MSOS of the ML expressions specified in this chapter.

Ex. 5.11 Consider an alternative construct for binary operation application, with abstract syntax:

```
Exp ::= app(Op,Exp,Exp).
```

Specify rules for the evaluation of `app(O,E1,E2)` such that:

(a) the evaluations of `E1` and `E2` are sequential, left to right;

(b) the evaluations of `E1` and `E2` are interleaved; or

(c) as (b), except that if one of the evaluations computes a zero for the operation, the other evaluation may be abandoned.

Ex. 5.12 Consider the following constructs, corresponding directly to SML's concrete syntax '`E1 andalso E2`' and '`E1 orelse E2`':

```
Exp ::= cond-conj(Exp,Exp) | cond-disj(Exp,Exp)
```

(a) Give MSOS rules for these constructs.

(b) Validate your rules by translating them to Prolog and testing them (changing the DCG to generate the new constructs).

(c) Are there any significant advantages of introducing and using these abstract constructs, compared to the technique used in Module 5.11? Motivate your answer.

Chapter 6

Declarations

This chapter focusses on the fundamental concepts of declarations: *bindings* of identifiers to values, and the computation and *scopes* of these bindings. Section 6.1 introduces several abstract constructs for declarations, and relates them informally to the concrete syntax of ML. Section 6.2 gives the MSOS of the abstract constructs. It also extends the concrete sublanguage of ML specified in Chapter 5 with declarations. Recursive declarations are deferred to Chapter 8, where they are considered in connection with so-called abstractions.

Sets of bindings are represented by *environments*. Declarations compute environments,[1] and the label on a transition has an environment as a fixed component (as in Chapter 5). Although the environment remains fixed for adjacent steps during the computation of a particular construct, the environment for steps for component constructs can be adjusted to reflect local bindings. The MSOS rules show clearly the relationship between the outer environment and the inner one.

The files specific to this chapter are available online in the MSOS directory at `Lang/ML/Dec`.

6.1 Fundamental Concepts

▷ Declarations are constructs which compute *bindings*.

```
D: Dec
```

In most programming languages, including ML, declarations are clearly distinguished from expressions, and bindings are not included in the set of values computed by expressions.

[1] Informally, we may also say that a declaration computes *bindings*, thinking of the set of bindings represented by the environment.

▷ A binding is an association between an identifier and a bindable value.

The association in a binding is one-way: from the identifier, to the value to which it is bound. In ML, all expressible values are bindable, but this quite exceptional: in most programming languages, the difference between the sets of bindable values and expressible values is quite large, and reveals fundamental features of the language design. For instance, functions are often bindable but not expressible, and constant values are sometimes not bindable.

▷ The *scope* of a binding is the part of a program where it is *current*.

The scope of a binding is the part of a program where the identifier may be used to refer to the associated value. In most languages, the outer limits of the scope of a binding are determined by the context-free structure of the program. For instance, the scope of the binding of `x` to the value 0 below has the scope indicated by '...'.

```
let val x=0 in ... end
```

Holes in the scope of a binding arise where it is *overridden* by another binding for the same identifier, e.g.:

```
let val x=0 in ... let x=1 in ... end ... end
```

▷ *Binding occurrences* of identifiers are distinguished from *bound* and *free* occurrences.

A binding occurrence of an identifier is where it is used to *create* a binding, and a bound occurrence is where it is used to *refer* to a binding. For instance, the first occurrence of '`x`' below is a binding occurrence, whereas the second is a bound occurrence:

```
... let val x=0 in ...x... end ...
```

A non-binding occurrence is called *free* in some construct when it is does not refer to a binding provided in that construct. For instance, assuming that there is no binding occurrence of '`y`' in '`let val x=0 in ...y... end`', its occurrence is free in the let-expression. In general, complete programs do not have any free occurrences of identifiers: each non-binding occurrence refers either to a binding given in the program, or to an initial binding determined by the language. A part of a program that has no free occurrences of identifiers is said to be *closed*.

▷ The same identifier can sometimes refer to more than one binding in the same part of the program.

This can happen when the context of the bound occurrence of an identifier determines which of the available bindings it refers to. In ML, for instance, the same identifier could be bound simultaneously to a value and to a module (i.e., structure or signature), but it is always clear from the context of a bound occurrence just which of the two bindings is concerned.

6.1.1 Value Declarations

▷ A value declaration binds an identifier to the value of an expression.

```
Dec ::= bind(Id,Exp)
```

A value declaration `bind(I,E)` is written '`val I=E`' in ML. The declaration computes the binding of `I` to the value of `E`, but does not itself affect the scopes of bindings. In particular, the current bindings are also those used for the evaluation of `E`, so any free reference to `I` in `E` has to refer to these current bindings: simple value declarations are not recursive.

Modelling the binding of an identifier to a value does not entail representing how the value might be *stored*, although in some imperative languages (such as C) value identifiers are regarded as variables that cannot be updated.

6.1.2 Local Declarations

▷ Local declarations may override nonlocal bindings, and have limited scope.

```
Exp ::= local(Dec,Exp)
```

```
Dec ::= local(Dec,Dec)
```

The concept of a local declaration is a generic one, and can be instantiated with expressions, declarations, and other kinds of constructs. A local declaration expression '`local(D,E)`' is written '`let D in E end`' in ML, whereas the corresponding declaration construct '`local(D1,D2)`' is written '`local D1 in D2 end`'.

The declaration '`local(D,E)`' restricts the scope of the bindings computed by `D` to `E`. The nonlocal bindings (i.e., those current before the declaration) can be referenced in `D`; and those which are not overridden by the bindings computed by `D` can also be referenced in `E`. For instance, consider this ML expression:

```
let val c = 1
in  let val c = c+1
    in  c+1
    end
end
```

The first occurrence of 'c+1' is in the scope of the binding of c to 1, and the second in the scope of the binding of c to 2.

Regarding flow of control, a local declaration is sequential: the computation of the local bindings by D has to be completed before the computation for E starts. It computes whatever E computes, and similarly when the scope of the local bindings is itself a declaration.

6.1.3 Accumulating Declarations

▷ Accumulating declarations may override previous bindings, and have unlimited scope.

```
Dec ::= accum(Dec,Dec)
```

An accumulating declaration 'accum(D1,D2)' is written 'D1;D2' in ML (in fact the semicolon is optional). Although it might not be apparent, this construct is closely related to 'local(D1,D2)', in that the scope of the bindings computed by D1 includes D2, and that D2 may refer to the current bindings prior to D1, except when these are overridden. The only difference is that whereas the local declaration finishes by discarding the bindings computed by D1, and computes just the same bindings as D2, the accumulating declaration combines the bindings computed by D1 and by D2 (letting the latter override the former), which means that their scope is not restricted by the construct. However, the combination 'local(accum(D1,D2),E)' restricts the scope of both bindings to E, and should be equivalent to 'local(D1,local(D2, E))' (but of course not to 'local(local(D1,D2),E)').

6.1.4 Simultaneous Declarations

▷ Simultaneous declarations compute bindings independently.

```
Dec ::= simult(Dec,Dec)
```

In this construct, the computations of the bindings by the declarations may be interleaved. It has a sequentialized variant:

```
Dec ::= simult-seq(Dec,Dec)
```

A sequentialized simultaneous declaration '`simult-seq(D1,D2)`' is written '`D1 and D2`' in ML.

These constructs *exclude* `D2` from the scope of the bindings computed by `D1` (and vice versa): both `D1` and `D2` can refer to the current bindings, which remain unchanged. As in an accumulating declaration, the bindings computed by `D1` and `D2` are subsequently combined, but here the natural default is to use a symmetrical combination: union. If the domains of the bindings to be combined overlap, their union is undefined. To let the bindings computed by `D2` override those computed by `D1` would give further variants.

For example, consider the following ML expression:

```
let val c = 1
in  let val c = c+1 and d = c+1
    in  c+d
    end
end
```

In both occurrences of '`c+1`', `c` refers to the binding of `c` to 1.

6.2 Formal Semantics

The foregoing informal explanation of fundamental concepts and constructs attempted to be precise and complete, but clearly it is not easy to be sure that this has been achieved. In contrast, we shall see that the formal semantics of these constructs is remarkably straightforward.

▷ Declarations compute environments.

Module 6.1 Cons/Dec

```
1  State ::= Dec
2
3  Final ::= Env
4
5  Dec ::= Env
```

As in Chapter 5, environments are maps:

```
ENV: Env = (Id,Bindable)Map
```

We shall need to use various operations on maps in connection with the semantics of declarations:

- `I |-> BV` is the map taking `I` to `BV`, and otherwise undefined. As in the definition of `init-env` at the end of Chapter 5, we may also specify a larger map in a concise way: `{I1 |-> BV1,..., In |-> BVn}`.
- `ENV1 / ENV2` is the mapping obtained by letting `ENV1` override `ENV2`.
- `ENV1 + ENV2` is the union of the mappings `ENV1`, `ENV2`, being undefined whenever the domains of the mappings overlap.
- `ENV - IS` is the mapping obtained from `ENV` by removing any value for the identifiers in the set `IS`.

6.2.1 Value Declarations

▷ A value declaration binds an identifier to the value of an expression.

Module 6.2 Cons/Dec/bind

```
1  |             E --{...}-> E'
2  |---------------------------------
3  |bind(I,E):Dec --{...}-> bind(I,E')
4  |
5  |bind(I,BV):Dec ---> {I |-> BV}
```

The above construct does *not* require that all values computed by expressions `E` are bindable: the second rule simply cannot be instantiated when the value computed by `E` is not in `Bindable`. The exact relationship between the sets of expressible and bindable values is left open by our reusable modules. In an MSOS of ML, we shall include `Value` in `Bindable`, reflecting that all values are so-called 'first-class citizens', having the same rights.

6.2.2 Local Declarations

▷ The environment component of a label always represents the current bindings.

Module 6.3 Cons/Exp/local

```
1  Label = {env:Env,...}
2
3                D --{...}-> D'
4  ------------------------------------
5  local(D,E):Exp --{...}-> local(D',E)
6
7     (ENV/ENV0) = ENV',  E --{env=ENV',...}-> E'
8  ------------------------------------------------
9  local(ENV,E):Exp --{env=ENV0,...}-> local(ENV,E')
10
11 local(ENV,V):Exp ---> V
```

The current environment for `E` is the current environment `ENV0` for '`local(D,E)`', overridden by the environment `ENV` computed by `D`. After the computation for `E` has terminated with final state `V`, the local bindings are no longer needed, and the last rule above eliminates them. For instance, `V` might be a number, which is obviously independent of any bindings. In Chapter 8 we shall see how to deal with a less obvious case, where `E` is a function abstraction with free identifiers.

We see that although the environment remains fixed for adjacent steps of a computation, it can be adjusted as required to reflect changes to the current environment in connection with transitions for *components*.

N.B. The condition `(ENV/ENV0)=ENV'` is a mathematical equation, *not* an assignment command! It simply names the result of applying the overriding operation to `ENV` and `ENV0`.[2]

The rules for '`local(D1,D2)`' are completely analogous, but let us include them for the sake of completeness:

[2]The current version of the software accompanying these notes requires the results of applying all operations (other than constructors) to be named.

Module 6.4 Cons/Dec/local

```
Label = {env:Env,...}

              D1 --{...}-> D1'
----------------------------------------
local(D1,D2):Dec --{...}-> local(D1',D2)

   (ENV1/ENV0) = ENV',   D2 --{env=ENV',...}-> D2'
-------------------------------------------------------
local(ENV1,D2):Dec --{env=ENV0,...}-> local(ENV1,D2')

local(ENV1,ENV2):Dec ---> ENV2
```

6.2.3 Accumulating Declarations

▷ The semantics of accumulating declarations is closely related to that of local declarations.

Assuming that one has understood the above rules for local declarations, the rules for accumulating declarations specified in the following module should be clear too:

Module 6.5 Cons/Dec/accum

```
Label = {env:Env,...}

              D1 --{...}-> D1'
----------------------------------------
accum(D1,D2):Dec --{...}-> accum(D1',D2)

   (ENV1/ENV0) = ENV',   D2 --{env=ENV',...}-> D2'
-------------------------------------------------------
accum(ENV1,D2):Dec --{env=ENV0,...}-> accum(ENV1,D2')

     (ENV2/ENV1) = ENV
-----------------------------
accum(ENV1,ENV2):Dec ---> ENV
```

The only difference from the rules for '`local(D1,D2)`' is in the last rule above.

6.2.4 Simultaneous Declarations

▷ The computations of simultaneous declarations may be interleaved.

Since the current bindings are unaffected by simultaneous declarations, their computations can be interleaved:

Module 6.6 Cons/Dec/simult

```
1   |                D1 --{...}-> D1'
2   |-----------------------------------------
3   |simult(D1,D2):Dec --{...}-> simult(D1',D2)
4   |
5   |                D2 --{...}-> D2'
6   |-----------------------------------------
7   |simult(D1,D2):Dec --{...}-> simult(D1,D2')
8   |
9   |     (ENV1 + ENV2) = ENV
10  |-----------------------------
11  |simult(ENV1,ENV2):Dec ---> ENV
```

The rules for the sequential variant '`simult-seq(D1, D2)`' differ only in having `ENV1` instead of `D1` in the second rule:

Module 6.7 Cons/Dec/simult-seq

```
1   |                   D1 --{...}-> D1'
2   |-------------------------------------------------
3   |simult-seq(D1,D2):Dec --{...}-> simult-seq(D1',D2)
4   |
5   |                   D2 --{...}-> D2'
6   |-------------------------------------------------------
7   |simult-seq(ENV1,D2):Dec --{...}-> simult-seq(ENV1,D2')
8   |
9   |         (ENV1 + ENV2) = ENV
10  |---------------------------------
11  |simult-seq(ENV1,ENV2):Dec ---> ENV
```

6.2.5 ML Declarations

As in Chapter 5, we use MSDF to indicate the required abstract constructs, and a DCG to specify the concrete syntax of ML declarations and their mapping to abstract constructs.

Since we are simply extending the previous specifications, we can refer to them explicitly, thereby avoiding repeating them.

Module 6.8 Lang/ML/Dec

```
see     Lang/ML/Exp

see     Prog, Prog/Dec

see     Dec, Dec/bind, Dec/simult-seq, Dec/accum, Dec/local

see     Exp, Exp/local
```

Module 6.9 Lang/ML/Dec/SYN

```
:- multifile prog/3, atexp/3, reserved/1.

:- ensure_loaded('../Exp/SYN.pro').

% Programs

prog(D:'Prog') -->
        i, topdecs(D), i.

% Top-level declarations:

topdecs(accum(D1,D2)) -->
        topdec(D1), i, ";", i, topdecs(D2).

topdecs(D) -->
        topdec(D), i, ";".

topdec(D) -->
        dec1(D).

topdec(bind(id(it),E)) -->
        exp(E).

% Declarations:

dec(D) -->
        dec1(D).

dec(accum(D1,D2)) -->
        dec1(D1), i, ";", i, dec(D2) ;
        dec1(D1), i, dec(D2).

dec1(D) -->
        "val", i, valbind(D).

dec1(local(D1,D2)) -->
```

```
45          "local", i, dec(D1), i, "in", i, dec(D2), i, "end".
46
47  % Value bindings:
48
49  valbind1(bind(I,E)) -->
50          vid(I), i, "=", i, exp(E).
51
52  valbind(D) -->
53          valbind1(D).
54
55  valbind(simult_seq(D1,D2)) -->
56          valbind1(D1), i, "and", i, valbind(D2).
57
58  % Expressions:
59
60  atexp(local(D,E)) -->
61          "let", i, dec(D), i, "in", i, exp(E), i, "end".
62
63  reserved("and").
64  reserved("end").
65  reserved("in").
66  reserved("let").
67  reserved("local").
68  reserved("val").
```

Summary

In this chapter:

- We considered some fundamental concepts regarding declarations and bindings.
- We introduced basic abstract constructs for value declarations, local declarations, accumulating declarations, and simultaneous declarations.
- We saw how to express changes to the label component which represents the current bindings.
- We gave rather simple MSOS rules for the basic abstract constructs, clearly exhibiting the similarities and differences between the intended semantics constructs.
- Finally, we gave an MSOS for a simple sublanguage of ML including declarations, extending the expression language defined in Chapter 5.

Exercises

Ex. 6.1 Let x and y be identifiers, and consider the ML expression:

```
x * let val y = x * y
    in  y + let val x = 2 * y
            in  x + let val y = 3 * x
                    in  x + y
                    end
            end
    end
```

(a) Indicate which occurrences of x and y above are binding, and for each binding occurrence, which bound occurrences refer to it.

(b) Which occurrences of x and y are free in the whole expression?

(c) How many closed sub-expressions does the above expression have?

