MODULAR DENOTATIONAL SEMANTICS

Roberto S. Bigonha

Modular Denotational Semantics

Roberto S. Bigonha

Belo Horizonte

Minas Gerais – Brazil

05 de junho de 2021

Cataloging Data

B594	Bigonha, Roberto S. Modular Denotational Semantics / Roberto S. Bigonha — Belo Horizonte, MG, 2021 256p Bibliography. ISBN 978-65-00-22442-9 1. Computer Science 2. Computing Engineering 3. Informatics 4. Formal Semantics 5. Denotational Semantics 6. Separation of Concerns I. Bigonha, Roberto S. CDD-004.07

Prologue

Denotational semantics is a powerful and elegant formalism for describing the meaning of programming language constructs. However, it is used less than it should be. Apparently, the lack of popularity stems from the difficulties that most programmers and even computer scientists have to understand formal semantics definitions. And probably these difficulties are inherent to the way formal descriptions have been organized so far.

This book aims to contribute to the denotational model by offering means to enhance the comprehensibility of semantic descriptions of languages of realistic size. It seeks to answer the quest for legible formal semantics by proposing a semantics-definition style based on the syntactic structure of the language and on the concept of separation of concerns.

The book's thesis is that a disciplined and well structured semantic presentation will enhance comprehensibility of descriptions in the formalism, so that they would not be more complicated than ordinary computer programs written in a high level language.

This book has been designed for computer science students who have good knowledge of programming computers in high level languages. No advanced mathematics is required. Acquaintance with sets and functions is enough to grasp the basic concepts on denotational semantics.

Chapter 1 describes $\mathcal{M}$, a meta-language specially designed and proposed to support the methodology herein described for developing modular and comprehensible presentations of

denotational semantics.

Chapter 2 presents a complete description of the architecture of a hypothetical computer.

Chapter 3 presents the definition of a compiler for translating programs written in a small imperative language into the code of the machine described in the preceeding chapter.

Chapter 4 presents a review of the techniques and foundations of Standard Denotational Semantics, so as to make this textbook self-contained.

Chapter 5 introduces a technique to mix direct and continuation semantics, in order to get the best of both worlds.

Chapter 6 describes in detail a complete and modular definition of a simple and yet revelatory imperative programming language, with the objective of highlighting the methodology proposed for structuring formal definitions.

Chapter 7 shows how to improve the presentation of the description developed in the preceding chapter by reorganizing it to promote even deeper separation of concerns by means of denotational components.

Chapter 8 addresses the foundations of Dana Scott's theory of domains, which underlies the denotational semantics model.

Chapter 9, the epilogue, gives the due credits to some of the pillars of the Field.

This is deliberately a very concise book on a vast and complex subject. Hopefully, this conciseness is in conformance with the ancient idea that the primordial purpose of the words is to improve the silence.

Acknowledgments

First and foremost, I would like to express my gratitude to late professor David F. Martin for his high-quality lectures and remarkable guidance skills that he offered to his students at the University of California, at Los Angeles, and for having introduced me to the fundations of programming language semantics. It was a privilege to have worked closely with him.

I would also like to thank all my undergraduate, master and doctorate students who did an immense amount of constructive work in the development of their research on subjects closely related to this book. I have learned a lot from them.

I wanted to avoid singling out their names, for they are many, and any omission would be unforgivable. It is always unfair to single out individuals in such a context.

However, it would be even more unfair not to cite those who directly cooperated with me over many parts of my search for legible semantics. So I would like to specially mention Elaine Gouvea Pimentel, Fabio Tirelo and Guilherme Henrique Souza Santos, as important collaborators.

Roberto S. Bigonha

Contents

1 The Meta-Language for Semantics Definitions 1

1.1 Basic structures 2
1.2 Built-in domains 6
1.3 Declarations 9
1.4 Domains 12
1.4.1 Domain of enumerations 13
1.4.2 Domain constants 14
1.4.3 Domain of tuples 14
1.4.4 Domain of lists 14
1.4.5 Domain of tree nodes 15
1.4.6 Domain of functions 15
1.4.7 Union of domains 16
1.4.8 Domain equivalence and compatibility 17
1.5 Expressions 19
1.5.1 Functional expressions 19
1.5.2 Pattern expressions 22
1.5.3 Conditional expressions 23
1.5.4 Basic expressions 24
1.5.5 Logical expressions 26
1.5.6 Integer expressions 27
1.5.7 Quotations 28
1.5.8 Fixpoint operator 30

1.5.9 Tuple expressions 30
1.5.10 List expressions 31
1.5.11 Node expressions 33
1.5.12 Mapping expressions 34
1.6 Compilation units 35
1.7 Interface modules 36
1.7.1 Imports section 37
1.7.2 Privates e publics sections 39
1.8 Definition modules 41
1.8.1 Lexis section 42
1.8.2 Syntax section 47
1.8.3 Functions section 54
1.9 The main module 61
1.10 Module `System` 64

2 The Description of a Computer Architecture 65
2.1 The machine architecture 65
2.1.1 The environment 66
2.1.2 The store 68
2.1.3 The stack 69
2.1.4 The dump 70
2.1.5 Files 71
2.1.6 Continuations 73
2.2 Program execution 76
2.3 Machine instructions 77
2.3.1 Flow of execution 78
2.3.2 Instruction set 78
2.4 Concluding remarks 85

3 The Specification of a Compiler 87
3.1 Machine instructions 87
3.2 The compiler specification 88
3.2.1 Concrete and abstract syntax 89
3.2.2 Translation rules 90
3.3 Concluding Remarks 93

4 Standard Denotational Semantics 95
4.1 Direct semantics of a simple language 97
4.1.1 Concrete and abstract syntaxes 98
4.1.2 Informal semantics 99
4.1.3 Semantic domains 99
4.1.4 Semantic equations 100
4.1.5 A worked example 102
4.2 Standard semantics model 103
4.2.1 Standard environments and stores . . 104
4.2.2 Domains of standard values 105
4.2.3 The notion of continuations 106
4.2.4 Standard continuations 109
4.3 Continuation semantics of Simple 111
4.3.1 Semantic domains 112
4.3.2 Semantic equations 113

5 Retractile Continuations 117
5.1 Conciliation of semantics styles 118
5.2 An example 120
5.3 Conclusion 124

6 Syntax-Driven Methodology 125
6.1 Syntax-directed module structure 127

6.2 The semantics of Small 129
6.2.1 Semantic infrastructure 130
6.2.2 Function main 140
6.2.3 Small programs 141
6.2.4 Small declarations 142
6.2.5 Small commands 146
6.2.6 Small expressions 150
6.2.7 Small tokens 154
6.3 Evaluation 156

7 Component-Based Style 157
7.1 The fundamental principle 159
7.2 Context removal 160
7.3 Component-based semantics of Small 164
7.3.1 Main function 165
7.3.2 Small programs 166
7.3.3 Small declarations 167
7.3.4 Small commands 170
7.3.5 Small expressions 174
7.3.6 Small tokens 177
7.4 The denotational components for Small . . . 178
7.4.1 Components for declarations 178
7.4.2 Components for commands 179
7.4.3 Components for expressions 180
7.5 Discussion 182

8 Domain Theory 185
8.1 Theoretical problems 186
8.1.1 Recursive definition of functions . . . 188
8.1.2 Recursive definition of sets 189

8.1.3 Program nontermination 189
8.2 The work of Scott 190
8.3 Foundations 191
8.3.1 Functions 191
8.3.2 Relations on sets 192
8.3.3 Partially ordered sets 193
8.3.4 Complete partial order 194
8.4 Scott domains 195
8.5 Flat domains 198
8.6 Cartesian product 199
8.7 Domain union 200
8.8 Domain of sequences 201
8.9 Domain of functions 201
8.9.1 Monotonic functions 201
8.9.2 Continuous functions 202
8.9.3 Domain mapping 204
8.9.4 Composition of functions 208
8.10 LAMBDA functions 208
8.11 $\mathcal{M}$ functions 209
8.12 Fixpoints 210
8.12.1 Calculation of fixpoints 212
8.12.2 The paradoxal operator Y 217
8.13 Final comments 217

9 Epilogue 219

Chapter 1

The Meta-Language for Semantics Definitions

Entia non sunt multiplicanda praeter necessitatem.[1]
William of Ockham (1285-1349)

$\mathcal{M}$ is a pure functional domain-specific language aimed to provide a well-suited notation for conveying comprehensible denotational descriptions of programming language semantics. To that end, issues such as description modularization, structural type equivalence, control of visibility, encapsulation, and information hiding have been incorporated in the language's structure.

Formal semantic definitions in $\mathcal{M}$ are composed of modules, each containing one or more definition elements, which can be concrete grammars, definition of tokens, declaration of variables and domains, and the specification of semantic equations. There are two types of modules: interface modules, which contain declarations and import/export specifications, and definition modules, which contain definitions of concrete

[1] Entities must not be multiplied beyond necessity.

grammars, tokens, and semantic functions.

In a complete language definition, there must be a special module that contains the definition of the standard function `main` (*§1.9, page 61*), and that usually establishes the environment in which the semantics is defined, including the program source code file, input and output files, whose names are collected from the command line that activates the execution of the semantic definition.

The concrete grammar also contains the definition of the associated abstract syntax of the language being defined. The concrete grammar is used to generate translators for rendering programs in the defined language into an internal notation for abstract syntax tree (AST). The abstract syntax is used to convey semantics to language constructs.

The best module organization is user-defined. Indeed, a formal definition may be composed of several modules, each of which focusing on a particular syntactic or semantic aspect of the language being defined. Sometimes it is convenient to encapsulate the definition of the concrete syntax and the corresponding semantics of each important construct into a distinct module in order to permit a better separation of concerns. The language $\mathcal{M}$ is flexible enough to permit the user to choose the most appropriate module organization.

1.1 Basic structures

The grammar of $\mathcal{M}$ is presented in an extended BNF notation [59] with the convention that nonterminals are sequences of letters, possibly with embedded underline _ symbols.

The production rules of the grammar are encoded according to the following notation:

notation	meaning
`X*`	list with zero or more occurrences of X
`(Y)*`	list with zero or more occurrences of Y
`X+`	list with one or more occurrences of X
`(Y)+`	list with one or more occurrences of Y
`(Y)`	same as Y
`"t"`	a terminal symbol, `t` can be anything but "
`"\""`	the terminal symbol "
`A ::= B	···	C`	a cfg production rule
`A === "x" .. "y"`	define A as a character in the interval `[x,y]`
`A =/= "x"`	define A as any character but `x`
ϵ	empty string

where `X` represents a nonterminal, `Y` is a non-empty string of terminal and/or nonterminal symbols; `x` and `y` stand for one-character symbols. The letter `A` represents a nonterminal symbol, and `B` and `C` represent strings of terminals, nonterminals or one of the lists defined above.

Identifiers

$\mathcal{M}$ identifiers have different structures depending on their uses in the program. For instance, identifiers denoting domain names always start with a capital letter, e.g., `Store`, `Command` and `Environment`. Those denoting domains of lists are suffixed by `*` or `+`, and the identifiers of variables or functions are generally initiated with a small letter, and may have digits, `*` or `+` at their end.

The basic identifiers can be classified as decorated or undecorated. The undecorated identifiers are proper nouns or common nouns. The decorated ones are indexed proper nouns,

indexed common nouns, proper nouns for lists or common nouns for lists.

The syntax of $\mathcal{M}$ identifiers are as follows:

```
identifier      ::= proper_id   | proper_id_list
                  | proper_idx  | proper_idx_list
                  | common_id   | common_id_list
                  | common_idx  | common_idx_list
proper_id       ::= uppercase anycase*
proper_idx      ::= proper_id digit+
proper_id_list  ::= proper_id rep_op+
proper_idx_list ::= proper_id digit+ rep_op+
common_id       ::= lowercase anycase*
common_idx      ::= common_id digit+
common_id_list  ::= common_id rep_op+
common_idx_list ::= common_id digit+ rep_op+
anycase         ::= lowercase | uppercase | "_"
uppercase       === "A" .. "Z"
lowercase       === "a" .. "z"
rep_op          ::= "*" | "+"
digit           === "0" .. "9"
```

The case of the letters that occur in an identifier is meaningful, e.g., the identifier `readq` is distinct from `readQ`.

Note that the symbols `+` and `*` may be part of identifiers. In this case, they must occur at the end of character's sequences that compose identifiers. In these circumstances, they are not delimiters. In other situations, the symbols `+` and `*` are just binary operators. To be recognized as such, they must be preceded by a delimiter or white spaces. Note the differences:

```
s* t  -- identifier s* (list) followed by identifier t
s * t -- product of (integer) s by (integer) t
s *t  -- same s * t
(s)*t -- same as s * t
```

The following identifiers are reserved, that is, they have special meaning, and thus cannot be used for other purposes:

and	becomes	end	File	false
functions	imports	interface	is	lexis
main	module	N	Nonterminal	privates
publics	Q	return	Start	syntax
T	Token	true	where	Y

The following identifiers denote standard functions with predefined meaning:

append	ascii	close	compile	cond	eof
flatten	getarg	getchar	head	open	putchar
size	tail	toN	toQ	toT	ungetchar
value					

Layout

The newline (`"\n"`) and the carriage return (`"\r"`) characteres are delimiters in $\mathcal{M}$:

```
newline ::= "\n" | "\r"
```

White spaces, including form feed (`"\f"`) character are treated as delimiters of $\mathcal{M}$'s tokens, and are not part of any token.

Comments

Comments in $\mathcal{M}$ start with the symbol `--`, and end with the end of the line, i.e., when the newline mark is encountered.

```
comments ::= "--" anychar* newline
anychar  =/= newline
newline  ::= defined in §1.1, page 5
```

1.2 Built-in domains

$\mathcal{M}$ domains are complete partial orders with a minimal element $\perp$ (*bottom*) [44, 47, 51, 57], and also with an undefined polymorphic value, represented by `?`. Domains and types are treated as synonymous (*§1.4, page 12*).

Domains or types have properties that guarantee that solutions of all domain equations always exist up to isomorphism. The special value $\perp$ (*bottom*), which is not directly representable in $\mathcal{M}$, serves to model the semantics of nontermination.

The built-in domains in $\mathcal{M}$ are:

```
builtin_dom ::= "N" | "Q" | "T" | "File"  | "?"
              | "Nonterminal"   | "Start" | "Token"
```

`N` is the domain of 32-bit integer numbers, `Q` is the domain of quotations or strings, `T` is the domain of truth-values, `File` is the domain of files (*§1.2, page 6*), `Nonterminal` is the domain of all nonterminal symbols used or defined in the syntax part of a module, `Start` is the domain of the grammar starting symbol, and the symbol `?` represents the domain of undefined values.

The built-in domains are standard and flat [47], and thus directly available in every semantic definition, and each of which has a number of constants and predefined operations, which produce $\perp$ whenever any of their operands is $\perp$. And these predefined operations produce `?`, when any of their operands is `?` and none is $\perp$. Otherwise, they have the usual expected behavior. Saying that an operation produces $\perp$ is tantamount to say that it does not terminate, and not that it returns this special *value*.

The built-in domain `?` contains the special *undefined* value ?, which is polymorphic, i.e., it is member of all domains, be they built-in or user-defined. The context should provide enough information for resolving the overloading of the symbol ?, whenever necessary. The undefined value is used to indicate the value of semantically *non-sensical* expressions. Any operator or predefined operation can be applied to the undefined value, and the result is always undefined. The undefined value may be passed as argument to a ordinary function. In this case, the function result depends on the evaluation of its body, which may evaluate or not to the undefined value.

The domain `File` represents files containing a list of characters stored in an external media. A character at a time can be read from these files or written into them by means of the built-in functions `getchar` and `putchar`, respectively.

The domain `File` defines the structure of file descriptors, which hold all information that is necessary to read from or write into a given file, namely: the file current read/write head position, end-of-file condition indicator, the contents of the file's buffer and the file's physical address.

The standard functions associated with files are:

- `open : Q -> File`
  The function application `open(filename)` returns the descriptor of the file whose name has been passed as argument. The file name must be a quotation structured according to the local execution system conventions. If the specified file cannot be opened, the undefined value `?` is returned. When a file is opened, an empty buffer is associated with it. No files can be read or written before

it had been opened.

- `close : File -> File`
  The function application `close(file)` removes the vinculum between the file descriptor that is associated to given argument `file` and the file itself in the external media. Before closing, data in the file's input buffer are recorded in the associated file. No files can be read or written after it had been closed, unless it is opened again.
- `getchar : File -> (File,N)`
  The function application `getchar(file)` reads the next input character from the file's buffer, if the buffer is not empty, otherwise it reads the next portion of the given file into the file's buffer, and returns the integer value that represents the ASCII code associated with the first character now in the buffer. If no character is encountered on the file current reading position, the undefined value (?) is returned. An updated file descriptor, which records the reading head position and the current buffer contents, is also returned. If the end-of-file mark is unexpectedly encountered while attempting to read a character, the undefined value `?` is returned instead, and the end-of-file condition is set for this file, and further attempts to read it will always return a pair containing the current file descriptor and the undefined value.
- `ungetchar : (File,N) -> File`
  The function application `ungetchar(file,n)` puts back to the buffer of the specified file the character represented by the integer `n`, so that the next `getchar(file)` on the same file will return this character.

- `eof : File -> T`
  If the end-of-file mark of the given file has been detected in a previous operation or if it will be read in next operation, the function `eof(file)` returns `true`, otherwise, it returns `false`.
- `putchar : (File,N) -> File`
  The function application `putchar(file,n)` prints the contents of the given integer value `n` as a character on the file associated with the specified file descriptor. An updated file descriptor is returned because this operation changes the current position of the writing head.

1.3 Declarations

Declarations serve the purpose of associating variables with domains, and, if necessary, providing names for new user-defined domains. Any $\mathcal{M}$ interface module may have a declaration part, which introduces names used or defined in the corresponding module definition.

The syntax of the declaration part of a module is defined by the following syntactic rules:

```
var_dom_dcls ::= var_dom_dcls ";" var_dom_dcl
               | var_dom_dcl
var_dom_dcl  ::= new_domain_id "=" domain_exp
               | new_vars ":"  domain_exp
               | new_vars ":" domain_id "=" domain_exp
               | empty
new_vars     ::= new_var_id ("," new_var_id)*
new_var_id   ::= identifer | "main"
domain_id    ::= proper_id
identifier   ::= defined in §1.1, page 3
```

```
proper_id     ::= defined in §1.1, page 3
domain_exp    ::= defined in §1.4, page 12
```

Note that domain names are always proper nouns, and both common and proper names with or without decoration can be used in a declaration of variables. However, it is recommended that proper names be only used to designate certain special functions, and that nonfunctional variables and ordinary functions are preferentially named with common names.

Names of variables can be decorated with numbers, in which case they are said be indexed, or marked with the symbols + or *, which make them list designators. The domains of indexed variables and of list variables are inferred automatically in the way described in the sequel.

The following variable declarations illustrate some cases:

```
1    a, b : A;
2    A = N -> N;
3    d, g : (N,A*);
4    B = Q -> Q;
5    x, y, z : (A);
6    C : Cmd -> Env -> Cc -> Store -> Ans;
```

In the above program code, note that:

- Variables `a` and `b`, in line 1, are in domain A, which is the domain of functions from `N` to `N`.
- Variables `d` and `g`, in line 3, are two-component tuples of type `(N,A*)`.
- The new domain `B`, defined in line 4, is the domain of functions from type `Q` to `Q`.
- Variables `x, y` and `z`, declared in line 5, are functions of type `A`, i.e., functions that map members of `N` to `N`. The

domain expression `(A)` is not a tuple domain, for tuples must have at least two components.

- And `C` in line 6 is a curried function.

All variables must have their types known in the scope they are used. Variables designated by proper nouns must always be declared, however, not all variables designated by common nouns need be explicitly declared. Their types are deduced from the structure of their identifiers, according to the following convention:

- Any nondeclared undecorated variable is assumed to be in the domain whose name is that of the variable with the first letter capitalized, e.g., `exp` is implicitly in `Exp`.
- Any nondeclared variable decorated with decimal digits is implicitly in the same domain as its corresponding undecorated version. For example, variables `s1` and `s2` are by default in the domain of `s`.
- If `a` is in domain `A`, then the occurrences of nondeclared `a+` and `a*` are interpreted as members of domains `A+` and `A*`, respectively.
- Variables that are formal parameters of a function are implicitly declared according to the type of the corresponding parameter in the declaration of the function heading.

For instance,

```
1 privates
2    A = N -> N;
3    B = Q -> Q;
4    a : N;          -- from this point on a has type N
5    x : T;
6    f : N -> Q;
```

```
7     g : Q -> T;
8     h : B* -> Q
9  functions
10    f(x)   = ... a ... x ... -- a and x have type N
11    g a    = ... a ... x ... -- a has type Q, and x, T
12    h(a1*) = ...  a1*  ...   -- a1* has type B*
```

These conventions are intended to contribute to description compactness and convenience of writing. Since they are very simple and uniform, they could help producing readable definitions.

1.4 Domains

There exists a variety of domain operators for creating more complex domains to model semantic properties of programming languages. Domain expressions that can be associated with new domains according to the following syntax:

```
domain_exp    ::= domain_exp "|" domain_a
                | domain_a
domain_a      ::= domain_b "->" domain_a
                | domain_b
domain_b      ::= simple_domain | tuple_domain
                | node_domain   | enum_domain
                | list_domain   | const_domain
simple_domain ::= domain_id | builtin_dom
tuple_domain  ::= "(" field_domains ")"
field_domains ::= field_domain ("," field_domain)*
field_domain  ::= domain_exp
node_domain   ::= "[" domain_c+ "]
domain_c      ::= simple_domain | const_domain
enum_domain   ::= "{" enum_elems "}" | "{" "}"
enum_elems    ::= constant ("," constant)*
list_domain   ::= simple_domain rep_op+
```

```
                  | tuple_domain rep_op+
const_domain  ::= quotation
builtin_dom   ::= defined in §1.2, page 6
domain_id     ::= defined in §1.3, page 9
constant      ::= defined in §1.5.4, page 24
quotation     ::= defined in §1.5.4, page 24
```

A domain expression may be a domain name, a quotation, in which case it is assumed to denote a singleton domain whose only proper element is that constant, a list of enumerated elements, or a combination of simpler domain expressions by means of domain operators. Domain combinations denote union of domains, domain of tuples, domain of nodes, domain of lists or domain of functions.

A domain expression may contain names of domains that are only defined later in the module scope. This should not cause any problem or error unless the yet undefined domains are effectively used in an expression in the module before the undefined elements are properly declared. In order to cope with this situation, after processing each block of domain declarations, all domain definitions in the current module should be re-checked to remove any dependency on undefined domains that have been defined in the block after their uses. At the end of the module, all theses dependencies must have been solved, so as to not incur a situation of missing declarations.

1.4.1 Domain of enumerations

A domain can be created by enumerating its elements, such as `{"integer", "float", "undefined"}`. The enumerated elements must be constants of the same type.

1.4.2 Domain constants

All quotations are in domain `Q`. However, for technical reasons, any quotation occurring in places where a domain is expected is considered to represent the domain whose only proper non-bottom element is the quotation itself. The name of this domain is the quotation itself. For instance in the declaration

```
Mode = "int"
```

the string `"int"` denotes a domain whose name is `"int"`, and that contains only the quotation `"int"`, the bottom element $\perp$, and the undefined value ?.

1.4.3 Domain of tuples

A domain expression of the form $(d_1, \ldots, d_n)$ represents the domain of n-tuples whose i-th component is in the domain denoted by d_i, for $1 \leq i \leq n$ and $n \geq 2$. This notation represents the cartesian product of domains. When $n = 1$, the domain expression is not a domain of tuple, it is just a domain expression enclosed in parentheses.

1.4.4 Domain of lists

Domain expressions of the form $d*$ denote domains of finite and possibly empty lists whose components are in d. Domain expression of the form $d+$ denotes domain of lists with at least one element.

Assume that $e_1, \ldots, e_n$, for $n \geq 1$, are expressions of the same type d. An instance of a non-empty list is created by

$(e_1, e_2, \cdots, e_n)$, for $n \geq 1$. The domain of the list just created is the domain d+.

An empty list is represented by keyword `nil`, which is a polymorphic value, i.e., the exact type of `nil` depends on the context it occurs.

1.4.5 Domain of tree nodes

A domain expression of the form `[`$D_1 \ldots D_n$`]` represents the domain of abstract syntax tree nodes, each of which consists of a label and a tuple of emanating branches.

The polyadic operator `[ ··· ]` requires that each D_i, for $1 \leq i \leq n$ and $n \geq 1$, be either a quotation or a domain identifier possibly followed by a sequence of `*` and/or `+` symbols. Quotations that occur in domain expressions denote constant domains.

1.4.6 Domain of functions

The domain expression d_1 `->` d_2 denotes the domain of **continuous functions** [44, 47] from domain d_1 to domain d_2. The operator `->` is right-associated and has precedence over the operator `|`.

For instance,

d_1 `->` d_2 `|` d_3 `->` d_4 `->` d_5 `|` d_6 `->` d_7 `->` d_8

is equivalent to

d_1 `->` d_2 `|` (d_3 `->` (d_4 `->` d_5)) `|` (d_6 `->` (d_7 `->` d_8))

1.4.7 Union of domains

The domain expression d_1 `|` ... `|` d_n represents the **union** of the domains denoted by domain expressions $d_1, d_2, \ldots, d_n$. Each d_i, for $1 \leq i \leq n$, is called a summand of the union. Alternatively, a definition of a union can be unfolded into a sequence of simpler definitions with the same left hand side.

Distinct definitions of the same domain introduce a union definition. Both styles lead to equivalent domain definitions. For instance,

```
A = X;
A = Y;
A = Z
```

is equivalent to

```
A = X | Y | Z
```

The domain operator `|` is left-associated and has the lowest priority. It generates a separated union like the *separated sum* of D. Scott [44]. The user has to cope with *projections* and *injections* between a union and its summands throughout denotational descriptions. So the domain of all values are carried at run-time and can be inspected. The domain of any operand of a union to which a given value belongs can be ascertain by means of the enquiry operation `is` (*§1.5.5, page 26*) or via pattern-matching applications (*§1.5.2, page 22*).

In pattern-abstraction applications, domain projections are performed implicitly. However, they can be done explicitly, if desired. For instance, if `A = A`$_1$ `|` ⋯ `|` `A`$_n$, `a:A`, and if `a is A`$_1$, the projection of the value `a` to the summand domain `A`$_1$ is achieved via the expression `A`$_1$`(a)`. On the other hand, if `a` is not in the domain `A`$_1$, the expression `A`$_1$`(a)` returns `?`.

The reverse operation, the injection, is automatically performed whenever a value in the domain of any summand is used where the value of the summation domain is expected. Alternatively, the explicit injection of a value of a summand `x:X` into a union domain `C = ... | X | ...` can be expressed as `C(x)`.

As two union domains may intersect, the elements in the intersection may be explicitly converted from one domain to another. For instance, in the code fragment below,

```
1 privates
2    A = N | T;
3    B = Q | N
4 functions
5    a  = 1;
6    b1 = a is N => N(a), ?;
7    b2 = B(a)
```

lines 6 and 7 produce the same effect, including the case in which `a` is not in `N`.

1.4.8 Domain equivalence and compatibility

$\mathcal{M}$ type discipline is based on **structural compatibility** [3], and is defined as follows:

A domain A is **structurally compatible** to a domain B if and only if A and B are **equivalent** or B is a union of domains, and A is one of its summands.

Two domains A and B are **equivalent** if and only if at least one of the following conditions applies:

1. A and B are identical domain names.
2. A and B are the same constant domain.
3. A and B are the domain ? of undefined values.

4. B is a domain expression of the form `(`A`)` or A is of the form `(`B`)`.
5. B is a domain expression and A is a domain name, whose definition is $A = d$ or $A = A_1 = A_2 = ... = A_n = d$, where A_i, for $1 \leq i \leq n$, are domain names, and domain expression d is equivalent to domain expression B.
6. A is a domain expression and B is a domain name whose definition is $B = d$ or $B = B_1 = B_2 = ... = B_n = d$, where B_i, for $1 \leq i \leq n$, are domain names, and domain expression A is equivalent to domain expression d.
7. A and B are domains of lists, A is of the form a`*`, B is of the form b`*`, and a is equivalent to b.
8. A and B are domains of lists, A is of the form a`+`, B is of the form b`+`, and a is equivalent to b.
9. A and B are domains of tuples, domain A is of the form `(`$a_1, \cdots, a_n$`)`, B is of the form `(`$b_1, \cdots, b_n$`)`, for $n \geq 2$, and, for $1 \leq i \leq n$, a_i is equivalent to b_i.
10. A and B are domains of nodes whose labels are identical.
11. A and B are domains of continuous functions, A is of the form a_1 `->` a_2, B is of the form b_1 `->` b_2, a_1 is equivalent to b_1, and a_2 is equivalent to b_2.
12. A and B are unions of domains, A is of the form a_1 `|` a_2 `|` $\cdots$ `|` a_n, B is of the form b_1 `|` b_2 `|` $\cdots$ `|` b_n, and, there is a permutation b'_i of the elements b_i, for $1 \leq i \leq n$, such that a_i is equivalent to b'_i.
13. A and B are distinct domain names, each being the recursive reference to a reflexive domain definition occurring in exactly the same corresponding position in their domain structures.

1.5 Expressions

Expressions are classified as functional, conditional, or basic:

```
expression      ::= functional_exp | conditional_exp
                  | basic_exp
functional_exp  ::= defined in §1.5.1, page 19
conditional_exp ::= defined in §1.5.3, page 23
basic_exp       ::= defined in §1.5.4, page 24
```

Functional expressions denote new anonymous functions that can be applied to arguments or used as parameters to other functions.

Conditional expressions allow strict selection of one between two expressions of equivalent types (*§1.5.3, page 23*).

Basic expressions essentially contain (monadic and dyadic) operators, variables, constants, lists, tuples, tokens, and nodes (*§1.5.4, page 24*).

1.5.1 Functional expressions

Anonymous non-recursive functions are specified by the notation $\backslash x\,.\,e$, in which the symbol \ is the typewriter's rendition of the traditional λ of λ-calculus, x is pattern expression, e.g., an identifier or a tuple of identifiers, and e is an arbitrary expression. The scope of x is the expression e, and $\backslash x\,.\,e$ has type `A->B`, if x`:A` and e`:B`.

Functional expression is an abstraction that is derived from the notation of the λ-expression of λ-calculus. The basic idea is to allow patterns (*§1.5.2, page 22*) to occur in binding contexts, such as p in the functional-expression `\p.e`, where `p` is a pattern-expression. The syntax of functional expression is as follows:

```
functional_exp ::=  "\" pattern_exp+ "." expression
pattern_exp    ::= defined in §1.5.2, page 22
expression     ::= defined in §1.5, page 19
```

The functional-expression binding mechanism provides a powerful device for extracting components of compound values, such as nodes, lists or tuples. For example, if e represents a tuple of two elements, the application of the functional expression `(\(x1,x2).`e_1`)` to e will cause `x1` and `x2` to be bound to the first and second components of tuple e, respectively, during the evaluation of the expression e_1 in the body of the abstraction. On the other hand, if e does not have the form specified by the corresponding formal parameter, the value of this function application is `?`.

Another illustration is the application

`(\[x1 x2].`e_1`)(`e_2`)`

that binds `x1` and `x2` to the immediate subtree components of node e_2 throughout the evaluation of e_1. If e_2 is not a node of the form `[x1 x2]`, this application returns `?`.

In summary, the binding mechanism associated withe the application `\`p`.`e to argument a falls in one of two cases:

1. If pattern p is an identifier, possibly decorated by indices, `*`'s and/or `+`'s, `(\`p`.`e`)(`a`)` is just a regular function application, in which a is evaluated immediately and bound to p to provide the scope to evaluate e.

2. On the other hand, when pattern p is not an identifier, then the argument a must be evaluated immediately in order to perform the required pattern-matching, which is conducted as follows:

(a) If a has the form or structure defined by pattern p, the identifiers in p are bound to corresponding values in the structure of a, as described in the sequel, and then the body e is evaluated.
(b) If a does not have the form defined by pattern p, the result of the function application `(\`p`.`e`)(`a`)` is `?`.

More specifically, if p is a pattern-expression, $p_1, \cdots, p_n$ are pattern elements, and a has the structure defined by p, then the bindings produced by the application `(\`p`.`e`)(`a`)` or `(\(`p`).`e`)(`a`)` are defined as follows:

1. If p is a literal constant or the empty list symbol (`nil`), no bindings result, but a and p must be equal.
2. If p is of the form $(p_1, \cdots, p_n)$, for $n \geq 2$, then the identifiers p_i, for $1 \leq i \leq n$, are properly bound to the corresponding components of a.
3. If p is of the form $p_1 : p_2$, then the identifier p_1 is properly bound to the first element of the list a, and the identifier p_2, to the tail of a.
4. If p is of the form `[`$p_1 \cdots p_n$`]`, then p_i, for $1 \leq i \leq n$, that are identifiers are bound to the corresponding parts of node a, and those that are constants must match the corresponding element of a.

A *recursive* functional expression is defined via `Y(\`$p.e$`)`, with the restriction that the value of e `is` p must be always "manifestly" `true` [30] and `Y` is the *Paradoxical Combinator* [9].

1.5.2 Pattern expressions

Pattern-matchings occur during parameter passing operations, as part of the process of applying functions to arguments. The patterm-matching operation permits checking whether the function's arguments have the particular *form* or structure that is described by the corresponding formal parameter.

It permits to investigate the structure of a value rather than the value itself or its domain. In essence, if the value denoted by the argument can be structured according to the pattern dictated by the formal parameter, then the matched elements are bound accordingly.

Function's formal parameters can be pattern expressions, whose syntax is defined as follows:

```
pattern_exp      ::= simples_pattern | node_pattern
                   | compound_pattern
simples_pattern  ::= variable_id | constant
node_pattern     ::= "[" simples_pattern+ "]"
compound_pattern ::= "(" pattern_elems ")"
pattern_elems    ::= pattern_elem ("," pattern_elem)*
pattern_elem     ::= simples_pattern
                   | list_pattern
                   | node_pattern
list_pattern     ::= simple_pattern ":" simple_pattern
constant         ::= defined in §1.5.4, page 24
variable-id      ::= defined in §1.5.4, page 24
```

These grammar productions show that pattern-expression can be an identifier, which matches expressions of the identifier's type, a literal constant, which matches itself, or a combination of simpler pattern-expressions and pattern construction operators. If $p, p_1, \cdots, p_n$ are identifiers or constants, then new patterns can be built up as follows:

1. $(p_1, \cdots, p_n)$ - to match tuples with $n \geq 2$ components. If $n = 1$, this pattern matches a single expression.
2. p`*` - to match lists with zero or more components.
3. p`+` - to match lists with at least one component.
4. `nil` - to match empty lists.
5. $(p_1 : p_2$`*`$)$ - to match lists with at least one component.
6. `[`$p_1 \cdots p_n$`]` - to match nodes whose label is equal to this pattern's label.
7. `(`p`)` - the same as p, i.e., anything that matches pattern p matches `(`p`)`.

1.5.3 Conditional expressions

A conditional expression has the structure t`=>`e_1`,`e_2, where t is an expression that evaluates to `true`, `false`, `?` or does not terminate ($\perp$), and e_1 and e_2 are arbitrary expressions of equivalent types. The expression t`=>`e_1`,`e_2 is equivalent to e_1, if t denotes `true`; equivalent to e_2, if t stands for `false`; equivalent to `?`, if t evaluates to `?`, and equivalent to $\perp$, if t is $\perp$. The syntax of conditional expressions is as follows:

```
conditional_exp ::= basic_exp
                        "=>" expression "," expression
basic_exp       ::= defined in §1.5.4, page 24
expression      ::= defined in §1.5, page 19
```

Conditional expressions can be nested, so the expression

```
x = a => b => c, d, e
```

produces e, when a is `false`, c, if a and b are simultaneously `true`, and d, if a is `true` and b is `false`. That is, the expression

```
x = a => b => c, d, e
```

is the same as

```
x = a => (b => c, d), e.
```

Conditional expressions can also be specified by means of the polymorphic built-in function `cond`, whose generic type is defined as `cond:(D,D)->T->D`, for any valid domain `D`. The expression `cond(d1,d2)b` is equivalent to `b=>d1,d2`.

1.5.4 Basic expressions

The most simple expressions are variables, integer constants, boolean constants, quotations, the undefined value `?`, and the empty-list constant `nil`.

Variables are used to denote members of domains. Those which denote lists are usually, but not necessarily, suffixed by sequences of `*` or `+` symbols. For instance, `x*` denotes a finite list of arbitrary size, and `x+` represents a non-empty finite list. The syntax of basic expressions is defined as:

```
basic_exp       ::= exp_a ":" basic_exp | exp_a
exp_a           ::= exp_b rel_op exp_b  | exp_b
rel_op          ::= "is" | "==" | "!=" | "<"
                  | "<=" | ">"  | ">="
exp_b           ::= exp_b add_op exp_c | exp_c
add_op          ::= "||" | "+" | "-"
exp_c           ::= exp_c mul_op exp_d | exp_d
mul_op          ::= "&&" | "*" | "/" | "%"
exp_d           ::= mon_op exp_e | exp_e
mon_op          ::= "!"  | "-"
exp_e           ::= exp_e exp_f | exp_f
exp-f           ::= constant    | variable_id
                  | function_id | nonterm_id
                  | domain_id   | lex_nonterm_id
                  | builtin_dom | list_exp
                  | node_exp    | mapping_exp
```

```
                  | tuple_exp   | "(" expression ")"
variable_id     ::= common_id      | common_idx
                  | common_id_list | common_idx_list
                  | proper_id      | proper_idx
                  | proper_id_list | proper_idx_list
function_id     ::= common_id   | common_idx
                  | proper_id   | proper_idx
                  | builtin_fun
builtin_fun     ::= "append"    | "ascii"
                  | "close"     | "compile"
                  | "cond"      | "eof"
                  | "flatten"   | "getarg"
                  | "getchar"   | "head"
                  | "main"      | "open"
                  | "putchar"   | "size"
                  | "tail"      | "toN"
                  | "toQ"       | "ungetchar"
                  | "value"     |  "Y"
constant        ::= truth_const | quotation | empty_list
                  | integer_const | undefined_const
integer_const   ::= digit+
quotation       ::= """ quotation_ch* """
                  | """ quotation_ch* newline
quotation_ch    ::= ordinary_char | special_char
ordinary-char   =/= """
special-char    ::= "\b" | "\\" | "\ddd" | "\r"
                  | "\t" | "\n" | "\""   | "\f"
truth_const     ::= "true" | "false"
undefined_const ::= "?"
empty_list      ::= "nil"
digit           ::= defined in §1.1, page 3
builtin_dom     ::= defined in §1.2, page 6
domain_id       ::= defined in §1.3, page 9
expression      ::= defined in §1.5, page 19
tuple_exp       ::= defined in §1.5.9, page 30
list_exp        ::= defined in §1.5.10, page 31
```

```
node_exp          ::= defined in §1.5.11, page 33
mapping_exp       ::= defined in §1.5.12, page 34
nonterm_id        ::= defined in §1.8.2, page 47
lex-nonterm_id    ::= defined in §1.8.1, page 42
common_id         ::= defined in §1.1, page 3
proper_id         ::= defined in §1.1, page 3
common_idx        ::= defined in §1.1, page 3
proper_idx        ::= defined in §1.1, page 3
newline           ::= defined in §1.1, page 5
```

1.5.5 Logical expressions

The meaningful constants in the domain `T` are `true` and `false`. The most simple logical expressions are these constants or variables of type `T`, and more complex terms may be put together with the following operators: `!` (negation), `||` (or), `&&` (and), `==` (equal), `!=` (not equal), `is` (inspect), `<` (less than), `<=` (less than or equal), `>` (greater), and `>=` (greater or equal).

Expressions of the form e_1`==`e_2 are used to test whether two expressions, e_1 and e_2, denote the same value. The expression e_1`==`e_2 evaluates to `true` if e_1 and e_2 have the same non-functional value. Otherwise, if e_1 and e_2 are functional values or denote distinct values, it evaluates to `false`. Of course, the result is $\perp$ if either e_1 or e_2, or both are $\perp$, and, *mutatis mutandis*, similar observation holds for the undefined value `?`. The negation of e_1`==`e_2 is written as e_1`!=`e_2.

The binary relational operators `<`, `<=`, `>`, and `>=` are applicable to integer values and quotations. And logical operations **not**, **or**, and **and** are performed by the operators `!`, `||`, and `&&`, respectively.

The binary operator `is` allows inspecting the domain of any value of a union domain. An element of a union domain may

be in more than one member of the union, and the `is` operator allows checking whether it is in a given member. For instance, if `x:A`, and `A = A1 | ··· | Ai | ··· | An`, the expression `x is Ai` returns `true` if the value currently associated with `x` is in the domain `Ai`, and returns `false` otherwise.

The built-in function `toQ:T->Q` converts `true` into quotation `"true"`, and `false` into `"false"`. All other quotations are converted to `?`.

1.5.6 Integer expressions

Integer expressions are built upon variables and constants of type `N`, function calls, and possibily using some of the following operators:

`n1 + n2`	integer addition
`n1 - n2`	integer subtraction
`n1 * n2`	integer multiplication
`n1 / n2`	integer division
`n1 % n2`	integer division remainder
`- n`	sign change of integer

where `n`, `n1` and `n2` are elements of `N` or expressions whose values are in `N`.

If the result of any of the above operations falls outside the interval of 32-bit integer numbers, $[-2^{32}, 2^{31} - 1]$, the value undefined `?` is returned.

Integer constants are nonnegative integer numbers in the range of 32-bit integers, i.e., values in the interval $[0, 2^{31} - 1]$.

The built-in function `toQ:N->Q` converts integer values into quotations containing the decimal digits that compose the number, whereas the function `ascii:N->Q` returns a quota-

tion containing the character whose internal ASCII code is given. In case of invalid ASCII code, the value `?` is produced.

1.5.7 Quotations

Constants in domain `Q` are sequences of any ASCII characters enclosed in quotes (") or delimited by a quote and newline mark (`\n` or `\r`), i.e., all quotation start with a quote and finish with another quote or the end of the line mark. Special characters can be encoded in quotations as shown below:

Name	Code	Name	Code	Name	Code
backspace	`\b`	carriage return	`\r`	newlines	`\n`
backslash	`\\`	form feed	`\f`	null character	`\0`
bit pattern	*ddd*	horizontal tab	`\t`	quote	`\"`

where `\ddd` denotes a character whose internal code is the decimal `ddd`. An example of special character encoding in a quotation is:

```
"She said: \"This is a quotation\""
```

which denotes the quoting of the text

```
She said: "This is a quotation"
```

Quotations may be concatenated by operator `+`. The operator `:` produces a quotation that is the concatenation of a character, its first operand, with a quotation, its second operand.

If `k` is an integer value, c is a character, `q`, `q1`, and `q2` are in domain `Q`, the following operations are defined:

`q1 < q2`	returns true if `q1 < q2` in lexicographic order, else false
`q1 <= q2`	returns true if `q1 <= q2` in lexicographic order, else false
`q1 > q2`	returns true if `q1 > q2` in lexicographic order, else false
`q1 >= q2`	returns true if `q1 >= q2` in lexicographic order, else false
`q1 == q2`	returns true if `q1` is equal to `q2`, otherwise returns false
`q1 != q2`	returns true if `q1` is not equal to `q2`, otherwise false
`q1 + q2`	returns the concatenation of `q1` and `q2`
`q(k)`	returns a quotation containing the `k`-th character of `q`
`c : q`	returns a quotation where `c` is the head and `q` is the tail

The built-in function `toN` converts a quotation containing only decimal digits into an integer number, `toT` converts `"true"` into `true`, `"false"` to `false`, and any other quotation to ?. The function `size` returns the number of characters that are within a quotation, and `flatten` transforms a list of quotations into a single quotation, containing all the characteres in the given quotations. In summary, the following built-in functions are applicable to quotations:

- `flatten : Q* -> Q`
  concatenates quotations from a list of quotations.
- `toN : Q -> N`
  converts a quotation containing only decimal digits into an integer value, if the quotation contains any other character, it returns ?.
- `toT : Q -> T`
  converts `"true"` into `true`, `"false"` into `false`, otherwise, returns ?.
- `size : Q -> N`
  gives the number of characters of a quotation.

1.5.8 Fixpoint operator

`Y` is the Curry's *Paradoxical Combinator* [9], which computes the fixpoint of recursive equations and is defined as:

`Y = \f.(\x.f(x x))(\x.f(x x))`

which is the $\mathcal{M}$'s rendition for the Church's λ-calculus expression:

$\texttt{Y} = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$

For instance, in the program text below, f is a fixpoint of the recursive equation `f = g(f)` computed by the combinator `Y` in line 5:

```
1  F: A -> B;
2  g: F -> F;
3  ...
4  g  = \f.  ... f ... ;
5  f  = Y(g);
```

1.5.9 Tuple expressions

Tuples are constructed by explicitly enumerating its components by means of the notation $(e_1, \cdots, e_n)$, where expressions e_i, for $1 \leq i \leq n$ and $n \geq 2$, define the values of the tuple's components. The syntax of tuple expressions is as follows:

```
tuple_exp    ::= "(" tuple_elems ")"
tuple_elems  ::= expression ("," expression)+
expression   ::= defined in §1.5, page 19
```

Selection of fields of a tuple is achieved by means of pattern-matching or by element indexing. To illustrate the selection of tuple's fields, consider the program fragment:

```
1 privates
2    X = (A,B,C);
```

```
3     f : X -> D;
4     g : D -> D
5  functions
6     x        = (... , ... , ...);
7     h(...)   = ... x(2) ... ;
8     f(a,b,c) = ... a ... b ... c.... ;
9     g(...)   = ... f(x) ...  ;
```

The expression `x(2)` in line 7 exemplifies the selection of a tuple's second field by means of the indexing operation. Tuple-indexing operations have the general form e`(`k`)`, where k is an integer expression whose value is in the interval $1 \leq k \leq$ *number of components of* e. This operation returns the value of the k-th component of tuple `e`. In case k is outside limits, the undefined value `?` is returned.

The pattern-matching mechanism is associated with functional-abstraction applications (*§1.5.2, page 22*). When executing the body of function `f` in line 8, due to the function call in line 9, parameters `a`, `b`, and `c` are bound to the first, second and third fields of `x`, respectively.

1.5.10 List expressions

A list is a sequence of one or more expressions of equivalent types, separated by commas and enclosed in the pair of symbols `(` and `)`. Lists differ from tuples by requiring that all expressions in a list must be of equivalent types.

```
list_exp     ::= "(" list_elems ")"
list_elems   ::=  expression ("," expression)*
expression   ::= defined in §1.5, page 19
```

Note that the use of pair of parentheses in $\mathcal{M}$ is threefold: they are used to group sub-expressions, to construct

lists, and also to construct tuples. A single expression inside parentheses is not a tuple, for tuples must have at least two components, but it may be, according to context, a list of one element or just an expression enclosed in parentheses.

The main operations on lists, where `D` is any domain, `e:D`, and k is an integer expression whose value must be in the interval $1 \leq k \leq$ `size e*`, are as follows:

`e : e*`	produces new list whose head is `e` and tail `e*`
`e1* + e2*`	constructs a list by concatenating e_1* and e_2*
`e*(k)`	returns the `k`-th element of the list `e*`

The list-indexing operation `e*(k)`, when k is outside limits, returns the undefined value `?` to indicate this error condition.

The other operations on lists are as follows:

- `append : (D*,D) -> D+`
  the operation `append(`$e*$`,`e`)` returns a new list by appending e to the end of a copy of the list $e*$.
- `flatten : D** -> D*`
  the `flatten(e**)` operation generates a list by concatenating the elements of the list `e**`, preserving the order of the elements in the given list.
- `head : D+ -> D`
  the operator `head` returns the first element of a list. Another way to retrieve the head of a list is by means of application of pattern-matching (*§1.5.1, page 19*). The application of function `head` to an empty list produces the undefined value `?`.
- `size : D* -> N`
  the operation `size(e*)` returns the number of components in the list `e*`.

- `tail : D+ -> D*`
  the operator `tail` returns a list containing all the elements of the given list but the first one. Applying function `tail` to an empty list produces the undefined value `?`. Another way to retrieve the tail of a list is by means of application of pattern-matching (*§1.5.1, page 19*).

1.5.11 Node expressions

Nodes are the building blocks of abstract syntax trees (AST). Members of the domain `[`$D_1 \cdots D_n$`]`, for $n \geq 1$, are represented as `[`$e_1 \ldots e_n$`]`, where $e_1, \ldots, e_n$ are identifiers or constants in the corresponding domains $D_1, \ldots, D_n$. The syntax of AST nodes is defined as follows:

```
node_exp     ::= "[" node_elem+ "]"
node_elem    ::= variable_id | constant
variable_id  ::= defined in §1.5.4, page 24
constant     ::= defined in §1.5.4, page 24
```

The components of a tree node can be retrieved by means of the pattern-matching mechanism associated with the applications of adequately devised functions. (*§1.5.1, page 19*).

All nodes are represented internally as having a label and a tuple of emanating tree branches. The label of a node serves to distinguish it from other nodes and is implicitly defined by `flatten (`$q_1, ..., q_n$`)`, where q_i, for $1 \leq i \leq n$, are the names of the domains of the elements occurring inside the node. For instance, the label of the node `[id ":=" exp]`, for `id:Id` and `exp:Exp`, is the quotation `"Id:=Exp"`. The label of a node is used to ease the pattern-matching process (*§1.5.1, page 19*).

The tuple of emanating branches from a node whose declared domain is `[`$D_1 \ldots D_n$`]` contains the values that are in

the non-constant domains specified. For instance, in the previous example, the tuple is `(id,exp)`. A tree node can be viewed as a labeled tuple.

1.5.12 Mapping expressions

It is common in denotational semantic descriptions to define certain functions in a stepwise fashion. For instance, initially a function is defined to only return the undefined value `?`, or another constant, for all values of its argument. Then, other elements of this function are gradually defined for certain values of the argument as its definition progresses. The function definition is thus *updated* step-by-step.

Note that the term *updating function* is just an abuse of terminology. A new function value is always produced whenever the function is *updated*, as expected in the functional paradigm.

Mapping expressions are a feature that facilitate the modeling of environments and stores, quite common in semantic definitions. Consider, for example, the domain `S` of stores commonly used in standard denotational semantics [15], and suppose they are modeled as:

```
S = Loc -> Sv
```

where `Loc` is the domain of locations, and `Sv` that of storable values.

Initially, the store is assumed empty and the definition of function `s:S` reflects that fact:

```
s loc = "unused"
```

where `"unused"` is a special value in the domain `Sv`.

Later, when the value denoted by a given expression `e` is to

be associated with a given location, say `a:Loc`, in the mapping `s`, function `s` is *updated* to `s{a<-e}` such that, for any `x:Loc`,

$$\texttt{s\{a<-e\}(x)} = \begin{cases} \texttt{e} & \text{if } \texttt{x == a} \\ \texttt{s(x)} & \text{otherwise} \end{cases}$$

And this updating process may proceed by aggregating new binding pair to the function definition.

The syntax for mapping expressions is defined as follows:

```
mapping_exp   ::= "{" binding_pairs "}"
                |  "{" function_id "}"
binding_pairs ::= binding_pair ("," binding_pair)*
binding_pair  ::= expression "<-" expression
expression    ::= defined in §1.5, page 19
function_id   ::= defined in §1.5.4, page 24
```

Thus an updating function consists of a function identifier and one or more mapping expressions as the following expression illustrates:

`f{`x_1`<-`e_1`, ··· ,` x_n`<-`e_n`}{g}`

where `f` and `g` have type `A->B`; e_1, $\cdots$,e_n are arbitrary expressions in the domain `B`; and x_1, $\cdots$, x_n are expressions denoting members of `A`. The meaning of this *updating* function is :

`\x.(g(x)!=?) => g(x),(x==`x_1`) =>` e_1 `, ··· ,(x==`x_n`) =>` e_n`,f(x)`

1.6 Compilation units

A complete $\mathcal{M}$ definition of a programming language consists of a main module along with zero or more other modules. Each module is a compilation unit, i.e., a unit that can be compiled separately, although not independently, from the units it depends on and from those that depend on it. Formally, its syntax is defined as:

```
M_definition     ::= compilation_unit+
compilation_unit ::= interface_module
                   | definition_module
```

An $\mathcal{M}$ module is a mechanism for encapsulating declaration of variables, domains, grammars and functions that are related to each other, and for allowing exercising some degree of discretionary information hiding, in the sense that details of selected domains and of functions can be abstracted away, while highlighting some group of domains and functions for outside use.

There are two types of modules: the definition module, which must be held in file with extension ".m", and the interface module, which must be in file with extension ".i". Related interface and definition modules form a pair that bears the same principal name, e.g, `Command.i` and `Command.m` denote the files that hold the interface and definition of a module called `Command`.

1.7 Interface modules

The interface module should contains all declarations of the entities used in the corresponding definition modules. That is, everything declared in an interface module or imported to it is automatically avaliable for use in the corresponding definition module.

An interface module may contain up to three sections, namely **imports**, **privates** and **publics**, always presented in this order according to the following syntax:

```
interface_module  ::= "interface" interface_id
                                  interface_body "end"
```

```
interface_id      ::= proper_id
interface_body    ::= imports privates publics
imports           ::= imports_section  | ϵ
privates          ::= privates_section | ϵ
publics           ::= publics_section  | ϵ
proper_id         ::= defined in §1.1, page 3
imports_section   ::= defined in §1.7.1, page 37
privates_section  ::= defined in §1.7.2, page 39
publics_section   ::= defined in §1.7.2, page 39
```

The section named imports serves to bring to the current scope domains, variables and functions that are declared in other modules. The section named privates allows declarations of entities whose scopes are limited to the proper interface module and to the corresponding definition module. And the section named publics permits the declarations of variables, domains and functions that are usually defined in the corresponding definition module and that can be imported by other modules.

1.7.1 Imports section

The import section of an interface module allows bringing to the scope of the module pair entities defined in other modules, so they can be used in this interface module and in the corresponding definition module of the pair. The syntax of the importation mechanism is defined as follows:

```
imports_section ::= "imports" import_clauses
import_clauses  ::= import_clauses ";" windows
                  | windows
window          ::= module_id "(" imported_elems ")"
                  | empty
imported_elems  ::= imported_item ("," imported_item)*
```

```
imported_item   ::= item | item "becomes" new_item
new_item        ::= item
item            ::= domain_id   | variable_id
                  | function_id | nonterm_id
domain_id       ::= defined in §1.4, page 12
variable_id     ::= defined in §1.5, page 19
function_id     ::= defined in §1.5.4, page 24
module_id       ::= defined in §1.6, page 35
nonterm_id      ::= defined in §1.8.2, page 47
```

Name conflicts may arise when a module imports entities from other modules. However, name conflict is only a problem when the function overloading mechanism (*§1.8.3, page 59*) is not able to resolve it. In such cases, conflicts must be explicitly resolved by means of the `becomes` clause, which gives a local name for an imported entity.

The example below shows the use of the `becomes` device:

```
1 interface A   | interface B   | interface C
2 publics       | publics       | imports
3   .......     |   .......     |   A(x,z,r);
4      w:H;     |    w:H;       |   B(x becomes y,r);
5      x:N;     |    x:N;       |   -- y is the x from B
6      z:Q;     |    z:Q;       |   -- x is the x from A
7      r:Q;     |    r:Q;       |   -- z is the z from A
8   .......     |   .......     |   -- r is ambiguous
9 end           | end           | end
```

The internal structure of imported domains and types of imported functions become automatically available in the importing modules. However, only the names of entities used in their internal structure and that have been also explicitly imported may be used in the importing unit. The code fragment below should make these rules clear.

```
1 interface A            | interface B
```

```
2   imports                 |   imports
3     ...                   |     A(f,D,g);
4   privates                |     ...
5     h = D -> D            |   privates
6   publics                 |     h : D -> D;
7     f : D -> D;           |     s : D -> D
8     g : D = P -> P;       |   publics
9     P = N                 |     ...
10 end                      | end
11 module A                 | module B
12   functions              |    functions
13     ...                  |      ...
14     g p = ... ;          |      r (d) = ...;
15     n = g(1);            |      n = g(1); -- wrong
16     d = h(f(g))          |      d = h(f(d))
17 end                      | end
```

Note that, although function `f` and domain `D` have been properly imported to the body of module `B`, the operation in line 15 of `B` does not work properly because domain `P` has not been imported along, so the type of the value `d(1)` is unknown in body of `B`. To fix this, `P` should be included in the importation list of module `B`. On the other hand, line 16 of `B` is correct, for all needed information for the operation are available. Also everything in the definition module `A` is correct, because a definition module has visibility to all entities declared in the corresponding interface module.

1.7.2 Privates e publics sections

The privates and publics sections of an interface module have the following syntax:

```
privates_section ::= "privates" var_dom_dcls
```

```
publics_section  ::= "publics"  var_dom_dcls
var_dom_dcls     ::= defined in §1.3, page 9
```

The visibility rules between a interface module and its corresponding definition module are as follows:

- All entities declared in the interface module or imported to it are automatically available in the definition unit that bears the same name.
- The entities declared in the publics section of an interface modules can be imported by other modules.
- The entities declared in the privates section of an interface modules are only visible in this interface and in the corresponding definition module. They can not be imported by other modules, for they are private to the module.
- All domains of nonterminals defined in the syntax part of a definition module may be included in the publics section of corresponding interface module, so they can be imported by any other module. In such case, these nonterminals should be mentioned in the publics section of the interface module as member of the built-in domains `Nonterminal` or `Start`.

The example below illustrates a few situations.

```
1 interface A    |module A  |interface B|module B
2 import         |lexis     |imports    | v not available
3  B(z)          | v::=id   | A(a,id,D) | id available
4 publics        |syntax    |publics    | x not available
5  a,b:N;        | ...      | z:N;      | a available
6  id:Token;     | x:D::=...| r:N;      | b not available
7  D:Nonterminal | ...      | w:N       | z,r,w available
8 end            |end       |end        |end
```

The visibility rules for the above corresponding module are as follows:

- `D` of definition module `A` is visible to interface module `A`, because it is defined in a syntax section of module `A`, so it can be exported by the interface module `A`.
- `z` of `B` is visible to definition module `A`, because it has been imported to interface `A`.
- `z`, `r` and `w` of B is visible to definition module `B` because they are declared in the interface module `B`.
- `x`, `b` and `v` of `A` are not visible to module `B`, for they have not been imported to `B`. Notice that `b` could have been imported, but `x` and `v` could not, because they are note visible in the interface module `A`.

1.8 Definition modules

Definition modules are also organized into up to three sections, namely **lexis**, **syntax** and **functions**, always presented in this order. The syntax and lexis sections define the grammatical rules for the concrete and abstract syntactic structure of language constructs. The functions section of a module is used to provide the definition of semantic functions.

Declarations of variables, domains, or types of functions are not allowed in definition modules. These entities, if needed in the definition module, must be declared in the corresponding interface module or imported to this interface. Thus, most definition modules must have a companion interface module, which performs all needed importation of variables, domains, or functions from other modules. The interface module must

also declare all entities that are used or defined in the corresponding definition module, and specifies the collection of the elements defined in the module and that can be imported by other modules.

Definition modules have the following syntax:

```
definition_module ::= "module" module_id
                                  module_body "end"
module_id         ::= proper_id
module_body       ::= lexis syntax functions
lexis             ::= lexis_section | ϵ
syntax            ::= syntax_section | ϵ
functions         ::= functions_section | ϵ
proper_id         ::= defined in §1.1, page 3
lexis_section     ::= defined in §1.8.1, page 42
syntax_section    ::= defined in §1.8.2, page 47
functions_section ::= defined in §1.8.3, page 54
```

One of the modules of a complete $\mathcal{M}$ denotational specification must have the definition of the distinguished function with the reserved name `main` (*§1.9, page 61*). This user-defined function must map elements of type `Q*` to some type defined by the user according to the value produced by its body. The arguments of `main` are usually information on the execution environment. The module containing the function `main` is recognized as the main module of the formal definition.

1.8.1 Lexis section

This part of a definition module copes with the specification of the set of special tokens that can be used in syntax sections. All terminal symbols, i.e., quotations appearing in production rules of the syntax section of the modules that form a

semantic definition, are automatically collected and properly incorporated in the implicit or explicit lexis section of each respective module. The programmer only has to specify the lexical definitions of the additional tokens.

The production rules that occur in the lexis section of modules are restricted to be very simple: no recursion is allowed neither is element grouping.

The left-hand side of each production rules is an undecorated common identifier (*§1.1, page 3*), possibly accompanied by its domain name, which is implicitly a member of the domain `Token`, and can thus be exported to other modules.

The right hand side of a lexis production must be regular expressions consisting of just a sequence of terminals and nonterminals, possibly including nonterminals decorated indices and/or * and +.