Ex. 6.2 List all the properties that you expect the various operations on maps to satisfy, including:

(a) associativity

(b) commutativity

(c) unit laws.

Ex. 6.3 Consider the state

```
local(bind(id(x),2), tup-seq(id(x),id(x))) .
```

(a) Write out the entire computation starting from this state.

(b) Check your answer using the Prolog code supplied with these notes.

(c) Follow the computations starting from some larger initial states, observing how the environments computed by components influence the values of identifiers in their scopes.

Ex. 6.4 Consider Module 6.2. Explain why the evaluation rules for identifiers are not involved here.

Ex. 6.5 Suppose that `D, D', D''` are arbitrary value declarations in ML. Argue for or against each of the following semantic equivalences:[3]

(a) `D ; (D' ; D'')` ≡ `(D ; D') ; D''`.

(b) `local D in (D' ; D'')` ≡ `(local D in D') ; (local D in D'')`.

Ex. 6.6 A *structure declaration* in ML can be written:

```
struct I = structure D end
```

where the declarations `D` may include nested structure declarations. The bindings computed by `D` can be referenced using expressions of the form '`I.I1`', and similarly for bindings computed by nested structures. The declaration '`open I`' computes the same bindings as `D`.

(a) Introduce abstract syntax for declaration and use of structures, indicating its relationship to the concrete syntax indicated above.

(b) Introduce values which represent structures.

(c) Specify transition rules for each of the introduced constructs.

Ex. 6.7 Using the Prolog code accompanying these notes, validate the MSOS of the ML declarations specified in this chapter.

[3]The parentheses merely indicate the intended grouping, they are not part of ML concrete syntax.

Chapter 7

Commands

This chapter introduces fundamental concepts which are familiar from conventional imperative programming languages: commands with sequential flow of control and updatable variables.[1] It shows that although commands are not explicit in ML, some particular forms of expressions in ML are closely related to commands. Modelling commands involves labels with components representing changeable information. Remarkably, adding imperative features to expressions does not require any reformulation of the rules given in the foregoing chapters, thanks to the high degree of modularity of MSOS.

The files specific to this chapter are available online in the MSOS directory at `Lang/ML/Cmd`.

7.1 Fundamental Concepts

▷ A command is a construct which is executed for its *effects*.

```
C: Cmd
```

In contrast to expressions, commands do not compute any values at all; alternatively (and equivalently), we could regard them as computing a fixed, null value. This implies that constructs for commands are primarily concerned with flow of *control*, rather than flow of data, between their subcommands.

[1] Output is deferred to Chapter 9, since it is a special case of the reactive behaviour of concurrent processes.

▷ ML expressions of unit type are similar to commands.

ML expressions having the unit type also compute a fixed value, and there is not usually much point in evaluating an expression of unit type unless it can have some side-effects. Thus such expressions strongly resemble commands. However, ML provides constructs for sequencing and iterating evaluations of arbitrary expressions, not just for those of unit type. Below, we shall consider the exact relationship between expressions and commands in more detail.

7.1.1 Sequencing

▷ Sequencing of commands can be binary or n-ary

```
Cmd ::= seq(Cmd,Cmd)
```

```
Cmd ::= seq(Cmd+)
```

A sequence `seq(C1,...,Cn)` simply determines the order in which its component commands are to be executed. In Chapter 5 we introduced the construct `tup-seq(E1,...,En)` for sequencing evaluations of expressions, but there, the values of the subexpressions are collected to give a tuple as resulting value; command sequencing lacks this complication.

Notice that we require the sequence of commands to be non-empty, by using `Cmd+` rather than `Cmd*`. The empty sequence of commands would represent that there is nothing left to do, but we represent this by a constant construct:

```
Cmd ::= skip
```

▷ An expression can be turned into a command.

```
Cmd ::= effect(Exp)
```

The above construct represents the obvious way of obtaining a command from an expression: by evaluating it for any effects that it might have, and then discarding its value.

▷ Sequencing a command and an expression provide a general way for turning commands into expressions.

To obtain an expression from a command, we need to provide a value. The simplest would be to let the value be the unique value of the unit type—but few languages have such a type. The following constructs give a general way of forming expressions from commands:

```
Exp ::= seq(Cmd,Exp)
```

```
Exp ::= seq(Exp,Cmd)
```

In both `seq(C,E)` and `seq(E,C)`, the subexpression `E` determines the value of the enclosing expression: the only difference is whether this happens before or after the execution of the command `C`. When `E` is a constant, the two constructs are equivalent.

ML's sequencing of expressions, written '`(E1;...;En;E)`', corresponds to `seq(seq(effect(E1),...,effect(En)), E)`. We see that `E1,...,En` are all treated as commands, their values being discarded, in contrast to `E`, which gives the value of the expression. ML has an alternative sequencing of expressions, written '`(E1 before E2`', which corresponds to the opposite of '`(E1;E2)`', namely `seq(E1,effect(E2))`, where we see that it is `E1` which determines the value.

▷ The usual form of loop has a boolean condition.

```
Cmd ::= while(Exp,Cmd)
```

As with conditional choice in Chapter 5, we assume that the expression tested here is boolean; Chapter 1 illustrated variants where a numerical condition is tested for being (non)zero.

We could reduce other kinds of loops to combinations involving the basic while-command—although for those which occur in many languages, such as for-loops, it would usually be preferable to introduce further abstract constructs, and define the semantics of those constructs directly. ML's while-expression '`while E1 do E2`' corresponds to `seq(while(E1,effect(E2)),tuple())` which is simple enough, since the value of `E2` is always discarded, the entire expression having the unit type (which is implied by the possibility that `E2` never gets evaluated at all).

There are further abstract constructs concerned primarily with flow of control, such as throwing and catching exceptions, but these are out of the scope of these

notes. Some hints about them will however be given in Chapter 9, in connection with abrupt termination of concurrent threads.

The remaining constructs that we consider here involve processing of changeable information, rather than flow of control directly—although the changes made do in fact follow the flow of control.

7.1.2 Storing

▷ We distinguish *variables* from expressions.

```
VAR: Var
```

Syntactically, variables usually resemble simple kinds of expressions. A variable determines where a value can be stored, so it can be used as the target of an assignment. Of course, a variable can also be considered as an expression in a natural way, computing the value that it is currently storing. In contrast, expressions do not correspond directly to variables, except when they evaluate to so-called references.

▷ Variables for storing values need to be allocated.

```
Var ::= alloc(Exp)
```

The variable `alloc(E)` allocates a cell and initializes it to the value given by `E`. The value computed by the variable is the cell, determining where the value of the variable is stored. Cells represent *simple* variables, such that assigning a value to a particular variable can never affect the values of other variables; *compound* variables, such as arrays, have subvariables, and can be represented by data structures built from simple variables.

▷ The value of a variable is stable between assignments.

The value last assigned to a variable is expressed by the following abstract construct:

```
Exp ::= assigned(Var)
```

Stored values are assumed to be stable, remaining constant between assignments to that cell.

```
Exp ::= assign-seq(Var,Exp)
```

In general, variables may involve expressions as components—for instance, to compute the indices of components of array variables. The order of computation of `VAR` and `E` can be significant, so we have both interleaved and sequential variants of the assignment construct. We let them compute the value of `E`; the corresponding command can be derived using `effect`.

▷ Identifiers are bound to cells.

We do *not* treat identifiers themselves as cells, but rather let variable identifiers be *bound* to cells. Thus the value of a variable identifier is that which is stored at the cell to which it is bound. Different identifiers bound to the same cell are called *aliases*, and assigning a value to one of them affects the value of the other(s). (Aliasing can arise for instance when the same variable is supplied twice as a parameter of an abstraction.)

▷ Our model of storage is quite abstract.

Let us disregard whether cells for variables are allocated in a stack-like manner, or from a 'heap', and assume that cells are never recycled. Such details are relevant to those implementing programming languages, but can be ignored when giving abstract models.

▷ Variables are represented by reference values in ML.

```
Exp ::= ref(Var)
```

```
Var ::= deref(Exp)
```

ML has a somewhat unusual treatment of variables as so-called 'references', which correspond directly to simple variables. A reference is itself a value, and needs to be explicitly 'de-referenced' to obtain the value that was last assigned to it. Let us think of the reference value as constructed from the simple variable that was allocated when the reference was created: de-referencing then consists of obtaining the variable component of the reference, and inspecting its stored value.

In most other languages, variables are not themselves are not usually directly represented by values, and an occurrence of a variable identifier in an expression

is implicitly de-referenced, in general (as in bc). References can also be used as pointers, being stored in other references, i.e., pointer variables.

ML is unusual also in not having syntactic constructs for allocation of, assignment to, and dereferencing variables. Instead, it provides initial bindings for the identifiers `ref`, `:=`, and `!`, and requires the use of a general application construct.

7.2 Formal Semantics

In MSOS, the termination of a computation is indicated by a final state, which generally represents a computed value. Commands do not compute values as such, but their terminating computations still need to have final states. This final state is always the same, and corresponds to the null command, whose execution takes not even a single step):

Module 7.1 Cons/Cmd

```
State ::= Cmd

Final ::= skip

Cmd ::= skip
```

7.2.1 Sequencing

▷ The transition rules for sequencing do not reveal what kinds of effects commands might have.

Module 7.2 Cons/Cmd/seq

```
            C1 --{...}-> C1'
-----------------------------------
seq(C1,C2):Cmd --{...}-> seq(C1',C2)

seq(skip,C2):Cmd ---> C2
```

The rules for n-ary sequencing are only slightly less simple:

Module 7.3 Cons/Cmd/seq-n

```
           C+ = (C1,C2*),
         C1 --{...}-> C1',
          (C1',C2*) = C'+
------------------------------
seq(C+):Cmd --{...}-> seq(C'+)

     C+ = (skip,C2+)
-------------------------
seq(C+):Cmd ---> seq(C2+)

seq(skip):Cmd ---> skip
```

Notice that we use S+ instead of S* in the last rule above, since we have not allowed commands of the form `seq()`.

The rules for the remaining constructs concerning sequencing of commands and expressions are all very straightforward indeed:

Module 7.4 Cons/Cmd/effect

```
            E --{...}-> E'
---------------------------------
effect(E):Cmd --{...}-> effect(E')

effect(V):Cmd ---> skip
```

Module 7.5 Cons/Exp/seq-Cmd-Exp

```
           C --{...}-> C'
-------------------------------
seq(C,E):Exp --{...}-> seq(C',E)

seq(skip,E):Exp ---> E
```

Module 7.6 Cons/Exp/seq-Exp-Cmd

```
           E --{...}-> E'
-------------------------------
seq(E,C):Exp --{...}-> seq(E',C)

           C --{...}-> C'
-------------------------------
seq(V,C):Exp --{...}-> seq(V,C')

seq(V,skip):Exp ---> V
```

An alternative to specifying the last two rules above would be to exploit the fact that this construct is equivalent to the previous one when the expression is simply a value, and make an unobservable transition from `seq(V,C)` to `seq(C,V)` In general, it is preferable to specify each construct independently, without relying on the inclusion of other constructs. In the following case, however, there is simply *no alternative*:

Module 7.7 Cons/Cmd/while

```
1  see Cmd/cond, Cmd/seq
2
3  while(E,C):Cmd --->
4   cond(E,seq(C,while(E,C)),skip)
```

The point is that it is essential to duplicate `E` before starting to evaluate it, since it will have to be evaluated again if its value turns out to be true; similarly, a copy of `C` may be needed for subsequent iterations. The specified expansion is fortunately very natural—and it is difficult to imagine a language that has a while-command but no conditional command... Incidentally, notice that the above rule cannot be applied again until after the rules for '`cond`' and '`seq`' allow control to reach the copy of the while-command.

7.2.2 Storing

▷ The rules for constructs concerned with allocation, updating, and inspection of variables involve abstract stores.

```
S: Store = (Cell,Storable)Map
```

Stores are modelled as finite mappings from a set `Cell` of cells to a set `Storable` of storable values. Whereas `Cell` can be chosen freely, `Storable` should include only those values which are actually storable in (simple) variables in the language being described. For ML, we will let `Storable` include all expressible values; for modelling other languages, the relationship between the sets `Value`, `Bindable`, and `Storable` is usually not so straightforward.

Module 7.8 Cons/Var/alloc

```
Label = {store,store':Store,...}

           E --{...}-> E'
-------------------------------
alloc(E):Var --{...}-> alloc(E')

    new-cell(S) = CL, {CL |-> SV}/S = S'
-------------------------------------------
alloc(SV):Var --{store=S,store'=S',---}-> CL
```

The above specification of `Label` reflects that stores are changeable information. Recall the definitions of composability and (un)observability from Chapter 2: when the labels on adjacent transitions have stores `S1, S1'` and `S2, S2'`, the labels are composable only when `S1'` is the same as `S2`, and for a label to be unobservable, its two stores must be identical.

The last rule above illustrates how one can specify that a label is *almost* unobservable, i.e., unobservable except for a possible change to the store. Formally, the three dashes '`---`' are equivalent to a fixed meta-variable that ranges over unobservable sub-labels.

That rule also illustrates the use of the operation `new-cell(S)`, indicating the choice of any cell not in use in `S`. Assuming that there are no rules for removing cells from the store, each allocation in a particular computation computes a different cell.

Module 7.9 Cons/Exp/assign-seq

```
Label = {store,store':Store,...}

                  VAR --{...}-> VAR'
-------------------------------------------------
assign-seq(VAR,E):Exp --{...}-> assign-seq(VAR',E)

                   E --{...}-> E'
-----------------------------------------------
assign-seq(CL,E):Exp --{...}-> assign-seq(CL,E')

       def lookup(CL,S),  (CL|->SV)/S = S'
-------------------------------------------------------
assign-seq(CL,SV):Exp --{store=S,store'=S',---}-> SV
```

The final transition for obtaining the value of a variable below has a completely unobservable label.

Module 7.10 Cons/Exp/assigned

```
Label = {store,store':Store,...}

              VAR --{...}-> VAR'
-----------------------------------------
assigned(VAR):Exp --{...}-> assigned(VAR')

                lookup(CL,S)= SV
-------------------------------------------
assigned(CL):Exp --{store=S,store'=S,---}-> SV
```

Notice how none of the rules concerning variables involve bindings. Of course an expression could be simply an identifier, then its evaluation to a cell would require labels to contain a component representing the current bindings—but otherwise, the treatment of bindings has no impact on the rules for variables.

7.2.3 ML Commands

As in the preceding two chapters, we use MSDF to indicate the required abstract constructs, and a DCG to specify the concrete syntax of ML expressions that correspond to commands and their mapping to abstract constructs.

Since we are simply extending the previous specifications, we can refer to them explicitly, thereby avoiding repeating them.

Module 7.11 Lang/ML/Cmd

```
see     Lang/ML/Dec

see     Cmd, Cmd/seq-n, Cmd/effect, Cmd/while

see     Exp, Exp/seq-Cmd-Exp, Exp/seq-Exp-Cmd,
        Exp/assign-seq, Exp/ref, Exp/assigned

see     Var, Var/alloc, Var/deref

Storable ::= Value
```

Module 7.12 Lang/ML/Cmd/SYN

```
:- ensure_loaded('../Dec/SYN.pro').

% Expressions:

exp(seq(while(E1,effect(E2)),tup(null))) -->
        "while", i, exp(E1), i, "do", i, exp(E2).

atexp(seq(effect(E1),E2)) -->
        "(", i, exp(E1), i, ";", i, exp(E2), i, ")".

atexp(seq(seq(Cs),E2)) -->
        "(", i, expseq2(Cs), i, ";", i, exp(E2), i, ")".

infexp(seq(E1,effect(E2))) -->
        atexp(E1), i, "before", i, atexp(E2).

infexp(seq(effect(assign_seq(deref(E1),E2)),tup(null))) -->
        atexp(E1), i, ":=", i, atexp(E2).

appexp(ref(alloc(E))) -->
        "ref", i, atexp(E).

atexp(assigned(deref(E))) -->
        "!", i, atexp(E).

% Commands:

expseq([effect(E)]) -->
        exp(E).

expseq([effect(E)|Cs]) -->
        exp(E), i, ";", i, expseq(Cs).

expseq2([effect(E)|Cs]) -->
        exp(E), i, ";", i, expseq(Cs).

reserved("before").
reserved("do").
reserved("ref").
reserved("while").
reserved(":=").
reserved("!").
```

Summary

In this chapter:

- We considered fundamental concepts of commands, introduced the corresponding basic abstract constructs, and related them to ML;
- We gave MSOS rules for the basic abstract constructs, illustrating the use of labels involving changeable information.

Exercises

Ex. 7.1 In SML, give a declaration of a function `while_do` such that

```
while_do(fn()=>E1, fn()=>E2)
```

gives the same value as '`while E1 do E2`' for all `E1`, `E2`.

Ex. 7.2 Map the following loops to combinations of constructs that we have already introduced:

(a) The command '`repeat C until E`'.

(b) A corresponding expression, '`repeat E1 until E2`'.

(c) As for (b) above, but letting the value computed by the expression on termination be the last value computed by `E1`.

(d) For-loops (e.g., from Java or C).

Ex. 7.3 Indicate how Module 7.6 could be simplified by making use of Module 7.5.

Ex. 7.4 Introduce abstract constructs corresponding directly to the following loops, and specify their transition rules, validating your answers:

(a) The command '`repeat C until E`'.

(b) A corresponding expression, '`repeat E1 until E2`'.

(c) As for (b) above, but letting the value computed by the expression on termination be the last value computed by `E1`.

(d) For-loops (e.g., from Java or C).

Ex. 7.5 Specify the operation `new-cell(S)`, assuming `Cell ::= cell(Integer)`.

Ex. 7.6 Indicate what changes would be needed to Module 7.9 would be needed so that the computed result is `ref(CL)` rather than `SV`.

Ex. 7.7 Using the Prolog code accompanying these notes, validate the MSOS of the ML commands specified in this chapter.

Omitted Modules

Module 7.13 Cons/Exp/ref

```
Value ::= ref(Cell)

          VAR --{...}-> VAR'
--------------------------------
ref(VAR):Exp --{...}-> ref(VAR')
```

Module 7.14 Cons/Var/deref

```
Value ::= ref(Cell)

            E --{...}-> E'
-------------------------------
deref(E):Var --{...}-> deref(E')

deref(ref(CL)):Var ---> CL
```

```
CL : Cell ::= cell(Integer)
```

```
SV : Storable
```

Chapter 8

Abstractions

This chapter introduces fundamental concepts concerning so-called *procedural abstractions*, and considers *recursive declarations* involving such abstractions. Static scopes for bindings are modelled by *closing* abstractions. The MSOS of recursive declarations involves a gradual unfolding of the recursion.

The files specific to this chapter are available online in the MSOS directory at `Lang/ML/Abs`.

8.1 Fundamental Concepts

▷ An abstraction encapsulates a part of a program.

An abstraction encapsulates a part of the program (called the *body* of the abstraction) which is executed each time the flow of control reaches an *application* of it.

▷ When abstractions have *parameters*, their applications have to supply appropriate *arguments*.

The arguments of the current application are referred to in the body of an abstraction by means of parameter identifiers. The bindings for the parameter identifiers are made at the beginning of the application.

▷ The arguments are usually evaluated at the beginning of applications.

Although most languages insist on the evaluation of all arguments of an application before starting to execute the body of an abstraction, this isn't essential. Some languages provide so-called *lazy evaluation*, whereby each argument is evaluated

only when its value is actually *needed*; this is known as 'call-by-need'. Lazy evaluation has the advantage of avoiding the evaluation of arguments that are never needed at all during an application.

A less efficient way of getting a similar effect to lazy evaluation is to evaluate each argument every time its value is needed. This mechanism is known as 'call-by-name', and was introduced by Algol60. It can be simulated by letting the arguments be parameterless function abstractions.

▷ The association between parameter identifiers and arguments may be direct or indirect.

The parameter identifier may be bound *directly* to the corresponding argument. Alternatively, it may be a local variable identifier, and bound to a new cell; the value of the argument is then assigned to the cell. The argument for a so-called *reference parameter* must be a variable, and the parameter identifier is then bound directly to the cell determined by the variable, so that assignment to the parameter identifier affects the value of the argument variable.

Further variants involve separate cells for the argument and the corresponding parameter, with copying of the argument value to the parameter variable at the beginning of the application, and/or the other way round at the end of the application. Some languages allow different means of associating parameter identifiers with arguments to be used together, in the same abstraction.

▷ The correspondence between parameter identifiers and arguments may be *positional* or *by name*.

When an abstraction has more than one parameter, the correspondence between the individual parameter identifiers and the arguments has to be determined. Usually, the correspondence is given by the order in which the parameter identifiers and arguments are listed; default values for arguments can be supplied to allow some (or even all) arguments to be omitted. Conceptually, both the parameters and the arguments can here be regarded as *tuples*.

Alternatively, each argument can specify the parameter *identifier* to which it corresponds. The 'command-line switches' of Unix commands can be understood as an example of this mechanism. Here, it is common to have large numbers of parameters, all having default values, with most of the corresponding arguments being omitted in applications. Conceptually, the parameters and arguments correspond to *records*. The language Ada even allows a *mixture* of positional and named correspondence, with all the positional arguments preceding the named arguments.

▷ Most programming languages allow *declaration* of function or procedure abstractions; some also allow *expression* of abstractions.

A function or procedure declaration binds an identifier to a procedural abstraction; an application of a function or procedure identifier activates the abstraction to which it is bound. Functional programming languages generally allow an abstraction to be written as an expression, independently of declarations, without binding an identifier to it.

▷ The concepts of abstraction and recursion are closely related.

An abstraction is *recursive* when execution of its body involves an application of that same abstraction.

Let us now introduce some simple abstract constructs for abstraction and application, focussing on constructs that represent concepts underlying ML.

8.1.1 Abstractions

▷ Abstractions are values, but involve syntactic components.

We shall regard abstractions as syntactic constructs, rather than data.

```
ABS: Abs
```

▷ A *match* in ML involves function abstraction.

```
Abs ::= abs(Par,Exp)
```

The abstraction '`abs(PAR, E)`' is written '`PAR=>E`' in ML; it can occur in a so-called case expression '`case E' of PAR1=>E1|...|PARn=>En`', in an exception handler '`E' handle PAR1=>E1|...|PARn=>En`', or in a function abstraction expression '`fn PAR1=>E1|...|PARn=>En`'.

▷ *Parameters* of abstractions in ML include single identifiers and tuples.

```
PAR: Par
```

Parameters of abstractions are called *patterns* in ML. Here, we shall focus on the special cases of patterns that correspond to ordinary parameter lists, which are common to many programming languages.

```
Par ::= bind(Id)
```

A parameter '`bind(I)`' simply binds `I` to the value supplied in its application.

In ML, arguments are evaluated at the beginning of applications, and the parameter identifier is bound directly to the value.[1] Call-by-reference can be simulated simply by letting the argument be a reference to a variable.

```
Par ::= tup(Par*)
```

The parameter '`tup(PAR1,...,PARn)`' requires the argument supplied by the application to be a tuple with components '`V1,...,Vn`'. The identifiers bound by the `PARi` have to be distinct.

With only the above two constructs for forming parameters, the possibility of nesting tuples of parameters isn't so useful. However, it is simpler here to deal with the general case (which would be needed for a full treatment of patterns in ML) than to prohibit nesting of tuples.

▷ To ensure static scopes for bindings, abstractions need to be *closed*.

```
Exp ::= close(Abs)
```

Scopes for bindings are called *dynamic* when the initial bindings for executing the body of an abstraction are taken to be the bindings current at the point of its application; they are called *static* when they are the bindings which were current where the abstraction itself was introduced. An abstraction with no free occurrences of identifiers in its body is called *closed*, and is independent of whether scopes are static or dynamic.

Static scopes are modelled simply by embedding the current bindings in each abstraction to form a so-called *closure*.

[1] In ML, when the parameter identifier is already bound directly to a constructor (e.g., `nil`), the argument has to match exactly.

```
Abs ::= closure(Dec,Abs)
```

Often, the declaration component of a closure is just an environment, but we shall need the full generality in connection with recursive declarations.

The expression construct '`close(ABS)`' computes the closure of `ABS`. The expression '`close(abs(PAR,E))`' is written '`fn PAR=>E`' in ML. Moreover, the declaration '`fun I PAR = E`' abbreviates '`val rec I = fn PAR=>E`'. (Abstractions that occur in case-expressions and exception-handlers in ML are applied *in situ*, so closing them would make no difference.)

▷ Application can be generalized.

Chapter 5 introduced applications of operations (and of identifiers bound to operations) to values. Now we need to allow also applications of function abstractions. The simplest (and most appropriate for ML) is to let expressions include function abstractions, and generalize application to take two arbitrary expressions:

```
Exp ::= Abs
```

```
Exp ::= app(Exp,Arg)
```

```
Exp ::= app-seq(Exp,Arg)
```

The sequentialized application '`app-seq(E1,E2)`' is written simply '`E1 E2`' in ML.

▷ The inclusion of function abstractions as values makes the language *higher-order*.

Clearly, the possibility of function abstractions being both actual parameters and results of (other) abstractions implies that the language includes higher-order functions as values.

▷ Parameters correspond to *abstractions of declarations*, and can also be applied.

```
Dec ::= app(Par,Arg)
```

For instance, when `PAR` is simply `bind(I)`, its application to the value `V` of an argument expression `E` computes the environment that binds `I` to `V`, exactly as in `bind(I,E)`.

One can obtain parameters corresponding to other kinds of declarations in a systematic way, by removing the expression involved in the declaration. This is known as *The Correspondence Principle*; there is also an *Abstraction Principle*, saying that one can have procedural abstraction for each kind of construct that involves computation, and a related *Qualification Principle*, allowing local declarations for the same kinds of constructs.

These principles were originally advocated by Tennent in connection with language design; we shall adopt them for our basic abstract constructs (which are independent of the concrete syntactic concerns that may have discouraged full adherence to these principles in the design of many practical programming languages).

8.1.2 Recursion

▷ With *dynamic scopes* for bindings, declared abstractions are generally recursive by default; with *static scopes*, *recursive declarations* are needed to declare recursive abstractions.

When a declaration binds an identifier to an abstraction with dynamic scopes, and a subsequent application refers to this identifier, a (free) occurrence of the same identifier in the body of the abstraction obviously refers to the abstraction itself. For example, suppose that ML had dynamic scopes for bindings, and consider the declaration '`val f = fn x=>f(x+1)`'. An application '`f(0)`' in the scope of this declaration leads to the recursive application '`f(1)`', where `f` refers to the same binding as in the original application.

With static scopes, however, the current bindings for the declaration do not yet include the binding that is to be computed by the declaration, and any occurrence of the identifier bound to the abstraction in the body of the abstraction can only refer to some *previous* binding for the same identifier. To obtain a recursive abstraction with static scopes, it appears that we must either:

- manage to find a binding in advance, such that when given that binding, the declaration of the abstraction computes that *same* binding, or

- take the binding computed by a non-recursive declaration of an abstraction, and then convert it to a binding to an abstraction whose effect is the same as the desired recursive abstraction.

The first approach might seem the more attractive. However, the bindings required are in general infinite objects (although they can be represented as finite cyclic graphs), and moreover, one has to consider whether the bindings are uniquely defined. These concerns are elegantly addressed in denotational semantics (see the overview in a later chapter), but quite incompatible with the aims of operational semantics.

So we shall follow the second approach.[2] In any case, we need a new declaration construct:

```
Dec ::= rec(Dec)
```

In its most general form, '`rec(D)`' should compute the appropriate recursive bindings for any declaration `D`. Here, we shall restrict this construct to the case that `D` corresponds to simultaneous function declarations, since this will allow us to specify the formal semantics more straightforwardly.

8.2 Formal Semantics

8.2.1 Abstractions

▷ Function abstractions are treated as expressible values.

Module 8.1 Cons/Exp/Abs

```
1 Value ::= Abs
```

In connection with ML we shall only need *closures* formed from function abstractions, as computed by the following expression:

Module 8.2 Cons/Exp/close

```
1 see Abs/closure
2
3 Label = {env:Env,...}
4
5 close(ABS):Exp --{env=ENV,---}-> closure(ENV,ABS)
```

[2] A further possibility would be to introduce *indirect* bindings between identifiers and abstractions.

Module 8.3 Cons/Exp/app-seq

```
                E --{...}-> E'
-----------------------------------------
app-seq(E,ARG):Exp --{...}-> app-seq(E',ARG)

               ARG --{...}-> ARG'
-----------------------------------------
app-seq(V,ARG):Exp --{...}-> app-seq(V,ARG')

app-seq(V,PV):Exp ---> app(V,PV)

see Exp/app
```

Module 8.4 Cons/Exp/app

```
             E --{...}-> E'
-----------------------------------
app(E,ARG):Exp --{...}-> app(E',ARG)

           ARG --{...}-> ARG'
-----------------------------------
app(E,ARG):Exp --{...}-> app(E,ARG')
```

All rules for computing the result of the application depend on the value computed by `E1` above, and involve `app`, not `app-seq`.

The following rules for applications of function abstractions cover both static and dynamic scopes for bindings. When `E` is actually closed, the current bindings at the point of application are simply ignored.

Module 8.5 Cons/Abs/abs-Exp

```
see Dec/local, Dec/app

app(abs(PAR,E),PV):Exp ---> local(app(PAR,PV),E)
```

Module 8.6 Cons/Dec/app

```
             ARG --{...}-> ARG'
---------------------------------------
app(PAR,ARG):Dec --{...}-> app(PAR,ARG')
```

Module 8.7 Cons/Abs/closure

```
see Dec/local, Abs

app(closure(D,ABS),PV):Exp ---> local(D,app(ABS,PV))
```

The rules for applying a parameter to a value to compute bindings are quite straightforward:

Module 8.8 Cons/Par/bind

```
see Dec/app

app(bind(I),BV):Dec ---> {I |-> BV}
```

Module 8.9 Cons/Par/tup

```
see Dec/app, Dec/simult

PAR+ = (PAR,PAR*),  BV+ = (BV,BV*)
---------------------------------
app(tup(PAR+),tup(BV+)):Dec --->
   simult(app(PAR,BV),
          app(tup(PAR*),tup(BV*)))

         void = ENV
-----------------------------
app(tup(),tup()):Dec ---> ENV
```

An alternative to the first rule above might be to let the (big-step) computations of the bindings for `app(PAR,BV)` and `app(tup(PAR*),tup(BV*))` be conditions. Do you see why that would be undesirable?

8.2.2 Recursion

Finally, let us see how to make bindings for abstractions recursive:

Module 8.10 Cons/Dec/rec

```
Dec ::= reclose(Dec,Dec)

see Dec/bind, Dec/simult, Dec/simult-seq, Exp/close, Abs/closure

rec(D):Dec ---> reclose(rec(D),D)

```

```
7  reclose(rec(D),bind(I,close(ABS))):Dec --->
8                 bind(I,close(closure(rec(D),ABS)))
9
10 reclose(rec(D),simult-seq(D1,D2)):Dec --->
11                simult-seq(reclose(rec(D),D1),reclose(rec(D),D2))
12
13 reclose(rec(D),simult(D1,D2)):Dec --->
14                simult(reclose(rec(D),D1),reclose(rec(D),D2))
```

Notice how `rec(D)` is inserted into all the abstractions declared by `D`. When a recursively-defined function abstraction is applied, the computation starts by using the above rules again, so as to obtain a new environment where `rec(D)` has again been inserted. This unfolding can continue indefinitely.

An example may help to follow how it goes. Consider the ML declaration `fun f x = f x`. This corresponds to the following abstract syntax tree:

```
rec(bind(id(f),
         close(abs(bind(id(x)),
                   app(id(f),id(x))))))
```

The rules for `rec` compute the following environment:

```
id(f) |->
closure(void,
closure(rec(bind(id(f),
                 close(abs(bind(id(x)),
                           app(id(f),id(x))))),
        abs(bind(id(x)),
                 app(id(f),id(x)))))
```

In the scope of the above binding, an application of `f` leads to `closure` being replaced by `local`, which gives a state where the rules for `rec(D)` have to be applied again, before proceeding to evaluate the body of the abstraction.

8.2.3 ML Abstractions

As in the foregoing chapters, we use MSDF to indicate the required abstract constructs, and a DCG to specify the concrete syntax of ML abstractions and their mapping to abstract constructs.

Since we are simply extending the previous specifications, we can refer to them explicitly, thereby avoiding repeating them. This extension is independent of whether commands (or expressions with side-effects) are included.