The syntax of the lexis section of a definition module is defined as follows:

```
lexis_section      ::= "lexis" token_defs
token_defs         ::= token_defs ";" token_definition
                     | token_definition
token_definition   ::= regular_production
                     | range_production
                     | exception_production
                     | empty
regular_production ::= lhs "::=" reg_exps
reg_exps           ::= reg_exp ("|" reg_exp)*
reg_exp            ::= regular_factor* "=>" lexis_exp
                     | regular_factor*
regular_factor     ::= terminal
                     | lex_nonterm_id
range_production   ::= lhs "===" character_ranges
lhs                ::= new_nonterm_id ":" domain_id
```

```
                      | new_nonterm_id
character_ranges   ::= char_range ("|" char_range)*
char_range         ::= char ".." char
except_production  ::= lhs "=/=" char
char               ::=  any ASCII character enclosed in '
lexis_exp          ::= "return" syntactic_unit
                      | token_value
syntactic_unit     ::= "(" token_code "," token_value ")"
token_code         ::=  common_id
token_value        ::= basic_exp
lex_nonterm_id     ::= common_id  | common_id_list
                      | common_idx | common_idx_list
common_id          ::= defined in §1.1, page 3
common_idx         ::= defined in §1.1, page 3
common_id_list     ::= defined in §1.1, page 3
common_idx_list    ::= defined in §1.1, page 3
new_nonterm_id     ::= defined in §1.8.2, page 47
```

Range production is a special production rule whose syntax has been borrowed from P. Mosses [30]. They are production rules whose alternatives can only be an interval of ASCII characters. The value produced is always the character recognized.

Ranges are distinguished from normal production rules by the use of the symbols `===` or `=/=`, instead of `::=`, to separate the production sides. The `===` symbol defines the recognition of characters that belong to a given set of characters defined by a lexicographical order interval, and the `=/=` symbol defines the recognized character as those in the complement of the specified set. In the example

```
digit === '0' .. '9'
comment_char =/= ';'
```

the range defined by `digit` is equivalent to:

```
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6'
        | '7' | '8' | '9'
```

and `comment_char` is any ASCII character but a semicolon.

As for the associated semantics, the recognition of token `digit` by means of the rule

```
digit  === '0' .. '9'
```

automatically associates with `digit` the string containing the digit encountered during the parsing process.

In the case of a rule like `num ::= digit+`, the nonterminal `digit+` contains the concatenation of the digits encountered during the recognition process, and this string of digits (of type `Q`) becomes the value associated with `num`. A different value, though, may be associated with a lexis nonterminal by attaching a lexis expression to its defining rule. The value of this expression supersedes the rule default value.

The syntactic unit returned by the lexical analyzer specified by a lexis section is a token expression that denotes the construction of a token given its code and its value. The token code is the grammar symbol properly declared in a lexis production, and the token value is a value of type Q.

For example, the expression `(id,"abc")` that is a syntactic unit associated with a return expression occurring in a lexis section, denotes a pair whose first component is the token code of `id`, and the second component is the token value defined as the string `"abc"`.

A lexis expression of the form `return (id,v)` concludes the recognition of the token `id`. The value returned by the lexical analyzer is the specified tuple. In syntactic contexts, references to the token identifier `id` refer to its token code. In other contexts, the associated quotation value, i.e., the token value is automatically retrieved.

The standard built-in function `value:`*Token*`->Q` is such that `value(`t`)`, where t is a token code, may be used to retrieve the value associated with token `t`, but, most of the time, this is not necessary. For instance, in the following code, where terminal `id` has been properly declared elsewhere as `id:Id`,

```
1 syntax
2    stmt:Stmt ::= id ":=" exp => [id ":=" exp]
3 functions
4    exec[id ":=" exp] r c s =
5                    ... r(id) ... q ...
6    where q = id
```

the occurrences of token `id` in lines 2 and 4 need the token code, in order to perform pattern-matchings, while those in lines 5 and 6 refer to the value associated with the token, i.e., these latter occurrences of `id` are equivalent to `value(id)`.

The token `id` must have been declared in some lexis section as in:

```
1 lexis
2    id: Id   ::= letter+ => return (id,letter+);
3    letter   === 'a' .. 'z' ;
4    num: Num ::= digit+  => return (num,digit+);
5    digit    === '0' .. '9'
6 end
```

Each definition module has a proper `lexis` part, be it explicit or implicit. Thus, there are a lexical analyzer and a syntactic analyzer for each module that has a syntax section The lexical and syntactic analyzers of all modules in a semantic definition of a language are automatically regrouped by the $\mathcal{M}$ execution environment to form the parser of the entire language.

Tokens defined in a module can be used in other modules as long as their domains are exported by the corresponding interface modules, in which case these domains must be declared of type `Token` in the publics section of the exporting module and properly imported to the context they are needed.

1.8.2 Syntax section

The purpose of the syntax section of a definition module is to provide the specification of the concrete syntax of an entire programming language or simply of a part of it, and also for indicating how the abstract syntax tree (AST) for programs can be derived from the concrete syntax.

From the syntax sections of all modules that comprise the definition of a language, the $\mathcal{M}$ execution system generates, among other results, the function `compile:File->AST` to map source code of programs written in the language being defined into AST code.

The syntax specification is a context-free grammar in which each production rule implicitly or explicitly has an associated directive for building the corresponding node of the abstract syntax tree for the program being parsed.

All nonterminals and their respective domains, which are defined in the syntax part of a module, are visible to the corresponding interface module, and thus can be listed in its public section so as to be exportable.

Usually, only the domains of nonterminals need be exported, since all the information regarding the members of a domain is exported along with the domain. In fact, the structure of these exported domains is their definition as nodes or values

of the associated AST derived from the production specifications. The domain of all nonterminals is given by the keyword `Nonterminal`, which must be used to declare the domains that are exported.

Syntatic definition of the language being defined may be encapsulated in a module unit, or composed of separate pieces of the grammar definition scattered throughout several modules. The $\mathcal{M}$ execution environment collect all of them in order to produce the complete grammar that is necessary to generate the corresponding lexical analyzer and the parser.

The start symbol of each piece of grammar is the left hand-side of its first rule, while the start symbol of the complete grammar is the nonterminal declared to be in a domain that is declared as `Start` in the publics section of the corresponding interface module.

A parse table should be associated with each piece of grammar, and the start symbol of each piece is used to connect all the parse tables produced by the collection of modules.

The syntax section of a module is defined by:

```
syntax_section      ::= "syntax" productions
productions         ::= production ";" production
                      | production
production          ::= rule_lhs "::=" alternatives
                      | empty
rule_lhs            ::= new_nonterm_id ":" domain_id
                      | new_nonterm_id
alternatives        ::= alternative ("|" alternative)*
alternative         ::= element* "=>" ast_info
                      | element*
element             ::= nonterm_id | terminal
terminal            ::= quotation
ast_info            ::= basic_exp
```

```
new_nonterm_id      ::= common_id
nonterm_id          ::= common_id  | common_id_list
                      | common_idx | common_idx_list
common_id           ::= defined in §1.1, page 3
common_idx_list     ::= defined in §1.1, page 3
domain_id           ::= defined in §1.3, page 9
basic_exp           ::= defined in §1.5.4, page 24
```

This grammar specification defines that each production serves the purpose of defining a nonterminal and its corresponding domain. Thus, each production has a nonterminal and its domain on the left side of `::=`, and a list of alternatives, separated by `|`, on the right side.

The specification of the domain of nonterminals is optional. When the domain specification is omitted, the domain of the nonterminal on the left hand side is assumed to be the one whose name is that of the nonterminal with its first letter capitalized. All domains of nonterminals are themselves, by default, in the built-in domain `Nonterminal`. Of course, domains of nonterminals need not to be declared, except to mention them in the public list of export entities of an interface module, for example, as in line 7 of the following code:

```
1  interface Expression
2  imports
3     Tokens(Id,Num,Aop,Mop);
4     Exp_Components(Ed,send,read,operate,fcall);
5     Exp_Components(choose,dereference);
6  publics
7     Exp : Nonterminal;
8     R   : Exp -> Ed ;  -- r-expressions
9     E   : Exp -> Ed ;  -- l-expressions
10 end
```

Each rule alternative may have an attached expression,

which defines the default action for constructing the corresponding abstract syntax tree node (*§1.8.2, page 51*).

Concrete syntax

Nonterminal symbols are written as common identifiers, and distinguished occurrences of the same nonterminal in a given production rule may be suffixed with a string of decimal digits in order to differentiate them for semantic purposes, while preserving their syntactic meaning.

A terminal symbol is a quotation or an identifier properly declared in the lexis section of a module. Recall that terminal symbols may be imported from other modules by importing their domain, which must be declared as `Token`. For example, given the following two interface modules:

```
interface A          | interface B
  publics            |   imports A(Id,Num);
    Id  : Token;     |   privates
    Num : Token;     |     id : Id; num : Num;
end                  | end
```

The tokens `id` and `num` may be used in module `B`.

A production alternative defines a possibly empty sequence of terminals and/or nonterminals. For example, the classical grammar for expressions can be specified as:

```
exp    : Exp ::= exp "+" term
               | term ;
term   : Exp ::= term "*" factor
               | factor ;
factor : Exp ::= id
               | constant
               | "(" exp ")" => exp ;
```

From the concrete syntax and lexis specifications, the $\mathcal{M}$ compiler generates LALR(1) parsing-tables, scanner routines, and ultimately produces a compiler that translates programs in the specified language into the corresponding abstract syntax tree code.

Abstract syntax

An abstract syntax tree (AST) is generated during program parsing and it is composed of nodes corresponding to the productions involved in the recognition process. Unless otherwise specified, nodes are constructed from the values associated with each grammar symbol that occurs on the right hand side of the considered production. A tree node of the AST possesses a label and a structural value, which is a tuple of emanating branches that are the values of the symbols on the production alternative. The label is built by the concatenation of the terminal and domain names of the grammar symbols that occur on the corresponding production alternative. A terminal is the quotation that represents the terminal itself or its identifier when it is declared to have type `Token`. The domains of nonterminals are those implicitly or explicitly declared for each one of them.

The structural value associated with terminal symbols is the terminal itself, which is a quotation or an identifier. The value associated with a nonterminal is the one assigned to it when the considered nonterminal was produced by the *reduce* action of the parser. Those values are put together to form a node according to the structure of the production alternative. The resulting node becomes the value associated with

the nonterminal on the left hand side of the production rule.

However, not all productions produce a tree node automatically. These exceptions are defined as follows:

1. Null-rules do not generate nodes. They just propagates the null string value to the nonterminal on the left hand side of the production rule.

2. Rules that possess just one symbol on the right hand side do not generate nodes. The value associated with this symbol becomes the value of the nonterminal on the left side.

3. Rules whose right hand side possesses an expression attached to it. In this case, the value of the expression attached to the alternative is associated with the nonterminal that is on the left hand side of the production.

In situations in which the alternatives have more the one symbol, or the alternatives have an attached value-expression of type node, a tree node is generated in one of two ways:

1. In productions without a value-expression attached, the tree node is built directly from the symbols on the production alternative.

2. In productions with a value-expression attached, the value produced is that of the attached expression, which must be a basic expression (*§1.5.4, page 24*). If the expression attached to the rule alternative is a node expression, the terminals and the nonterminals that compose this expression are used to construct the label and its value. The value-expression may only contain operands

which are terminal or nonterminal symbols occurring in the corresponding production alternative. The value of each operand is that of the value expression implicitly or explicitly associated with it. Nonterminals that occur more than once in a production alternative may be indexed to avoid ambiguity when referenced. Indices do not change the domain of nonterminals.

This mechanism for producing abstract syntax tree allows the following transformations, which are carried out according to the value-expression attached to the production alternative:

1. Elimination of precedence information present in concrete syntax specifications by an appropriate domain specification. For example, given the production rules:

```
exp    : Exp ::= exp "+" term
               |  term
               ;
term   : Exp ::= term "*" factor ;
factor : Exp ::= id => [id] ;
```

   where `exp`, `term` and `factor` are declared to be in the domain `Exp`, the resulting abstract syntax produced would be:

```
Exp = [ Exp "+" Exp ]
    | [ Exp "*" Exp ]
    | [id]
```

2. Placement of constructs with similar semantic properties in a single syntactic domain, while making sure that domains with different semantic roles are distinct.

3. Elimination of production rules that have lost their semantic significance after the operations described in the

items above had been performed. These are, in general, productions that would only yield extra *chain-reduction* nodes in the parse tree. Thus, nodes that correspond to alternatives containing only one nonterminal symbol, such as `term` above, are automatically eliminated.

4. Addition of terminal symbols to the node structure, e.g., parentheses, as in:

```
x:X ::= x "t" c => [x  "t" "(" c ")"] ;
c:C ::= "n" ;
```

5. Changing the occurence order of the constituents of constructs. For example, if `a+b`, `+ab`, and `ab+` are different representations of the same expression, it might be desirable to reorder the components of some of them so as to have just one abstract form. Operations of this type are useful to reduce the number of semantic equations.

1.8.3 Functions section

The functions section of a definition module is initiated by the keyword `functions` and contains definitions of semantic functions. The syntax of the functions section is as follows:

```
functions_section ::= "functions" definition_list
definition_list   ::= definition_list ";" fun_definition
                    | fun_definition
fun_definition    ::= main_definition
                    | main_definition where_clause
                    |empty
main_definition   ::= lhs "=" expression
where_clause      ::= "where" where_definitions
where_definitions ::= main_definition
                              ("and" main_definition)*
```

```
lhs               ::= pattern_exp
                    | function_name pattern_exp*
function_name     ::= common_id | common_idx
                    | proper_id | proper_idx
                    | "main"
pattern_exp       ::= defined in §1.5.2, page 22
exp               ::= defined in §1.5, page 19
```

The scope of all first-level definitions is the entire collection of defined functions. The order definitions are listed is not relevant. That is, one can freely use any functions defined on the same level. A where-clause introduces a new hierarchical definition level. Functions defined inside a where-clause are local to this clause and to the function the `where` is attached. In order to keep the language simple, just one level of where clause is allowed, i.e., where clauses inside a where-clause are not allowed. Functions in a where-clause may call functions defined in sibling clauses or in enclosing levels.

In the following program fragment the names visible in the bodies of functions `f1`, `f2`, `f3`, `g1` and `g2` are only those listed in their respective bodies.

```
1    f1 a = f1 f2 f3 g1 g2 a
2       where g1 x = f1 f2 f3  g1 g2  a  x
3       and   g2 y = f1 f2 f3  g1 g2  h1 a y
4       and   h1 z = f1 f2 f3  g1 g2  h1 a y z;
5    f2 b =  f1 f2 f3 b;
6    f3 c =  f1 f2 f3 c;
```

For instance, in the body of `f2`, the only visible names are `f1`, `f2`, `f3` and `b`, and in the body of `h1`, only `f1`, `f2`, `f3`, `g1`, `g2`, `h1`, `a`, and `z` are visible.

Notice that only `f1`, `g1` and `g2` see `h1`, which sees all the functions specified on the same or in enclosing levels.

The left hand side of a definition may be a pattern expression or a function heading. The former provides a mechanism for decomposing a structured value. For instance, in the following program fragment,

```
1 privates
2     S = (A,B)
3     ...
4 functions
5     s = (a,b);
6     ...
7     (a1,b1) = s;
8     ...
```

the definition in line 7 decomposes the value of `s` and binds its components to `a1` and `b1`.

In fact, an *assignment* `p = e`, where `p` is a pattern and `e`, an expression, is equivalent to `(\p.c)e`, where `c` is the context in which the binding produced is to be available.

The body of a function is an expression, which may be literal constants, variables, integer expressions, quotation expressions, logical expressions, lists, tuples, nodes, patterns, inquiry expressions, conditional expressions, functional expressions, functional applications or any well-formed combination of simpler expressions and operators. The body is evaluated to produce the value to be returned, whose domain must be compatible (*§1.4.8, page 17*) with the declared return type.

The types of all functions used in a module unit must be those properly imported from other units or explicitly declared in the corresponding interface module.

However, since the domains of formal parameters may be of union type, there may be, in a same scope, several definitions of the same function, each identified by specific members of

these unions.

In fact these definitions just constitute distinct cases of a single function definition whose body selects the specific case based on the values or types of the parameters that are passed to it.

For example, given the fragment of an interface module `A`,

```
1 interface A
2     ...
3 privates
4     X = N | Q | T;
5     f,g : X -> N;
6     ...
7 end
```

Suppose that in the corresponding definition module, function `f` is defined four times, three are special cases, and the fourth is a general definition. Thus, the function application `f(10)` in line 8 below performs the function defined in line 4. The function call `f("abc")` refers to function definition on line 5. The application `f(x)` activates the function of line 6.

```
1 module A
2 functions
3     f(0) = 0;
4     f(n) = n;
5     f(q) = 1;
6     f(x) = 2;
7     ...
8     g(x) = ... f(10) ... f("abc") ... f(x) ... f(0) ... ;
9     ...
```

The application `f(0)` also in line 8 is, in principle, ambiguous in the sense that definitions on lines 3 and 4 are equally applicable. This kind of ambiguity is resolved favoring the definition that textually comes first. So, in this case, the

definition on line 3 is chosen. Note that this only works for function definitions occurring intra-module, because there is no ordering relation between definitions occuring in different modules. The relation *come first textually* is a partial order, it only holds within a module.

A more meaningful example of the use of the order of presentation of the function definitions to resolve conflicts is:

```
1 functions
2    apply: Q -> (Rv,Rv) -> Ec -> Store -> Ans;
3    apply("+")(v1,v2) k s = k (v1 + v2) s;
4    apply("-")(v1,v2) k s = k (v1 - v2) s;
5    apply("*")(v1,v2) k s = k (v1 * v2) s;
6    apply("/")(v1,v2) k s = k (v1 / v2) s;
7    apply(op)(v1,v2)  k s = "error";
```

in which function definitions are processed in the order they are presented, so the function definition in line 7 is selected only if all previous definitions do not apply.

Generally speaking, formal parameters of an $\mathcal{M}$ function are indeed pattern-expressions, which may contain identifiers to be bound to corresponding components of the function's argument. Thus, in order to identify the proper definition of the function being called, a pattern-matching of the function's arguments must be performed with the patterns defined by the formal parameters. And these pattern-matching tests are ordered according to the textual occurrences of the function's definitions on the module.

$\mathcal{M}$ is a pure, strict functional language, so that the evaluation of function arguments is performed immediately before function call. Note that the pattern-matching mechanism forces the function's arguments to be evaluated before parameter passing, and that curried functions have just one

argument, which is the only one evaluated before function call.

Overloading resolution

As different functions may share the same name in the same scope, that is, names of functions can be overloaded, more than one function definition may be elegible for application as a result of a given function call.

In order to resolve name overloading, the type and value of each argument of the function application, and those of the corresponding formal parameters of the candidate function definitions must provide enough information to tell these functions apart.

The basic rule is that, in a function call, the domain of each actual parameter must be compatible to the domain of the corresponding formal parameter (*§1.4.8, page 17*).

If, after this compatibility test, more than one function definition are still candidates, the function application is inherently ambiguous.

More specifically, the overloading resolution procedure to identify the function to be called in a functional application of the form `f(`$a_1 \ \cdots \ a_n$`)`, where a_i is in domain A_i, for $i \geq 1$, follows the steps:

1. From the function designator `f`, define the set C of candidate functions as containing all functions with the same name `f` in the considered scope.
2. For each element a_i of the actual parameter tuple, for $1 \leq i \leq n$, eliminate from C functions that have a corresponding formal parameter p_i in domain P_i, such that A_i

is not compatible (*§1.4.8, page 17*) with P_i.

3. If, at this point, the cardinality of C is still greater than 1, perform the following additional steps:
   (a) Assign to each function in C the number of *nominal-matchings* occurring between the domains of the components of the tuple argument and those of the corresponding parameters. A nominal-matching occurs when both domains has identical names.
   (b) Keep in C only the functions to which the greatest nominal-matching number has been assigned.
4. If C becomes empty, the called function has not been declared in the calling scope.
5. If cardinality of C is 1, C contains the function to be called, otherwise, the function call is ambigous.

For example, given the following definition of function types and applications:

```
1  interface M
2  privates
3     A = N; B = Q; C = N; D = Q;
4     f : A -> B -> A;  -- function 1
5     f : N -> B -> N;  -- function 2
6     f : Q -> N -> A;  -- function 3
7  end
8  ...
9  module M
10 functions
11    ...
12    f a b = a;          -- function 1 definition
13    f n b = n;          -- function 2 definition
14    f q n = q;          -- function 3 definition
```

```
15     g(...) = ...f(a)...f(n)...f(q)...f(c)...f(d)... ;
16     ...
17 end
```

The overloading resolution is as follows:

- `f(a)` refers to function 1 defined at lines 4 and 12
- `f(n)` refers to function 2 defined at lines 5 and 13
- `f(q)` refers to function 3 defined at lines 6 and 14
- `f(c)` is ambiguous: functions 1 and 2 are candidates
- `f(d)` refers to function 3

Curried functions are one-argument strict functions, so only one argument is processed at a time. The argument, which may be a tuple of expressions, must bring enough information to resolve the overloading.

For instance, in the evaluation of `f a b c`, first `a` is evaluated, and the application `f a` is performed. Then `b` is evaluated to execute `r b`, where `r` is the function returned by `f a`, and so for the argument `c`.

1.9 The main module

All complete semantic descriptions must have a main module, which should establishes the description context, such as, the source program file, data input files, the initial environment and the initial store. The main module is the one from which the execution of the semantic definition must start.

The main module has the same structure as the other modules, except that it is the one containing the definition of a special monadic function named `main`.

The type of the argument of function `main` is predefined to be `Q*`, but the type of the return value and the body of `main` must be defined by the user, as in the following example.

```
1  interface miniL
2  imports
3     Program(Ans);
4  publics
5     main : Q* -> Ans;
6  end
7  module miniL
8  functions
9     main(arg*) = ... ;
10 end
```

The name of the module containing the function `main` is used as the external name of `main`, that is, a call for execution of the main module causes the activation of the corresponding function `main`. The way main modules are called for execution is implementation dependent. However, typically, this is accomplished via a command dispatched from the command line of the computer terminal. This command line should passes to function `main` the necessary argument, which usually is a list of quotations, designating tags and file names.

For instance, the command line

```
>miniL -f source.m -i in.txt -o out.txt
```

where `source.m` is the source program file, `in.txt` is the program input file and `out.txt` is the program output file. will produces the call

```
main( ("-f", "source.m", "-i", "in.txt", "-o","out.txt") )
```

The elements of the list of quotations passed as argument to `main` may be retrieved by means of the standard function

`getarg(q,q*)`, which has type `getarg:(Q,Q*)->Q`. So that the application `getarg(q,q*)` returns the first string that follows string `q` in the list of quotations `q*`. Usually, `q` is a tag, such as "-f", "-i", and "-o" in the command line shown above.

The definition module below illustrates the use of this and other built-in functions to initiate a program execution. Note that in order to keep the example clean, there is no provision for error detection in the body of `main`:

```
1  interface miniL
2  imports Program(Prog,exec,Ans) ;
3  privates
4     arg, filename : Q;
5     source, input : File;
6  publics
7     main : Q* -> Ans;
8  end
9  module miniL
10 functions
11    main(arg*) =  exec(prog)(input)
12    where filename1 = getarg("-f", arg*)
13    and   source    = open(filename1)
14    and   filename2 = getarg("-i", arg*)
15    and   input     = open(filename2)
16    and   prog      = compile(source);
17 end
```

There can be more than one main module per semantics definition. So there may be more than one semantic description sharing modules, and the execution of each definition should be started from different main module.

1.10 Module `System`

Module `System` defines a collection of built-in functions, which are automatically imported to all context. This module possesses the following interface, in which `D` is any domain, and *Token* represents the domain of grammar symbols:

```
1  interface System
2  publics
3     append    : (D*,D) -> D*;          --(§1.5.10, page 31)
4     ascii     : N  ->  Q;              --(§1.5.6, page 27)
5     close     : File -> File;          --  (§1.2, page 6)
6     compile   : File -> AST;           --(§1.8.2, page 47)
7     cond      : (D,D) -> T -> D;       --(§1.5.3, page 23)
8     eof       : File -> T;             --  (§1.2, page 6)
9     flatten   : Q* -> Q;               --(§1.4.4, page 14)
10    getarg    : (Q,Q*) -> Q;           --(§1.9, page 61)
11    getchar   : File -> (File,N);      --  (§1.2, page 6)
12    head      : D+ -> D;               --(§1.4.4, page 14)
13    main      : Q* -> D;               --(§1.9, page 61)
14    open      : Q -> File;             --  (§1.2, page 6)
15    putchar   : (File,N) -> File;      --  (§1.2, page 6)
16    size      : Q -> N;                --(§1.5.7, page 28)
17    size      : D* -> N;               --(§1.4.4, page 14)
18    tail      : D+ -> D*;              --(§1.4.4, page 14)
19    toN       : Q -> N;                --(§1.5.7, page 28)
20    toQ       : N -> Q;                --(§1.5.6, page 27)
21    toT       : Q -> T;                --(§1.5.7, page 28)
22    ungetchar : (File,N) -> File;      --  (§1.2, page 6)
23    value     : Token -> Q;            --(§1.8.1, page 45)
24    Y         : (D -> D) -> D;         --(§1.5.8, page 30)
25 end
```

*****

Chapter 2

The Description of a Computer Architecture

Simplicity is the ultimate sophistication.
Leonardo da Vinci (1452-1519)

Denotational semantics may be used to precisely describe computer architecture as this chapter demonstrates with the formal definition of a computer that features a high-level language architecture.

This virtual computer, named $\mathcal{SC}$, possesses a stack to hold the operands and the results of the execution of its instructions. Memory administration and scope control are performed automatically by high-level machine instructions.

2.1 The machine architecture

The execution of a program in the machine $\mathcal{SC}$ produces a sequence of state transformations. Each state in this sequence consists of a quintet defined by the tuple

`State=(Env, Stack, Store`, `Dump`, `Output)`,

where

- `Env` is the domain of environments.
- `Stack` is the of domain execution stacks.
- `Store` is the domain of computer memories or stores.
- `Dump` is the domain of activation record stack.
- `Output` is the domain of printable values.

2.1.1 The environment

The environment takes care of the scope mechanism and provides associations of variables with their corresponding denotable values, which can be a location, a right-value, a procedure value or a function value.

The scope mechanism implements the traditional static scoping of imperative languages of the Pascal family, i.e., procedures and functions are executed in their declaration environments, not in the environment from which they are called. Upon returning from functions and procedures, the environment that existed prior the corresponding call must be restored.

```
1  interface Menvironment
2  imports Minstructions(Code,Ecode);
3  privates
4     d : Dv | "unbound";
5  publics
6     Env       = Alist*;                  --machine env
7     Alist     = Pair* ;                  --association list
8     Pair      = (Var,Dv);                --an association
9     Dv        = Loc | Rv | Proc | Fun;   --denotable values
10    Var       = Q;                       --instr operands
11    Loc       = N;                       --store locations
12    Rv        = Bv | Bool;               --r-values
13    Bool      = T;                       --logical values
```

```
14    Bv        = N;                          --basic values
15    Proc      = (Code,Env);                 --procedure value
16    Fun       = (Ecode,Env);                --function value
17 -- operations
18    env0      : Env;
19    empty     : Env -> T;                   --is env empty?
20    empty     : Alist -> T;                 --is alist empty?
21    bind      : Pair -> Env -> Env;         --binds a var
22    search    : Var -> Env  -> (Dv | "unbound");--gets dv
23    search    : Var -> Alist-> (Dv | "unbound");--gets dv
24    pop       : Env -> Env;                  --discards alist
25    push      : Alist -> Env -> Env;        --pushes an alist
26    top       : Env -> Alist;               --gets topmost alist
27    locations : Alist -> Loc*;              --gets locations
28    locations : Env -> Loc*;                --gets locations
29 end
```

An environment is a stack of *Association Lists*. An association list, a member of `Alist`, represents an environment layer containing a list of pairs that associate variables with their denotable values. The definition of the operations over environments is detailed in the following module unit:

```
1  module Menvironment
2  functions
3     env0 = nil;
4     empty(env)   = (size env == 0);
5     empty(alist) = (size alist == 0);
6     bind(var,d)(env) = (empty env) => (((var,d))),
7                              ((var,d):head env):(tail env);
8     search(var)(env) = (empty env) => "unbound",
9             (d != "unbound") => d, search(var)(tail env)
10    where d = search(var)(head env);
11    search(var)(alist) = (empty alist) => "unbound",
12            (var1 == var) => d1, search(var)(tail alist)
13    where (var1,d1) = head alist;
```

```
14    pop env = (empty env) => ?, tail env;
15    top env = (empty env) => ?, head env;
16    push(alist,env) = alist:env;
17    locations(env)   = empty(env) => nil,
18             locations(head env) + locations(tail env);
19    locations(alist) = empty(alist) => nil,
20         (d is Loc) => (Loc(d)) + locations(tail alist),
21                       locations(tail alist)
22    where (var,d) = head(alist);
23 end
```

2.1.2 The store

The state component of type `Store` models the computer memory, which associates locations with values. For practical purpose, the memory has a fixed size, so memory overflow may happen during execution.

```
1  interface Mstore
2  imports Mcontinuations(Merror); Menvironment(Bv,Bool);
3  privates
4     next    : Loc -> (Loc | Merror);
5     getFree : (Store,Loc) -> (Loc | Merror);
6     s       : Store; a : Loc; v: Sv;
7  publics
8     Store  = Loc -> (Sv | "unused"); -- machine memory
9     Loc    = N;                      -- store locations
10    Sv     = Bv + Bool;              -- storable values
11    maxLoc : Loc;                    -- highest location
12 -- operations
13    new  : Store -> (Loc | Merror); -- gets free cell
14    free : Loc* -> Store -> Store; -- frees locations
15    get  : Loc -> Store -> Sv;      -- reads from location
16    save : (Loc,Sv)->Store->Store;  -- stores at location
17    s0   : Store;
18 end
```

The definition of the above operations over machine stores is detailed in the following module unit.

```
module Mstore
functions
   maxLoc = 32767;
   s0 a = "unused";
   new s = getFree(s,0);
   where getFree(s,a) = (s(a) == "unused") => a,
         (a == maxLoc) => "Storage-Full", getFree(s,a + 1);
   free a* s = empty(a*) => s , free tail(a*) s1
   where s1 = s{head a* <- "unused"};
   get a s = s(a);
   save(a,v)s = s{a<-v};
end
```

2.1.3 The stack

All expression evaluations are carried out on the machine stack, i.e., operands of any operation are retrieved and popped from the stack and the result of all operations are pushed back on the stack.

```
interface Mstack
imports Menvironment(Bv, Bool, Proc, Fun); Mstore(Loc)
privates
   e : Ev;
publics
   Stack = Ev*;
   Ev    = Bv|Bool|Loc|Proc|Fun;  -- expressible values
-- operations
   stack0   : Stack;
   empty    : Stack -> T;
   tooShort : Stack -> T;
   pop      : Stack -> Stack;
   push     : Ev -> Stack -> Stack;
```

```
14    top      : Stack -> Ev;
15    isBool   : Ev -> T;
16    areBool  : (Ev,Ev) -> T;
17    areBv    : (Ev,Ev) -> T;
18 end
```

The definition of the above operations over machine execution stack is detailed in the following module unit.

```
1  module Mstack
2  functions
3     isBool(e)      = e is T;
4     areBool(e1,e2) = (e1 is T) & (e2 is T);
5     areBv(e1,e2)   = (e1 is Bv) & (e2 is Bv);
6     stack0         = nil;
7     empty stack    = (size stack == 0);
8     tooShort stack = (size stack < 2);
9     pop stack      = (empty stack) => ?, tail stack;
10    top stack      = (empty stack) => ?, head stack;
11    push e stack   = e:stack;
12 end
```

2.1.4 The dump

Upon entering a procedure or a function body, the current environment must be pushed on the dump component of the state, and a new environment layer must be pushed on the environment stack. The dump stack allows the environment to be restored when the procedure or function in execution returns.

```
1  interface Mdump
2  imports Menvironment(Env)
3  publics
4     Dump  = Env*;  -- environment stack
5     dump0  : Dump;
```

```
6     empty  : Dump -> T;
7     pop    : Dump -> Dump;
8     push   : Env -> Dump -> Dump;
9     top    : Dump -> Env;
10 end
```

The definition of the above operations over dump stack is detailed in the following module unit.

```
1  module Mdump
2  functions
3     dump0          = nil;
4     empty(dump)    = (size dump == 0);
5     pop dump       = (empty dump) => ?, tail dump;
6     top dump       = (empty dump) => ?, head dump;
7     push env dump  = env:dump;
8  end
```

2.1.5 Files

The input and output operations are defined in module `Mio`. The output file records the execution results.

```
1  interface Mio
2  imports Menvironment(Rv); Mcontinuations(Merror);
3          Mstore(Loc,Store,save);
4  privates
5     readN   : File     -> (N,File); -- reads an integer
6     readN   : (File,N) -> (N,File); -- auxiliary function
7     zero    : N;                    -- code for ASCII "0"
8     nine    : N;                    -- code for ASCII "9"
9     a       : Loc;
10 publics
11    Output = nil | (Output,Ov);       --output file
12    Ov     = Rv  | ("stop" + Merror); --outputable values
13 -- operations
14    readint : (Loc,Store) -> (N,Store);
```

```
15    put     : (Ov,Output) -> Output;
16    o0 : Output;
17 end
```

In case of an execution error is detected, the output file records the appropriate error message, otherwise, at the end of execution, the message recorded is `"stop"`, which signalizes the success of the execution run.