Module 8.11 Lang/ML/Abs

```
see     Lang/ML/Dec

see     Exp, Exp/Abs, Exp/close, Exp/app-seq

see     Abs, Abs/abs-Exp, Abs/closure

see     Par, Par/bind, Par/tup

see     Dec, Dec/app, Dec/rec

```

Module 8.12 Lang/ML/Abs/SYN

```
:- ensure_loaded('../Dec/SYN.pro').

% Compatible with ../Cmd/SYN

% Application expressions:

appexp(app_seq(E1,E2)) -->
        atexp(E1), i, atexp(E2).

% Abstraction expressions:

exp(close(abs(PAR, E))) -->
        "fn", i, par(PAR), i, "=>", i, exp(E).

% Parameters:

par(bind(I)) -->
        vid(I).

par(tup(null)) -->
        "(", i, ")".

par(bind(I)) -->
        "(", i, vid(I), i, ")".

par(tup([bind(I)|PARs])) -->
        "(", i, vid(I), i, ",", i, parids(PARs), i, ")".

parids([bind(I)|PARs]) -->
        vid(I), i, ",", i, parids(PARs).

parids([bind(I)]) -->
        vid(I).
```

```
45
46  % Recursive declarations:
47
48  valbind(rec(D)) -->
49          "rec", i, valbind(D).
50
51  % Function bindings:
52
53  dec1(rec(bind(I,close(abs(PAR,E))))) -->
54          "fun", i, vid(I), i, par(PAR), i, "=", i, exp(E).
55
56  reserved("fn").
57  reserved("fun").
58  reserved("rec").
59  reserved("=>").
```

Summary

In this chapter:

- We introduced abstract constructs representing function abstractions and applications with simple parameter lists, and gave MSOS specifications of them.
- We considered techniques for interpreting recursive declarations of functions, and specified the MSOS of recursive declarations corresponding to those allowed in ML.

Exercises

Ex. 8.1 Suppose that one could choose whether bindings in ML were to have static or dynamic scopes.

(a) Give a simple example of ML code that would be type-correct with both static and dynamic scopes, but produce different results according to which kind of scopes are used.

(b) Give ML code containing a declaration of a function whose activations are non-recursive with static scopes, but which would become recursive with dynamic scopes.

Ex. 8.2

(a) Give some possible reasons for the restriction of recursive declarations in ML to function declarations.

(b) Is there any practical need to mix recursive declarations of functions with those of other kinds of values? If so, give an example of a program involving such a mixture, and show how it would have to be written in ML.

(c) (Challenging.) Give an MSOS for unrestricted recursive declarations of functions and other kinds of values, allowing the evaluation of the latter to involve applications of the former.

Ex. 8.3

(a) Give an example of an ML function whose applications are *always* infinite computations, regardless of its actual parameters.

(b) Give an example of a function whose applications would *always* terminate if evaluation of actual parameters were to be delayed until the corresponding formal parameters are first used.

Ex. 8.4 Consider the (useless) ML function declaration `fun f x = f x`. Using the Prolog code accompanying these notes, follow a few recursive applications of `f` starting from the call `f(0)` according to the specified rules, observing how the state grows with each recursive call. Could the state growth be avoided? Justify your answer.

Ex. 8.5 Using the Prolog code accompanying these notes, validate the MSOS of the ML abstractions specified in this chapter.

Chapter 9

Concurrency

Section 9.1 first considers some general concepts concerning concurrent processes. It then introduces various abstract constructs for concurrent processes, and for commands and expressions involving synchronous, channel-based communication. Section 9.2 presents the MSOS of the abstract constructs. It concludes by defining concrete syntax corresponding to a simple sublanguage of Concurrent ML [16], together with a mapping from concrete to abstract constructs.

9.1 Fundamental Concepts

In the foregoing chapters, we have considered programs that consist of single commands, declarations, or expressions. A computation for such a program can generally be regarded as a single sequential process, with interleaving merely allowing the steps to be carried out in different orders. In this chapter, we shall consider programs consisting of *systems of sequential processes* that are executed *concurrently*, i.e., concurrent processes. Here, the possibility of interleaving reflects that the steps of the different processes can happen independently. Moreover, communication between processes involves a different kind of information flow than that involved in the constructs presented in the previous chapters.

Early proposals for languages for communicating processes include CCS [17] and CSP [18]. More recently, Concurrent ML (CML) is an extension of ML proposed (and implemented) by Reppy [16]. Syntactically, CML merely adds some new types, and declares some function identifiers: function application is used to express all the new effects involved with concurrency, including the creation of processes, and synchronous communication between processes. CML supports also asynchronous and multi-cast communication. No new forms of expressions or other syntactic constructs are introduced at all, so CML programs can be compiled by ordinary ML compilers.

▷ Each process in a system has its own *thread of control*.

Conceptually, concurrent processes are regarded as being executed in parallel, so that any instant, all of them can be active. The progress of each process does not (directly) affect the progress of the other processes. Thus each process resembles a separate program with its own flow, or thread, of control. The difference between processes and programs is that information can flow between processes, and this information can (indirectly) affect the flow of control.

▷ Processing can be *distributed*.

An obvious way to implement concurrent processes is to let each of them be executed on a separate computer. Such processing is called distributed. Each distributed process has its own resources, such as stores and communication buffers; shared resources are accessed and affected via auxiliary server processes.

No assumptions are made about the relative speeds of the processes, except that no process can be *infinitely* faster than any other. This is to avoid so-called Zeno-computations, where one process 'finishes' a nonterminating computation while another process makes only a finite number of steps of another infinite computation.

▷ Threads within a process are scheduled *fairly*.

An alternative way of implementing concurrent processes is by time-sharing on a single computer. Each process can still have exclusive access to part of the overall store, but now there is also the possibility of common access to a shared part of the store. Processes that do not have their own resources are called threads. (Here, we shall refer to combinations of concurrent processes as systems, although combinations of threads are often themselves regarded as higher-level processes.)

Disregarding the possibility of exploiting several CPUs simultaneously, time-sharing usually involves *scheduling*, whereby each process is given a 'fair' share of the computation: in any nonterminating computation, each process which is ready to make a transition infinitely often should indeed make infinitely many transitions. Notice that 'fair' scheduling allows finite computations where one process is executed to termination before another one even starts—it is only infinite computations which are restricted by fairness constraints.[1]

[1] Since termination is usually undecidable, in practice implementations cannot avoid treating also finite computations in an asymptotically fair manner.

▷ A process may be able to initiate new processes.

Sometimes, a system of concurrent processes is *static*, in that the number of processes is fixed. All the processes are active from the start of the system, and no further processes can be created.

Dynamic systems of concurrent processes allow new processes to be added to the system at any time. The system can then start from a single initial process, and the number of new processes can even be unbounded. Terminated processes can be removed, so the system may ultimately revert to a single process, and subsequently terminate itself. Many enhancements of this basic idea are possible. For instance, a process may have to await the termination of all the new processes it has initiated before it terminates itself. Or one process may be able to cause (premature) termination of other processes.

▷ Processes may be able to access *shared resources*.

Regardless of whether systems of processes are static or dynamic, different processes may have direct access to the same resources (provided that they are not intended to be distributed). For instance, two processes may be able to assign values to, and inspect, some particular shared variable, providing a primitive means of communication between the processes.

▷ *Locks* on shared resources allow *mutual exclusion*.

In practice, programming becomes very difficult when access to shared resources is unrestricted. Modern languages supporting concurrent processes provide constructs which ensure exclusive access (for limited periods) to particular resources. A primitive kind of synchronization can arise when one process is waiting for another process to relinquish its access to the shared resource.

▷ Processes may be able to *communicate*.

An alternative to the use of shared resources for communication between processes is *message-passing*, whereby one process sends some information in a message and another one reads the message to obtain that same information.

▷ Communication may involve *synchronization*.

Sending a message may cause a process to wait until the message has been received by another process; this is called a *rendezvous*. As with waiting for processes to release locks on resources, a rendezvous can provide synchronization between the processes. An *extended rendezvous* lets the sending process wait for a subsequent response from the reading process before proceeding.

The main alternative, called *asynchronous* communication, lets the sending process proceed immediately, regardless of the receipt of the message by another process. Usually, the sent message is inserted in the *buffer* of the reading process, and remains there until read. When buffers are bounded, and messages are sent faster than they are read, either the sending process has to be suspended when the buffer is full, or some messages have to be discarded.

▷ Communication may be *direct* or *indirect*.

Another source of diversity in connection with communication between concurrent processes concerns the identification of senders and receivers. For example, the process sending a message may identify directly which other process(es) may read the message; this requires processes to have unique identifiers. Similarly, the reading process may accept messages only from particular processes. A message that could be read by all other processes is called a *broadcast*, and readers may *subscribe* to broadcasts based on the senders or the contents of the messages. Many variations on the basic concept of sending messages are possible, and found in different languages.

▷ *Channels* support synchronized, indirect communication.

Channels allow indirect communication between processes: any process with access to the channel may offer to send or read a message on it, and communication happens only when matching offers have been made. The details of matching can vary, but the basic idea is that any offer to send a message matches any offer to read a message on the same channel. A process may wait patiently for an offer to be matched, or it may make several alternative offers (then when one offer is accepted, the others have to be withdrawn). In any case, the matched offers are removed from the channel, and any processes waiting for the communication to take place can continue.

▷ Communication between processes is scheduled *fairly*.

In general, some fairness assumptions are made about the speed of communication between processes, and about the frequency of acceptance of the various offers of communication. The main principle is that a matching offer cannot be ignored indefinitely, but complications arise due to the possibility of offers being withdrawn, and of assigning priorities to offers.

As should be evident from the above, concurrent processes are especially rich with regard to variations on the fundamental concepts. Let us now introduce some simple abstract constructs corresponding to dynamic systems of processes with channel-based rendezvous communication, leaving formal exploration of the many variations to exercises and to more advanced studies.

9.1.1 Concurrent Processes

```
SYS: Sys
```

Systems form complete programs, and are not themselves components of other constructs.

▷ Systems can be sets or trees of processes.

```
Sys ::= conc(Sys,Sys)
```

Here, we shall regard a system as a set of processes (more precisely: a multiset, since duplication of processes is allowed). The constructor '`conc`' represents concurrency, and it will be both associative and commutative.

▷ A command can start a new process.

```
Cmd ::= start(Exp)
```

When `E` evaluates to a parameterless abstraction, `start(E)` initiates a process to evaluate the application of the abstraction to no arguments. If we needed to identify the new process (e.g., in connection with communication or termination), we would let this construct be an expression that evaluates to a process identifier (as in Concurrent ML, where the corresponding construct is written '`spawn E`') rather than a command. It would be straightforward to generalize `start` to deal with arguments for parametrized abstractions.

▷ A system might start with a single command.

```
Sys ::= Cmd
```

The commands which occur as components of concurrent systems correspond to individual sequential processes. A degenerate case of a concurrent system is a single command, which might occur at the start of a dynamic system, or just before its termination. We don't allow expressions as processes, since it isn't clear how to combine the values computed by different processes.

An initial command that starts a system of processes can involve local declarations. The resulting bindings can then be made available to processes whose abstractions are closed in the scope of those bindings. This allows particular bindings to be restricted to particular sets of processes, which can be important in connection with security issues.

Here, we do not include any means to give processes exclusive access to resources, and our concurrent processes will essentially be just threads.

9.1.2 Communication

▷ New *channels* can be allocated.

```
CHAN: Channel ::= chan(Integer)
```

```
Exp ::= alloc-chan
```

For prototyping, we represent channels in the same way as cells of storage. A channel declaration merely binds a channel identifier to a fresh channel given by evaluating `alloc-chan`; it is written '`chan I`' in Concurrent ML.

▷ Communications on channels are called *events*.

```
EVT: Event
```

An event is an offer of a communication.

▷ Sending and receiving values on channels are events.

```
Cmd ::= send-chan-seq(Exp,Exp)
```

```
Exp ::= recv-chan(Exp)
```

A match takes place when one process offers the event of sending a value on a particular channel, and another process offers to receive that same value on the same channel. In practice, the receiving process doesn't know in advance which value will be sent on the channel, so it offers to receive *any* value; the result of evaluating the `recv-chan` expression is the actual value that was sent. The expression corresponding to `send-chan-seq(E1,E2)` is written '`send(E1,E2)`' in Concurrent ML, and `recv-chan(E)` is written '`receive E`' (although both can also be expressed by applying the function '`sync`' to expressions of event types).

▷ Unmatched events are not allowed.

```
Sys ::= quiet(Sys)
```

We shall allow communication to take place between any pair of processes. Thus an offer to communicate on a channel is valid not only for the smallest subsystem that includes both the processes, but also for larger subsystems, including the entire system. The construct `quiet(SYS)` is used primarily to enclose an entire system, so that unmatched offers within `SYS` can be simply ignored, as in Concurrent ML; it could also be used to delimit independent subsystems.

9.2 Formal Semantics

The modules concerned with concurrent processes below are independent of whether processes can communicate or not.

9.2.1 Concurrent Processes

Module 9.1 Cons/Sys

```
1 State ::= Sys
```

Notice that concurrent systems might not have any final states at all, in general. In any case, the final states depend on the kind of constructs that can be used as individual processes.

Module 9.2 Cons/Sys/conc

```
                SYS1 --{...}-> SYS1'
-------------------------------------------
conc(SYS1,SYS2):Sys --{...}-> conc(SYS1',SYS2)

                SYS2 --{...}-> SYS2'
-------------------------------------------
conc(SYS1,SYS2):Sys --{...}-> conc(SYS1,SYS2')
```

The above rules correspond to *unrestricted* interleaving, and do *not* ensure any fairness of scheduling. For simplicity, we shall not bother with the extra information required to model fair scheduling of process execution; interested readers are encouraged to work out the details for themselves. (Hint: introduce limits on the number of steps that each process can take.)

Module 9.3 Cons/Cmd/start

```
Label = {starting':Abs*,...}

Value ::= Abs

            E --{...}-> E'
-------------------------------
start(E):Cmd --{...}-> start(E')

start(ABS):Cmd --{starting'=ABS,---}-> skip
```

We model the possibility of starting a new process from an arbitrary subcommand of an existing process by exploiting labels with *produced* information. When the `starting` component of a label is an abstraction, it indicates that a new process is to be started by applying that abstraction. Since each of our conditional rules ensures that unmentioned components of labels are automatically propagated from the transition in the condition to the transition in the conclusion, the `starting` component of the label will be the same for the sequential process enclosing the command, which can therefore detect the need to start the new process.[2]

[2] A modular treatment of throwing and catching exceptions can be given by exploiting produced components of labels in much the same way. Notice that the small-step nature of the transitions is essential: no analogous technique is available for big-step transitions.

Module 9.4 Cons/Sys/Cmd

```
Final ::= skip

Sys ::= skip

Label = {starting':Abs*,...}

    C --{starting'=ABS,...}-> C'
--------------------------------
C:Sys --{starting'=(),...}->
    conc(C',effect(app(ABS,tup())))

    C --{starting'=(),...}-> C'
------------------------------
C:Sys --{starting'=(),...}-> C'

conc(skip,SYS):Sys ---> SYS

conc(SYS,skip):Sys ---> SYS
```

The above rules specify how new processes are added to a system, and how terminated processes removed. A new process in inserted at a new conc node, combining the process that initiated the new process with the new process itself. Terminated processes can be removed at any stage of the computation (their presence does not affect the enclosing system in any way).

9.2.2 Communication

Module 9.5 Cons/Exp/alloc-chan

```
Label = {chans,chans':Channels*,...}

Value ::= Channel

    new-chan(CHANS) = CHAN,  CHANS + {CHAN} = CHANS'
-----------------------------------------------------
alloc-chan:Exp --{chans=CHANS,chans'=CHANS',---}-> CHAN
```

The set of allocated channels obviously has to be augmented to record the new channel. (The operation '+' on sets gives their union.)

Module 9.6 Cons/Cmd/send-chan-seq

```
Label = {event':Event*,...}

Event ::= sending(Channel,Value)

Value ::= Channel

                        E1 --{...}-> E1'
-------------------------------------------------------------
send-chan-seq(E1,E2):Cmd --{...}-> send-chan-seq(E1',E2)

                          E2 --{...}-> E2'
-------------------------------------------------------------
send-chan-seq(CHAN,E2):Cmd --{...}-> send-chan-seq(CHAN,E2')

send-chan-seq(CHAN,V):Cmd --{event'=sending(CHAN,V),---}-> skip
```

Module 9.7 Cons/Exp/recv-chan

```
Label = {event':Event*,...}

Event ::= receiving(Channel,Value)

Value ::= Channel

                E --{...}-> E'
---------------------------------------
recv-chan(E):Exp --{...}-> recv-chan(E')

recv-chan(CHAN):Exp --{event'=receiving(CHAN,V),---}-> V
```

We exploit labels with *produced* information to model offers of communication events. When considering a step for the sequential process containing the sending or receiving construct, our rules for commands, expressions, etc., ensure that the `event` component of a label is still present (just as with starting new processes). Moreover, the rules for `conc` in Module 9.2 preserve all label components.