The definition of the above operations over the input and output files is detailed in the following module unit, in which the built-in function `getchar(file)` reads the next input character from the input file and returns the integer value that represents the ASCII code associated with character read.

```
1  module Mio
2  functions
3     zero      = 48;
4     nine      = 58;
5     isdigit n = (n >= zero) & (n <= nine);
6     digit n   = n - zero;
7
8     readint(a,s)= eof(file) => (?,s),(n,save(a,file1)s)
9     where file = s a; (n,file1) = readN(s a);
10
11    readN(file)   = eof(file) => (?,file), readN(file,0);
12    readN(file,n) = eof(file)    => (n,file),
13                    !isdigit(n1) => (n,file),
14                    readN(file1,10 * n + digit(n1))
15    where (n1,file1) = getchar(file);
16
17    o0 = nil;
18    put(o,ov) = (o,ov);
19 end
```

2.1.6 Continuations

Suppose that function `MC : Code -> State -> State` give the meaning of a sequence of $\mathcal{SC}$ instructions, i.e., `MC` applied to an instruction sequence and the current machine state produces a new state. And, naturally, this operation is carried out executing, in order, each instruction in the sequence, according to the equations:

```
MC(nil)q = q
MC(I:c)q = MC(c)(MI(I)q)
```

where function `MI: Instr -> State -> State` executes an individual instruction, such as `I:Instr`, followed by a sequence `c:Code`, being `q:State` the current machine state.

Clearly, the flow of execution is modeled via function composition: the state produced by the execution of an instruction is passed to the execution of the next one, i.e., the above equation may be rewritten as:

```
MC(I:c) = MC(c) o MI(I)
```

This encoding makes clear that the execution of a sequence of instructions is defined as the execution of the first instruction to produce a machine state that is to be passed to the function that computes the meaning of the remaining instructions in that sequence.

However, when the execution of a instruction causes an error, the machine state should not be passed to the instructions that follow in the control flow. In this case, an appropriate error messagem must be issued instead. In other words, if `MI(I)` produces an error message instead of a new state, its composition with `MC(c)` must be avoided. Of course, to implement this, the types of `MC` and `MI` must be changed accordingly:

```
MC: Code  -> State -> State | "error"
MI: Instr -> State -> State | "error"
```

and the equation above becomes:

```
MC(I:c)q = (MI(I)q) == "error" -> "error",MC(c)(MI(I)q)
```

with evident damages to the comprehensibility of the description.

A clever solution for this composition problem is provided by the notion of *continuations*. The basic idea is to define special functions of type `Cont`, such that,

```
Cont   = State -> Ans
Ans    = State | "error"
```

and pass them to semantic functions, which must now have the types:

```
MC : Code  -> Cont -> State -> Ans
MI : Instr -> Cont -> State -> Ans
```

Functions of type `Cont` normally map intermediate values, such as the machine state, to the domain of `Ans` of final execution answer, which becomes the destination domain of all semantic functions. In this example, the final answer is the final state reached by the program execution or simply an error message.

Continuation functions must be constructed to embodied the semantics of the rest of the program to executed next. Thus, the new equations for `MC` become:

```
MC(nil)z q = z q
MC(I:c)z q = MI(I)(MC(c)z)q
```

where `z:Cont` is the continuation function.

These equations tell that the semantics of an empty list of instructions is obtained by passing the current state `q` to

a given program continuation `z`. And the semantics of a sequence of $\mathcal{SC}$ instructions consists of executing the first instruction `I` with a continuation that indicates that the remaining instructions are to be executed next.

In summary, to cope with error handling, functions of type `Cont` must be passed to semantic functions that check for error conditions, so that they can have the choice of either aborting execution with an error message or transmitting their intermediate results to the instructions that follow. For these reasons, functions that map intermediate values to final answers are called continuations.

```
interface Mcontinuations
imports Menvironment(Env); Mstack(Stack); Mstore(Store);
        Mdump(Dump); Mio(Output);
privates
   q: State;
   z: Cont;
publics
   State  = (Env,Stack,Store,Dump,Output)
   Cont   = State -> Ans;
   Ans    = State | Merror;
   Merror = {"Empty-Input"    , "Unbound-Var"   ,
             "Wrong-Opn"      , "Storage-Full"  ,
             "Stack-Underflow", "Misplaced-ret" ,
             "Stack-Overflow" , "Non-Return"    ,
             "Non-Bool-Value" , "Non-Num-Value" ,
             "Non-Rv-Value"   , "Non-Sv-Var"    ,
             "Non-Loc-Var"    , "Non-Fun-Value" ,
             "Non-Proc-Value" , "Misplaced-end" ,
             "No-File"        , "Invalid-Input"};
-- operation
   z0 : Cont;
end
```

The sole operation defined in this module unit is the initial continuation `z0`, which is the identity function, because after the execution of the entire program nothing is left to be done.

```
1  module Mcontinuations
2  functions
3     z0 q = q;
4  end
```

2.2 Program execution

The execution of a program in the computer $\mathcal{SC}$ starts by the application of the function `main` that is defined in the module `Machine` and described in this section.

```
1  interface Machine
2  imports Compiler(KP,Pro); Mcode(Mc);
3          Minstructions(Code);
4          Mcontinuations(Ans,State,Cont,Merror,z0);
5          Menv(Env,env0); Mstore(Store,s0,save);
6          Mstack(Stack,stack0); Mdump(Dump,dump0);
7          Mio(Output,o0,Input,Output);
8  privates
9     P      : Pro -> Input -> (Output | Merror);
10    out    : Ans -> (Output + Merror);
11    z      : Cont; q: State; s: Store;
12    source : File;   -- source program file descriptor
13    i      : File;   -- input file descriptor
14    arg    : Q;      -- argument of the main function
15    infile : Loc;    -- location of the infile descriptor
16 publics
17    main : Q* -> Ans;
18 end
```

The source code file, which contains the Small program to be compiled, and its input file have their names collected from

the command line of the execution environment and passed as a list of quotations to the function `main`.

In order to be executed, source programs in the language Small are translated into the AST format by the built-in function `compile` (*§1.8.2, page 47*), which is generated from the syntax specification of Small.

Then, the AST code is compiled via the function `KP` that is defined in the module `Compiler` (*§3.2, page 88*). The $\mathcal{SC}$ code produced by `KP` is then executed by function `MC` defined in module `Mcode` (*§2.3, page 77*).

```
1  module Machine
2  functions
3     main(arg*) = (pro!=? & i!=?) => P(pro)(i),"No-File"
4     where source = open(getarg("-f",arg*))
5     and   pro    = (source!=?)=>compile(source),?
6     and   i      = open(getarg("-i",arg*));
7
8     P(pro)(i) = out(MC(code)z0 q0)
9     where code   = KP(pro)
10    and   infile = new s0
11    and   env1   = bind("input_file",infile) env0
12    and   s1     = save(infile,i,s0)
13    and   q0     = (env1,stack0,s1,dump0,o0);
14
15    out(ans) = (ans is Merror) => ans, State(ans)(5);
16 end
```

2.3 Machine instructions

The meaning of a sequence of $\mathcal{SC}$ instructions is defined by semantic functions `MC` or `ME`. The meaning of each individual instruction is given by a proper definition of function `MI`.

2.3.1 Flow of execution

Module `Mcode` defines the flow of execution of the machine code. Function `MC` executes an ordinary sequence of code, and `ME`, a sequence of code that leaves a value on the top of the machine stack.

```
interface Mcode
imports Minstructions(MI,Code,Ecode,Instr);
        Mcontinuations(Cont,State,Ans);
publics
   MC : Code  -> Cont -> State -> Ans;
   ME : Ecode -> Cont -> State -> Ans;
   I  : Instr;     -- machine instruction
   c  : Code;      -- machine code for commands
   t  : Ecode;     -- machine code for expressions
   z  : Cont;      -- machine continuations
   q  : State;     -- machine execution states
end
```

The definition of the above operations is detailed in the following module unit.

```
module Mcode
functions
   MC(nil)z q = z q;
   MC(I:c)z q = MI(I)(MC(c)z)q;

   ME(nil)z q = z q;
   ME(I:t)z q = MI(I)(ME(t)z)q;
end
```

2.3.2 Instruction set

Module `Minstruction` describes the meaning of each machine instruction.

```
interface Minstructions
imports
     Mcontinuations(Cont,State,Ans,Merror);
     Menvironment(Var,Bv,Env,Dv,bind,search,Rv,Proc,Fun);
     Mstack(Stack,Ev,push,top,pop,isBool,areBool,areBv);
     Mstore(Store,Loc,Sv,new,free,get,save);
     Mdump(Dump,push,pop);
     Mio(Input,read,Output,put,readfile,eof);
privates
   a   : Loc;      -- memory addresses
   b   : Bv;       -- basic values
   c   : Code;     -- machine intruction sequence
   d   : Dv;       -- machine denotation values
   f   : Fun;      -- function value
   i   : Input;    -- input file
   o   : Output;   -- output file contents
   p   : Proc;     -- procedure values
   q   : State;    -- machine state
   s   : Store;    -- machine store
   t   : Ecode;    -- machine code for expression
   v   : Ev;       -- expressible values
   z   : Cont;     -- machine continuation
publics
   Instr  = ["halt"]  | ["loadv" Bv]    | ["loadt"]
          | ["loadf"] | ["store"]       | ["load" Var]
          | ["read"]  | ["output"]      | ["deref"]
          | ["add"]   | ["minus"]       | ["mult"]
          | ["div"]   | ["and"]         | ["or"]
          | ["not"]   | ["begin"]       | ["end"]
          | ["alloc"] | ["bind" Var]    | ["fcall"]
          | ["pcall"] | ["mkproc" Code] | ["ret"]
          | ["mkfun" Var Code] | ["cond" Code "," Code]
          | ["loop" Ecode "," Code];
   Code   = Instr*; Ecode  = Instr*;
   MI     : Instr -> Cont -> State -> Ans;
end
```

Function MI, which is implemented in module unit named Minstructions, defines the meaning of each $\mathcal{SC}$ instruction by first checking its preconditions, and either produces an appropriate error message or computes the effect of the instruction and passes it to the continuation function provided as parameter.

Therefore, the function MI shows the effect that each instruction would produce in the final answer of a program.

```
1  module Minstructions
2  functions
3     MI["read"](z)(env,stack,s,dump,o) =
4                      (n == ?) => "Invalid-Input", z(q)
5     where a      = search("input_file") env
6     and   (n,s1) = readint(a,s)
7     and   stack1 = push(n)stack
8     and   q      = (env,stack1,s1,dump,o);
9
10    MI["output"](z)(env,stack,s,dump,o) =
11          (empty stack)      => "Stack-Underflow",
12          !(top stack is Rv) => "Non-Rv-Value",z(q)
13    where stack1 = pop stack
14    and   o1     = put(o,top stack)
15    and   q      = (env,stack1,s,dump,o1);
16
17    MI["store"](z)(env,stack,s,dump,o)=
18          (tooShort stack) => "Stack-Underflow",
19          !(a is Loc) => "Non-Loc-Value",
20          !(v is Sv)  => "Non-Sv-Value",z(q),
21    where v      = top stack
22    and   a      = top (pop stack)
23    and   stack1 = pop(pop stack)
24    and   s1     = save(a,v,s)
25    and   q      = (env,stack1,s1,dump,o);
26
```

```
27    MI["loadv" b](z)(env,stack,s,dump,o) = z(q)
28    where stack1 = push(b)stack
29    and   q      = (env,stack1,s,dump,o);
30
31    MI["loadt"](z)(env,stack,s,dump,o) = z(q)
32    where stack1 = push(true)stack
33    and   q      = (env,stack1,s,dump,o);
34
35    MI["loadf"](z)(env,stack,s,dump,o) = z(q)
36    where stack1 = push(false)stack
37    and   q      = (env,stack1,s,dump,o);
38
39    MI["load" var](z)(env,stack,s,dump,o) =
40         (d = "unbound") => "Unbound-Var", z(q)
41    where d      = search(var)(env)
42    and   stack1 = push(d)stack
43    and   q      = (env,stack1,s,dump,o);
44
45    MI["deref"](z)(env,stack,s,dump,o) =
46         (empty stack) => "Stack-Underflow",
47         (a is Loc) => (v = "unused") => "Uninit-Loc",
48         !(v is Rv) => "Non-Rv-Value",z(q),
49         (a is Rv)  => z(env,stack,s,dump,o),
50                       "Non-Rv-Value"
51    where a      = top stack; v = get(a,s)
52    and   stack1 = push(v)(pop stack)
53    and   q      = (env,stack1,s,dump,o);
54
55    MI["not"](z)(env,stack,s,dump,o) =
56          (empty stack)       => "Stack-Underflow",
57          !(isBool(top stack))=> "Wrong-Opn" , z(q)
58    where stack1 = push(!top stack)(pop stack)
59    and   q      = (env,stack1,s,dump,o);
60
61    MI["add"]z(env,stack,s,dump,o) =
62          (tooShort stack) => "Stack-Underflow",
```

```
63          (!areBv(v1,v2)   => "Wrong-Opn" , z(q)
64      where v1     = top stack; v2 = top(pop stack)
65      and   stack1 = push(v1 + v2)(pop(pop stack))
66      and   q      = (env,stack1,s,dump,o);
67
68      MI["minus"]z (env,stack,s,dump,o) =
69          (tooShort stack) => "Stack-Underflow",
70          (!areBv(v1,v2)   => "Wrong-Opn", z(q)
71      where v1     = top stack
72      and   v2     = top(pop stack)
73      and   stack1 = push(v1 - v2)(pop(pop stack))
74      and   q      = (env,stack1,s,dump,o);
75
76      MI["mult"]z (env,stack,s,dump,o) =
77          (tooShort stack) => "Stack-Underflow",
78          (!areBv(v1,v2)   => "Wrong-Opn", z(q)
79      where v1     = top stack
80      and   v2     = top(pop stack)
81      and   stack1 = push(v1 * v2)(pop(pop stack))
82      and   q      = (env,stack1,s,sump,o);
83
84      MI["div"]z (env,stack,s,dump,o) =
85          (tooShort stack) => "Stack-Underflow",
86          (!areBv(v1,v2)   => "Wrong-Opn", z(q)
87      where v1     = top stack
88      and   v2     = top(pop stack)
89      and   stack1 = push(v1 / v2)(pop(pop stack))
90      and   q      = (env,stack1,s,dump,o);
91
92      MI["and"]z (env,stack,s,dump,o) =
93          (tooShort stack) => "Stack-Underflow",
94          (!areBool(v1,v2) => "Wrong-Opn", z(q)
95      where v1     = top stack
96      and   v2     = top(pop stack)
97      and   stack1 = push(v1 & v2)(pop(pop stack))
98      and   q      = (env,stack1,s,dump,o);
```

```
99
100   MI["or"]z (env,stack,s,dump,o) =
101         (tooShort stack) => "Stack-Underflow",
102         (!areBool(v1,v2) => "Wrong-Opn", z(q)
103   where v1     = top stack
104   and   v2     = top(pop stack)
105   and   stack1 = push(v1 | v2)(pop(pop stack))
106   and   q      = (env,stack1,s,dump,o);
107
108   MI["cond" c1 "," c2](z)(env,stack,s,dump,o) =
109         (empty stack)       => "Stack-Underflow",
110         !(head(stack) is T) => "Non-Bool-Value",
111         (head stack)        => MC(c1)(z)(q),
112                                MC(c2)(z)(q)
113   where stack1 = pop(stack)
114   and   q      = (env,stack1,s,dump,o);
115
116   MI["loop" t "," c](z)(q) = ME(t)(z1)(q)
117   where z1 q1 =
118         (empty stack)     => "Stack-Underflow",
119         !(top stack is T) => "Non-Bool-Value",
120         (top stack)       => MC(c)(z2)q1,z(q1)
121   and (env,stack,s,dump,o) = q1
122   and z2 q2 = MI["loop" t "," c](z)(q2);
123
124   MI["begin"](z)(env,stack,s,dump,o) = z(q)
125   where q = (push(nil)env,stack,s,dump,o);
126
127   MI["end"](z)(env,stack,s,dump,o) =
128               (empty env)=>"Misplaced-end",z q
129   where a* = locations(top env)
130   and   s1 = free(a*,s)
131   and   q  = (pop env,stack,s1,dump,o);
132
133   MI["bind" var](z)(env,stack,s,dump,o) =
134         (empty stack) => "Stack-Underflow",z(q)
```

```
135     where env1 = bind(var,top stack)env
136     and   q    = (env1,pop stack,s,dump,o);
137
138     MI["alloc"](z)(env,stack,s,dump,o) =
139           (empty stack) => "Stack-Underflow",
140           !(a is Loc)   => "Storage-Full", z(q)
141     where a      = new(s)
142     and   stack1 = push(a)(pop stack)
143     and   s1     = save(a,head stack)s
144     and   q      = (env,stack1,s1,dump,o);
145
146     MI["mkproc" var c](z)(env,stack,s,dump,o) = z(q)
147        where  q = (bind(var,(c,env))env,stack,s,dump,o);
148
149     MI["mkfun" var t](z)(env,stack,s,dump,o) = z(q)
150     where q = (bind(var,(t,env))env,stack,s,dump,o);
151
152     MI["pcall"](z)(env,stack,s,dump,o) =
153           (tooShort stack) => "Stack-Underflow",
154           !(p is Proc)     => "Non-Proc-Value",
155                               MC(c)(z)(q)
156     where p = top stack; (c,env1) = p
157     and   q = (push(nil)env1,stack,s,push(env)dump,o);
158
159     MI["fcall"](z)(env,stack,s,dump,o) =
160           (tooShort stack) => "Stack-Underflow",
161           !(f is Fun)      => "Non-Fun-Value",
162                               ME(t)(z)(q)
163     where f = top stack; (t,env1) = f
164     and   q = (push(nil)env1,stack,s,push(env)dump,o);
165
166     MI["ret"](z)(env,stack,s,dump,o) =
167                  (empty dump) => "Misplaced-ret",z(q)
168     where a* = locations(env)
169     and   s1 = free(a*,s)
170     and   q  = (top dump,stack,s1,pop dump,o);
```

```
171
172     MI["halt"](z)(env,stack,s,dump,o) =  q
173     where q = (env,stack,s,dump,put(o,"stop"));
174 end
```

2.4 Concluding remarks

Denotational semantics is powerful enough to permit a description of the arquitecture of a computer in a notation that most programmer can read and understand. The semantic model can be as detailed as desired, and the definition can be organized in a modular fashion.

*****

Chapter 3

The Specification of a Compiler

Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.[1]
Antoine de Saint-Exupéry (1900-1944)

Denotational semantics may also be used to precisely describe the specification of a compiler for programming languages. In order to illustrate this capability, this chapter presents the formal definition of a translator from Small programs into $\mathcal{SC}$ code. This translator may be viewed as the a formal definition of the Small language. Another definition of this language is the denotational semantics specification presented in Chapter 6.

3.1 Machine instructions

In order to make this chapter self-contained, relevant parts of the interface of module `MInstructions` (*§2.3.2, page 78*), defined in Chapter 2, are repeated below.

[1] It seems that perfection is attained not when there is nothing more to add, but when there is no longer anything to remove.

Please, refer to Chapter 2 for the precise definition of the meaning of each machine instruction, which is given by function MI, the architecture of the computer $\mathcal{SC}$, and the definition of the machine continuations and execution environments.

```
interface Minstructions
imports Mcontinuations(Cont,State,Ans);
        Menvironment(Var,Bv);
publics
   Instr  = ["halt"]  | ["loadv" Bv]    | ["loadt"]
          | ["loadf"] | ["store"]       | ["load" Var]
          | ["read"]  | ["output"]      | ["deref"]
          | ["add"]   | ["minus"]       | ["mult"]
          | ["div"]   | ["and"]         | ["or"]
          | ["not"]   | ["begin"]       | ["end"]
          | ["alloc"] | ["bind" Var]    | ["fcall"]
          | ["pcall"] | ["mkproc" Code] | ["ret"]
          | ["mkfun" Var Code]
          | ["cond" Code "," Code]
          | ["loop" Ecode "," Code];
   Code   = Instr*;
   Ecode  = Instr*;
   MI     : Instr -> Cont -> State -> Ans;
end
```

3.2 The compiler specification

The specification of the compiler for Small programs is presented in two complementary parts.

Initially the concrete syntax of Small programs is defined so programs in this language can be properly parsed and translated to the AST format. This is accomplished by the module called Parser.

In the second part, the translation of AST nodes into $\mathcal{SC}$ code sequences is presented in the interface and definition modules named `Compiler`.

3.2.1 Concrete and abstract syntax

```
1  interface Parser
2  publics
3     Pro, Dec, Cmd, Exp, Op : Nonterminal;
4     id  : Token;
5     num : Token;
6  end
7  module Parser
8  lexis
9     id     ::= letter+ => return (id,letter+) ;
10    letter === "A" .. "Z" | "a" .. "z";
11    num    ::= digit+  => return (num,digit+);
12    digit  === "0" .. "9";
13
14 syntax
15    pro:Pro   ::= "program" cmds;
16    cmd:Cmd   ::= exp ":=" exp
17                | exp "(" exp ")"
18                | "while" exp "do" cmds "end"
19                                  => ["while" exp cmds]
20                | "if" expr "then" cmds1 "else" cmds2 "end"
21                                  => ["if" exp cmds1 cmds2]
22                | "output" exp
23                | "begin" decs ";" cmds "end"
24                                  => ["begin" decs cmds]
25                ;
26    cmds:Cmd* ::= cmds ";" cmd  => append(cmds,cmd)
27                | cmd           => (cmd)
28                ;
29    exp:Exp   ::= num => [num] | "true" => ["true"]
30                | "false" => ["false"]
```

```
31                  | "read" => ["read"]
32                  | id => [id]
33                  | exp op exp | exp "(" exp ")"
34                  | "not" exp
35                  | "if" exp "then" exp1 "else" exp2
36                                  => ["if" exp exp1 exp2]
37                  ;
38    op:Op      ::= "+"   => ["+"]   | "-"  => ["-"]
39                  | "*"   => ["*"]   | "/"  => ["/"]
40                  | "and" => ["and"] | "or" => ["or"]
41                  ;
42    dec:Dec    ::= "const" id "=" exp
43                  | "var" id "=" exp
44                  | "proc" id "(" id1 ")" cmds "end"
45                                  => ["proc" id id1 cmds]
46                  | "fun" id "(" id1 ")" exp
47                                  => ["fun" id id1 exp]
48                  ;
49    decs:Dec*   ::= decs ";" dec => append(decs,dec)
50                   | dec          => (dec)
51                   ;
52 end
```

3.2.2 Translation rules

Using the parser generate from module `Parser`, the AST of small programs may be generated, so the the function `KP` that is exported by interface module `Compiler` may translate source programs in AST format into sequences of the machine instructions that are defined in module units named `Minstructions`.

Function `KP` uses local functions called `KD`, `KC` and `KE`. The function `KD` elaborates variable declarations, functions `KE` and `KC` serve to compile expressions and commands, respectively.

```
interface Compiler
imports
    Minstructions(Code, Ecode);
    Parser(Pro, Dec, Cmd, Exp, Op, id, num);
privates
   -- Declarations:
   KD : Dec  -> Code;
   KD : Dec* -> Code;
   -- Commands:
   KC : Cmd  -> Code;
   KC : Cmd* -> Code;
   -- Expressions:
   KE : Exp -> Ecode;
   KR : Exp -> Ecode;
   KO : Opr -> ECode;
publics
   Pro : Start;
   KP  : Pro ->  Code;
end
```

The functions KE, KC, KD and KP are defined by cases in the following definition module.

```
definition Compiler
functions
   KE[num] = (["loadv" toN(num)]);
   KE["true"]  = (["loadt"]);
   KE["false"] = (["loadf"]);
   KE["read"] = (["read"]);
   KE[id] = (["load" id]);
   KE["not" exp] = KR(exp) +  (["not"]);
   KE[exp1 op exp2] = KR(exp1) + KR(exp2) + KO(op)
   KE[exp "(" exp1 ")"] =
          KE(exp1) + KE(exp) + (["fcall"]);
   KE["if" exp exp1 exp2] =
          KR(exp) + (["cond" KE(exp1) "," KE(exp2)]);
   KR(exp) = KE(exp) + (["deref"]);
```

```
15
16      KO["+"]   = (["add"]);
17      KO["-"]   = (["minus"]);
18      KO["*"]   = (["mult"]);
19      KO["/"]   = (["div"]);
20      KO["and"] = (["and"]);
21      KO["or"]  = (["or"]);
22
23      KC[exp ":=" exp1] = KE(exp) + KR(exp1) + (["store"]);
24      KC["output" exp] = KR(exp) + (["output"]);
25      KC[exp "(" exp1 ")"] =
26             KE(exp1) + KE(exp) + (["pcall"]);
27      KC["if" exp cmd1 cmd2] =
28             KR(exp) + (["cond" KC(cmd1) "," KC(cmd2)]);
29      KC["while exp cmd*] =
30             (["loop" KR(exp) "," KC(cmd*)]);
31      KC["begin" dec* cmd*] =
32             (["begin"]) + KD(dec*) + KC(cmd*) + (["end"]);
33      KC(cmd:cmd*) = KC(cmd) + KC(cmd*);
34      KC(nil) = nil;
35
36      KD["const" id "=" exp] =
37             KR(exp) + (["bind" value(id)]);
38      KD["var" id "=" exp] = KR(exp) + (["alloc"]) +
39                                       (["bind" value(id)]);
40      KD["fun" id id1 exp]= (["mkfun" value(id) t]);
41      where t = (["bind" value(id)]) + (["bind" value(id1)]) +
42                KE(exp) + (["ret"]);
43      KD["proc" id id1 cmd*] = (["mkproc"  value(id) c]);
44      where c = (["bind" value(id)]) + (["bind" value(id1)]) +
45                KC(cmd*) + (["ret"]);
46      KD(dec:dec*) = KD(dec) + KD(dec*);
47      KD(nil) = nil;
48
49      KP["program" cmd*] = KC(cmd*) + (["halt"]);
50  end
```

3.3 Concluding Remarks

In the next chapters, the emphasis is to provide a standard way to select meaningful domains and appropriate module structure in order to produce descriptions that are both effective and comprehensible by programmers.

*****

Chapter 4

Standard Denotational Semantics

Le bon sens est la chose du monde mieux partagée.[1]
René Descartes (1596-1590)

In denotational semantics, the meaning of a language is given by associating with each construct in the language a corresponding semantic object. A denotational definition consists of the specification of the syntactic and semantic domains together with a collection of mappings that associate syntactic elements with their denotations, as depicted in Fig. 4.1.

The syntactic domain of a language is the domain of all programs in the form of *Abstract Syntax Tree* (AST), whose nodes exhibit the constituents of each construct.

The abstract syntax tree is thus an encoding of the source program in which only the main constituents of each construction are exhibited, and details regarding their concrete syntaxes are left out. Nodes of the AST are defined by production rules of the form $\texttt{A} = r_1 B_1 r_2 B_2 \cdots r_n B_n$, where r_i and

[1] Common sense is the best distributed asset.

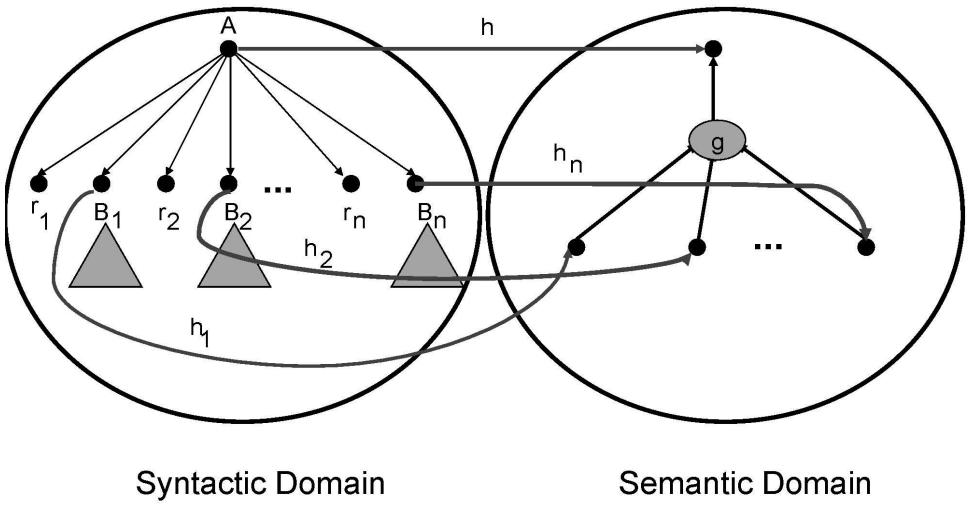

Figure 4.1: The Semantic Model

B_i, for $1 \leq i \leq n$, are terminal symbols and nonterminal symbols, respectively.