Module 9.8 Cons/Sys/conc-chan

```
Label = {event':Event*,...}

Event ::= sending(Channel,Value) | receiving(Channel,Value)

SYS1 --{event'=  sending(CHAN,V),---}-> SYS1',
SYS2 --{event'=receiving(CHAN,V),---}-> SYS2'
-----------------------------------------------------------
conc(SYS1,SYS2):Sys --{event'=(),---}-> conc(SYS1',SYS2')

SYS2 --{event'=  sending(CHAN,V),---}-> SYS2',
SYS1 --{event'=receiving(CHAN,V),---}-> SYS1'
-----------------------------------------------------------
conc(SYS1,SYS2):Sys --{event'=(),---}-> conc(SYS1',SYS2')
```

The above rules allow communication between any two processes. Clearly, a single step modelling synchronous communication for a system involves simultaneous steps by two of its processes. In contrast with all our other rules for small-step semantics, we here need rules with more than one transition as conditions.

For simplicity, we assume that an offer of a communication event cannot be made in the same step as an effect on changeable information, nor be combined with a nonempty `starting` component. Then the remaining components of all the labels involved are unobservable and identical, as indicated by the occurrences of '`---`'.

Notice that we need a separate rule for the case that the sending process occurs after the receiving process in the enclosing system. Moreover, in the event term `receiving(CHAN,V)`, the value assigned to the metavariable `V` is determined by that given by `sending(CHAN,V)`, so for prototyping purposes, we need to be careful about the order of the conditions.

Module 9.9 Cons/Sys/quiet

```
Label = {event':Event*,...}

           SYS --{event'=(),...}-> SYS'
-----------------------------------------------
quiet(SYS):Sys --{event'=(),...}-> quiet(SYS')

quiet(skip):Sys ---> skip
```

Finally, the first rule above for `quiet(SYS)` requires labels to have empty event components. This ensures that transitions with unmatched offers arising in `SYS` do *not* give rise to corresponding transitions for `quiet(SYS)`. The effect is that when a process reaches a sending or receiving construct, it has to stay in the same state until its offer gets matched.

9.2.3 Concurrent ML Processes

As in the foregoing chapters, we use MSDF to indicate the required abstract constructs, and a DCG to specify the concrete syntax of process creation and communication expressions and their mapping to abstract constructs.

Since we are simply extending the previous specifications, we can refer to them explicitly, thereby avoiding repeating them. This extension requires both commands and abstractions. To avoid ambiguity, we introduce a new reserved word '`cml`' to indicate that an expression is to be treated as a concurrent system.

Module 9.10 Lang/ML/Conc

```
see     Lang/ML/Cmd, Lang/ML/Abs

see     Cmd, Cmd/send-chan-seq, Cmd/start

see     Exp, Exp/recv-chan, Exp/alloc-chan

see     Sys, Sys/Cmd, Sys/conc, Sys/conc-chan, Sys/quiet
```

Module 9.11 Lang/ML/Conc/SYN

```
:- ensure_loaded('../Abs/SYN.pro').
:- ensure_loaded('../Cmd/SYN.pro').

prog(quiet(effect(E)):'Sys') -->
        i, "cml", i, exp(E), i.

% Application expressions:

appexp(seq(start(E),tup(null))) -->
        "spawn", i, atexp(E).

appexp(seq(send_chan_seq(E1,E2),tup(null))) -->
        "send", i, atexp(tup_seq([E1,E2])).

appexp(recv_chan(E)) -->
        "receive", i, atexp(E).

% Declarations:

dec1(bind(I,alloc_chan)) -->
        "chan", i, vid(I).
```

Summary

In this chapter:

- We considered fundamental concepts of concurrency, and introduced abstract constructs representing concurrent processes, channel allocation, and synchronous communication on channels.
- We specified the dynamic semantics of the abstract constructs, and gave a mapping from a sublanguage of CML to these constructs.

Exercises

Deferred to the next version of these notes.

Chapter 10

Types

This chapter introduces fundamental concepts concerning *types* and type-checking. In ML, types can be *polymorphic*, and types of identifiers usually do not need to be declared, since they can be *inferred* from the uses of the identifiers. A computation of a type for an expression is modelled by a computation consisting of a single transition, using the so-called 'big-step' style of MSOS.

10.1 Fundamental Concepts

▷ A type is a construct which indicates the set of possible values of other constructs.

```
T : Type
```

Abstractly, types correspond to *sets* of values. The type `T` of an expression determines a set including all the values that the expression might have, and thus how the expression can be used. For example, if the type is numerical, arithmetic operations and predicates can be applied to the expression, whereas if it is a list type, the relevant operations include head, tail, prefixing, and append. Operations themselves have types, indicating the values to which they are applicable, and the kind of results that they then compute.

```
Exp ::= typed(Exp,Type)
```

An explicitly-typed expression `typed(E,T)` is written '`E:T`' in ML. It requires `E` to have type `T`, and then has itself type `T`.

Type checking ensures that operations are applied only to expressions which are sure to compute acceptable arguments. Sometimes, operations are partial (e.g., division is undefined when its second argument is zero, and head and tail are undefined on the empty list) but type checking is usually not able to prevent application of an operation to arguments outside its domain of definition.

▷ A type may also indicate the representation of a value.

Implementations generally represent values according to their types. The representation of atomic values (booleans/bits, characters, integers, reals, etc.) is usually different from that of structured values such as lists and arrays. More efficient representations can be used when the range of values is further restricted, so differently-bounded types of integers may be distinguished, even though the distinctions do not affect which operations are applicable.

▷ Other kinds of constructs can be considered to have types too

For instance, declarations of value identifiers can have types that give bindings mapping the declared identifiers to their types. We could even consider declarations of *type identifiers* as having types (introducing an atomic type '`type`' standing for the set of all possible types). Such type declarations may bind type identifiers to existing types, or to *new*, generated types.

▷ Type expressions are simply *expressions* which evaluate to types.

Types themselves are essentially *constant* type expressions, not involving type identifiers and therefore independent of the current bindings. The evaluation of a type expression to a type is entirely straightforward: it consists of replacing any type identifiers that occur in it by the corresponding types. Here, we shall focus on the basic concepts of types themselves, and omit further consideration of type declarations and type expressions.

10.1.1 Atomic Types

```
Type ::= bool
```

```
Type ::= int
```

```
Type ::= real
```

```
Type ::= unit
```

The above atomic types correspond directly to some of ML's atomic types. Apart from 'unit', they are found in many other languages as well, generally standing for fixed sets of values. Often, 'int' is treated as a subtype of 'real', so that an integer-valued expression can be used wherever a real-valued expression is required; ML, however, has no subtypes, and to obtain a real number from an integer, a conversion function has to be applied. The type 'bool' is required in connection with conditional expressions, as well as being the type of the result of the operations for comparing values, such as '<'.

10.1.2 Compound Types

In ML, the compound types include product types (for tuples), list types, and function types.

```
Type ::= tup(Type*)
```

A product type tup(T1,...,Tn) (where $n \geq 2$) is written 'T1*$\cdots$*Tn' in ML, e.g., 'int*int'. A value tup(V1,...,Vn) is in the type tup(T1,...,Tn) iff for all i, Vi is in Ti, so it corresponds to the usual cartesian product of the sets given by the types Ti. Following ML, we shall regard the null value, written '()', as having the type 'unit', although it could also be taken to have the empty product type 'tup()'.

```
Type ::= list(Type)
```

A list type list(T) is written 'T list' in ML, e.g., 'int list', and a value list(V1,...,Vn) is in list(T) iff for all i, Vi is in T. (For simplicity, let us ignore here that lists in ML are merely abbreviations for iterated applications of prefixing, and that 'list' is just an identifier, initially bound to a so-called type constructor.)

```
Type ::= func(Type,Type)
```

A function type func(T1,T2) is written 'T1->T2' in ML, e.g., 'int->int'. For each argument type T1 and result value type T2, func(T1,T2) indicates the set of all functions FN in Func such that for all V in T1, the value computed by applying FN to V (whenever the evaluation terminates normally) is in T. As usual, it is

convenient to consider functions of more than one argument as unary functions on tuples of argument values, e.g., in `func(T1,T2)`, `T1` might be `tup(int,int)` for a binary function.

Notice that a function `FN` may be in several such function types for different argument types `T1`: the fact that `FN` is of type `func(T1,T2)` indicates something about some applications of `FN`, but does not exclude the possibility of other applications. For instance, the addition operation is applicable both to pairs of integers and to pairs of reals, being overloaded. Programs cannot declare new overloaded functions in ML, but declared functions can be polymorphic (see below), which leads to the same function having infinitely-many different types.

10.1.3 Polymorphic Types

```
Type ::= var(Id)
```

Simply by allowing *type variables* to occur in types, the types immediately become *polymorphic*. A type variable `var(I)` is written as an identifier starting with a prime in ML (this distinguishes it from a type identifier that refers to a declared type, which cannot start with a prime). A type variable starting with only a single prime, such as `'a`, allows values of all possible types; one starting with a double prime, e.g., `''a`, restricts the values to those of so-called *equality types*, which exclude real numbers, functions, and structured values with such values as components.

▷ Polymorphic types may be *specialized* and *unified*.

To specialize a type that contains one or more occurrences of a particular type variable, we simply replace all those occurrences *uniformly* by some type (which may also include type variables—even the same one that is being replaced). For instance, we may specialize the type written ‘`'a->'a`’ to get ‘`int->int`’ or ‘`''a*'a->'a*'a`’, but we cannot obtain ‘`int->real`’ from ‘`''a->'a`’ by specialization.

More symmetrically, given two types possibly containing type variables, we may try to *unify* them, specializing either or both of them to obtain the same type. Thus we can unify ‘`''a->'a`’ with ‘`int->int`’, but not with ‘`int->real`’. When both types include type variables, there may be more than one pair of specializations that unify them. The least arbitrary pair of specializations is called the *most general unifier* of the two types. The process of finding most general unifiers is the basis for the usual implementation of type inference and checking in ML; it also underlies the implementation of Prolog (and in fact Prolog can be used to implement ML type inference in a very straightforward way).

▷ Type schemes allow different specializations.

ML involves one further construct concerning polymorphism, which is called a *type scheme*, and is *not* itself regarded as a type:

```
TypeScheme ::= forall(Id+,Type)
```

The type scheme `forall(I1...In,T)` is *implicit* in value bindings '`val I=E`' when the type of `E` is `T`, and the type variables in `T` are the identifiers `I1,...,In`. (The full details in ML are actually a bit more complicated.)

The effect of the above treatment is to allow the types of different occurrences of `I` to be *different specializations* of `T`. For instance, the declaration '`val f = fn x=>x`' binds `f` to a type scheme such as `forall(I, func(var(I),var(I)))` for some type variable identifier `I`, so different occurrences of `I` might have types such as `func(int,int)` and `func(bool,bool)`.

Note that in '`fn I=>E`', `I` is bound to a *type*, not to a type scheme, so all occurrences of `I` in `E` must have the *same* type. The main reason for this difference is that in a value binding, we know the degree of polymorphism of the type of the expression involved, whereas in a function abstraction, we cannot tell whether the argument supplied by an application will be polymorphic or not.

10.2 Formal Semantics

To model type-checking of expressions, we shall use a special style of MSOS where every computation takes just one step, going straight from abstract syntax to a computed value. This style is known as 'big-step', in contrast to the usual 'small-step' style of MSOS. For various reasons, its use should be reserved for modelling computations that correspond to evaluation of terms in mathematics, without side-effects, raising of exceptions, and nontermination.

To avoid confusion between type-checking and evaluation, let us adopt notation such as `E===>T` for type-checking of expressions `E` computing types `T`, and similarly for the static semantics of other kinds of constructs.

The labels in static semantics are *always* unobservable, but usually include fixed components, such as environments that give the types of all currently-bound identifiers. We write `E===>T` as `E=={env=ENV,---}=>T` when we need to refer to `ENV`. When no labels are mentioned explicitly, the transitions in the conditions of a rule implicitly have the same label as the transition in the conclusion. For instance, consider the rule for type-checking typed expressions:

```
      E ===> T
--------------------
typed(E,T):Exp ===> T
```

`E===>T` will imply not only that `E` has type `T`, but also that all subexpressions of `E` have been successfully type-checked. An expression may have many different types, or none at all.

The remaining rules are still being developed and validated, and are deferred to the next version of these notes. See Chapter 1 for the validated type-checking rules for abstract constructs involved in a simple sublanguage of bc.

Chapter 11

Other Frameworks

▷ We have seen how to use one framework for specifying formal semantics

In the preceding chapters, we have considered some of the main fundamental concepts of programming languages, and given a conceptual analysis of various constructs taken from Standard ML and Concurrent ML. We have also specified a formal semantics for these constructs, using the Modular SOS (MSOS) framework; by now, those who have carefully studied the specified rules and worked through the exercises should have acquired the ability not only to *read*, but also to *write*, such specifications.

▷ Alternative frameworks include other forms of operational semantics, as well as denotational semantics, axiomatic semantics, and action semantics

Let us now sketch the main alternative frameworks, and compare them briefly with the framework that we have adopted here. The aim is, for each framework, to explain its main principles, to give an impression of how semantic descriptions look in it, and to draw attention to any major drawbacks that it might have.

11.1 Structural Operational Semantics

The Structural Operational Semantics (SOS) framework was proposed by Plotkin in 1981 [1]. The main aim was to provide a simple and direct approach, allowing concise and comprehensible semantic descriptions based on elementary mathematics. The basic SOS framework has since been presented in various textbooks (e.g. [3, 5]), and exploited in numerous papers on concurrency [17]; see also [19]. The big-step form of SOS (also known as Natural Semantics [20]) was used during the design of Standard ML, as well as to give the official definition of that language [7].

▷ The original SOS framework is closely related to the framework used here

Both involve the use of *rules* to give inductive specifications of transition relations, where the states involve both abstract syntax trees and computed values. When describing a purely functional programming language, SOS rules look quite similar to those given in the preceding chapters. For instance:[1]

$$\frac{E_1 \longrightarrow E_1'}{\texttt{cond}(E_1, E_2, E_3) \longrightarrow \texttt{cond}(E_1', E_2, E_3)} \tag{11.1}$$

$$\texttt{cond}(\texttt{true}, E_2, E_3) \longrightarrow E_2 \tag{11.2}$$

$$\texttt{cond}(\texttt{false}, E_2, E_3) \longrightarrow E_3 \tag{11.3}$$

Notice, however, that there are no labels (such as X or U) on the transitions in the above SOS rules: labels are used in SOS only in connection with communication and synchronization between concurrent processes, and don't occur at all with transitions for sequential programming constructs. Let us now focus on the main differences between the original SOS framework and the MSOS variant of it that we have been using in these notes.

▷ States are not restricted to syntax trees

In the original SOS framework, syntax is not so clearly separated from auxiliary semantic entities as it is in MSOS: the former allows both syntactic and semantic entities as components of states, whereas MSOS restricts states to pure syntax (together with computed values) and insists that all other entities are to be incorporated in the labels on transitions. So it might seem that SOS has the advantage of greater generality. However, we shall see below that this apparent relaxation leads to some unfortunate notational complications.

▷ Labels on adjacent transitions are not restricted to be composable

Recall that in MSOS, transitions can occur next to each other in a computation only when their labels are composable. When labels are used in SOS, no notion of composability is involved, nor is there any *a priori* distinction between observable and unobservable labels.

[1] When illustrating other frameworks, we shall use the same notation for abstract syntax as in the preceding chapters, to facilitate comparison. In general, however, specifications in the other frameworks use notation that is strongly suggestive of the concrete syntax of the language whose semantics is being described.

Bindings

▷ Bindings are usually represented by explicit components of states

In fact the treatment of bindings in SOS is somewhat awkward. Suppose that the states for expression evaluation include bindings: State = Exp × Env. This requires the specification of transitions $(E, ENV) \longrightarrow (E', ENV)$ where the bindings ENV remain *unchanged*. Clearly, it would be tedious to have to write (and read) ENV twice each time a transition is specified, and it is usual practice to introduce the notation $ENV \vdash E \longrightarrow E'$ as an abbreviation for $(E, ENV) \longrightarrow (E', ENV)$. Thus when the functional language being described involves bindings, the SOS rules given above would be reformulated as follows:

$$\frac{ENV \vdash E_1 \longrightarrow E_1'}{ENV \vdash \mathsf{cond}(E_1, E_2, E_3) \longrightarrow \mathsf{cond}(E_1', E_2, E_3)} \tag{11.4}$$

$$ENV \vdash \mathsf{cond}(\mathtt{true}, E_2, E_3) \longrightarrow E_2 \tag{11.5}$$

Etc.

Alternatively, bindings can be eliminated as soon as they have been computed by *substituting* the bound values for the identifiers throughout the scope of the bindings. However, an explicit definition of the result $[ENV]T$ of substitution of ENV throughout T requires the tedious specification of a defining equation for each non-binding construct T, for instance:

$$[ENV]\mathsf{cond}(E_1, E_2, E_3) = \mathsf{cond}([ENV]E_1, [ENV]E_2, [ENV]E_3) \tag{11.6}$$

as well as some rather more intricate equations for the binding constructs.