The meanings of AST nodes are abstract mathematical objects, which are referred to as denotations. The semantic mappings are defined by means of functions such as h and h_i, for $1 \leq i \leq n$, of Fig 4.1, and the denotation of a construct is defined in terms of the denotations of its constituents.

In order to define the structure of the semantic mappings, suppose that:

- the abstract production rule $p : A \rightarrow \ r_1 B_1 r_2 B_2 \cdots r_n B_n$ be the textual representation of the AST node depicted in Fig. 4.1;
- T_{B_i}, for $1 \leq i \leq n$, be the domain of AST nodes with root B_i, and $t_{B_i} \in T_{B_i}$;
- Π_p, for each AST production $\mathtt{p}$, is a function that put together AST subtrees to make an AST node according to p;
- $[\![t]\!]$, in which t is a sequence of grammar symbols, be the

so-called quasi-quotation [44] of the syntactic member t;

- S and S_i, for $1 \leq i \leq n$, be domains of denotations;
- g be a function that combines denotations.

Then the following diagram shows the structure of the mappings:

$$\begin{array}{ccccccccc} T_{B_1} & \times & T_{B_2} & \times & \cdots & \times & T_{B_n} & \stackrel{\Pi_p}{\rightarrow} & T_A \\ \downarrow\ h_1 & & \downarrow\ h_2 & & & & \downarrow\ h_n & & \downarrow\ h \\ S_1 & \times & S_2 & \times & \cdots & \times & S_n & \stackrel{g}{\rightarrow} & S \end{array}$$

The functions h and h_i, for $1 \leq i \leq n$, are homomorphisms from *Syntactic Domains* into *Semantic Domains*. The function h is defined as:

$$h(\Pi_p(t_{B_1}, t_{B_2}, \cdots, t_{B_n})) = g(h_1(t_{B_1}), h_2(t_{B_2}), \cdots, h_n(t_{B_n}))$$

or, using the definitions of Π_p and p,

$$h[\![A]\!] = g(h_1[\![B_1]\!], h_2[\![B_2]\!], \cdots, h_n[\![B_n]\!])$$

and, hence,

$$h[\![r_1 B_1 r_2 B_2 \cdots B_n r_n]\!] = g(h_1[\![B_1]\!], h_2[\![B_2]\!], \cdots, h_n[\![B_n]\!])$$

In summary, a denotational definition consists in defining the abstract syntax of the language, the semantic domains and the function h for each construct in the language. And, in theory, the semantics of a construct should only depend on the semantics of its constituents.

4.1 Direct semantics of a simple language

In order to illustrate the application of the denotational semantics model, consider the formal description of a toy language named Simple.

Simple is a very small imperative language whose programs

always read an integer value into a variable, then execute a list of statements and, at the end, print the value of an expression. For instance, the program

```
begin read x do x := succ x ; write suc suc x end
```

reads an integer value into the variable `x`, increments the value stored in `x` and prints the value of the second successor of current value of `x`.

4.1.1 Concrete and abstract syntaxes

The concrete and abstract syntaxes of Simple programs are defined in the following $\mathcal{M}$ module:

```
1  module Simple
2  lexis
3     id:Token  ::= letter+ => return id(letter+);
4     letter    === "A" .. "Z" | "a" .. "z";
5
6  syntax
7     pro :Pro  ::= "begin" "read" id "do" cmds
8                   "write" exp "end" => ["read" id cmds exp]
9                 ;
10    cmd :Cmd  ::= id ":="  exp
11                | "while" exp "do" cmds "end"
12                                  => ["while" exp cmds]
13                | "begin" dec "do" cmds "end"
14                                  => ["begin dec cmds "end"]
15                ;
16    cmds:Cmd* ::= cmds ";" cmd => append(cmds,cmd)
17                | cmd          => (cmd)
18                ;
19    dec :Dec  ::= "var" ids
20                ;
21    ids :Id*  ::= id           => (id)
22                | ids ; id     => append(ids,id)
```

```
23     exp :Exp  ::= "0"          => ["0"]
24                 | id           => [id]
25                 | "suc" exp
26                 ;
27 functions
28   -- definition of function to come soon in the sequel
29 end
```

4.1.2 Informal semantics

The Simple's statements can be an assignment command, a repetition command or a block.

The command list that occurs within a repetition statement is to be executed repeatedly until the command control expression evaluates to a zero value.

Blocks allow the introduction of local scope for variables, which are allocated upon entering the block and freed at block exiting.

Simple expressions can be the integer constant `0`, a variable designated by an identifier or an expression that computes the successor of an integer value.

Identifiers that are declared inside a begin block are automatically initialized with 0 (zero). All other identifiers are initially associated with `?`, and the only possible execution error in a Simple program is the attempt to use identifiers whose value is `?`, in which case, the result of the program is `"error"`.

4.1.3 Semantic domains

The states of a Simple program execution are modeled by the semantic domain `State`, which is the domain of functions

that associate variables with integer values. The evaluation of expressions needs a state to get the current values of the variables, and assignment commands update the current state.

In the initial state `s0`, all variables are *undefined*, that is, `s0(id) = ?`, for all `s:State` and `id:Id`. It is assumed that the memory is as large as needed, so there is no provision for checking memory overflow in the definitions that follow.

4.1.4 Semantic equations

The meaning of language Simple is defined via four semantic functions: `E`, for expressions, `C`, for commands, `D`, for declarations, and `P`, for programs. The types of these functions are defined in the following interface module:

```
1  interface Simple
2  privates
3     s    : State;
4     init : Id* -> State -> State;
5  publics
6     State = Id -> N;
7     E : Exp -> State -> N;
8     C : Cmd -> State -> State;
9     D : Dec -> State -> State;
10    P : Pro -> N -> (N | "error");
11 end
```

These semantic functions map their arguments to the result the associated Simple construction produces. This style of denotational semantics definition is called *direct semantics*.

All Simple programs produce a result that is either an integer value or an error message.

Function `E` takes as arguments an expression and the current state, in which variables are bound to their values, and

produces the integer value of the expression. No collateral effect is expected.

Function `C` takes as arguments a command and the current state, and produces a new state according to the command semantics.

Function `D` takes as arguments a declaration of a list of identifiers and the current state, and adds the association of these identifiers with value 0 (zero) to the current state.

And `P` maps Simple program and an input integer value into the integer value that is to be printed by the program.

In the initial state `s0`, all variables are bound to the undefined value ?.

The definition of these functions is exhibited in the following definition module.

```
1  module Simple
2   -- syntax and lexis parts already presented
3  functions
4     E["0"]s = 0;
5     E[id]s = s(id);
6     E["suc" exp]s = (E(exp)s == ?) => ?, E(exp)s + 1;
7     C[id ":=" exp]s = (E(exp]s == ?) => ?, s{id<-E(exp)s};
8     C["while" exp cmd*]s =
9       (E(exp)s == ?)  => ?, (E(exp)s == 0) => s,
10      (C(cmd*)s == ?) => ?, C["while" exp cmd*](C(cmd*)s);
11    C["begin" dec cmd* "end"] s = C(cmd*)(D(dec)s);
12    C(cmd:cmd*)s = (C(cmd)s == ?) => ?, C(cmd*)(C(cmd)s);
13    C(nil)s = s;
14    D["var" id*] s = init(id*)s;
15    where init(nil)s = s
16    and   init(id:id*)s = init(id*)(s[id:0]);
17    P["read" id cmd* exp]n = (s2!=?)&(v!=?) => v,"error"
18    where s0 = \id.?
19    and   s1 = s0{id<-n}
```

```
20     and   s2 = C(cmd*)(s1)
21     and   v  = E(exp)s1;
22 end
```

4.1.5 A worked example

A step by step execution of a program example might help revealing details of the formal definition of Simple.

Consider the folowing program, hereon called `p`,

```
1 begin
2    read x
3    do x := succ  x ;
4    write suc suc  x
5 end
```

The first step is to parse `p` and translate it into the AST format, which is be represented textually[2] as:

```
["read" x <[x ":=" "suc" x]> "write" ["suc" "suc" x]]
```

The step by step computation of the semantics `m` of this program with input `n` is as follows.

```
m=P<["read" x <[x ":=" "suc" x]>"write"["suc" "suc" x]])n
```

Applying the definition of `P`:

$$\texttt{m = E["suc" "suc" x]}\ \underbrace{(\texttt{C(<[x ":=" "suc" x]>)}(\overbrace{s0[x:n]}^{\texttt{s1}}))}_{\texttt{s2}}$$

Thus,

```
m = E["suc" "suc" x](s2)
```

Applying the definition of `E`:

```
m = E["suc" x]s2 + 1
m = E(x)s2 + 1 + 1
```

[2]The notation [...] denotes an AST node, and <..>, a list.

```
m = s2(x) + 2
```

On the other hand,

```
s2 = C(<[x ":=" "suc" x]>)s1
s2 = C(nil)(C[x ":=" "suc" x]s1)
s2 = C[x ":=" "suc" x]s1
s2 = s1{x <- E["suc" x]s1}
s2 = s1{x <- E(x)s1 + 1}
s2 = s1{x <- s1(x) + 1}
s2 = s1{x <- n + 1}
```

hence,

```
s2(x) = s1{x <- n + 1}(x) = n + 1
```

Given that

```
m = s2(x) + 2
```

then

```
m = n + 1 + 2
```

and

```
P["read" x<[x ":=" "suc" x]> "write"["suc" "suc" x]]n =
                                                    n + 3
```

As expected, the semantics of the program `p` is a function that maps the input integer value `n` into `n + 3`.

4.2 Standard semantics model

Discipline in the process of constructing denotational semantics definitions facilitates language comparison and the formulation of the semantic mappings. In fact, most languages share similar domain structure. For instance, in the imperative paradigm, there are important issues that must be addressed and treated uniformly among different languages, such

as memory allocation, memory sharing, error handling and scope management, which should follow a standardized structure.

4.2.1 Standard environments and stores

From the programmer viewpoint, it would be easier to assume that identifiers can be directly associated with their values as it has been done in the definition of Simple. However, it is quite common the case that more the one identifier share the same memory association, in such a way that changing the value associated with one of them would cause the updating of the value associated with the others, as it happens with C pointers holding the address of the same memory location, and thus creating an aliasing situation.

Thus, it may be convenient to split the binding of identifiers into two standard mappings: environments and stores. Standard environments are members of the domain `Env=Id->Dv`, and thus map identifiers to *denotable values* (`Dv`), among which are the *locations* (`Loc`). The standard stores, which are members of `Store=Loc->Sv`, map locations in `Loc` to *storable values* (`Sv`). The exact nature of the domains `Dv` and `Sv` is language dependent, and these domains form a dimension along which languages can be classified. This scheme facilitates expressing the semantics of identifier aliasing and dynamic allocation of memory space. Throughout the definitions in the sequel, `r`, `r1`, `r2`, and so forth, are elements of `Env`, and `s`, `s1`, `s2`, and so forth, are elements of `Store`.

Similar standardization should be applied to other domains of values and specially to the way errors and jumps are han-

dled. To that end, Michael Gordon [15] has defined *standard denotational semantics* as the semantic definition based on the following conditions:

1. The binding of identifiers that represent variables to values should be split into two parts: a mapping from identifiers to locations and a mapping from locations to values. The first mapping is the *environment* and the latter one, the *store*.
2. The machine state should be transmitted along the program elements via a device called *continuations*.
3. The language dependent domains of values, particularly those that go in the environments, the *denotable values*, and in the stores, the *storable values*, should be clearly defined and have their structures standardized.

4.2.2 Domains of standard values

In the imperative language paradigm, the evaluation of expressions produces values and may cause the collateral effect of updating the memory.

The values produced by expressions are called *expressible values* (`Ev`), and they also form a dimension to classify languages.

Expressible values can be a memory address, which is in the domain `Lv` of *left value*, or other types of values, which are said to be in the domain `Rv` of *right values*. What *left values* and *right values* are is also a language dependent issue and forms a classification dimension. A subdomain of the expressible values is the domain `Ov` of *outputable value*, which

is the domain of elements that can be sent to the program output file.

Environments and stores deal with *denotable* and *storable* values, as discussed in previous sections.

In summary, in addition to *Environments*, *Stores* and *Continuations*, the following language dependent domains should be defined in a standard denotational semantics description:

- *Storable Values* `Sv`: the domain of values that can be associated with locations in the store. Throughout the definitions, $v, v_1, v_2, \cdots \in$ `Sv`.
- *Denotable Values* `Dv`: the domain of values that can be associated with identifiers in the environment. Throughout the definitions, $d, d_1, d_2, \cdots \in$ `Dv`.
- *Expressible Values* `Ev`: the domain of values that can be produced by evaluating expressions. Throughtout the definitions, $e, e_1, e_2, \cdots \in$ `Ev`.
- *Left Values* `Lv`: the domain of values that can appear on the left-hand side of an assignment statement, usually they are elements of `Loc`.
- *Right Values* `Rv`: the domain of values that can be used on the right-hand side of an assignment statement.
- *Outputable Values* `Ov`: the domain of values that can be sent to the program output file.

4.2.3 The notion of continuations

In standard semantics, commands are supposed to alter the store, and not the environments. For instance, the assignment statement `[id ":=" exp]` has the effect of updating the memory location associated with the identifier `id`. Nor-

mally, the updated store must be transmitted to other parts of the program.

Declarations usually have effect on the environment and on the store, e.g., the declaration `["var" id "=" exp]` has the effect of allocating a new cell in the memory, associating, in the environment, the location of this cell with the identifier `id`, and updating this memory location with the value of `exp`. The updated environment and the updated store must be passed to declarations or commands that follow the declaration just processed.

In general, in the imperative language paradigm, the evaluation of expressions produces values and may cause collateral effects by updating the memory. The values produced by an expression and the possibly updated store must be transmitted to the program text that follows the expression.

Information, like environments and stores, is easily transmitted along program execution as parameters of the semantic functions, which are composed in order to produce the final result of the program. These parameters define the context in which the semantics of each construct is to be evaluated.

This device for transmitting context via parameters in function composition works fine and seems quite natural. However, it does not make good pair with error handling and with control transfer. Error handling demands an effort to cope with error propagation throughout remaining function compositions, and control transfer requires the discard of the normal function composition and the acquisition of a new one to continue execution.

Error propagation and the process of discarding certain

function compositions may be a nuisance to cope with, although yet feasible. A better solution is offered by Wadsworth & Strachey's *Continuations* [54]. The idea is to add a new parameter, the continuation, to all functions, so that instead of returning the value they compute, these functions pass the computed value to the continuation that has been passed as parameter. In this way, functions with continuation parameters have the option of passing their intermediate values to the given continuation or sending them to other destinations.

For instance, suppose that functions `f`, `g` and `h` be defined as follows.

```
1  declarations
2     f : N -> N
3     g : N -> (N | "error")
4     h : N -> (N | "error")
5     v : N | "error"
6  functions
7     f n = n
8     g n = (n == 0) => "error", 100/n
9     h n = (g(n) == "error") => "error", f(N(g(n)))
10    v1 = h(0)
11    v2 = h(100)
```

Note that function `h`, defined in line 9, is polluted, for it must be aware of the fact that `g(n)` may return `"error"`, and so propagates this value accordingly, and must be careful in casting to `N` the value returned by `g(n)`, before passing it to `f`. Thus, in the above example, the value of `v1`, defined in line 10, is `"error"`, and that of `v2` in line 11 is `1`.

Using continuations, one could rewrite, while keeping the intended meaning, the above functions to include a new parameter of type `K`, which is the continuation, so that these

functions may choose to return the value they compute or to pass it to the continuation, as it is illustrated in the following program fragment.

```
1  privates
2     f : K -> N -> N;
3     g : K -> N -> N;
4     h : K -> N -> N;
5     K = N -> N;
6  ...
7  functions
8     f k n = k(n);
9     g k n = (a==0) => "error", k(100/n);
10    h k n = g(\n.f k n) n;
11    v1 = h(\n.n)(0);
12    v2 = h(\n.n)(100);
```

Notice that the pollution caused by error handling has been removed from the new definition of `h`, which now simply invokes `g` with the appropriate continuation, and the *pollution* has been confined in the definition of `g` in line 9, which is the proper place for it. The evaluation of variables `v1` and `v2`, in lines 11 e 12, still produces `"error"` and `1`, respectively.

The definition of labels and goto statements of imperative languages follows similar pattern: all semantic functions must carry a continuation parameter that define the normal control flow and that could be discarded in favor other continuations when necessary.

4.2.4 Standard continuations

The continuation of a construct must model what follows it. The continuation takes the intermediate results produced by the construct and uses it to produce the result of executing

the *rest of the program*. Hence, the continuation is a mapping from intermediate results to the final result of the program. The standard domain of *final answers* of a program is `Ans`. What the members of `Ans` are depends on the language being defined.

Of course, the intermediate results depend on the construct with which the continuation is associated. For instance, declarations produce new environments and update the store, commands update the store, and expressions produce expressible values and may change the store. Therefore, the standard continuations for these constructs have the following types:

- Command Continuation (`Cc`):
  `Cc = Store → Ans`
  `z, z1, z2, ··· ∈ Cc`
- Expression Continuations (`Ec`):
  `Ec = Ev → Store → Ans`
  `Ec = Ev → Cc`
  `k, k1, k2,··· ∈ Ec`
- Declaration Continuations (`Dc`):
  `Dc = Env → Store → Ans`
  `Dc = Env → Cc`
  `u, u1, u2,··· ∈ Dc`

Continuation functions in the domains above should be used as parameter of the semantic functions `E, C` and `D` as follows:

- Semantics of Expressions (`Exp`):
  `E : Exp -> Env -> Ec -> Store -> Ans`

- Semantics of Commands (`Com`):
  `C : Com -> Env -> Cc -> Store -> Ans`
- Semantics of Declarations (`Dec`):
  `D : Dec -> Env -> Dc -> Store -> Ans`

The selection of the above parameter ordering is due to the frequency of modification of the structures: more stable arguments should come first. In this way, for instance, the expression `E(exp)r` can be used with different continuations and stores, and the expression `E(exp)r k` may be used with different stores, while its intermediate results are sent to the same continuation `k`. Also note that the destination domain `Ans` of the semantic functions must be the same as the destination of the associated continuation functions, for the latter functions are responsible to transmit intermediate results to the rest of the program in order to produce final answers.

Typically, the bodies of standard semantic functions follow the structure:

- `E(exp)r k s = (any error) => "error", k e s1`
- `C(com)r z s = (any error) => "error", z s1`
- `D(dec)r u s = (any error) => "error", u r1 s1`

where `e`, `r1` and `s1` are intermediate results, and "error", a member of domain of final answers `Ans`.

4.3 Continuation semantics of Simple

A new semantic definition of Simple is presented in order to illustrate the use of continuation. The direct and the continuation semantic definitions of Simple convey the same meaning,

and the concrete and abstract syntaxes of Simple are the same defined in module `Simple` (*§4.1.1, page 98*).

The new semantic domains and equations are detailed in the sequel.

4.3.1 Semantic domains

The states of a Simple program execution are modeled by the semantic domains `Env` and `Store`, which are domains of functions that associate variables with locations and locations with integer values, respectively. It is assumed that the store is as large as needed, so there is no need to check for memory overflow in the definition that follows.

Expressions need the state to get the current value of program variables, and assignment commands update the current state accordingly.

In the initial state, all variables and locations are *undefined*, that is, `r0(id)=?` and `s0(a)=?`, for `r:Env`, `s:State`, `a:Loc` and `id:Id`.

Declared variable must be associated with a free memory cell, which is then initialized with the integer value 0.

The semantics of Simple is built upon four semantic functions: `E`, for expressions, `C`, for commands, `D`, for declarations, and `P`, for programs, whose types are defined in the following interface module.

```
1 interface Simple
2 privates
3    r: Env   = Id  -> Dv;
4    s: Store = Loc -> Sv;
5    k: Ec    = Ev -> Store -> Ans;
6    z: Cc    = Store -> Ans;
```

```
7     u: Dc    = Env -> Store -> Ans;
8     Ans      = N | "error";
9     a: Loc   = N;
10    Dv       = Loc;
11    Sv       = N;
12    e: Ev    = N;
13    new : Store -> Loc;
14    getFree : (Store, Loc) -> Loc;
15    arg, filename : Q;
16    source, input : File;
17    dc    : Id* -> Env -> Dc -> Store -> Ans;
18    E     : Exp -> Env -> Ec -> Store -> Ans;
19    C     : Com -> Env -> Cc -> Store -> Ans;
20    D     : Dec -> Env -> Dc -> Store -> Ans;
21    P     : Pro -> N -> Ans;
22 publics
23    main : Q* -> Ans;
24 end
```

4.3.2 Semantic equations

Function `E` takes as arguments an expression and the current state, in which variables are bound, and produces the integer value of the expression.

Function `C` takes as arguments a command and the current state and updates the current state according to the given command.

Function `D` takes as arguments identifier declarations and the current state, allocates these identifiers in the current store, and updates the current state according to the declared identifiers.

Function `P` maps Simple program and an integer value into the integer value to be printed by the program, or returns an

error message.

```
1  module Simple
2  syntax
3     ...
4  lexis
5     ...
6  functions
7     new s  = getFree(s,0)
8     where getFree(s,a) = (s a==?) => a, getFree(s,a + 1);
9
10    E["0"]r k s = k 0 s;
11
12    E[id]r k s  = (r(id) == ?) => "error",
13                  (s r(id)) == ?) => "error",
14                                     k (s r(id)) s;
15    E["suc" exp]r k s = E(exp)r k1 s
16    where k1 e s =  k (e + 1) s;
17
18    C[id ":=" exp]r z s = (r(id) == ?) => "error",
19                                       E(exp)r k s
20    where k e s = z(s[r(id):e]);
21
22    C["while" exp cmd*]r z s = E(exp)r k s
23    where k e s = (e==0) => z s, C(cmd*)r z1 s
24    and   z1 s  = C["while" exp cmd*]r z s;
25
26    C["begin" dec cmd* "end"]r z s = D(dec)r u s
27    where u r s = C(cmd*)r z s;
28
29    C(nil)r z s = z s;
30    C(cmd:cmd*)r z s = C(cmd)r z1 s
31    where z1 s = C(cmd*)r z s;
32
33    D["var" id*]r u s =  dc(id*)r u s
34    where dc(nil)r u s = u r s
35    and   dc(id:id*)r u s = dc(id*)(r{id<-a}) u (s{a<-0})
```

```
36    and   a  = new s;
37
38    P["read" id cmd* exp]n = C(cmd*)r z s
39    where s0 a  = ?
40    and   a     = new s0
41    and   s     = s0{a<-n}
42    and   r0 id = ?
43    and   r     = r0{id<-a}
44    and   k     = \e s.e
45    and   z s   = E(exp)r k s;
46
47    main(arg*) =  P(pro)(input)
48    where filename1 = getarg("-f", arg*)
49    and   source    = open(filename1)
50    and   filename2 = getarg("-i", arg*)
51    and   input     = open(filename2)
52    and   prog      = compile(source);
53
54 end
```

Observe that the continuation mechanism allows that error conditions be checked only at the places (lines 12, 13 and 18) from where they are originated, preventing these concerns to be scattered throughout other equations.

***

Chapter 5

Retractile Continuations

Make everything as simple as possible, but not simpler.
Albert Einstein (1879-1955)

The notion of *retractile continuation* is the basis of a technique for accommodating, in a same denotational semantics definition, both the continuation and direct semantics styles.

The kind of semantics used in most denotational definitions is continuation semantics [17, 26, 42, 54, 57]. The continuation approach is generally chosen for its convenient way of dealing with error conditions and jumps. However, the sequential nature of continuations presents some practical difficulties. Take a list of mutually recursive equations as an example. In order to manufacture each one of the equations in the list, it is required that all others equations have already been defined so that the types of free variables occuring in the body in each equation are available in a common environment.

In the direct approach to semantics, a situation like this is easily modelled by defining a system of mutually recursive equations, each defining a partial environment that results from the evaluation of the associated equation. Each equation is then defined in the same environment, which should

be recursively defined in terms of those partial environments produced by the individual equations.

On the other hand, in the continuation approach, each semantic function is supposed to pass the intermediate value it produces, for example an environment, to the rest of the definition, i.e., to the normal continuation, which generally maps intermediate results to final answers. This means that intermediate values in the continuation approach are not returned by the semantic functions, and, therefore, they are not available locally to construct the desired system of recursive equations.

The conclusion is that the continuation approach makes it easier to cope with error conditions, whereas the direct approach facilitates the modeling of non-sequential evaluations. Hence, it would be very convenient to have a mechanism that allows a harmonious coexistence of both semantics styles, so that the *right* kind of semantics may be used where it works best.

5.1 Conciliation of semantics styles

The proposal is that the default semantics' style be continuation semantics and, in order to define systems of mutually recursive equations, for example, there must be means to perform a temporary switch to direct semantics style, so that each equation may return its intermediate result instead of sending it to the continuation.

The switching mechanism can be implemented by passing to the semantic functions involved in the process a conve-

niently manufactured continuation in place of the normal continuation. This new continuation has the purpose of forcing intermediate values to be returned by the function.

For example, let f be a semantic function of type

```
f : X -> C -> Ans
```

where `C = V->Ans` is a domain of continuations, `Ans` is that of final answers, and `V` is the domain of intermediate values that `f` passes to its continuation.

The first step is to make `f` return a value, say `v:V`, rather than the final answer. The idea is to make arrangements to force `f` to return the intermediate value `v` it produces instead of passing it to the normal continuation. And this must be achieved without invalidating the application of `f` in different contexts, which certainly assume that `f` still deals properly and systematically with its continuation.

The solution herein proposed consists of passing to `f` a special type of continuation, named **retractile continuation**, with the purpose of hoisting the intermediate value produced by `f` to the calling point. A retractile continuation works like a boomerang, which when correctly thrown (passed as parameter) glides back to a point near the thrower (the calling point).

A retractile continuation `w` has type `w:V->A`, where `V` is the domain of intermediate values accepted by normal continuations, and `A` is a new domain of final answers, which is the original domain `Ans` extended to incorporate the domain `V`. Retractil continuations must be always defined as an identity function such as `w=\v.v`.

Accordingly, the type of the semantic function `f` must be

changed from `f:X->C->Ans` to `f:X->C->A`, where `A=Ans|V` is the extended domain of final answers.

Hence, the value of `a:A` in `a = f(x)(w)` is the value which `f` passes to its continuation. It is an intermediate value, if `f` succeeds, otherwise it is some other value in the domain of final answers, such as an error message, for example.

Later, when comes the time to switch back to continuation semantics, the intermediate value `a` produced locally can be explicitly passed to the normal continuation as it would have been done under normal conditions.

5.2 An example

In order to illustrate the application of the switching mechanism just proposed, the denotational semantics of a toy language, herein called R, is presented in the sequel using a combination of continuation and direct semantics.

The syntactic structure of R is defined by the following grammar module:

```
1  module R
2  lexis ... id: Token; ...
3  syntax
4     pro:Prog ::= "write" exp;
5     exp:Exp  ::= defs "in" exp
6                | id "(" exp ")"  => [id exp]
7                | "\" id "." exp
8                | id   => [id]
9                | "0"  => ["0"];
10    defs:Def+ ::= defs def   => append(defs,def)
11                 | def       => (def);
12    def:Def   ::= "let" id "=" exp;
13 end
```

A program in the language R is simply the constant `0`, an identifier `id`, a function application, a λ-abstraction or a sequence of let-clauses ended by an in-expression.

The let-clauses serve to bind identifiers to R-expressions. Bound identifiers can be freely used in any expression of the let-clauses and in the corresponding in-expression. In order to capture this semantics, all let-clauses shall be evaluated in an environment containing all bindings they introduce, possibly in a mutually recursive fashion. The direct semantic approach is more suitable to model the meaning of this feature, and for the remaining constructs, continuation semantics is more appropriate, so errors can be easily handled.

To formulate the continuation semantics of R, the following semantic domains are defined:

```
1  interface R
2  privates
3     Ans = N | "error";              -- Final answers
4     A   = Ans | Env;                -- New final answers
5     Env = Id -> Dv;                 -- Environments
6     Dv  = N | G | "unbound";        -- Denotable values
7     Dc  = Env -> A;                 -- Decl continuations
8     Ec  = Dv -> A;                  -- Exp continuations
9     a: A; r: Env; v:Dv; ok: T; k: Ec;
10    u: Dc;  -- normal decl continuation
11    w: Dc;  -- retractile def continuation
12    G   = Dv   -> Ec  -> A;         -- One arg function
13    E   : Exp  -> Env -> Ec -> A;   -- semantics of exp
14    L   : Def* -> Env -> Dc -> A;   -- semantics of def*
15    F   : Def  -> Env -> Dc -> A;   -- semantics of def
16    P   : Prog -> Ans;              -- semantics of prog
17 end
```

The domain `Env` of environments is part of the domain `A`

of extended final answers in order to implement the switching mechanism proposed in the previous section.

The continuation semantics definition of R is detailed in the following definition module:

```
module R
syntax ...
functions
   E["0"]r k = k(0);

   E[id]r k = (r(id) == "unbound") => "error", k(r(id));

   E["\" id "." exp]r k = k(g)
   where g = \v k1.E(exp)(r + {id<-v})k1);

   E[id exp]r k = (r(id) is G) => E(exp)r k1, "error"
   where k1 v = r(id)v k;

   E[def+ "in" exp]r k = L(def+)r u
   where u = \r1. E(exp)r1 k;

   L(nil)r u = u(r);
   L(def:def*)r u = ok => u(r1),"error"
   where w  = \r.r
   and   a1 = F(def)r1 w
   and   a2 = L(def*)r1 w
   and   ok = (a1 is Env) & (a2 is Env)
   and   r1 = ok => r{a1}{a2}, r;

   F["let" id "=" exp]r u = u(r1)
   where r1 = r[id:(\k. E(exp)r1 k)];

   P["write" exp] = E(exp)r0 k
   where r0 id = "unbound"
   and    k v  = (v is N) => N(v), "error";
end
```

The initial environment is defined as `r0(id) = "unbound"`, in line 29, to indicate that initially all identifiers in R are unbound. An error should be indicated whenever, in the evaluation of an expression, a reference to identifier not bound by any of the let-clauses is encountered. Errors should also be indicated when non-functional values are applied to any values. Since continuation semantics works best in this situation, it has been chosen to define the language R.

The **retractile continuation** `w:Dc` needed by the switching mechanism is defined, in line 19, as `w = \r.r`, for `r:Env`.

Semantic function `E` (lines 4 to 14) defines the semantics of R-expressions.

Function `L` (lines 17 to 18) define the semantics of a list of let-clauses. The meaning of non-empty lists of let-clauses is easily defined in terms of a system of recursive equations in a pure **direct semantics** style.

Function `F` (line 25) and function `L` evaluate the elements of `def:def*`, and compute, in lines 23 and 26, the environment `r1` recursively. Environment `r1` contains all the bindings produced by `def` and `def*`, so recursive calls are dealt with appropriately.

In the definition of semantic function `L`, the equations defining environments `a1`, `a2` and `r1` form the following system of mutually recursive equations:

```
w  = \r.r
a1 = F(def)r1 w
a2 = L(def*)r1 w
ok = (a1 is Env) & (a2 is Env)
r1 = ok => r{a1}{a2}, r
```

Note that the use of retractile continuation `w` removes tem-

porarily the continuation nature of `F` and `L`.

5.3 Conclusion

This chapter has shown a method for accommodating in a same denotational semantics definition both continuation and direct semantics styles, and also has illustrated the application of the method by means of an example.

The claim is that the advantages of this method is that it permits the *right* kind of semantics to be used where it works best. The continuation approach makes it easier to cope with error conditions, and the direct approach facilitates the modelling of non-sequential evaluations.

Retractile continuation was used for the first time in 1981 in the formulation of the formal definition of a functional meta-language of realistic size and complexity[4].

*****

Chapter 6

Syntax-Driven Methodology

Metodologia é a arte de guiar o espírito na investigação da verdade.[1]
Michaelis - Dictionary (2014)

Denotational semantics is a very powerful and elegant formalism for describing the meaning of programming language constructs, but it is used less than it should be. In fact, in the industry, programming languages are generally described by means of a formal presentation of their syntaxes based on context free grammars together with an informal description of their semantics. The definition of $\mathcal{M}$ in Chapter 1 is an example of this style of presentation. Even when a formal definition of a language is publicly available, it is rarely read by programmers and computer scientists.

According to Peter Mosses[33], one of the reasons for this limited use of formal semantics in the industry is the difficulty most programmers and computer scientists have in dealing with the mathematical apparatus of formal definitions. Additionally, the difficulty of comprehending formal semantics

[1] Methodology is the art of guiding the mind in the search for truth.

definitions is also inherent to the way they have been organized so far.

One way to improve the presentation of denotation semantics definitions can be based upon the resources offered by the abstract-data-type and object-oriented methodologies [12, 21, 22, 25]. As in the object-oriented methodology, attention should be focused on *data* rather than *control*. Basically, a good strategy is to build modules to encapsulate data structure, i.e., domain definitions, and related semantic and auxiliary functions. This could help enhancing conceptual clarity while making the semantic definition more compact and elegant.

Clearly, the first step to enhance the *comprehensibility* of denotational definitions is to find a good module structure. The partitioning of the syntacic domains of constructs of a language into conceptually meaningful groups seems to be a good way to explore. Note that the classification of the constructs of a language into groups according to their semantic similarities is not a difficult task, as long as some knowledge about the semantics of the language is available in advance. In fact, at the time the definition of a programming language is being formulated, at least a major portion of the language must have already been designed, so that the definer should have good insight into its intended semantics.

The second step in the formulation process of a denotational definition is the characterization of the necessary semantic domains. In general, the structure of the semantic domains depends on the way the associated mappings are defined. For instance, standard denotational semantic defi-

nitions are modeled upon the notion of stores, environments and continuations. However, the need for these concepts and the details of their internal structures vary from language to language. Therefore, it seems quite natural to provide the specification of semantic domains incrementally as they are demanded by semantic functions.

Abstract-data-type and object-oriented methodologies are useful but they are not enough to achieve a satisfactory module partitioning in denotational semantics. This book addresses this issue in two complementary ways: this chapter presents a modular semantics definition style that separates concerns according to the syntactic structure of the language being defined, and the next chapter describes a complementary technique for promoting an even deeper separation of concerns.

6.1 Syntax-directed module structure

A key characteristic of denotational semantic definitions is that they are (abstract) syntax-directed. Therefore, unlike programs in a general purpose programming language, denotational definitions have their *control structure* more or less established in advance by the language's abstract syntax. It seems natural that the language's abstract syntax should play a very important role in the organization of denotational definitions, and thus a methodology for formulating semantic definitions should give special attention to the specification of abstract syntaxes and semantic domains rather than on the *control structure* of semantic functions.

The main idea behind syntactic-directed modular structure is that the formulation of a denotational semantic definition should start with the specification of the syntactic universe, i.e., the concrete and the abstract syntax of the language being defined, and use this syntactic structure as a guide to define and organize the semantic modules, and thus promote better separation and encapsulation of related concerns.

This approach has the advantage to automatically establish the module structure when the language's abstract syntax is defined. Therefore, solving the most difficult task in traditional modular programming methodology, namely, the problem of finding the *best* collection of modules.

Similar criteria have been widely used since long ago to provide BNF-based informal definitions of programming languages [2, 20, 37, 58]. These definitions were, in general, organized into chapters and sections that could be viewed as *modules*, each of which dedicated to some specific language concept, such as expressions, commands and declarations, while abstracting away details of others. Abstractions were informally used throughout these informal definitions. For instance, the section concerned with **for** statement only requires knowledge about the types of the expressions involved in the construction, leaving details on how expressions are constructed and evaluated to other sections of the defining document.

In summary, the methodology herein proposed for formulating modular denotational semantics definitions consists of the following steps:

1. From the concrete syntax of the language and from the

language's semantics the definer has in mind, the corresponding abstract syntax should be defined.

2. One or more syntactic domains should be encapsulated in modules, each of which should define:
   - the internal structure of the associated semantic domains
   - definition of the associated semantic functions
3. As the definition of modules progresses, the need for new domains or semantic functions may arise, which may be grouped to form new ad-hoc modules.

It should be emphasized that inherent properties of denotational semantics do not permit traditional information hiding principles to be used to their full extent. The internal structure of syntactic domains may not be completely hidden inside the modules they are defined because of the syntax-directed nature of the semantic definition style. Also, the internal structure of certain domains, e.g., the domains of continuations, cannot be completely encapsulated because their internal details are needed at the various points of the definition, mainly where the continuation functions are defined incrementally. This implies that the mechanism for controlling the visibility to information in semantic modules must be more flexible than those of modules that implement abstract data types in modern imperative languages.

6.2 The semantics of Small

In the sequel, the denotational definition of a variant of Michael Gordon's Small language [15] is presented to illustrate the ap-

plication of the technique just discussed. Small is a simple imperative language containing basic constructs for encoding expressions, statements and declarations.

Small expressions may contain variables, integer constants, logical values, function calls, file reading operations, conditional terms, and basic operators. They may produce collateral effect because any operand of an expression may be a `read` operation, which inputs an integer value from the standard input file, and changes accordingly the descriptor of the input file, which is part of the execution state.

Small statements are assignments, conditional commands, while-loops, procedure calls, and blocks. A block contains inner declarations and commands, including other inner blocks.

Small declarations just introduce and initialize variables and constants. There is no explicit type declaration, but variables and constants have the type of the values associated with them.

The body of a procedure is a list of statements, and that of functions is just an expression, which yields the function return value. Procedures and functions of Small are restricted to have just one parameter.

6.2.1 Semantic infrastructure

Domains defined in modules named `Environment`, `Storage` and `Continuations` model the run-time and compile-time standard infrastructure for imperative languages whose expressions may have collateral effects and several types of errors may be flagged during program execution.

Environment

Interface and definition modules named `Environment` model the context in which expressions should be evaluated and where values produced by expressions and declarations can be bound to identifiers.

```
1  interface Environment
2  imports Tokens(Id);
3          Storage(Store,Loc);
4          Continuations(Ec,Ans);
5  privates
6     r: Env; d: Dv; k: Ec; e: Ev; r: Rv; id:Id;
7  publics
8     Env    = Id -> Dv;
9     Dv     = Loc | Rv | Proc | Fun | "unbound";
10    Rv     = T | N;
11    Fun    = Ec -> Ev -> Store -> Ans;
12    Proc   = Cc -> Ev -> Store -> Ans;
13    Ev     = Loc | Rv | Proc | Fun | "unbound";
14    ro     : Env;
15    push   : (Id,Dv) -> Env -> Env ;
16    push   : Env -> Env -> Env;
17    bind   : (Id,Dv) -> Env;
18    get    : Id -> Env -> Dv;
19    isLoc  : Ec -> Dv -> Store -> Ans;
20    isLoc  : Ec -> Ev -> Store -> Ans;
21    isRv   : Ec -> Dv -> Store -> Ans;
22    isRv   : Ec -> Ev -> Store -> Ans;
23    isFun  : Ec -> Dv -> Store -> Ans;
24    isFun  : Ec -> Ev -> Store -> Ans;
25    isProc : Ec -> Dv -> Store -> Ans;
26    isProc : Ec -> Ev -> Store -> Ans;
27    isBool : Ec -> Rv -> Store -> Ans;
28    isN    : Ec -> Rv -> Store -> Ans;
29 end
```

The domains defined and exported by the interface module `Environment` have the following meaning:

- `Env` represents an abstract structure used to associate names with their meanings. It is a mapping from identifiers to denotable values. The environment works like a stack: new bindings introduced by a block are pushed down on its top, and are popped up when the execution of the block finishes.
- `Dv` is the domain of denotable values, with which Small identifiers can be associated in the environment. There is a special denotable value, named `"unbound"`, which is used to indicate that the associated identifier has not been bound yet.
- `Rv` represents the r-values of the language, i.e., values of the same nature as those produced in the evaluation of the operands of expressions occurring on the right-hand side of assignment statements.
- `Fun` is the domain of function values, which contain all the information needed to call the designated function, namely: the return address, the actual parameter, the function body, and the environment in which this body must be evaluated.
- `Proc` is the domain of procedure values, which contain all the information needed to call the designated procedure, namely: the return address, the actual parameter, the procedure body, and the environment in which this body must be executed.
- `Ev` is the domain of expressible values, i.e., values produced by the evaluation of expressions.

The domains `Dv` and `Ev` are union domains, so in order to performing the usual projection operations between the unions and their summands, the following operations, in which `k:Ec` and `s:Store`, are provided:

- `isLoc k v s` checks if `v` of type `Dv` or `Ev` holds a `Loc` value, and, if so, passes it to continuation `k`, otherwise stops execution with the message `"error"`.
- `isRv k v s` checks if `v` of type `Dv` of `Ev` holds an r-value, i.e., an member of `Rv`, and, if so, passes it to continuation `k`, otherwise stops execution with the message `"error"`.
- `isBool k v s` checks if `v` of type `Rv` holds a boolean value, and, if so, passes it to continuation `k`, otherwise stops execution with the message `"error"`.
- `isN k v s` checks if `v` of type `Rv` holds an integer value, and, if so, passes it to continuation `k`, otherwise stops execution with the message `"error"`.
- `isProc k v s` checks if `v` of type `Dv` or `Ev` holds a `Proc` value, and, if so, passes it to continuation `k`, otherwise stops execution with the message `"error"`.
- `isFun k v s` checks if `v` of type `Dv` or `Ev` holds a `Fun` value, and, if so, passes it to continuation `k`, otherwise stops execution with the message `"error"`.

Note that these functions are defined as continuation transforming mapping, i.e., they all have type `Ec->Ec`, so as to encapsulate the error condition handling.

The following operations, which are exported by interface module `Environment`, and in which `r:Env`, `id:Id` and `d:Dv`, deal with binding and scope handling:

- `r0(id)` returns the value `unbound` for any `id`.
- `push(id,d)r` pushes the binding value `{id<-d}` on the top of the environment `r`.
- `push(r1)r2` combines environments `r1` and `r2`, pushing `r1` on the top of `r2`.
- `get(id)r` retrieves the denotable value associated with `id` in environment `r`.
- `bind(id,d)` constructs a little environment `{id<-d}`.

The definition of the functions exported by the interface module `Environment` is as follows.

```
1  module Environment
2  functions
3     push(id,d)r  = r{id<-d};
4     push(r1)r2   = r2{r1};
5     bind(id,d)   = {id<-d};
6     get id r     = r(id);
7     r0(id)       = "unbound";
8     isLoc  k d s = (d is Loc)  => k Loc(d) s,"error";
9     isLoc  k e s = (e is Loc)  => k e s,"error";
10
11    isRv   k d s = (d is Rv)   => k Rv(d) s,"error";
12    isRv   k e s = (e is Rv)   => k e s,"error";
13
14    isProc k d s = (d is Proc) => k Proc(d) s,"error";
15    isProc k e s = (e is Proc) => k e s,"error";
16
17    isFun  k d s = (d is Fun)  => k Fun(d) s,"error";
18    isFun  k e s = (e is Fun)  => k e s,"error";
19
20    isBool k r s = (r is T)    => k r s,"error";
21    isN    k r s = (r is N)    => k r s,"error";
22 end
```

Storage

Modules named `Storage` define the structure of the machine memory and functions that deal with stores and storable elements.

```
1  interface Storage
2  imports Continuations(Ec,Cc,Ans); Environment(Ev);
3  privates
4     s: Store; k: Ec; e: Ev; a: Loc; v: Sv; z: Cc;
5     n, minLoc, maxLoc: N;
6     getFree: (Store,N) -> (N | "Storage-Full");
7  publics
8     Store   = Loc -> Sv;
9     Loc     = N | "input";
10    Sv      = Rv | File | "unused";
11    isRv    : Ec -> Sv -> Store -> Ans;
12    new     : Store -> Loc;
13    free    : Loc* -> Store -> Store;
14    get     : Loc  -> Store -> Sv;
15    s0      : Store;
16    ref     : Ec  -> Ev -> Store -> Ans;
17    update  : (Loc,Sv) -> Store -> Store;
18    update  : Loc -> Cc -> Ev -> Store -> Ans;
19    deref   : Ec  -> Ev -> Store -> Ans;
20    contents: Ec  -> Ev -> Store -> Ans;
21 end
```

The domains exported by the interface `Storage` are:

- `Store` is the machine-memory domain, which associates locations with Storable values.
- `Loc` is the domain of store locations, with which storable values can be associated in the machine store. There is a special location, named `"input"`, which is used to hold the descriptor of the input file of Small programs.

- `Sv` is the domain of storable values, i.e., the domain of values that can be associated with locations in the store. There is a special storable value, named `"unused"`, to mark store locations that are not in use.

The operations that are exported by module `Storage` have the following meaning, in which `a:Loc`, `s:Store`, `k:Ec`, `z:Cc`, `v:Sv`, and `e:Ev`:

- `new s`, if there a free cell in `s`, returns the location of a free cell from the store `s`, otherwise returns `"error"`.
- `free a* s` returns a new store that is a copy of `s`, except that the store locations in the list `a*` are marked as `"unused"` in the store returned.
- `get a s` retrieves the value associated with location `a` in store `s`.
- `s0` is the initial store, where all locations are associated with `"unused"`.
- `ref k e s`, if there is a free cell in `s`, stores the value `e` in a newly allocated location in store `s` and passes this location and the updated store to the expression continuation `k`, otherwise it stops execution with message `"error"`.
- `update(a,v)s` stores the storable value `v` on location `a` of store `s`, and returns the store just updated.
- `update a z e s`, if the value `e`, an `Ev` value, is storable, stores it into location `a`, and passes the updated store to the command continuation `z`, otherwise it stops execution with `"error"`.
- `deref k e s`, if `e` is not a location, passes it to the continuation `k`, otherwise it passes the dereferenced value of `e` to this continuation.

- `contents k e s`, if `e` is a location, passes the contents of this location in the store `s` to the expression continuation `k`, otherwise it stops execution with `"error"`.

The functions exported by the interface `Storage` are implemented as follows:

```
1  module Storage
2  functions
3     s0(a)  = "unused";
4     minLoc = 0;
5     maxLoc = 32767;
6
7     isRv k v s = (v is Rv) => k Rv(v)s,"error";
8
9     new s = getFree(s,minLoc)
10    where getFree(s,n) = (s n == "unused") => n,
11           (n == maxLoc)=>"Storage-Full",getFree(s,n + 1)
12    and   free a* s = (size a*==0) => s,
13                    free(tail a*,s{head a* <- "unused"});
14    get a s = s(a);
15    update(a,v)s = s{a<-v};
16
17    ref k e s = (new s == "error")=>"error",
18                                update(new s)(k(new s))e s;
19    update a z e s = e is Sv => z(s{a<-Sv(e)}),"error";
20
21    contents k e s = !(e is Loc) => "error",
22                     (s e == "unused") => "error",k(s e)s;
23    deref k e s = (e is Loc) => contents k e s , k e s;
24 end
```

Continuations

Modules named `Continuations` encapsulate the basic apparatus to deal with error conditions and jumps.

```
1  interface Continuations
2  imports Environment(Env,Ev,Rv); Storage(Store);
3  privates
4     s : Store;
5  publics
6     Dc  = Env -> Store -> Ans;
7     Cc  = Store -> Ans;
8     Ec  = Ev -> Store -> Ans;
9     Ans = {"error","stop"} | (Rv,Ans);
10    z0 : Cc;
11 end
```

The main domains exported by the interface `Continuations` are:

- `Dc` is the domain of functions that implement declaration continuations. These functions map environments and stores, produced by declarations, to final answers.
- `Cc` is the domain of functions that implement command continuations. These functions map stores, produced by command execution, to final answers.
- `Ec` is the domain of functions that implement expression continuations. These functions map expressible values and stores, produced by expression evaluation, to final answers.
- `z0` is the initial continuation function, which models the end of an execution without errors.
- `Ans` is the domain of Small program final answers, which consist of a sequence of numerical and/or truth values ended either by the message `"stop"`, for correct programs, or `"error"`, for programs containing any semantic error.

The initial continuation function `z0` is the only function defined in the definition unit of `Continuations`.

```
1 module Continuations
2 functions
3    z0 s = "stop" ;
4 end
```

Module units `Files`

Small programs can be fed with input values by reading a file from the domain `File`. This input file is modelled in the semantic definition by associating it with a special location, named `"input"` in the machine store, which holds the file descritor and its contents.

The module `Files` describes the function `readint`, declared in line 11 of the interface module below and defined in line 3 of the corresponding definition module, that perfoms the desired input file operation.

The function `readint` reads the next integer value from the input file and returns the value just read and the store updated with the file descriptor to be used in the next reading operation. In case of reading error detection, the undefined value `?` is returned instead.

```
1  interface Files
2  imports Storage(Store,Loc,get,update)
3  privates
4     in      : File;                  -- input file
5     readN   : File -> (N,File);      -- reads an integer
6     readN   : (File,N) -> (N,File); --
7     isdigit : N -> T;                -- check if a digit
8     s       : Store;
9     acc     : N;
10 publics
11    readint : Store -> (N,Store);
12 end
```

The definition of `readint` is in the definition module `Files`:

```
1  module Files
2  functions
3     readint(s)= !(in is File)=>(?,s),eof(in)=>(?,s),(n,s1)
4     where in      = get("input")s
5     and   (n,in1) = readN(in)
6     and   s1      = update("input",in1)s;
7
8     readN(in) = eof(in) => (?,in), readN(in,0);
9
10    readN(in,acc) = eof(in) => (acc,infile),
11         !isdigit(n1) & (n==0) => (?,ungetchar(n1,in1)),
12         !isdigit(n1) =>(acc,ungetchar(n1,in1)),
13                         readN(in1,acc1)
14    where (n1,in1)  = getchar(in)
15    and   digit n   = n - 48
16    and   isdigit n = (n >= 48) & (n <= 58)
17    and   acc1      = 10 * acc + digit(n1);
18 end
```

6.2.2 Function main

The interface and definition modules, named `Small`, that are presented in the sequel define the body of the function `main`, which when automatically activated from the command line

```
>Small -f smallprog.txt -i data.txt
```

performs the necessary checks, and, if all succeed, generates the AST for the Small program that is stored in file named `smallprog.txt`, using the built-in function `compile`.

The AST code produced by this function and the program's input file `data.txt` are passed to the semantic function `P` that is imported from module `Program` in order to initiate the execution of the small program.

```
interface Small
imports Continuations(Ans); Programs(P,Pro);
privates
   source, in : File;
   arg : Q;
publics
   main : Q* -> Ans;
end
```

The module below shows how the information from the command line is passed to the function `main`:

```
module Small
functions
   main(arg*) = (pro!=?)&(in!=?) => P(pro)(in), "error"
   where source = open(getarg("-f",arg*))
   and   pro    = (source!=?) => compile(source), ?
   and   in     = open(getarg("-i", arg*));
end
```

6.2.3 Small programs

The abstract syntax of Small programs is derived from the grammar defined in the modules `Programs`, `Declarations`, `Commands` and `Expressions`.

The semantic processing starts with module units `Program`, which export the domain `Pro` and semantic function `P` that maps Small programs in the abstract syntax domain `Pro` into their denotations in domain `Pd`.

The semantic function `P`, activated by the function `main`, initiates the execution of the commands of the given program in the initial context, in which (i) the descriptor of the input file is stored into the special location `"input"` of the initial and unused store, (ii) all identifiers in the environment are bound

to "unbound", and (iii) the initial continuation z0 models a successful execution.

The interface of Programs is:

```
interface Programs
imports Continuations(Cc,z0,Ans);
        Commands(Cmd,C);
        Environment(Env,r0);
        Storage(Store,s0,update);
privates
   r: Env; z: Cc; s: Store;
   infile : File;
   cmds   : Cmd*;
publics
   Pro : Start;
   P   : Pro -> Pd;
   Pd  = File -> Ans;
end
```

And the definition module Programs is as follows:

```
module Programs
syntax
   pro:Pro ::= "program" cmds "end";
functions
   P["program" cmd* "end"](infile) = C(cmd*)r0 z0 s1
   where s1 = update("input",infile)s0;
end
```

Recall that domain of functions Store has a special location named "input" that is used to hold the program input file.

6.2.4 Small declarations

The syntax and semantics of Small declarations are presented in the following module units:

```
1  module Declarations
2  syntax
3     decs:Dec* ::= decs ";" dec  => append(decs,dec)
4                 | dec           => (dec);
5     dec:Dec   ::= "const" id "=" exp
6                 | "var"   id "=" exp
7                 | "proc" id "(" id1 ")" ":" cmds
8                         => ["proc" id "(" id1 ")" cmds]
9                 | "fun" id "(" id1 ")" ":" exp
10                        => ["fun" id "(" id1 ")" exp];
11    -- semantic functions defined here
12 end
```

The semantic functions `D`, whose types are declared at lines 16 and 17 of modules `Declarations`, map Small declarations to their denotations, whose domain is defined at line 18 of the following interface module:

```
1  interface Declarations
2  imports
3     Environment(Env,Dv,Proc,Fun,Ev,bind,push);
4     Storage(Store); Continuations(Dc,Ans);
5     Tokens(Id);
6     Expressions(Exp);
7     Commands(Cmd,C);
8     Expressions(E,R);
9     Commands(C);
10 privates
11    r: Env; u: Dc; s: Store; k: Ec;
12    z: Cc;  v: Ev; p: Proc;  f: Fun;
13    cmds: Cmd*;
14 publics
15    Dec: Nonterminal;
16    D  : Dec  -> Dd;
17    D  : Dec* -> Dd;
18    Dd = Env -> Dc -> Store -> Ans;
19 end
```

The definition of the the cases of the functions named `D` is detailed in the definition module `Declarations` below.

```
1  module Declarations
2  -- here comes the grammar definition of declarations
3  functions
4     D["const" id "=" exp]r u s = R(exp)r k1 s
5     where k1 v s = u(bind(id,v))s;
6
7     D["var" id "=" exp]r u s = R(exp)r k1 s
8     where k1 v s = ref k2 v s
9     and   k2 a s = u(bind(id,a))s;
10
11    D["proc" id "(" id1 ")" cmd*] r u s = u bind(id,p) s
12    where p z v s = C(cmd*)r1 z s
13    and   r1      = push(id,p)(push(id1,v)r);
14
15    D["fun" id "(" id1 ")" exp]r u s = u bind(id,f) s
16    where f k v s = E(exp)r1 k s
17    and   r1      = push(id,f)(push(id1,v)r);
18
19    D(dec:dec*)r u s = D(dec)r u1 s
20    where u1 r1 s = D(dec*)(push r1 r)u2 s
21    and   u2 r2 s = u(push r2 r1)s;
22
23    D(nil)r u s  = u r s;
24 end
```

These definitions of `D`, in which `r:Env`, `u:Dc`, and `s:Store`, can be informally read as follows.

- `D["const" id "=" exp]r u s` evaluates `exp` to obtain a denotable value, and then associates this value with `id` to form a little environment, which together with the current store, is passed to declaration continuation.

- `D["var" id "=" exp]r u s` evaluates `exp`, stores the value obtained at a new location in the current store and associates this location with `id` to form a little environment, which together with the store just updated, is passed to the declaration continuation.
- `D["proc" id "(" id1 ")" cmd*]r u s` computes a procedure value and associates it with `id` to form a little environment, which is passed to the given declaration continuation. This procedure value defines that, whenever it is applied, the body `cmd*` of the procedure is to be executed in the environment in which the procedure has been declared, and that the value of the actual parameter and the procedure value itself are bound to `id1` and `id`, respectively. The continuation of `cmd*` and the store to be used must be passed to the procedure value at the moment the procedure is called.
- `D["fun" id "(" id1 ")" exp]r u s` computes a function value and proceeds, `mutatis mutandis`, as described in the semantics of procedure declaration above.
- `D(dec:dec*)r u s` elaborates `dec` and `dec*` in sequence, and passes to the declaration continuation the little environment formed by pushing the environment produced by `dec` on that produced by `dec*`.
- `D(nil)r u s` ends the elaboration of a declaration list, passing the current environment and store to the normal declaration continuation.

6.2.5 Small commands

The syntax and semantics of Small commands are encapsulated in the module units called `Commands`, which is shown below in a stepwise fashion.

Initially, the concrete and abstract syntax of commands are defined and, in the sequel, the associates semantic functions are detailed.

```
1  module Commands
2  syntax
3     cmd:Cmd   ::= id ":=" exp
4                 | "output" exp
5                 | "while" exp "do" cmds "end"
6                                   => ["while" exp cmds]
7                 | "if" exp "then" cmds1 "else" cmds2 "end"
8                                   => ["if" exp cmds1 cmds2]
9                 | "begin" decs ";" cmds "end"
10                                  => ["begin" decs cmds]
11                | exp1 "(" exp2 ")"
12                                  => ["call" exp1 exp2];
13    cmds:Cmd* ::= cmds ";" cmd    => append(cmds,cmd)
14                | cmd             => (cmd);
15 functions
16    -- semantic functions defined here (see below)
17 end
```

The two semantic functions named `C`, declared at lines 12 and 13 in the module below, map Small commands into their denotations, whose domain is defined at line 14.

```
1  interface Commands
2  imports
3     Continuations(Cc,Ec,Ans);
4     Storage(Store,Loc,update); Tokens(Id);
5     Expressions(E,R,Exp); Declarations(D,Dec);
```

```
6     Environment(Env,Ev,isLoc,Proc,isProc,push);
7  privates
8     r:Env; z:Cc; k:Ec; s:Store; p:Proc; a:Loc; v:Ev;
9     decs : Dec*;
10 publics
11     Cmd: Nonterminal;
12     C  : Cmd  -> Cd;
13     C  : Cmd* -> Cd;
14     Cd = Env -> Cc -> Store -> Ans;
15 end
```

The semantic functions of Small commands are detailed in the following definition module unit.

```
1  module Commands
2  syntax
3    -- here comes the grammar of commands as shown above
4  functions
5     C[id ":=" exp]r z s = E(id)r k1 s
6     where k1 v1 s = isLoc k2 v1 s
7     and    k2 v2 s = R(exp)r k3 s
8     and    k3 v3 s = update Loc(v2) z v3 s;
9
10    C["output" exp]r z s = R(exp)r k s
11    where k v s = (v,z s);
12
13    C["if" exp cmd1 cmd2]r z s = R(exp)r k1 s
14    where k1 v s = isBool k2 v s
15    and   k2 v s = T(v) => C(cmd1)r z s, C(cmd2)r z s;
16
17    C["while" exp cmd*]r z s = f
18    where f      = Y(\f.R(exp) r k1 s)
19    and   k1 v s = isBool k2 v s
20    and   k2 v s = T(v) => C(cmd*)r z1 s, z s
21    and   z1 s   = f r z s;
22
23    C["call" exp1 exp2]r z s = E(exp1)r k1 s
```

```
24      where k1 v s = isProc k2 v s
25      and   k2 p s = E(exp2)r(p z)s;
26
27      C["begin" dec* cmd*]r z s = D(dec*)r u s
28      where u r1 s = C(cmd*)(push r1 r)z s;
29
30      C(cmd:cmd*)r z s = C(cmd)r z1 s
31      where z1 s = C(cmd*)r z s;
32
33      C(nil)r z s = z(s);
34  end
```

In the informal explanation presented in the sequel for the equations defined in the module above, it is implicit that whenever the value obtained in the evaluation process does not have the expected type, the normal continuation is abandoned, an error message is sent to the final answer, and the execution stops. This is the semantics incorporated in all type checking functions used in the above equations. Thus, for `r:Env`, `z:Cc`, and `s:Store`, the equations for `D` have the folowing interpretations:

- `C[id ":=" exp]r z s` evaluates `id` to obtain the denotable value associated with it, checks if this value is a location, in which case evaluates `exp` to obtain an r-value, stores this r-value at this location and passes the store just updated to the normal continuation.
- `C["output" exp]r z s` evaluates `exp` to obtain an r-value, sends it to the final answer, and then proceeds computing the remaining elements of the final answer by executing the normal continuation of the command in the current store.

- `C["if" exp cmd1 cmd2]r z s` evaluates `exp` to obtain an r-value, which must be of type boolean, in which case, uses this boolean value to determine whether to execute next, in the current store, `cmd1` or `cmd2`.

- `C["while" exp cmd*]r z s` evaluates `exp` to obtain an r-value, which must be boolean, in which case, if this value is `false` just follows the normal continuation in the current store, otherwise executes the command list in the body of the `while` statement to obtain a new store, which is used to recursively re-executes the same `while` statement.

- `C["call" exp1 exp2]r z s` evaluates `exp1` to obtain the associated procedure value, then evaluates `exp2` to produce the necessary parameter value, and then performs the procedure call by applying the procedure value just obtained to the normal continuation and the parameter produced. The normal continuation is used as return address for the procedure call.

- `C["begin" dec* cmd*]r z s` elaborates `dec*` to obtain a little environment, which is pushed on the top of the current environment, and the combined environment and the updated store are then used to execute `cmd*`.

- `C(cmd:cmd*)r z s` executes `cmd` to get a new store, which is used to execute the command list `cmd*`.

- `C(nil)r z s` passes the current store `s` to the continuation `z`.