Stores

▷ Effects on storage are represented by explicit store components of states

When the described language isn't purely functional, and expression evaluation can have side-effects, the states for expression evaluation include the current store as well as the bindings: State = Exp × Env × Store. Transitions between such states are written $ENV \vdash (E, S) \longrightarrow (E', S')$, so the rules given above would be reformulated as follows:

$$\frac{ENV \vdash (E_1, S) \longrightarrow (E_1', S')}{ENV \vdash (\mathsf{cond}(E_1, E_2, E_3), S) \longrightarrow (\mathsf{cond}(E_1', E_2, E_3), S')} \tag{11.7}$$

$$ENV \vdash (\mathsf{cond}(\mathtt{true}, E_2, E_3), S) \longrightarrow (E_2, S) \tag{11.8}$$

Communications

▷ Communication between concurrent processes is represented by labels on transitions

Finally, suppose that expression evaluation can involve process creation and communication, as in Concurrent ML. The conventional technique in SOS is here to add labels to transitions—but leaving the bindings and stores in the states. The SOS rules given above would be reformulated thus:

$$\frac{ENV \vdash (E_1, S) \xrightarrow{L} (E_1', S')}{ENV \vdash (\mathtt{cond}(E_1, E_2, E_3), S) \xrightarrow{L} (\mathtt{cond}(E_1', E_2, E_3), S')} \tag{11.9}$$

$$ENV \vdash (\mathtt{cond}(\mathtt{true}, E_2, E_3), S) \xrightarrow{\tau} (E_2, S) \tag{11.10}$$

where τ is some fixed label that indicates a silent, uncommunicative step.

▷ Rules require reformulation when components of states or labels on transitions are added, changed, or removed

As illustrated above, the formulation of rules in conventional SOS has to change whenever the components of the model involved in transitions (i.e. states and labels) are changed. This is in marked contrast to the situation with MSOS, where the formulation of transitions in rules is stable, allowing the rules for each programming construct to be given definitively, once-and-for-all. Of course, if one is actually working with a fixed definition of states and transition labels in SOS, the stability ensured by MSOS is of no advantage, and the SOS style of exhibiting all the components may be preferable—especially when needing to refer to them frequently. But in fact, it is straightforward to move from MSOS to SOS:

Conversion of MSOS specifications to SOS

▷ Any MSOS specification can be converted systematically into an equivalent SOS

Clearly, given a specification of the components of labels in an MSOS, the set of states of an SOS can be defined as the product of the set of abstract syntax trees of the MSOS with the sets of values of the fixed and changeable components; if there are any observable components, the labels on transitions in the SOS may be defined accordingly. It is then straightforward to reformulate the MSOS rules as SOS rules.

For example, suppose that we have in an MSOS:

$$\begin{array}{lcl} \text{State} & ::= & \text{Exp} \mid \ldots \\ \text{Label} & = & \{\texttt{env} : \text{Env}, \texttt{store}, \texttt{store}' : \text{Store}, \texttt{out}' : \text{Value}^*, \ldots\} \end{array}$$

The we'd get in the corresponding SOS:

$$\begin{array}{lcl} \text{State} & = & \text{Exp} \times \text{Env} \times \text{Store} \mid \ldots \\ \text{Label} & = & \text{Value}^* \end{array}$$

and each MSOS transition $E \xrightarrow{X} E'$ would give rise to a corresponding SOS transition of the form $(E, ENV, S) \xrightarrow{O^*} (E', ENV, S')$. The labels on the MSOS transitions are composable if and only if the the resulting SOS transitions can be adjacent, so there is a 1-1 correspondence between the computations specified by the MSOS and by the derived SOS.

Small-Step and Big-Step Styles

▷ In conventional SOS, the small-step and big-step styles are usually regarded as alternatives

Formally, the big-step style can be regarded as a special case of the small-step style: computations in the big-step style simply don't involve any intermediate states, only initial and final states. Note also that if one has defined a small-step SOS, the transitive closure of the small-step relation provides the corresponding big-step relation (as exploited by Plotkin to let a single step of command execution depend on the complete evaluation of the component expressions). In practice, however, authors of SOS descriptions usually choose one style or the other—and then stick to it, since changing styles involves major reformulation.

In MSOS, as illustrated in the preceding chapters, it seems better to mix the small-step and big-step styles, choosing the more appropriate style for each kind of construct by consideration of the nature of its computations:

Big-step SOS is better for constructs whose computations are *pure evaluation*, with no side-effects, no exceptions, and always terminating—e.g., for evaluating decimal numerals to numbers, for matching patterns against values, and for types;

Small-step SOS is better for all other constructs, since it makes explicit the order in which the steps of their computations are made, which is usually significant. Moreover, small-step SOS copes more easily with specifying interleaving, exception handling, and concurrency than big-step SOS does.

Informal Conventions

▷ The official Definition of Standard ML is not entirely formal

A major example of an SOS in the pure big-step style is the Definition of Standard ML [7]. The description covers the static and dynamic semantics of the entire language (both the core and module levels), and has been carefully written by a group of highly qualified authors. Nevertheless, its degree of formality still leaves something to be desired—especially in connection with two "conventions" that were adopted:

The "store convention" allows the store to be left implicit in rules where it is not being extended, updated, or inspected.

The "exception convention" allows the omission of rules that merely let uncaught exceptions preempt further sub-expression evaluation.

For instance, consider the following rule for the evaluation of conditional expressions:

$$\frac{ENV \vdash E_1 \longrightarrow \texttt{true} \qquad ENV \vdash E_2 \longrightarrow V}{ENV \vdash \texttt{cond}(E_1, E_2, E_3) \longrightarrow V} \tag{11.11}$$

By the conventions, the above rule abbreviates the following three rules:

$$\frac{\begin{array}{c} ENV \vdash (E_1, S) \longrightarrow (\texttt{true}, S'), \\ ENV \vdash (E_2, S') \longrightarrow (V, S'') \end{array}}{ENV \vdash (\texttt{cond}(E_1, E_2, E_3), S) \longrightarrow (V, S'')} \tag{11.12}$$

$$\frac{ENV \vdash (E_1, S) \longrightarrow (\texttt{raised}(EX), S')}{ENV \vdash (\texttt{cond}(E_1, E_2, E_3), S) \longrightarrow (\texttt{raised}(EX), S')} \tag{11.13}$$

$$\frac{\begin{array}{c} ENV \vdash (E_1, S) \longrightarrow (\texttt{true}, S'), \\ ENV \vdash (E_2, S') \longrightarrow (\texttt{raised}(EX), S'') \end{array}}{ENV \vdash (\texttt{cond}(E_1, E_2, E_3), S) \longrightarrow (\texttt{raised}(EX), S'')} \tag{11.14}$$

where $\texttt{raised}(EX)$ indicates that the evaluation of a sub-expression has raised an exception with value EX. Such conventions are completely unnecessary when using MSOS. The store convention is obviously redundant, since stores are already hidden as components of labels. Moreover, the novel technique hinted at in Chapter 9 for the semantics of throwing and catching exceptions eliminates the need for further rules, and hence for the exception convention as well. The only convention at all that we have introduced in these notes is the one allowing us to omit unobservable labels when writing transitions, which is a trifling detail in comparison to the conventions adopted in the Definition of Standard ML.

11.2 Reduction Semantics

This framework was developed by Felleisen and his colleagues towards the end of the 1980's [21]. It has been used primarily in theoretical studies, where it is sometimes preferred to SOS; for instance, Reppy used Reduction Semantics to define (parts of) Concurrent ML.

▷ States are abstract syntax trees, corresponding to well-formed terms

They don't involve abstract mathematical values (numbers, sets, maps, etc.) at all: they are purely syntactic. For example, numerical expressions compute decimal numerals rather than abstract mathematical numbers. If needed, however, auxiliary constructs may be added to the abstract syntax (as in SOS).

▷ Transitions are term rewriting steps, called reductions

Term Rewriting is an interesting and well-developed topic in its own right [22]. A rewriting step is called a *reduction* (regardless of whether the resulting term is actually smaller than the previous one or not). The sub-term that gets rewritten in a reduction is called a *redex*, and the resulting sub-term is called a *reduct*. Reductions may normally be made in any sub-term, and continue until no more are possible—perhaps never terminating.

Redexes are restricted to occurrences in evaluation contexts When the term being reduced corresponds to the abstract syntax of a program, the location of the redexes of the reductions should be restricted to follow the flow of control of the computation (otherwise reductions could be prematurely made in parts of the program that were not even supposed to be executed, leading to unintended results).

An evaluation context C : Ctx is a term with a single *hole*. If t is a term, $C[t]$ is the result of replacing the hole in C by t. A *reduction in a context* C is written $C[t] \longrightarrow C[t']$ (with a longer arrow), and can be made whenever there is an ordinary reduction $t \to t'$ (written with a shorter arrow). In fact $C[t]$ here is generally the entire program context of t, assuming that there are no further rules that would allow a reduction in a context to be regarded as an ordinary reduction.

Rules may be given also for rewriting the context as well as the term in that context: $C[t] \longrightarrow C'[t']$; in this case it is not required that $t \to t'$ should be an ordinary reduction.

The evaluation contexts for use in a reduction semantics are specified by a context-free grammar.

▷ Simple SOS rules correspond to reductions

Comparing SOS with Reduction Semantics, the simple rules of an SOS generally correspond directly to rules for ordinary reductions. For example, consider the following reduction rules for continuing with the evaluation of a conditional expression after its condition has been evaluated:

$$\texttt{cond}(\texttt{true}, E_2, E_3) \rightarrow E_2 \tag{11.15}$$

$$\texttt{cond}(\texttt{false}, E_2, E_3) \rightarrow E_3 \tag{11.16}$$

▷ Conditional SOS rules correspond to productions for evaluation contexts

Many conditional SOS rules simply express the flow of control of the computation, such as indicating which sub-expression is to be evaluated first. These SOS rules correspond not to reduction rules, but rather to productions in the grammar for evaluation contexts. For example, the rule for evaluating the condition of a conditional expression corresponds to the following production for contexts:

$$\text{Ctx} ::= \texttt{cond}(\text{Ctx}, \text{Exp}, \text{Exp})$$

▷ Reductions that replace the evaluation context do not correspond directly to SOS rules

A significant advantage of Reduction Semantics is that it is straightforward to specify rules that affect the entire context of the sub-expression being evaluated. For example, the following rule specifies clearly that when *exit* is evaluated, the remaining evaluation of the entire program is terminated.

$$C[\texttt{exit}] \longrightarrow \texttt{null} \tag{11.17}$$

Exceptions can be specified in a similar way. The novel technique available for the MSOS of exceptions provides similar benefits for SOS.

▷ Computed values are simply canonical terms in normal form

The computed values in a Reduction Semantics for a language like Standard ML would include not only numerals and booleans, but also tuples, lists, and records with values as components. The syntax of values is specified by a context-free grammar—for example, by taking some of the productions for expressions *Exp* from the grammar for the full abstract syntax, and replacing *Exp* by *Val* (except within abstractions).

▷ Bindings are represented by substitution, which is itself tedious to specify

Substitution replaces identifiers by the values to which they are bound, and can be specified by reduction rules (or defined equationally). In Reduction Semantics, there is unfortunately no alternative to the use of substitution to deal with the bindings that arise in the semantics of local declarations.

▷ Effects on storage are represented by rewriting a store term

A term representing a store is a sequence of *canonical* assignments, i.e. assignments where the location and the value have already been evaluated. There is only one level of store—in contrast to the situation with local bindings—so it can be kept as a separate component of the entire program context:

$$\begin{array}{lll} \text{ProgCtx} & ::= & \texttt{prog-ctx}(\text{Ctx}, \text{Store}) \\ \text{Store} & ::= & \texttt{skip} \mid \texttt{seq}(\text{Store}, \texttt{update}(\text{Loc}, \text{Val})) \end{array}$$

When the left- and right-hand sides of an assignment expression (or command) have been evaluated, the effect of the assignment is simply added to the store, by giving a reduction that replaces the entire context:

$$\begin{array}{c} \texttt{prog-ctx}(C[\texttt{update}(L,V)], S) \longrightarrow \\ \texttt{prog-ctx}(C[\texttt{null}], \texttt{seq}(S, \texttt{update}(L,V))) \end{array} \tag{11.18}$$

Inspecting the value stored at a particular location also involves the context, but does not change it:

$$\frac{S = \texttt{seq}(S', \texttt{update}(L,V))}{\texttt{prog-ctx}(C[\texttt{stored}(L)], S) \longrightarrow \texttt{prog-ctx}(C[V], S)} \tag{11.19}$$

$$\frac{\begin{array}{c} S = \texttt{seq}(S', \texttt{update}(L',V)), L \neq L', \\ \texttt{prog-ctx}(C[\texttt{stored}(L)], S') \longrightarrow \texttt{prog-ctx}(C[V], S') \end{array}}{\texttt{prog-ctx}(C[\texttt{stored}(L)], S) \longrightarrow \texttt{prog-ctx}(C[V], S)} \tag{11.20}$$

▷ Reduction rules for communication involve separate evaluation contexts for the concurrent processes involved

For example, suppose that a system of concurrent processes is represented as a map from thread identifiers to states of threads, then synchronous communication can be specified thus:

$$\begin{array}{c} \{I_1{=}C_1[\texttt{send}(K,V)]\} + \{I_2{=}C_2[\texttt{receive}(K)]\} + TM \longrightarrow \\ \{I_1{=}C_1[\texttt{null}]\} + \{I_2{=}C_2[V]\} + TM \end{array} \tag{11.21}$$

11.3 Abstract State Machine Semantics

Abstract State Machines (ASM) is an operational semantics framework that was proposed by Gurevich in the late 1980's [23]. The main aim was to specify the individual steps of computations at the proper level of abstraction; issues such as control flow and scopes of bindings were regarded as of secondary importance. The framework has been applied to several major languages, including Standard ML and Java. However, the details and general style of ASM specifications varies considerably between different publications; here, we shall follow [24], which appears to be competitive with SOS in its accessibility.

▷ States interpret static and dynamic function symbols

The interpretation of a function symbol is a map from arguments to results. The function is called *static* when the map doesn't change during a computation. In contrast, the values of *dynamic* functions on particular arguments can be initialized, changed, or made undefined. Static functions of no arguments correspond to ordinary constants, whereas dynamic functions of no arguments correspond to updatable variables.

For example, functions corresponding to arithmetic operations are static, and so is the no-argument function *body* that gives the abstract syntax of the initial program; the no-argument function *pos* gives the position of the phrase currently being executed in the tree representing what remains to be executed, which is itself represented by the 1-argument dynamic function $restbody : Pos \to Phrase$, where the set *Phrase* contains not only all possible abstract syntax trees, but also computed values, and trees where some nodes have been replaced by their computed values. [2]

▷ Transitions assign values to functions for particular arguments

A transition may simultaneously assign values for several functions on various arguments. Each assignment may be conditional, depending on the values of terms formed from the function symbols. All the terms involved in a simultaneous set of assignments are evaluated before any of the assignments are actually made, so the testing of the conditions and the resulting state are independent of the order of the assignments. The values of functions on particular arguments only change due to explicit assignment: their values on other arguments remain stable.

[2] The idea of gradually replacing phrases by their computed values, familiar from SOS, has only recently been adopted in the ASM framework: in earlier ASM specifications, the original tree was static, and a separate dynamic function was used to associate computed values with the nodes of the tree.

▷ ASM specifications often introduce auxiliary notation

The introduction of appropriate auxiliary notation allows transition rules to be specified rather concisely. However, ASM specifications of different languages tend to introduce different auxiliary notation, which leads to quite varied specifications of the same construct, and makes it difficult to reuse a transition rule from one ASM directly in another ASM. For example, the auxiliary notation introduced in the ASM specification of Java [24] includes:

- $context(pos)$, returning either $restbody(pos)$ or $restbody(up(pos))$; and
- $yieldUp(V)$, abbreviating the transition $restbody := restbody[V/up(pos)]$ performed simultaneously with $pos := up(pos)$, thus combining the replacement of a phrase by its computed result V with the adjustment of pos.