6.2.6 Small expressions

The syntax and the semantics of Small expressions are defined by module pair named `Expressions`.

```
1  module Expressions
2  syntax
3     exp:Exp     ::= term
4                   | exp rop term
5                   ;
6     term:Exp    ::= factor
7                   | term aop factor
8                   ;
9     factor:Exp  ::= primary
10                  | factor mop primary
11                  ;
12
13    primary:Exp ::= id => [id] | "true" => ["true"]
14                  | "false" => ["false'] | num => [num]
15                  | "read"  => ["read"]
16                  | exp1 "(" exp2 ")" => [exp1 exp2]
17                  | "if" exp "then" exp1 "else" exp2 "end"
18                                  => ["if" exp exp1 exp2]
19                  ;
20 functions
21    -- here comes the semantic functions
22 end
```

The semantic functions `E` and `R`, declared at lines 14 and 15 in the module below, map Small expressions into their denotations, which are in the domain `Ed` that is defined at line 16 of the following module:

```
1  interface Expressions
2  imports
3      Continuations(Ec,Ans);
4      Storage(Store);
```

```
5      Tokens(Id, Rop, Aop, Num);
6      Environment(Env,Ev,Rv,isFun,get,isN);
7      Storage(Loc,get,update);
8      Files(readint);
9  privates
10     r: Env; k: Ec; s: Store; v: Ev; op: Q; a: Loc;
11     apply : Q -> (N,N) -> Ec -> Store -> Ans;
12 publics
13     Exp : Nonterminal;
14     E   : Exp -> Ed;
15     R   : Exp -> Ed;
16     Ed  = Env -> Ec -> Store -> Ans;
17 end
```

The semantic functions for Small expressions are detailed in the following definition module unit:

```
1  module Expressions
2  syntax
3   --- here comes the grammar of expressions
4  functions
5     apply("+")(n1,n2)  k s = k (n1 +  n2) s;
6     apply("-")(n1,n2)  k s = k (n1 -  n2) s;
7     apply("*")(n1,n2)  k s = k (n1 *  n2) s;
8     apply("/")(n1,n2)  k s = k (n1 /  n2) s;
9     apply("<")(n1,n2)  k s = k (n1 <  n2) s;
10    apply(">")(n1,n2)  k s = k (n1 >  n2) s;
11    apply(">=")(n1,n2) k s = k (n1 >= n2) s;
12    apply("<=")(n1,n2) k s = k (n1 <= n2) s;
13    apply(op)(n1,n2)   k s = "error";
14
15    E["true"]r k s  = k(true)s;
16    E["false"]r k s = k(false)s;
17    E[num]r k s = k(toN num)s;
18
19    E[id]r k s  = (a == "unbound") => "error", k(a)s
20    where a = get(id)r;
```

```
21
22      E["read"]r k s = (n==?) => "error", k n s1
23      where (n,s1) = readint(s);
24
25      E[exp1 rop exp2]r k s = R(exp1)r k1 s
26      where k1 v1 s = isN v1 => R(exp2)r k2 s, "error"
27      and   k2 v2 s = isN v2  =>
28                      apply(rop)(N(v1),N(v2))k s, "error";
29
30      E[exp1 aop exp2]r k s = R(exp1)r k1 s
31      where k1 v1 s = isN v1 => R(exp2)r k2 s, "error"
32      and   k2 v2 s = isN v2 =>
33                      apply(aop)(N(v1),N(v2))k s, "error";
34
35      E[exp1 mop exp2]r k s = R(exp1)r k1 s
36      where k1 v1 s = isN v1 => R(exp2)r k2 s, "error"
37      and   k2 v2 s = isN v2 =>
38                      apply(mop)(N(v1),N(v2))k s, "error";
39
40      E[exp1 exp2]r k s = E(exp1)r(isFun k1)s
41      where k1 f s = E(exp2)r(f k) s;
42
43      E["if" exp exp1 exp2]r k s = R(exp) r k1 s
44      where k1 v s = isBool k2 v s
45      and   k2 v s = T(v) => R(exp1)r k s, R(exp2)r k s;
46
47      R(exp)r k s = E(exp) r k1 s
48      where k1 v s = deref k2 v s
49      and   k2 v s = isRv k v s;
50  end
```

In the explanation below, it is implicit that, whenever the value obtained in the process of evaluating an expression does not have the expected type, the normal expression continuation is abandoned, an error message is sent to the final answer,

and the execution stops.

Abstracting from this error handling mechanism, the semantics of the above equations, whose parameters are `r:Env`, `k:Ec`, and `s:Store`, can be construed as:

- `E["true"]r k s` passes `true` and the current store to the normal expression continuation.
- `E["false"]r k s` passes `false` and the current store to the normal expression continuation.
- `E[num]r k s` converts the quotation associated with the token `num` into integer, and passes the converted value and the current store to the normal expression continuation.
- `E[id]r k s` retrieves from the given environment the denotable value associated with `id`, and, if this value is not `"unbound"`, passes it and the current store to the normal expression continuation.
- `E["read"]r k s` reads the next string of digit from the input file and returns the corresponding integer value and a store containing the updated file descriptor of the input file. The value just read and the store updated are passed to the normal expression continuation.
- `E[exp1 rop exp2]r k s` evaluates expressions `exp1` and `exp2`, in this order, to get the operands for the indicated binary relational operation. If they are both of integer type, executes the operation `rop`, and passes the result and the current store to the normal expression continuation.
- `E[exp1 aop exp2]r k s` evaluates expressions `exp1` and `exp2`, in this order, to get the operands for the indicated binary arithmetic operation. If they are both of integer

type, executes the operation `aop`, and passes the result and the current store to the normal expression continuation.

- `E[exp1 mop exp2]r k s` evaluates expressions `exp1` and `exp2` to get the operands for the indicated binary arithmetic operation. If both operands are of integer type, executes the operation `mop`, and passes the result and the current store to the normal expression continuation.
- `E[exp1 exp2]r k s` evaluates `exp1` to obtain the associated function value, evaluates `exp2` to produce the necessary parameter value, and then perform the function call, applying the function value just obtained to the normal expression continuation and the parameter produced. The normal expression continuation is used as return address for the function call.
- `E["if" exp exp1 exp2]r k s` evaluates `exp` to obtain a boolean r-value, which is used to select one of the expressions `exp1` and `exp2`, which is evaluated in the sequel with the normal expression continuation.
- `R(exp)r k s` evaluates the expression `exp` to produce an expressible value. If the value obtained is a location, gets the storable value associated with this location. Either way, if the value obtained is an r-value, passes it and the current store to the normal expression continuation.

6.2.7 Small tokens

The terminal symbols of the Small language are those collected in the grammar sections of the various modules provided by this semantic definition. In addition to these termi-

nal symbols, there are still other five tokens that are exported by the interface module `Tokens`:

```
interface Tokens
syntax
   Id, Aop, Mop, Rop, Num: Token;
end
```

Note that the elements exported are the domains of the token, not the tokens themselves, and each of these tokens must be properly specified as shwown in the following definition module:

```
module Tokens
lexis
   id:Id   ::= letter+ => return id(letter+)
               ;
   letter  === 'A' .. 'Z' | 'a' .. 'z' ;
   aop     ::= "+"     => return (aop,"+")
             | "-"     => return (aop,"-")
               ;
   mop     ::= "*"     => return (mop,"*")
             | "/"     => return (mop,"/")
               ;
   rop     ::= "<"     => return (rop,"<")
             | ">"     => return (rop,">")
             | "<="    => return (rop,"<=")
             | ">="    => return (rop,">=")
             | "="     => return (rop,"=")
               ;
   num:Num ::= digit+  => return (num,digit+)
               ;
   digit   === '0' .. '9'
               ;
end
```

6.3 Evaluation

The encapsulation of fundamental and intricate concepts of programming languages may contribute to make formal definitions popular and turn formal descriptions of the semantics of large programming languages comprehensible by programmers and computer scientists.

Accordingly, the formal definition of Small has been decomposed into small pieces which are more or less independent from each other. Basic semantic concepts such as stores, environments and continuations have been isolated into separate modules so that the choice of a particular model for them does not affect the structure of the rest of the definition.

The syntactic structure of the language was used to guide the partitioning of the definition into small modules. Each module encapsulates the details of the definition of some domains and related functions, and makes their names and types available for use in other modules.

This partitioning of a semantic definition into *syntactic* modules corresponds to common practice in informal specifications of programming languages [2, 20, 37, 58].

The claim is that such a partitioning of a denotational definition produces satisfactory results in the sense that the interfaces among the various modules are kept reasonably small.

*****

Chapter 7

Component-Based Style

Ne sutor supra crepĭdam.[1]
Apelles (circa 330 B.C.)

The process of organizing denotational semantics definitions according to the syntactic structure of the language being defined produces a good set of modules. However, the comprehension level of these modules can still be further improved, for there are complexities in the denotational model that still need to be tamed.

If Backus-Naur form has been established as a universal notation for defining programming language syntax, formal semantics methods have never achieved similar success. Probably this is due to the fact no formal semantics method has the simplicity of the syntactic formalisms.

There are many explanation for that, and a good one would be that, in denotational semantics, although the semantics of a language construct should depend only on the semantics of its immediate constituents, there are always, in the semantic equations, explicit dependencies on other elements, such as

[1] The shoemaker should not go beyond the sandals (apud [40]).

the context of the construct.

The context of language constructs is usually composed of three elements: *antecedents*, *destination* and *locality*, as proposed by Tirelo et al. [52].

The *antecedents* of a construct's context comprises the effects of what has been executed previously in the program. In general these effects are propagated to the construct by entities like store and environment. For example, the values previously assigned to variables are part of the context in which an expression is evaluated.

The *destination* of a construct provides the context to which the effects of its execution are to be sent. This is usually modeled by the notion of continuations. For example, in a statement sequence, the destination of the results produced by the execution of a statement are usually the commands that follow it.

And the *locality* is the context given by the construct's enclosing structure in the abstract syntax tree. The semantics of the Java `break` statement, for instance, depends on whether it occurs inside or outside a `try-finally` statement [1].

In order to cope with all these facets of the context, the semantic equation of a construct must be provided with appropriate parameters, in addition to those that specify its constituents. These parameters allow context information be transmitted to the denotations of the construct's constituents, and also to its destination. The need to explicitly deal with context produces an undesirable dependence relation among otherwise independent equations, and complicates the related domain apparatus in which they are defined, impairing the

comprehensibility of the equations.

Moreover, the abstraction mechanisms of λ-calculus, considered insufficient to properly encapsulate definition details of semantic domains, aggravate even more the comprehensibility issue [19, 32, 33, 60]. In summary, context dependence and the lack of appropriate modularization mechanisms in λ-calculus make semantic equations very intricate.

It would be nice to rescue the idea that the semantics of a construct only depends on the meanings of its immediate constituents. Thus, the semantic equations should only specify the direct mappings from language constructs to their denotations by means of denotational components in an easy and readable way, and the details, such as context handling, should be encapsulated away.

7.1 The fundamental principle

An important property of denotational semantic definitions that derives from that idea that the meaning of a language construct should only depend on the meaning of its immediate constituents is called *referentially transparent property* [15]. This property is related to the fact that the denotation of a construct is intended to be a complete representation of its semantics, and, the semantics of a construct should only depend on the denotations of its constituents and on nothing else. Requiring referential transparency is tantamount to requiring that if two constructs have the same denotation, then they are semantically indistinguishable.

Moreover, the dependency between the denotation of a con-

struct and those of its constituents need only be on the types and names of the associated semantic functions, and not on their complete definition.

In terms of formulating language definitions, the property of referential transparency provides the basis for applying *information hiding* techniques as suggested by modern object-oriented programming methodology.

A module can then be used as the adequate apparatus to abstract away details of syntactic domains and associated semantic functions. However, a higher level of abstraction could be achieved if the context dependence is removed from all semantic equations and encapsulated in separate modules.

7.2 Context removal

The process of removing context information from semantics equations is inspired by Peter Mosses' concept of components [34, 35, 36] that he has used to improve reusability of action semantics and structured operational semantics descriptions.

The basic idea of this process applied to denotational semantics is illustrated with the following development of a comprehensible description for the semantics of the classical **if-then-else** statement of an imperative language. Traditionally, the semantics of this type of construction is defined as follows:

```
C["if" exp cmd1 cmd2]r z s =
     R(exp)r(\v s.isBool(\v s.                                    (7.1)
             v => C(cmd1)r z s, C(cmd2)r z s)v s)s
```

Notice that context information in equation (7.1) is defined by the environment `r:Env`, the command continuations `z:Cc`,

and by the machine store `s:Store`. This equation can be read as:

> *the semantics of the* **if-then-else** *statement in the presence of an environment* `r`*, command continuation* `z` *and store* `s` *comprises the evaluation of the statement condition to obtain an r-value, which is passed to the continuation to be checked if it is a boolean and, if so, to selected one of the branches of the statement to continue the execution.*

Clearly the use of context information is fundamental to convey the desired meaning to the above command, but it certainly impairs definition comprehensibility.

The context, represented by `r`, `z` and `s`, is mentioned 17 times in the equation (7.1), and it would be nice to be able to cancel out all these occurrences.

In fact, this can be achieved by creating a combinator [18], named `choose`, which permits transforming equation (7.1) into equation (7.2):

```
C["if" exp cmd1 cmd2]r z s =
   choose(R(exp),C(cmd1),C(cmd2))r z s
```
(7.2)

where `choose` is defined as:

```
choose(e,d1,d2) r z s =
   e r (\v s.isBool(\v s.v=>d1 r z s, d2 r z s)v s)s
```
(7.3)

Next step is to abstract away the definition of the combinator `choose` from the reader's eyes, placing it in a library of denotational components, and apply η reduction to equation (7.2) to reduce it to:

```
C["if" exp cmd1 cmd2] = choose(E(exp),C(cmd1),C(cmd2))
```

This equation emphasizes the following semantics:

> *the semantics of the* **if-then-else** *command is such that the value of the command's condition is to be evaluated first, and if its value is* `true`*, the execution proceeds with command* `cmd1`*, otherwise command* `cmd2` *is to be executed.*

Hopefully, to have this level of understanding of the meaning of the defined construction it is enough to know the interface of the component `choose`, which can be presented in a high level of abstraction, relieving the reader from learning the details of the definition of this combinator.

To generalize this structuring process, consider the semantic function h of a generic construct A defined in a context *c:Context*:

$$\begin{gathered} A \to r_0 B_1 r_1 B_2 \ldots B_n r_n \\ h : A \to \mathit{Context} \to \mathit{Ans} \\ h[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!]\, c = g(h_1[\![B_1]\!], h_2[\![B_2]\!], \ldots, h_n[\![B_n]\!], c) \end{gathered}$$

In the above equations, functions h_i, for $1 \leq i \leq n$, give the semantics of the A's constituents; r_j, for $0 \leq j \leq n$, are terminals, and B_k, for $1 \leq k \leq n$, are nonterminals.

In order to remove the context `c` from the parameter list of function g, and to set up the basis to encapsulate the flow of context information, consider a generic combinator K, defined as:

$$K\,(d_1, d_2, \ldots, d_n)\, c = g(d_1, d_2, \ldots, d_n, c)$$

where d_i, for $1 \leq i \leq n$, are denotations.

The use of the combinator K to rewrite the above definition of h produces:

$$h[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!]c = K(h_1[\![B_1]\!], h_2[\![B_2]\!], \ldots, h_n[\![B_n]\!])c$$

which may be simplified to:

$$h[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!] = K(h_1[\![B_1]\!], h_2[\![B_2]\!], \ldots, h_n[\![B_n]\!])$$

To keep the formulation simple, each semantic equation typically must use just one combinator, avoiding the enticing idea of using combinator composition. In fact, to achieve comprehensibility, discipline and standardization are mandatory. Thus, it seems reasonable to require that all denotational combinators like K must have the standard type:

K: $(D_1\ ,\ D_2\ ,\ \cdots\ ,\ D_n)$ -> *Context* -> *Ans*

where D_i, for $1 \leq i \leq n$, are domains of denotations or of special values associated with the production, and *Ans* the usual domain of final answers.

This discipline permits rescuing the central idea of the denotational semantics formalism in which the meaning of a construct only depends on the meaning of its immediate constituents. In this sense, the parameters of a combinator should be only denotations of constructs because the semantics model should follows the premises that the combinator K must combine the semantics of the immediate constituents of `A` to produce its meaning.

Additionally, the function K should not contain any references to the terminal symbols that occur on the right hand side of the production defining A. Only the nonterminals are allowed to take part in the formulation. This means that, to keep the model clean, each right hand side of a grammar production must imply in a new g, i.e., dependences on terminal symbols are to be forged into the structure of function g.

However, in order to favor reuse of components and yet preserving comprehensibility, sometimes it may be convenient to

pass to a combinator special values to determine some specific behavior, instead of writing several similar combinator functions. The need for this arises when more than one production have the same nonterminals on their right hand sides, being distinguished only by the terminal symbols involved. The use of the resource, though, should be an exception rather than a rule.

This process of encapsulating context should be applied to the semantic equations of all constructs in the language, producing a clean set of combinators whose use is much more comprehensible than their counterparts, which exhibit all context dependencies.

A special attention should be paid to the initial semantic equation, which should not be object of componentization because this equation is the place in which the execution context must be established.

7.3 Component-based semantics of Small

A component-based denotational definition of Small, the toy programming language presented in Chapter 6, is used to illustrate the technique proposed for separation of concerns.

The following module units, defined in Chapter 6, are used in this definition:

- `Environment` (*§6.2.1, page 131*), which defines and implements domains that are specific to Small environments.
- `Storage` (*§6.2.1, page 135*), which defines the domains and functions dealing with program stores.

- `Continuations` (*§6.2.1, page 137*), which defines the various types of continuations.
- `Files` (*§6.2.1, page 139*), which defines the domain of file and i/o operations.

7.3.1 Main function

Module `small` defines the function `main`, which is implicitly activated from the command line

```
>small -f source.small -i in.txt
```

where `source.m` is the source code file containing the Small program to be compiled and executed, and `in.txt`, the program input file. The activation of `main` is carried out as:

```
main( ("-f", "source.small", "-i", "in.txt") )
```

The interface module `small` is as follows:

```
1  interface small
2  imports
3     Continuations(Ans);
4     Programs(P,Pro);
5  privates
6     source, in : File;
7     arg : Q;
8  publics
9     main : Q* -> Ans;
10 end
```

And the corresponding definition is:

```
1  module small
2  functions
3     main(arg*) = (pro!=?)&(in!=?)=> P(pro)(in), "error"
4     where source = open(getarg("-f",arg*))
5           pro    = (source!=?) => compile(source), ?
6           in     = open(getarg("-i", arg*))
7  end
```

Note that `main` retrieves the program input and the source program file descriptors from its argument list, generates the AST for the program stored in file `source.small` using the built-in function `compile` and passes the program AST and the input file to the semantic function `P` of Small programs to produce the final answer of its execution.

7.3.2 Small programs

The semantic function `P` maps Small programs represented as elements of the abstract syntax domain `Pro` into their denotations `Pd`, which are mappings from program input file to final answers, as defined in the following module:

```
1  interface Programs
2  imports Continuations(Cc,z0,Ans); Commands(C,Cmd);
3          Environment(Env,r0); Storage(Store,s0,update);
4  privates
5     r : Env ; z : Cc; s : Store;
6     cmds : Cmd*
7  publics
8     P  : Pro -> Pd;
9     Pd = File -> Ans;
10 end
```

Function `P` initiates the execution of the given program in the initial context, in which the descriptor of the input file is plugged to the special location `"input"` of the initial store.

```
1  module Programs
2  syntax
3     pro:Pro ::= "program" cmds "end" ;
4  functions
5     P["program" cmd*]file = C(cmd*)r0 z0 s1
6     where s1 = update("input",file)s0;
7  end
```

7.3.3 Small declarations

The syntax and the semantic functions for Small declarations are detailed in the following module units:

- `interface Dec_Components`, which defines the interface of the denotational components that implement the details of the semantics of declarations.
- `module Dec_Components`, which defines the definition of the denotational components for the semantics of Small declarations (*§7.4.1, page 178*).
- `interface Declarations`, which contains the interface of the high level semantic functions for Small declarations.
- `module Declarations`, which contains the definition of the semantic equations for Small declarations.

Declaration component interface

```
1  interface Dec_Components
2  imports
3     Exp_Components(Ed); Cmd_Components(Cd);
4     Storage(Store); Environment(Env,Fun,Proc); Tokens(Id);
5     Declarations(Ev); Continuations(Dc,Cc,Ec,Ans);
6     Exp_Components(Ed); Cmd_Components(Cd);
7  privates
8     r : Env; s : Store; d : Dd;  c : Cd; e : Ed; v: Ev;
9     f : Fun; p : Proc;  u : Dc;  z : Cc; k : Ec;
10 publics
11    Dd    = Env -> Dc -> Store -> Ans;
12    cbind, vbind : (Id,Ed) -> Dd;
13    pbind : (Id,Id,Cd) -> Dd;
14    fbind : (Id,Id,Ed) -> Dd;
15    elab  : (Dd,Dd) -> Dd;
16    continue: Dd;
17 end
```

In the above interface module, `Dd` is the domain of command denotations, and the denotational components specified in the module are, for `r:Env`, `u:Dc`, and `s:Store`:

- `cbind(id,e)r u s`, which elaborates the expression denotation `e:Ed`, binds the result to the identifier `id` in the current environment, which is passed to the continuation.
- `vbind(id,e)r u s`, which elaborates the expression denotation `e:Ed`, stores the result in a new location of the current store, binds this location to the identifier `id` in the current environment, which is passed to the continuation.
- `pbind(id,id1,c)r u s`, which builds a procedure value with parameter `id1` and body denotation `c:Cd`, and associates this value with identifier `id` in the current environment, which is passed to the continuation.
- `fbind(id,id1,e)r u s`, which builds a function value with parameter `id1` and body denotation `e:Ed`, and associates this value with the identifier id in the current environment, which is passed to the continuation.
- `elab(d1,d2)r u s`, which elaborates the two declaration denotations `d1` and `d2`, in this order, combines their results and pushes the little environment produced on the top of the current environment, which is passed to the continuation.
- `continue r u s`, which passes the current environment to the declaration continuation.

The context defined by `r`, `u` and `s` are not mentioned in the semantic equations for Small declaration definitions presented in the sequel, since the context handling of these components is hidden and detailed in modules defined in Section 7.4.1.

Semantic equations for declarations

The types of the semantic functions for Small declarations are declared in the interface module:

```
interface Declarations
imports
   Expressions(E,R,Exp); Commands(C,Cmd); Tokens(Id);
   Dec_Components(Dd,cbind,vbind,pbind,fbind);
   Dec_Components(continue,elab);
privates
   cmds: Cmd*
publics
   D : Dec  -> Dd;
   D : Dec* -> Dd;
end
```

The syntax of Small declarations and the associates semantics are defined in the following module:

```
module Declarations
syntax
   decs:Dec* ::= decs ";" dec   => append(decs,dec)
               | dec            => (dec);
   dec:Dec   ::= "const" id "=" exp
               | "var" id "=" exp
               | "proc" id "(" id1 ")" ":" cmds
                          => ["proc" id "(" id1 ")" cmds]
               | "fun" id "(" id1 ")" ":" exp
                          => ["fun" id "(" id1 ")" exp];
functions
   D["const" id "=" exp] = cbind(id,R(exp));
   D["var" id "=" exp]   = vbind(id,R(exp));
   D["proc" id "(" id1 ")" cmd*] = pbind(id,id1,C(cmd*));
   D["fun" id "(" id1 ")" exp] = fbind(id,id1,E(exp));
   D(nil) = continue;
   D(dec:dec*) = elab(D(dec),D(dec*));
end
```

7.3.4 Small commands

The syntax and semantics definition of Small commands comprise the modules:

- `interface Cmd_Components`, which defines the interface of the denotational components that implement the details of the semantics of commands.
- `module Cmd_Components`, which defines the definition of the denotational components for the semantics of Small commands (*§7.4.2, page 179*).
- `interface Commands`, which contains the interface of the high level semantic functions for Small commands.
- `module Commands`, which contains the definition of the semantic equations for Small commands.

The interface of module `Commands`, which exports the semantic function `D`, is:

```
1  interface Commands
2  imports
3     Declarations(Dec,D);
4     Expressions(Exp,E,R);
5     Tokens(Id);
6     Cmd_Components(Cd,assign,toAns,choose,loop);
7     Cmd_Components(newblock,execute,continue);
8  privates
9     decs : Dec*;
10 publics
11    C : Cmd  -> Cd;
12    C : Cmd* -> Cd;
13 end
```

The syntax and semantics of Small commands are detailed in the following definition module:

```
1  module Commands
2  syntax
3     cmd:Cmd   ::= id ":=" exp
4                 | "output" exp
5                 | "while" exp "do" cmds "end"
6                                    => ["while" exp cmds]
7                 | "if" exp "then" cmds1 "else" cmds2 "end"
8                                   => ["if" exp cmds1 cmds2]
9                 | "begin" decs ";" cmds "end"
10                                   => ["begin" decs cmds]
11                | exp "(" exp1 ")" => ["call" exp exp1]
12                ;
13    cmds:Cmd* ::= cmds ";" cmd     => append(cmds,cmd)
14                | cmd              => (cmd)
15                ;
16 functions
17    C[id ":=" exp] = assign(E(id),R(exp));
18    C["output" exp] = toAns(R(exp));
19    C["if" exp cmd1 cmd2]=choose(R(exp),C(cmd1),C(cmd2));
20    C["while" exp "do" cmd] = loop(R(exp),C(cmd));
21    C["call" exp exp1] = pcall(E(exp),E(exp1));
22    C["begin" dec* cmd*] = newblock(D(dec*),C(cmd*));
23    C(cmd:cmd*) = execute(C(cmd),C(cmd*));
24    C(nil) = continue;
25 end
```

Note that the context of command execution, i.e., the parameters `r:Env`, `z:Cc` and `s:Store`, is not mentioned in the above equations in order to enhance comprehensibility.

Command component interface

The domains and the types of the functions used in the definition of the semantic equation of commands are hidden and defined in the modules for command components as follows:

```
1  interface Cmd_Components
2  imports
3     Exp_Components(Ed);
4     Storage(Store);
5     Environment(Env,Ev,isBool,isLoc,isProc);
6     Storage(Store,update);
7     Continuations(Dc,Cc,Ec,Ans);
8     Dec_Components(Dd);
9     Exp_Components(Ed);
10 privates
11    r : Env; u : Dc; z : Cc; s : Store;
12    d : Dd;  c : Cd; e : Ed; v : Ev;
13    p : Ev;  f : Cd;
14 publics
15    Cd = Env -> Cc -> Store -> Ans;
16    assign   : (Ed,Ed) -> Cd;
17    toAns    : Ed -> Cd;
18    choose   : (Ed,Cd,Cd) -> Cd;
19    loop     : (Ed,Cd) -> Cd;
20    pcall    : (Ed,Ed) -> Cd;
21    execute  : (Cd,Cd) -> Cd;
22    continue : Cd;
23    newblock : (Dd,Cd) -> Cd;
24 end
```

In the interface above, `Cd` is the domain of command denotations, and the specified denotational components are as follows:

- `assign(e1,e2)r z s`, which elaborates the expression denotation `e1:Ed` to obtain a location, say `a:Loc`, then evaluates the expression denotation `e2:Ed` to obtain a value which is associated with location `a` in the current store.

- `toAns(v)r z s`, which sends value denoted by `v:N|T` to the program final answer.
- `choose(e,c1,c2)r z s`, which checks whether the evaluation of expression denotation `e:Ed` leads to a boolean value, in which case, elaborates the command denotation `c1:Cd`, if `e` denotes `true`, otherwise elaborates command denotation `c2:Cd`.
- `loop(e,c)r z s`, which repeatedly checks whether the evaluation of expression denotation `e:Ed` produces the boolean value `true`, in which case, command denotation `c:Cd` is evaluated in the current store to produce a new store. The repetition process ends when the boolean value produced is `false`, and then the current store is passed on to the normal continuation.
- `pcall(e1,e2)r z s`, which elaborates expression denotation `e1` to obtain a procedure denotation, and then applies it to `e2`, current environment and store.
- `execute(c1,c2)r z s`, which elaborates command denotation `c1:Cd`, and, in the sequel, elaborates command denotation `c2:Cd` in the current environment and store.
- `continue r z s`, which passes the current store to the continuation.
- `newblock(d,c)r z s`, which starts a new scope layer, adding declarations encountered in the evaluation of the declaration denotation `d:Dd` to the current environment, and then evaluates the command denotation `c:Cd` in the extended environment.

7.3.5 Small expressions

The concrete and abstract syntaxes of Small expressions are defined in the following interface and definition modules:

- `interface Exp_Components`, which defines the interface of the denotational components that implement the details of the semantics of expressions.
- `definition Exp_Components`, which defines the definition of the denotational components for the semantics of Small expressions (*§7.4.3, page 180*).
- `interface Expressions`, which contains the interface of the high level semantic functions for Small expressions.
- `definition Expressions`, which contains the definition of the semantic equations for Small expressions.

Semantic equations for expressions

The interface module `Expressions` is:

```
1  interface Expressions
2  imports
3     Exp_Components(Ed,send,read,operate,fcall);
4     Exp_Components(choose,dereference);
5     Continuations(Ec,Ans);
6     Environment(Env,Rv,isFun,isRv,isBool,isN);
7     Storage(Store,deref);
8     Tokens(Id,Num,Aop,Mop);
9  privates
10    r: Env; k: Ec; s: Store; e: Ed; op: Q; v: Ev; r: Rv;
11    apply : Q -> (N,N) -> Ec -> Store -> Ans;
12 publics
13    R : Exp -> Ed;    -- r-expressions
14    E : Exp -> Ed;    -- l-expressions
15 end
```

And the corresponding definition module, which contains the syntax and semantic definitions of Small expressions, is:

```
module Expressions
syntax
   exp:Exp     ::= term
                 | exp rop term
                 ;
   term:Exp    ::= factor
                 | term aop factor
                 ;
   factor:Exp  ::= primary
                 | factor mop primary
                 ;
   primary:Exp ::= id => [id] | "true" => ["true"]
                 | "false" => ["false"] | num => [num]
                 | "read"  => ["read"]
                 | exp "(" exp1 ")" => [exp exp1]
                 | "if" exp "then" exp1 "else" exp2 "end"
                                    => ["if" exp exp1 exp2]
                 ;
functions
   E["true"]  = send(true);
   E["false"] = send(false);
   E[num] = send(toN num);
   E[id]  = send(id);
   E["read"] = read;
   E[exp1 rop exp2] = operate(rop,R(exp1),R(exp2));
   E[exp1 aop exp2] = operate(aop,R(exp1),R(exp2));
   E[exp1 mop exp2] = operate(mop,R(exp1),R(exp2));
   E[exp exp1] = fcall(E(exp),E(exp1));
   E["if" exp  exp1 exp2] =
                     choose(R(exp),E(exp1),E(exp2));
   R(exp) = dereference(E(exp));
end
```

Expression component interface

The modules named `Exp_Components` define the components that hide the details of the semantics of expressions:

```
1  interface Exp_Components
2  imports
3      Environment(Env); Storage(Store);
4      Continuations(Ec,Ans);
5  publics
6     Ed = Env -> Ec -> Store -> Ans;
7     send        : T -> Ed;
8     send        : N -> Ed;
9     read        : Ed;
10    operate     : (Q, Ed,Ed) -> Ed;
11    fcall       : (Ed,Ed) -> Ed;
12    choose      : (Ed,Ed,Ed) -> Ed;
13    dereference: Ed -> Ed;
14 end
```

In the above interface module, `Ed` is the domain of expression denotations, and the specified denotational components have the following description, in which `e`, `e1` and `e2` are elements of `Ed`, and `r:Env`, `k:Ec`, and `s:Store` are the context:

- `send(t)r k s`, which transmits value `t:T` to the current expression continuation.
- `send(n)r k s`, which transmits value `n:N` to the current expression continuation.
- `send(id)r k s`, which retrieves the value associated with `id` in the current environment and, if the retrieved value is not "unbound", passes it to the current expression continuation.
- `read r k s`, which reads the integer number from the head of the standard input file, advances the reading po-

sition accordingly, and passes the value just read to the current continuation.