To "streamline the notation", positions are indicated directly in patterns for phrases, the current value of pos being written ▶. After these preliminaries, transition rules for evaluating Java's conditional expressions can be specified as follows:

$$
\begin{array}{l}
execJavaExp_I = \textbf{case}\ context(pos)\ \textbf{of} \\
\quad \ldots \\
\quad \texttt{cond}({}^{\alpha}E_1, {}^{\beta}E_1, {}^{\gamma}E_1) \quad \rightarrow \quad pos := \alpha \\
\quad \texttt{cond}({}^{\blacktriangleright}V_1, {}^{\beta}E_2, {}^{\gamma}E_3) \quad \rightarrow \quad \textbf{if}\ V_1\ \textbf{then}\ pos := \beta\ \textbf{else}\ pos := \gamma \\
\quad \texttt{cond}({}^{\alpha}\texttt{true}, {}^{\blacktriangleright}V_2, {}^{\gamma}E_3) \quad \rightarrow \quad yieldUp(V_2) \\
\quad \texttt{cond}({}^{\alpha}\texttt{false}, {}^{\beta}E_2, {}^{\blacktriangleright}V_3) \quad \rightarrow \quad yieldUp(V_3) \\
\quad \ldots
\end{array}
\qquad (11.22)
$$

Note that transitions are specified by assignments written with ':=', and the '→'s above are merely part of the '**case**' notation for pattern-matching (which is not itself provided by the ASM framework, but introduced *ad hoc*) in [24].

▷ Bindings are modelled by stacks of frames in ASM

The dynamic no-argument function $locals : (Id, Val)Map$ gives maps from local variable identifiers directly to the values that they are storing. To cope with redeclaration of local variables and with recursive procedural activation (both of which may require different values for the same variable identifier to coexist), a stack of frames is maintained, each frame storing the relevant $locals$. Thus the transition for an assignment expression $\texttt{update}(I, {}^{\blacktriangleright}V)$, where the new value V has already been computed, can be specified by $locals := \{I{=}V\}/locals$, without overwriting the value of other active local variables having the same identifier I (the notation used for maps in [24] is actually slightly different).

It might seem more natural to treat *locals* as a unary dynamic function from variable identifiers to values, but the ASM framework is first-order, and doesn't allow functions themselves to be used as values.

▷ Exceptions are modelled by propagation

In the Definition of Standard ML, an informal "exception convention" was introduced, so that a lot of tedious transition rules could be left implicit. In the ASM specification of Java, the propagation of raised exceptions is specified explicitly by introducing a predicate *propagatesAbr* on phrases, then using a special pattern $phrase(^{\blacktriangleright}A)$ which matches arbitrary phrases that have a raised exception A as any immediate component.

▷ Multiple threads can be modelled in various ways

Separate ASM agents can be set up to execute threads independently, with access to the same storage. Synchronization between threads can be achieved using dynamic functions which indicate that particular threads have locked particular storage areas (since a lock can be both tested and conditionally set in a single transition).

In the cited ASM for Java, however, a different approach is used—motivated mainly by the requirement that the model should be executable using a recently-developed prototyping system called AsmGofer. The idea is that at every transition during a computation for a multi-threaded Java program, an active thread is selected arbitrarily, using a so-called choice function, which is itself left unspecified. In contrast to the situation with Distributed ASMs, this modelling of thread on a single ASM allows computations that perpetually switch between threads, without making any actual progress.

11.4 Denotational Semantics

The framework of Denotational Semantics was developed by Scott and Strachey at Oxford in the late 1960's [25, 26, 27]. One of the main aims was to provide a proper mathematical foundation for reasoning about programs and for understanding the fundamental concepts of programming languages. Denotational Semantics has since been used in teaching [3, 5, 28] as well as in research. It has also been used to define the functional programming language Scheme; attempts to give denotational semantics for larger programming languages have generally been less successful, although several major descriptions have been given using a notational variant of denotational semantics called VDM [29].

Denotations

▷ The denotation of a part of a program represents its contribution to overall behaviour

The denotation of a construct is typically a function from arguments that represent the information available before its execution, and the result represents the information available afterwards. The intermediate states during the execution of the construct are generally of no relevance (except when interleaving is allowed) and are thus not represented, cf. big-step SOS (Natural Semantics). Usually, non-termination is represented by a special value written $\perp$ (the bottom element in a partial ordering based on information content).

▷ Denotations are defined inductively, using λ-notation to specify how the denotations of components are to be combined

Semantic functions map constructs to their denotations. For example, let *Exp* be the abstract syntax of expressions, and let *Den* be the set of all (potential) denotations of expressions. A semantic function for expressions:

$$\mathcal{E} : \mathrm{Exp} \to Den$$

is defined inductively by semantic equations such as:

$$\mathcal{E}[\![\mathsf{cond}(E_1, E_2, E_3)]\!] = F(\mathcal{E}[\![E_1]\!], \mathcal{E}[\![E_2]\!], \mathcal{E}[\![E_3]\!]) \tag{11.23}$$

where $F : Den^3 \to Den$ is defined using so-called λ-*notation*, which is a mathematical notation for function abstraction (a function with argument x is written $\lambda x.t$), application, and composition, extended with a case construct and a few other useful features. Note that both $\mathcal{E}$ and F above are higher-order functions, assuming that *Den* is a set of functions.

▷ Denotations of loops and recursive procedures are least fixed-points of continuous functions on Scott-domains

To define the denotation d of a loop, for instance, we need to be able to provide a well-defined solution to $d = F(d)$, where $F(d)$ is a particular composition of d with the denotations of the loop condition and body. It turns out that such an equation always has a solution, and in particular it has a *least* solution—provided only that $F : Den \to Den$ is *continuous* in a certain sense on *Den*, which has to be a Scott-domain: a *cpo* (complete partially-ordered set). In fact F is *always* continuous when defined using λ-notation, so let us not bother here with further details of the mathematical foundations of Denotational Semantics.

Direct and Continuation-Passing Styles

▷ The denotational semantics of a purely functional language may be in *direct* or in *continuation-passing* style

Using the direct style, let $Den = \mathrm{Val}_\perp$; then the denotations of conditional expressions can be defined as follow:

$$\begin{aligned}\mathcal{E}[\![\texttt{cond}(E_1,E_2,E_3)]\!] = \\ \quad \text{case } \mathcal{E}[\![E_1]\!] \text{ of } \texttt{true} \Rightarrow \mathcal{E}[\![E_2]\!] \mid \\ \qquad\qquad \texttt{false} \Rightarrow \mathcal{E}[\![E_3]\!]\end{aligned} \tag{11.24}$$

Note that if the denotation of E_1 is $\perp$, then so is that of the whole conditional expression; this reflects that if the evaluation of E_1 never terminates, then neither does that of the enclosing expression. If the evaluation of E_1 does terminate, it should give either `tt` or `ff`, which are here the denotations of the corresponding boolean constants:

$$\mathcal{E}[\![\texttt{true}]\!] = \texttt{tt} \qquad\qquad \mathcal{E}[\![\texttt{false}]\!] = \texttt{ff} \tag{11.25}$$

(In Denotational Semantics, one generally avoids use of syntactic phrases such as `true` and `false` in the set of denotations.)

The so-called continuation style of denotational semantics looks rather different. Here one would take $Den = K \to A$, where $K = \mathrm{Val} \to A$ and A is some set of values representing the possible results of executing complete programs (e.g. for Standard ML, A would be Val together with some values representing unhandled exceptions). The idea is that the continuation given as argument to $\mathcal{E}[\![E_1]\!]$ is supposed to be applied to the value computed by E_1; if E_1 never terminates, or terminates exceptionally, the continuation is simply ignored. The continuation for E_1 involves the denotations of E_2 and E_3, which are both given the continuation k provided for the entire conditional expression.

$$\begin{aligned}\mathcal{E}[\![\texttt{cond}(E_1,E_2,E_3)]\!] = \\ \quad \lambda k.\mathcal{E}[\![E_1]\!](\lambda t.\text{case } t \text{ of } \texttt{tt} \Rightarrow \mathcal{E}[\![E_2]\!]k \mid \\ \qquad\qquad \texttt{ff} \Rightarrow \mathcal{E}[\![E_3]\!]k)\end{aligned} \tag{11.26}$$

If E_1 is simply `true`, its denotation applies the continuation k to the corresponding value:

$$\mathcal{E}[\![\texttt{true}]\!] = \lambda k.k(\texttt{tt}) \qquad\qquad \textit{etc.} \tag{11.27}$$

▷ Bindings are represented by explicit arguments of denotations

Regardless of whether denotations are in the direct style or using continuations, the dependency of the values of identifiers on the bindings provided by their context is represented by letting denotations be functions of environments. For instance, let $Den = \text{Env} \to \text{Val}_\perp$; then the direct semantics for conditional expressions would be formulated as follows:

$$\begin{aligned}\mathcal{E}[\![\mathsf{cond}(E_1,E_2,E_3)]\!] = \\ \quad \lambda\rho.\text{case } \mathcal{E}[\![E_1]\!]\rho \text{ of } \mathtt{tt} \Rightarrow \mathcal{E}[\![E_2]\!]\rho \mid \\ \qquad\qquad\qquad\qquad \mathtt{ff} \Rightarrow \mathcal{E}[\![E_3]\!]\rho\end{aligned} \tag{11.28}$$

The denotation of an identifier simply applies the environment to the identifier itself:

$$\mathcal{E}[\![I]\!] = \lambda\rho.\rho(I) \tag{11.29}$$

▷ Effects on storage are represented by letting denotations be functions from stores to stores

It might seem that the easiest would be to add stores as arguments and results to the direct-style denotations given above, taking $Den = \text{Env} \to (\text{Store} \to (\text{Val} \times \text{Store})_\perp)$. However, that would lead to the following semantic equation for conditional expressions, which is not as perspicuous as one might wish:

$$\begin{aligned}\mathcal{E}[\![\mathsf{cond}(E_1,E_2,E_3)]\!] = \\ \quad \lambda\rho.\lambda\sigma.(\lambda(t,\sigma').\text{case } t \text{ of } \mathtt{tt} \Rightarrow \mathcal{E}[\![E_2]\!]\rho\,\sigma' \mid \\ \qquad\qquad\qquad\qquad \mathtt{ff} \Rightarrow \mathcal{E}[\![E_3]\!]\rho\,\sigma') \\ \qquad (\mathcal{E}[\![E_1]\!]\rho\,\sigma)\end{aligned} \tag{11.30}$$

So let us instead try adding stores to the denotations used with the continuation-style semantics. The appropriate set of denotations is then $Den = \text{Env} \to K \to C$, where $K = \text{Val} \to C$, and $C = \text{Store} \to A$, and we may give a relatively straightforward-looking semantic equation—not even mentioning the stores, which automatically follow the flow of control in continuation semantics:

$$\begin{aligned}\mathcal{E}[\![\mathsf{cond}(E_1,E_2,E_3)]\!] = \\ \quad \lambda\rho.\lambda k.\mathcal{E}[\![E_1]\!]\rho(\lambda t.\text{case } t \text{ of } \mathtt{tt} \Rightarrow \mathcal{E}[\![E_2]\!]\rho\,k \mid \\ \qquad\qquad\qquad\qquad \mathtt{ff} \Rightarrow \mathcal{E}[\![E_3]\!]\rho\,k)\end{aligned} \tag{11.31}$$

▷ Nondeterminism and interleaving can be represented by letting denotations be set-valued functions

In operational frameworks based on transition relations, the possibility of nondeterministic computations doesn't make any difference to how rules are formulated. In denotational semantics, however, the use of functions as denotations means that the ranges of the functions have to be changed to allow them to return sets of possible results; moreover, other functions that are to be composed with these set-valued functions have to be extended to functions that takes sets as arguments. As one may imagine, the extra notation required leads to further complication of the specifications of denotations.

Since interleaving generally entails nondeterminism, its denotational description obviously requires the use of set-valued functions. However, a further problem arises: it simply isn't possible to compose functions that map initial states to (sets of) final states so as to obtain a function corresponding to their interleaving at intermediate states. So-called *resumptions* are needed: these are functions representing the points at which interleaving can take place, and correspond closely to the computations used in operational semantics.

Monadic Semantics

▷ Use of monadic notation and monad transformers gives good modularity

The straightforward use of λ-notation to specify how denotations are combined requires awareness of the exact structure of the denotations: whether they are functions of bindings, stores, continuations, etc. When new constructs are added to a language, it may be necessary to change the structure of the denotations, and reformulate all the semantic equations that involve those denotations. Thus it appears that use of λ-notation is a major hindrance to obtaining modularity in denotational descriptions.

However, suppose that we define some *auxiliary notation* for combining denotations, corresponding to fundamental concepts such as sequencing. We may then be able to specify denotations using the auxiliary notation, without any dependence on the structure of denotations. If we later change that structure, we shall also have to change the definition of the auxiliary notation—but the use of that notation in the semantic equations may remain the same.

Monadic Semantics provides a particular auxiliary notation for use in Denotational Semantics. It was developed by Moggi at the end of the 1980's [30, 31], and inspired by category-theoretic concepts. The basic idea is that denotations compute values of particular types; when two such denotations are sequenced, the value computed by the first one is made available to the second one, written

let $x = d_1$ in d_2 (where d_2 usually depends on x). The only other bit of essential notation is for forming a denotation that simply computes a particular value v, which is written $[v]$. A set of denotations equipped with this notation may be regarded as a mathematical structure called a *monad*. Here is how the semantic equation for conditional expressions looks in the monadic variant of Denotational Semantics:

$$\begin{aligned}\mathcal{E}[\![\texttt{cond}(E_1,E_2,E_3)]\!] = \\ \quad \text{let } t = \mathcal{E}[\![E_1]\!] \text{ in} \\ \quad \text{case } t \text{ of } \texttt{tt} \Rightarrow \mathcal{E}[\![E_2]\!] \mid \texttt{ff} \Rightarrow \mathcal{E}[\![E_3]\!]\end{aligned} \tag{11.32}$$

Note that as well as being independent of the structure of denotations, the monadic semantic equation is also more perspicuous and suggestive of the intended semantics than our previous semantic equations were.

How about further ways of combining denotations that might be needed, but which are not based on sequencing? Some *monad transformers* are available: fundamental ways of adding features to denotations (bindings, effects on storage, exceptions, nondeterministic choice, interleaving, etc.), together with appropriate notation. Unfortunately, it isn't always so straightforward to combine different monad transformers, and difficulties can arise when trying to redefine auxiliary notation in connection with applying a monad transformer. (We shall consider Action Semantics, a hybrid framework that doesn't suffer from such problems, at the end of this chapter.)

11.5 Axiomatic Semantics

Axiomatic Semantics was developed primarily by Hoare in the late 1960's [32]. The main aim was initially to provide a formal basis for the verification of abstract algorithms; later, the framework was applied to the definition of programming languages, and consideration of Axiomatic Semantics influenced the design of Pascal [33].

▷ A Hoare Logic gives rules for the relation between assertions about values of variables before and after execution of each construct

Usually, the constructs concerned are only commands C. Suppose that P and Q are assertions about the values of particular variables; then the so-called *partial correctness* formula $P\{C\}Q$ states that if P holds at the beginning of an execution of C and the execution terminates, Q will always hold at the end of that execution. Notice that $P\{C\}Q$ does not require C to terminate, nor does it require Q to hold after an execution of C when P didn't hold at the beginning of the execution.

A Hoare Logic specifies the relation $P\{C\}Q$ inductively by rules, in the same way that as an SOS specifies a transition relation.

▷ Expressions are assumed to have no side-effects

Expressions are used in assertions, so their interpretation has to be purely mathematical, without effects on storage, exceptions, non-terminating function calls, etc. For example, consider conditional commands with the following abstract syntax, where the conditions are restricted to pure boolean-valued expressions:

$$\text{Cmd} ::= \texttt{cond}(\text{Exp}, \text{Cmd}, \text{Cmd})$$

A typical rule given for this construct is:

$$\frac{(P \wedge E)\{S_1\}R, \qquad (P \wedge \neg E)\{S_2\}R}{P\{\texttt{cond}(E, C_1, C_2)\}R} \tag{11.33}$$

Notice the use of E as a sub-formula in the assertions. Similarly, the usual rule for assignment:

$$P[E/I]\{\texttt{update}(I, E)\}P \tag{11.34}$$

involves the substitution $P[E/I]$ of an expression E for an identifier I in an assertion P.

▷ Bindings can be represented by explicit environments

In many presentations of Hoare Logic, bindings are left implicit: the relation $P\{C\}Q$ is defined on the basis of a fixed set of bindings. To reflect local declarations, it is necessary to use more elaborate formulae such as $\langle ENV \mid P\{C\}Q\rangle$, where the current bindings ENV are made explicit.

▷ Hoare Logic for concurrent processes involves rules for interleaving

Hoare Logic is exploited in connection with the development and verification of concurrent processes. The rules can get rather complicated.

▷ Predicate transformer semantics is essentially denotational

In connection with a methodology for developing programs from specifications, Dijkstra defined, for each command C and postcondition Q, the weakest precondition P guaranteeing total correctness: if P holds at the beginning of the execution of C, then C always terminates, and Q holds at the end of the execution. Although the assertions used here are similar to those in Hoare Logic, the definition of the weakest precondition P is actually inductive in the structure of the command C, and Dijkstra's semantics is better considered as denotational (with the denotations being predicate transformers, i.e. functions on the interpretation of assertions) rather than axiomatic.

11.6 Action Semantics

The Action Semantics framework was developed by the present author, in collaboration with Watt, in the second half of the 1980's [34, 35, 36].

▷ Action Semantics is a hybrid of denotational and operational semantics

As in denotational semantics, inductively-defined semantic functions map phrases to their denotations, only here, the denotations are so-called *actions*; the notation for actions is itself defined operationally [34].

▷ Action Semantics avoids the use of higher-order functions expressed in λ-notation

The universe of pure mathematical functions is so distant from that of (most) programming languages that the representation of programming concepts in it is often excessively complex. The foundations of reflexive Scott-domains and higher-order functions are unfamiliar and inaccessible to many programmers (although the idea of functions that take other functions as arguments, and perhaps also return functions as results, is not difficult in itself).

▷ Action semantics provides a rich action notation with a direct operational interpretation

The universe of actions involves not only control and data flow, but also scopes of bindings, effects on storage, and interactive processes, allowing a simple and direct representation of many programming concepts.

Computed values are given by actions, and the action combination A_1 *then* A_2 passes all the values given by A_1 to A_2. For example, assuming `evaluate` : Exp $\rightarrow$ *Action*, the value computed by `evaluate` E_1 is the one tested by the action *given true* below:

$$\begin{array}{l}\texttt{evaluate cond}(E_1,E_2,E_3) = \\ \quad \texttt{evaluate}\ E_1\ \textit{then} \\ \quad (\ \textit{given true then}\ \texttt{evaluate}\ E_2\ \textit{otherwise}\ \texttt{evaluate}\ E_3\ )\end{array} \tag{11.35}$$

Bindings are implicitly propagated to the sub-actions of most actions, and can always be referred to, as illustrated below:

$$\texttt{evaluate}\ I = \textit{give the val bound to } I \tag{11.36}$$

Effects on storage implicitly follow the flow of control:

$$\begin{array}{l}\texttt{evaluate update}(E_1, E_2) = \\ \quad \texttt{evaluate } E_1 \textit{ and } \texttt{evaluate } E_2 \\ \quad \textit{then update}(\textit{the loc\#1}, \textit{the val\#2})\end{array} \tag{11.37}$$

Concurrent processes are represented by agents that perform separate actions, with asynchronous message-passing.

Summary

In this chapter, we have considered the main alternatives to our MSOS framework for semantic description, giving fragments to illustrate the different styles of specification that are used there.

Most of the alternative frameworks have significant disadvantages regarding the modularity of semantic descriptions, severely limiting the possibility of reusing parts of the specification of one language in the specification of another:

- The original SOS framework may require reformulation of transition rules when the described language is extended.
- Reduction Semantics has reasonable modularity, but uses substitution to deal with scopes of bindings, which is tedious to specify.
- Semantic descriptions using Abstract State Machines tend to introduce *ad hoc* auxiliary notation to abbreviate the many details (e.g. concerning stacks of frames), which hinders reuse of transition rules in specifications of different languages.
- The direct use of λ-notation in Denotational Semantics requires reformulation of semantic equations when the described language is extended. Use of monadic notation eliminates this problem, but the foundations of the framework may be regarded as too abstract for use by programmers (despite the recent popularity of monadic techniques in functional programming).
- Axiomatic Semantics becomes quite complicated when used to describe real programming languages such as Java, and in any case is a somewhat indirect way of defining models for programming languages.
- Action Semantics is as modular as MSOS, but it may be regarded as a disadvantage that it requires familiarity with Action Notation, which has somewhat more constructs than the notations underlying the other frameworks.

MSOS thus appears to have significant advantages over all the alternative frameworks, at least regarding the actual *specification* of semantics. Proving properties of programs and languages is quite demanding in all the frameworks (apart from in Axiomatic Semantics, which is closely related to the specification of *invariants* in programs); it remains to be seen whether MSOS has definite disadvantages over other frameworks in this respect.

Chapter 12

Conclusion

These lecture notes are about the fundamental concepts and formal semantics of programming languages. Let us conclude by summarizing the topics covered in the preceding chapters, noting some points where improvements are needed.

12.1 Fundamental Concepts

Chapters 1 and 5–10 introduced various linguistic and computational concepts associated with expressions, declarations, commands, abstractions, concurrency, and types. Although the collection of concepts covered was by no means comprehensive, it was adequate as a basis for a conceptual analysis of many constructs from Standard ML, as well as some from Concurrent ML.

Students at Aarhus take a course on functional programming in ML and (to a lesser extent) logic programming in Prolog before the course that is based on these notes. The illustrations of conceptual analysis of particular ML constructs given here may have deepened their understanding of these constructs. However, the main purpose was to increase their familiarity with the underlying fundamental concepts and abstract constructs, and facilitate the study of the formal semantics of the constructs. The treatment in the sections on fundamental concepts was deliberately rather informal.

12.2 Formal Semantics

Chapter 1 introduced the basic ideas of formal descriptions of programming languages, distinguishing between different kinds of syntax (concrete v. abstract, context-free v. context-sensitive) and semantics (static v. dynamic, operational v. denotational). Chapter 2 explained the foundations of MSOS, our modular variant

of the structural approach to operational semantics, and presented MSDF, a metanotation for MSOS specifications. Chapter 3 explained how MSOS descriptions of individual constructs and entire languages are organized so as to facilitate reuse. Chapter 4 introduced the techniques of structural induction and bisimulation for giving modular proofs of properties of MSOS models.

Chapters 5–9 illustrated the use of MSOS to describe the dynamic semantics of various common programming constructs. The semantics of expressions involved flow of control and computed values; fixed components of labels were needed for the semantics of declarations, and changeable components for commands. The semantics of procedural abstractions and recursion was based on forming closures. Our treatment of concurrency was quite conventional, exploiting produced information in labels to model process creation and communication.

The use of MSOS for static semantics is currently under investigation; Chapter 1 illustrated a monolithic static semantics for a sublanguage of bc, but the static semantics of the constructs introduced in the later chapters needs further development and validation before it can be included in Chapter 10.

Chapter 11 gave an overview of various alternative frameworks for formal semantics. The focus was on the "look and feel" of semantic descriptions, and the aim was merely to ensure awareness of the existence of the other frameworks, together with their main features.

In general, the claims made for the modularity of MSOS have been substantiated by the illustrative examples: descriptions of individual constructs were formulated quite independently, in general, and remained unchanged when combined with further constructs in connection with the description of particular languages.

New in the present version of these notes was the introduction of MSDF, a precise metanotation for MSOS, together with software to translate MSDF modules to Prolog. This relieved students (and the author) of having to translate rules manually from MSOS to Prolog, thereby substantially increasing the accuracy of validation. However, further exercises should be provided in writing MSOS descriptions of new constructs, and in using the provided software to validate the descriptions, together with better documentation.

It had been hoped that students would easily grasp how MSOS rules work by following the intermediate states of computations using the provided software, but this does not appear to have been the case, in general. Exercises involving manual derivation of entire computations seem to be essential.

Finally, the author had underestimated the extent of revision needed to the previous version of these notes in connection with the introduction of MSDF. As a consequence, the production of this version was delayed, and the final result was less polished than intended. In particular, many of the exercises in the later chapters were too hastily adapted from the previous version, and better exercises should be developed.

Bibliography

[1] Gordon D. Plotkin. A structural approach to operational semantics. *J. Logic and Algebraic Programming*, 60–61:17–139, 2004. Originally published as DAIMI FN-19, Dept. of Computer Science, Univ. of Aarhus, 1981.

[2] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley, New York, 1990.

[3] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, Chichester, UK, 1992.

[4] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.

[5] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[6] Peter D. Mosses. Modular structural operational semantics. *J. Logic and Algebraic Programming*, 60–61:195–228, 2004. Special issue on SOS.

[7] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[9] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. Learn Prolog now! `http://www.learnprolognow.org/`, 2003.

[10] Free Software Foundation. *GNU bc Manual*, 2001. `http://www.gnu.org/software/bc/`.

[11] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[12] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[13] Peter D. Mosses. Foundations of modular SOS. BRICS RS-99-54, Dept. of Computer Science, Univ. of Aarhus, 1999.

[14] Peter D. Mosses. Pragmatics of modular SOS. In *AMAST'02*, LNCS 2422, pages 21–40. Springer, 2002.

[15] SDF – modular syntax definition formalism. `http://www.program-transformation.org/Sdf/SdfLanguage`.

[16] John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[17] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[18] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[19] Luca Aceto, Wan Fokkink, and Chris Verhoef. Structural operational semantics. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, chapter 3. Elsevier Science, 2001.

[20] Gilles Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

[21] Matthias Felleisen and Dan P. Friedman. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 193–217. North-Holland, 1987.

[22] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, volume B, chapter 6. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[23] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[24] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.

[25] Dana S. Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*. Polytechnic Institute of Brooklyn, 1971.

[26] Robert D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19:437–453, 1976.

[27] Peter D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[28] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn & Bacon, 1986.

[29] Dines Bjørner and Cliff B. Jones, editors. *Formal Specification & Software Development*. Prentice-Hall, 1982.

[30] Eugenio Moggi. An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Computer Science Dept., Univ. of Edinburgh, 1990.

[31] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[32] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[33] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[34] Peter D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[35] Peter D. Mosses and David A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 135–166. North-Holland, 1987.

[36] David A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

Extra Exercises

Be sure to update your copy of the MSOS repository before doing these exercises! Check that the directories `Lang/bc` *and* `Test/bc` *both contain subdirectories named '*`2`*'.*

Ex. 12.1 Consider the following simple bc program:

```
a = 2; if(a){2+2}
```

(a) Draw a diagram of the abstract syntax tree T to which this program is mapped when parsed by the DCG in `Test/bc/SYN.pro` (which is shown in Table 1.3).

(b) Check that T has the same structure as the abstract syntax term given for this program by the Prolog code accompanying these notes (cf. Chapter 3). Hint: Load `Tool/load` then use the following query:

```
parse('Test/bc', "a = 2; if(a){2+2}").
```

(c) The MSOS in `Test/bc/RUN.msdf` (shown in Tables 1.7–1.9) determines the dynamic semantics of T. Suppose that labels on transitions have only the required components, that the initial `env` component only maps `a` to `cell(1)`, and that the initial `store` component only maps `cell(1)` to `0`. Carefully derive each step of the computation for T by instantiating the specified transition rules, writing down a term or diagram for each state, and indicating the values of `store'` and `out'` for each step. (There should be 10 steps.)

(d) Check that each intermediate state of your derived computation has the same structure as the corresponding state given for the computation of this program by the Prolog code accompanying these notes (cf. Chapter 3). Hint: Load `Tool/load` then use the following query:

```
run('Test/bc', "a = 2; if(a){2+2}", 10, 0).
```

Ex. 12.2 This exercise involves adding for-loops to the concrete syntax of our sublanguage of bc.

(a) Add context-free grammar rules for the concrete syntax of for-loops to the file `Lang/bc/2/CFG.pro`.

(b) Write some test programs involving for-loops, and store them as files in the directory `Test/bc/2`.

(c) Check that all your test programs are accepted. Hint: Load `Tool/load` then use the following query:

```
accept('bc/2', ...).
```

where the '`...`' is replaced by the name of the file containing a test program. (Unless the file name is a word starting with a lowercase letter or a decimal numeral, it should be enclosed in single quotes.)

Ex. 12.3 This exercise involves mapping the concrete syntax of for-loops (see the preceding exercise) to the abstract syntax of our original sublanguage of bc. As indicated in the bc reference manual [10], for-loops can be expanded to combinations involving while-commands and command sequences, so no new abstract constructs are required for the semantics of this extension.

(a) Add DCG grammar rules for mapping the concrete syntax of bc for-loops in the file `Lang/bc/2/SYN.pro`.

(b) Check that all your test programs (as required for an earlier exercise) are mapped correctly to abstract syntax. Hint: Load `Tool/load` then use the following query:

```
parse('bc/2', ...).
```

where the '`...`' is replaced by the name of the file containing a test program. (Ignore any warnings about redefinition of static procedures.)

(c) Check that all your test programs have the correct dynamic semantics. Hint: Load `Tool/load` then use the following query:

```
run('bc/2', ...).
```

If you wish to see a particular intermediate state of the computation, give the number of the step as an extra argument in the above query. To see a series of intermediate states, give also the number of the first step to be shown as a further argument. For example, '`run ('bc/2', ..., 99, 0).`' shows all the states and labels, up to at most 99 steps. The queries '`hide_label.`' and '`show_label.`' control whether the label is shown.

Ex. 12.4 This exercise involves extending the concrete syntax of our sublanguage of bc with expressions for incrementing and decrementing variables, extending our abstract expressions with corresponding abstract constructs, mapping the concrete expressions to abstract constructs, and specifying the dynamic semantics of the new abstract constructs.

(a) Add context-free grammar rules for the concrete bc expressions '++var', '--var', 'var++', and'var--' to the file Lang/bc/2/CFG.pro.

(b) Write some test programs involving the new expressions, and store them as files in the directory Test/bc/2.

(c) Check that all your test programs are accepted by your grammar.

(d) Invent new abstract constructs that can be used to represent the new expressions without duplicating occurrences of variables, and add a subdirectory to Cons/Exp for each of them.

(e) Add the MSDF specifications of the abstract syntax of the new constructs as files named ABS.msdf to the new subdirectories, and generate the files named ABS.pro using the Prolog query:

```
cons('Exp/...').
```

replacing '...' each time by the name of a subdirectory. (Ignore any warnings about CHK.msdf and RUN.msdf not being found.)

(f) Add the MSDF specification to import the abstract syntax of the new constructs in the file Lang/bc/2/ABS.msdf, and generate the file named ABS.pro using the Prolog query:

```
lang('bc/2').
```

(Ignore any warnings about CHK.msdf and RUN.msdf not being found.)

(g) Add DCG rules for mapping the new bc concrete expressions to abstract constructs in the file Lang/bc/2/SYN.pro.

(h) Check that all your test programs (as required above) are mapped correctly to abstract syntax. Hint: Load Tool/load then use the following query:

```
parse('bc/2', ...).
```

where the '...' is replaced by the name of the file containing a test program.

(i) Add the MSDF specifications of the dynamic semantics of the new abstract constructs as files named RUN.msdf to the new subdirectories, and generate the files named RUN.pro using the Prolog query:

```
cons('Exp/...').
```

replacing '`...`' each time by the name of a subdirectory. (Ignore any warnings about `CHK.msdf` not being found.)

(j) Add the MSDF specification to import the dynamic semantics of the new constructs in the file `Lang/bc/2/RUN.msdf`, and generate the file named `RUN.pro` using the Prolog query:

```
lang('bc/2').
```

(Ignore any warnings about `CHK.msdf` not being found.)

(k) Check that all your test programs have the correct dynamic semantics. Hint: Load `Tool/load` then use the following query:

```
run('bc/2', ...).
```

(l) Congratulate yourself, and take a well-deserved break!

Ex. 12.5 This exercise concerns semantic equivalence for bc constructs, based on their dynamic semantics as specified in Chapter 1.

For each potential equivalence below, indicate whether you expect the two constructs to be interchangeable in all bc programs. (Partially) justify your answer in each case by giving a strong or weak bismulation including the pair of constructs, or an example that shows they are not interchangeable. (The metavariable `N` ranges over arbitrary integers, `C` ranges over arbitrary commands, and `I` ranges over pre-declared variable identifiers.)

(a) `cond-nz(N,skip)` $\equiv$ `skip`.

(b) `while-nz(N,skip)` $\equiv$ `cond-nz(N, while(1,skip))`.

(c) `while-nz(1,skip)` $\equiv$ `seq(while-nz(1,skip),C)`.

(d) `tup-seq(assign-seq(I,N),I)` $\equiv$ `tup-seq(assign-seq(I,N),N)`.

(e) `tup-seq(assign-seq(I,N1),assign-seq(I,N2))` $\equiv$ `tup-seq(N1,assign-seq(I,N2))`.

(f) `effect(assign-seq(I,I))` $\equiv$ `skip`.

(g) `seq(effect(assign-seq(I,N1)),effect(assign-seq(I,N2)))` $\equiv$ `effect(assign-seq(I,N2))`.

(h) `seq(effect(assign-seq(I1,N1)),effect(assign-seq(I2,N2)))` $\equiv$ `seq(effect(assign-seq(I2,N2)),effect(assign-seq(I1,N1)))`.