- `operate (op,e1,e2)r k s`, which performs the binary operation designed by `op:Q` over the operands `e1:Ed` and `e2:Ed`, and passes the result to the current expression continuation.
- `fcall(e1,e2)r k s`, which elaborates `e1:Ed` to obtain a function denotation, applies it to the parameter `e2:Ed` and then passes the result to the normal expression continuation.
- `choose(e,e1,e2)r k s`, which if `e:Ed` evaluates to the value `true`, elaborates `e1:Ed`, otherwise elaborates `e2:Ed`, and then passes the result of either case to the current continuation.
- `dereference(e)r k s`, which dereferences the value denoted by `e:Ed` and if it is an r-value, passes the dereferenced value to the current expression continuation.

7.3.6 Small tokens

In addition to the terminal symbols, which are written within quotes in production rules of the several modules of this definition, there are additional tokens whose domains are exported by the following module:

```
1 interface Tokens
2 publics
3    Id, Aop, Mop, Rop, Num : Token
4 end
```

The tokens in these domains exported by module `Tokens` are defined in the following definition module:

```
1  module Tokens
2  lexis
3     id      ::= letter+ => return (id,letter+)
4                 ;
5     letter  === 'A' .. 'Z' | 'a' .. 'z'
6                 ;
7     aop     ::= "+"     => return (aop,"+")
8                 | "-"     => return (aop,"-")
9                 ;
10    mop     ::= "*"     => return (mop,"*")
11                | "/"     => return (mop,"/")
12                ;
13    rop     ::= "<"     => return (rop,"<")
14                | ">"     => return (rop,">")
15                | "<="    => return (rop,"<=")
16                | ">="    => return (rop,">=")
17                | "="     => return (rop,"=");
18    num     ::= digit+  => return (num,digit+)
19                ;
20    digit   === '0' .. '9'
21                ;
22 end
```

7.4 The denotational components for Small

In order to complete the semantic definition of Small, the following section details the implementation of the semantic components used in the definition of declarations, commands and expressions.

7.4.1 Components for declarations

The definition of the functions listed in the interface module for declaration components (*§7.3.3, page 167*) is given in the

following companion module:

```
module Dec_Components
functions
   cbind(id,e)r u s = e r k s
   where k v s = u bind(id,v) s;

   vbind(id,e)r u s = e r k1 s
   where k1 v s = ref k2 v s
   and   k2 a s = u bind(id,a) s ;

   pbind(id,id1,c)r u s = u bind(id,p) s
   where p z v s = c r1 z s
   and   r1      = push(id,p)(push(id1,v)r);

   fbind(id,id1,e)r u s = u bind(id,f) s
   where f k v s = e r1 k s
   and   r1      = push(id,f)(push(id1,v)r);

   elab(d1,d2)r u s = d1 r u1 s
   where u1 r1 s = d2 (push r1 r) u2 s
   and   u2 r2 s = u (push r2 r1) s;

   continue r u s  = u r s;
end
```

7.4.2 Components for commands

The definition of the functions listed in the interface module for command components (*§7.3.4, page 171*) is given in the following companion module:

```
module Cmd_Components
functions
   assign(e1,e2)r z s = e1 r k1 s;
   where k1 v s = isloc k2 v s
```

```
5     and   k2 a s = e2 k3 s
6     and   k3 v s = update a z v s;
7
8     toAns e r z s =  e r k s
9     where k v s = (v,z s);
10
11    choose(e,c1,c2) r c s = e r k1 s
12    where k1 v s = isBool k2 v s
13    and   k2 v s = T(v) => c1 r z s , c2 r z s;
14
15    loop(e,c) r z s = f
16    where f      = Y(\f.e r k1 s)
17    and   k1 v s = isBool k2 v s
18    and   k2 v s = T(v) => c r z1 s , z s
19    and   z1 s   = f r z s;
20
21    pcall(e1,e2)r z s = e1 r k1 s
22    where k1 v s = isProc k2 v s
23    and   k2 p s = e2 r (p z) s;
24
25    execute(c1,c2)r z s = c1 r z1 s
26    where z1 s = c2 r z s;
27
28    continue r z s  = z s;
29
30    newblock(d,c) r z s = d r u s
31    where u r1 s = c (push r1 r) z s;
32
33 end
```

7.4.3 Components for expressions

The definition of the functions listed in the interface module for expression components (*§7.3.5, page 176*) is given in the following companion module:

```
1  module Exp_Components
2  functions -- auxiliaries
3     apply("+")(n1,n2)  k s = k (n1 +  n2) s;
4     apply("-")(n1,n2)  k s = k (n1 -  n2) s;
5     apply("*")(n1,n2)  k s = k (n1 *  n2) s;
6     apply("/")(n1,n2)  k s = k (n1 /  n2) s;
7     apply("<")(n1,n2)  k s = k (n1 <  n2) s;
8     apply(">")(n1,n2)  k s = k (n1 >  n2) s;
9     apply(">=")(n1,n2) k s = k (n1 >= n2) s;
10    apply("<=")(n1,n2) k s = k (n1 <= n2) s;
11    apply(op)(n1,n2)   k s = "error";
12
13    send t r k s = k t s;
14    send n r k s = k n s;
15    send(id)r k s = (r id == "unbound") => "error",
16                                           k(r id);
17    read r k s = (n == ?) => "error", k n s1
18    where (n,s1) = readint(s);
19
20    operate(op,e1,e2)r k s = e1 r k1 s
21    where k1 v1 s = isN v1 => e2 r k2 s, "error"
22    and   k2 v2 s = isN v2 => apply(op,N(v1),
23                                    N(v2))k s;
24    fcall(e1,e2)r k s = e1 r k1 s
25    where k1 v s = isFun k2 v s
26    and   k2 f s = e2 r k3 s
27    and   k3 v s = f k v s;
28
29    choose(e,e1,e2)r k s = e r k1 s
30    where k1 v s = isBool k2 v s
31    and   k2 v s = v => e1 r k s , e2 r k s;
32
33    dereference(e)r k s = e r k1 s
34    where k1 v s = deref k2 v s
35    and   k2 v s = isRv k v s;
36 end
```

7.5 Discussion

The component-based style for denotational semantics is a complementary approach to other solutions to the comprehensibility problem of formal semantics. For instance, the incremental definition style of Tirelo at al. [52], which is based on the linguistic concept of vagueness can benefit from the use of components. In the incremental approach details are added one by one to a simpler definition by means of a mechanism named denotation transformation. The use of components may help separating concerns, which is very important to facilitate the integration of new elements to the definition.

Another important attempt to solve the comprehensibility problem is the monadic semantics proposed by Moggi[27, 28], which also removes context information from the equations and, consequently, reaches high level of modularity. However, monadic semantics requires complex and intricate monad transformation operations. Component-based semantics are much simpler, they can encapsulate fundamental concepts in a way that is easy to use.

Apparently P. Mosses [34, 35, 36] has avoided the use of denotational semantics as the basis for a technique based on components due to the low comprehensibility caused by the explicit use of context information. The present proposal overcomes these difficulties.

Comparison with other approaches to formal semantics [32, 60], such as action semantics and structured operational semantics, was not addressed at this moment because the focus of this work is the improvement of the comprehensibility of de-

notational semantics, not to make it supersede other models. Each formal method has its proper niche, in which it produces better results. Component-based semantics just brings value to the denotational formalism.

Judicious use of denotational components permits the removal of context dependence from the presentation of semantic equations, making them more comprehensible. To this purpose, the component signatures can be standardized and the flow of context information encapsulated within these components. The result are semantic definitions which are reduced to mappings from abstract syntax constructs to their respective denotations expressed as a combination of denotational components.

Encapsulation of fundamental and intricate concepts of programming languages may contribute to make formal semantics popular and turn descriptions of the semantics of large programming languages comprehensible by programmers and computer scientists.

The claim is that to understand a component-based denotational semantics description of a given programming language all that is required is to know the interface of the used components, without any concern regarding details of their definitions.

Due to the continuous evolution of programming languages, it would be interesting to have a library of generic components that allows easy incorporation of new constructs to the languages. The more generic are the components the better will be the library, because the same components could be used to define many languages.

The challenge is to find a set of generic components capable of modeling the semantics of the most important constructs of popular languages, thus reducing the need to define new components whenever defining new programming languages.

In this respect, Santos [41] has proposed and implemented a set of generic denotational components to provide scalable definitions. In this way, the construction of new descriptions could be simply an act of putting together predefined components from this library, without any concerns to context handling.

In summary, the claim is that the separation of concerns and the use of generic denotational components are a promissing way to go in order to make formal descriptions more limpid, comprehensible and scalable.

*****

Chapter 8

Domain Theory

> *Quelli che s'innamoran di pratica senza scienzia sono como il nocchieri che entra in navilio sanza timone o bussula, che mai ha certezza dove si vada.*[1]
>
> Leonardo da Vinci (1452-1519)

This chapter addresses elementary aspects of Dana Scott's theory of domains and continuous functions, and the use of fixpoint techniques to find solutions for recursive equations.

In formal definitions of semantics, `domains` should be used instead of `sets`. The traditional concepts of sets and total functions work properly in the formulation of the definition of constructs whose semantics, in essence, requires just mappings from a set to another. However, there are situations in which the notion of set without a more elaborated structure does not work properly. In fact, some equations based on ordinary sets and total functions may present problems and inconsistences, and thus, to assure soundness of the model, a more sophisticated mathematical apparatus is required.

[1] Those in love with practice without scientific knowledge are like the helmsman that enters a ship without rudder or compass, and is never certain to where he might go.[13]

8.1 Theoretical problems

There are indeed three separate problems that require the use of domains instead of sets:

- recursive definition of functions
- recursive definition of sets
- program nontermination

The definition of Circularity, a very small imperative language, highlights the need for domains and exhibits their differences from sets.

```
1  module Circularity
2  syntax
3     prog:Prog ::= "program" cmds "write" exp
4                 ;
5     cmds:Cmd* ::= cmds ";" cmd => append(cmds,cmd)
6                 | cmd          => (cmd)
7                 ;
8     cmd :Cmd  ::= id ":=" exp
9                 | "while" exp "do" cmds
10                | id
11                ;
12    exp:Exp   ::= "0" | "suc" exp | "proc" cmds
13                ;
14    id:Id     ::= token
15                ;
16 lexis
17    id      ::= letter+ => return (id,letter+)
18              ;
19    letter  === 'A' .. 'Z' | 'a' .. 'z'
20              ;
21 functions
22   -- here comes the definitions of semantic functions
23 end
```

Suppose that the following module defines *sets*, instead of *domains*, and that all functions are total:

```
1  interface Circularity
2  publics
3     Id : Token;
4     Exp, Cmd : Nonterminal;
5     Prog : Start;
6     Value = N | Proc;
7     Proc  = State -> State;
8     State = Id -> Value;
9     E : Exp  -> State -> Value;
10    C : Cmd  -> State -> State;
11    C : Cmd* -> State -> State;
12    P : Prog -> Value;
13 end
```

The execution of a Circularity program is modeled be a sequence of state transformations. Each state is represented by the set `State` that contains functions to associate identifiers with their values, which can be integers or procedure values. Initially, all functions in `State` associates all identifiers with 0.

No continuations are used so as to keep the example clean. The semantic functions `E`, for expressions, `C`, for commands, and `P`, for programs, are defined as follows:

```
1  module Circularity
2  syntax
3    -- here the syntax is defined
4  lexis
5    -- and so is lexis
6  functions
7     E["0"]s = 0;
8     E["suc" exp] s = E[exp]s is N => E[exp]s + 1, 0;
9     E["proc" cmd*]s = \s.C(cmd*)s;
```

```
10    C[id ":=" exp]s = s{id<-E(exp)s};
11    C[id]s = s(id)s;
12    C["while" exp "do" cmds] s = (E(exp)s==0)=>s,
13                        C["while" exp "do" cmds](C(cmds)s);
14    C(nil)s = s;
15    C(cmd:cmd*)s = C(cmd*)(C(cmd)s);
16    P["program" cmd* "write" exp]=E(exp)(C(cmd*)(\id.0));
17 end
```

8.1.1 Recursive definition of functions

Recursive function definition is a natural device to model the semantics of some language constructs such as the while statement, which is defined at lines 12 and 13 of the definition unit above by means of a recursive equation.

This recursive equation seems correctly formulated, but it raises relevant questions as to whether it has a solution, or, if so, whether the solution is unique.

Generally speaking, it is a fact that many recursive equations work fine with sets. For instance, the recursive equation

$$f(x) = (x = 0) \rightarrow 1, x \times f(x-1)$$

uniquely defines the factorial function $f(x) = x!$.

However, not always recursive equations are mathematically sound, because plain sets do not prevent bad equations to be written. For instance, the equation

$$f(x) = f(x) + 1$$

does not have any solution, and the equation

$$f(x) = f(x)$$

has an infinite number of solutions.

In order to keep the model consistent, all equations should have a solution or, when there are more than one solution,

there must be a systematic way to select one of them. Sets cannot give this guarantee.

8.1.2 Recursive definition of sets

Sets may be recursively defined also. For instance, in the interface module `Circularity`, `State` is defined, at line 8, in terms of `Value`, which is defined in terms of `Proc`, which is defined as `State -> State`. Therefore, `State` is defined in terms of `State -> State`. Ignoring other elements of `State`, the definition of `State`, for the sake of argument, may be viewed as:

```
State = State -> State
```

which is an equation that violates the Cantor's theorem [44]:

Theorem 8.1 *The space of all functions in the set* $V \to V$ *has more elements than set* V*, if the cardinality of* $V \geq 2$*.*

Clearly, the system of mutually recursive equations defined in the interface Circularity does not lead to consistent and sound results, since it implies in function spaces that have too many functions.

8.1.3 Program nontermination

The semantic function `P:Prog->Value` defined as

```
P["program" cmd* "write" exp] = E(exp)(C(cmd*)(\id.0))
```

does not cope with the case of program nontermination, for the type of `P` requires that it always returns a value, missing the fact that nonterminating programs do not return any

value. Therefore, `P` only gives the semantics for programs that terminate, which is not satisfactory.

8.2 The work of Scott

In order to solve the three problems just described, Dana Scott [43, 44, 45, 46] introduced the concept of *domains* as a special kind of sets with an internal structure that ensures that every definition based on domains is good, and that all equations have at least one solution. He also defined a method to select the best solution when there are more than one.

According to Dana Scott, domain equations always have a solution provided that:

1. the mapping between two domains be not the domains of all function with this type, but only those that preserve the structure of the domains involved and that are continuous;
2. the symbol `=` in a domain definition, such as `A=B`, for any domains `A` and `B`, be not interpreted as equality. It should be viewed as an isomorphism.

In the formulation of his domain theory, Scott has accomplished the following:

1. he defined a class of structured sets with cartesian product ($\times$), domain union (`|`), mapping ($\rightarrow$), and sequence ($*$) operators that are defined in a way that preserves the structure of these sets;
2. he showed how domain elements can be defined recursively;

3. he showed how domains can be recursively defined;
4. he included in all domains the semantics of nontermination.

The interesting thing is that not everyone needs to know the theory of domains to be able to formulate sound denotational semantics definitions. In fact, only those interested in performing rigorous proofs of program properties, designing new description techniques, e.g., modeling of concurrency, or to be an expert need deeper knowledge of this theory.

8.3 Foundations

The underlying idea of the domain theory is the notion of ordering data objects according to the information contents of their representations, from which the theory of partial and infinite objects with finite representation has been developed.

In this section, the notion of approximation and partial order relation are introduced to establish the basis to define Scott's domains.

The notation and definitions used in this section mostly come from David F. Martin's UCLA class notes [23].

8.3.1 Functions

Definition 8.1 *A function f from a domain A into a domain B is a rule that associates with each element of A a unique element of B. Symbolically, this statement says $f : A \rightarrow B$.*

Definition 8.2 *A function $f : A \rightarrow B$ is* total *if it has a value defined for all elements of domain A. Otherwise, it is a* partial *function.*

Definition 8.3 *The Graph(f) of a function $f : A \to B$ is a subdomain[2] of the cartesian product $A \times B$, such that*

$$Graph(f) = \{(a, b) \mid a \in A \ \wedge b \in B \ \wedge b = f(a)\}$$

8.3.2 Relations on sets

Definition 8.4 *A* binary relation *R on a set A is a two-argument predicate $R : A \times A \to \mathcal{B}$, where $\mathcal{B}$ is the set of boolean values.*

A binary relation R on a set A is a subset of the cartesian product $A \times A$, i.e., $R \subseteq A \times A$.

If there is a relation between $a, b \in A$, one may write $R(a, b)$ or $a \ R \ b$ or then $(a, b) \in R$.

A relation R on a set A is:

- **reflexive** if and only if
  $\forall a \in A, \ a \ R \ a$
- **transitive** if and only if
  $\forall a, b, c \in A, \ a \ R \ b \ \wedge b \ R \ c \Rightarrow a \ R \ c$
- **symmetric** if and only if
  $\forall a, b \in A, \ a \ R \ b \Rightarrow b \ R \ a$
- **asymmetric** if and only if
  not $(\forall a, b \in A, \ a \ R \ b \Rightarrow b \ R \ a)$
- **antisymmetric** if and only
  $\forall a, b \in A, \ a \ R \ b \ \wedge b \ R \ a \Rightarrow a = b$

Definition 8.5 *An equivalence relation is a relation that is reflexive, transitive and symmetric.*

Definition 8.6 *A partial order relation is a relation that is reflexive, transitive and antisymmetric.*

[2] Subdomains are the counterpart of subsets.

8.3.3 Partially ordered sets

Definition 8.7 *A partially ordered set (*poset*) is a set P equipped with a partial order relation $\leq$ on P.*

The fact that the relation $\leq$ on P is a partial order implies that, for x, y and $z \in P$, it is:

1. reflexive : $x \leq x$
2. transitive: $x \leq y \wedge y \leq z \Rightarrow x \leq z$
3. antisymmetric: $x \leq y \wedge \ y \leq x \Rightarrow x = y$

Definition 8.8 *An upper bound in a poset P for an $X \subseteq P$ is a $u \in P$, such that $\forall x \in X, x \leq u$.*

Definition 8.9 *The least upper bound* $\mathbf{lub}_P\, X$ *in a poset P, for a subset $X \subseteq P$, is an upper bound u for X, such that $\forall\ v \in P$, if v is an upper bound for X, then $u \leq v$.*

If $X = \{x_0, x_1, \dots\}$ is a denumerable set, then **lub** X, the least upper bound of X, is written as $\mathbf{lub}_{i \geq 0}\, x_i$. The term **lub** X means $\mathbf{lub}_P\, X$, when P is understood in the context.

Definition 8.10 *The bottom element of a poset P, written as $\perp_{\mathrm{P}}$, has the properties:*

1. the bottom element $\perp_{\mathrm{P}} \in P$;
2. the bottom element is such that $\forall x \in P, \perp_{\mathrm{P}} \leq x$;
3. if a poset has a bottom element, then this bottom element is unique;
4. the bottom element $\perp_{\mathrm{P}}$ represents no information or nontermination.

Definition 8.11 *A set $X \subseteq P$ is chain in a poset P if the relation $\leq$ is a total order on X, i.e., $\forall x, y \in X,\ x \leq y \ \vee\ y \leq x$.*

If the denumerable set $\mathrm{X} = \{x_0, x_1, x_2, \cdots\}$ is a chain, then it can be written as $x_0 \leq x_1 \leq x_2 \leq \cdots$, and the length of a finite chain X is $|X|$, the cardinality of X.

A poset may have many chains, some finite and others infinite. If a poset does not have infinite chains, then it has a finite height, which is the length of its longest chain.

8.3.4 Complete partial order

Definition 8.12 *A complete partial order (cpo) is a poset P (under the partial ordering $\leq$) with a bottom element, and in which every chain has a least upper bound in P.*

All finite chain $x_0 \leq x_1 \leq x_2 \leq \cdots \leq x_n$ has a **lub** x_n, which is its last element, thus a poset of finite height and that has a bottom element is a cpo.

Consider, as an example of the process of determining that a poset is not a cpo, the poset $P = \{a, b, c, d, e\}$ with the partial order relation defined as: $a \leq b$, $a \leq c$, $b \leq d$, $b \leq e$ and $c \leq d$, pictorially represented by Fig. 8.1. Note that d and e are not related.

The chains in P are $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{a, b\}$, $\{a, c\}$, $\{b, e\}$, $\{b, d\}$, $\{c, d\}$, $\{a, b, e\}$, $\{a, b, d\}$ and $\{a, c, d\}$.

The bottom element of the poset P is a. And all these chains, except $\{b\}$ and $\{a, b\}$, have a least upper bound in P, which is the last element of each chain.

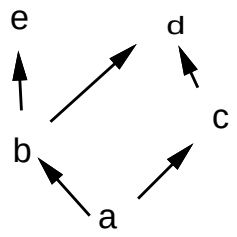

Figure 8.1: Partial order of a poset (from D. Martin's notes [23])

The elements d and e are upper bounds of the chains $\{b\}$ and $\{a, b\}$, but neither is a **lub** of these chains because they are unrelated. Therefore, according to definition 8.12, the poset P is a not cpo.

Theorem 8.2 *The empty poset is not a cpo.*

Proof: The empty poset does not have any element, so it cannot have a bottom element. □

8.4 Scott domains

Infinite mathematical entities need a finite representation to be computable, for computers operate on discret objects. Thus, in computing, either the objects are finite or they have a finite representation. For instance, $\pi, \{n^2 | n \geq 0\}, \lambda n \,.\, n + 2$ are infinite objects that must be treated in a finite way. Of course, infinite object cannot be totally computed in a finite time, so approximations must be used instead.

Several approximations of the same object O can be related by the relation $\sqsubseteq$, which reads *approximates*, such that $x \sqsubseteq y$ means that x has less information than y, or that y is a better approximation to O than x. Thus, approximations

are partially computed objects containing incomplete information. Each approximation may carry a different amount of information.

An infinite object O can be computed by a program by enumerating a sequence of better approximations, whose limit is the complete representation of O.

Consider, as an example, the infinite object defined by the quadratic function $f(x) = x^2,\ \forall x \geq 0,\ x \in \mathbb{N}$, and implemented by the program:

```
integer procedure p(x) is return x*x end
```

In general, during program execution, only a few pairs of the function graph must be computed, e.g., the function call `p(0)` produces `0`, the call `p(1)` returns 1, `p(2)` returns 4, etc.

Thus, successive calls to program `p` can compute, step by step, approximations of $f(x) = x^2,\ \forall x \geq 0,\ x \in \mathbb{N}$. For example:

1. `p(0)` produces `0`, thus `p` has computed the function graph
   $f = \{(0, 0)\}$
2. and then `p(2)` produces `4`, thus `p` has computed so far
   $f = \{(0, 0), (2, 4)\}$
3. and then `p(1)` produces `1`, thus `p` has computed so far
   $f = \{(0, 0), (1, 1), (2, 4)\}$
4. and then `p(3)` produces `9`, thus `p` has computed so far
   $f = \{(0, 0), (1, 1), (2, 4), (3, 9)\}$
5. and so forth, *ad infinitum*, `p` may be used to compute, stepwisely, more elements of the graph of the quadratic function.

Several graphs like this one may be computed, and they form sequences of better approximations of function f. One

chain of approximations could be:

$$\{(0,0)\} \sqsubseteq \{(0,0),(2,4)\} \sqsubseteq \{(0,0),(1,1),(2,4)\} \sqsubseteq$$
$$\{(0,0),(1,1),(2,4)(3,9)\} \sqsubseteq \cdots \{(x, x*x) \mid x \geq 0\}.$$

Consider now the problem of determining the function f that is implemented by the program `q` below, where `x` is an integer:

```
integer procedure q(x) is
   if x = 0 then 1 else x*q(x-1)
end
```

The first information that can be derived from this program is that at least the function $f = \{(0,1)\}$ is implemented by `q`. If $f = \{(0,1)\}$, then the second approximation could be the graph $f = \{(0,1),(1,1)\}$. The third approximation could be the graph $f = \{(0,1),(1,1),(2,2)\})$. The fourth approximation could then be $f = \{(0,1),(1,1),(2,2),(3,6)\}$.

And, therefore, the execution of `q` for other values of `x` enlarges the sequence of better approximations of f, such as:

$$\{(0,1)\} \sqsubseteq \{(0,1),(1,1)\} \sqsubseteq \{(0,1),(1,1),(2,2)\} \sqsubseteq$$
$$\{(0,1),(1,1),(2,2),(3,6)\} \sqsubseteq$$
$$\{(0,1),(1,1),(2,2),(3,6),(4,24)\} \sqsubseteq$$
$$\{(0,1),(1,1),(2,2),(3,6),(4,24),(5,120)\} \sqsubseteq \cdots$$

and provides, stepwisely, more precise definitions of the function. In fact, the above sequence of approximations suggests that its limit is the function $f = \{(x,y) \mid y = x! \ \wedge \ x \geq 0\}$.

This is a classical example, so it is easy to guess the limit of the sequence of approximations. Guessing like this may not be that easy for an arbitrary program.

However, the theory of domain comes to the rescue by mechanizing the process of finding this limit and provides:

- a special value to model nontermination, so that all semantic functions make sense in all cases;
- a partial order relation $\sqsubseteq$ to express the relationship among approximations;
- means to ensure the existence of limit in all approximation sequences;
- a method to compute this limit.

Definition 8.13 *A* **Scott domain** *is a cpo on which the partial order relation of approximations is* $\sqsubseteq$.

Henceforth, the word **domain** means Scott domain, and it denotes a complete partial order equipped with the partial order relation $\sqsubseteq$, and P, Q and R are, unless explicitly specified otherwise, assumed to be Scott domains, or simply domains, in which the respective partial orders are $\sqsubseteq_{\mathrm{P}}$, $\sqsubseteq_{\mathrm{Q}}$ and $\sqsubseteq_{\mathrm{R}}$, and the bottom elements are $\bot_{\mathrm{P}}$, $\bot_{\mathrm{Q}}$ and $\bot_{\mathrm{R}}$, respectively.

And, in order to keep the equations cleaner, frequently $\sqsubseteq$ is used instead of $\sqsubseteq_\alpha$, where α is the name of the domain that the context unambigously identifies as the one on which the relation $\sqsubseteq$ applies.

8.5 Flat domains

A domain can be constructed from an ordinary set A by creating a bottom element $\bot_{\mathrm{A}}$, such that $\bot_{\mathrm{A}} \notin \mathrm{A}$, and defining a new set $\mathrm{A}_\bot = A \cup \{\bot_{\mathrm{A}}\}$ equipped with a partial order relation $\sqsubseteq$ such that, for all $a, b \in \mathrm{A}_\bot$, $a \sqsubseteq b$, only and only if $a = \bot_{\mathrm{A}}$ or $a = b$. Domains constructed in this way are said to be *flat*.

The chains in $\mathrm{A}_\perp = \{\perp_\mathrm{A}, a_0, a_1, a_2, \cdots\}$ are $\{\perp_\mathrm{A}\}$, whose least upper bound is $\{\perp_\mathrm{A}\}$, $\{\perp_\mathrm{A}, a_i\}$, and $\{a_i\}$, for $i \geq 0$, whose least upper bounds are a_i.

Theorem 8.3 *The set* $A_\perp = A \cup \{\perp_\mathrm{A}\}$ *is a domain, for any set* A*, provided that, for all* $a, b \in A_\perp$*,* $a \sqsubseteq b$ *only and only if* $a = \perp_\mathrm{A} \vee \ a = b$*.*

Proof:
All chains in $\mathrm{A}_\perp$ have a least upper bound in $\mathrm{A}_\perp$, $\sqsubseteq$ on $\mathrm{A}_\perp$ is a partial order by definition, and the poset $\mathrm{A}_\perp$ has a bottom element by construction, therefore $\mathrm{A}_\perp$ is a domain. □

From flat domains, more complex domains can be constructed by the means of domain operators, such as cartesian product, union, sequencing and mappings.

8.6 Cartesian product

Definition 8.14 *The cartesian product* $P \times Q$ *is defined as*

$$P \times Q = \{(x, y) \mid x \in P, \ y \in Q\}$$

together with the partial order $\sqsubseteq_{\mathrm{P}\times\mathrm{Q}}$*, such that*

$$(x, y) \ \sqsubseteq_{\mathrm{P}\times\mathrm{Q}} (w, z), \text{ if and only if } x \sqsubseteq_\mathrm{P} w \wedge y \sqsubseteq_\mathrm{Q} z.$$

Theorem 8.4 *If* P *and* Q *are domains, the cartesian product* $P \times Q$ *is also a domain.*

Proof:

1. $P \times Q$ has a bottom element $<\perp_\mathrm{P}, \perp_\mathrm{Q}>$.
2. The partial order $\sqsubseteq_{\mathrm{P}\times\mathrm{Q}}$ is reflexive, antisymmetric and transitive because $\sqsubseteq_\mathrm{P}$ and $\sqsubseteq_\mathrm{Q}$ are.

3. All chains $(t_0, v_0) \sqsubseteq (t_1, v_1) \sqsubseteq \cdots$ in $P \times Q$ have a **lub** (t, v) in $P \times Q$, such that $t = \underset{i \geq 0}{\mathbf{lub}}\, t_i$ in P, and $v = \underset{i \geq 0}{\mathbf{lub}}\, v_i$ in Q.

4. Therefore, the cartesian product $P \times Q$ is a domain. $\square$

8.7 Domain union

Definition 8.15 *The separate union* $U = Q_1|Q_2|\cdots|Q_n$ *of domains* Q_i*, for* $1 \leq i \leq n$*, is defined as*

$U = \{\bot_{\mathrm{U}}\} \cup \{(q_j, j) \mid q_j \in Q_j, \textit{for integer } j, 1 \leq j \leq n\}$*, together with the partial order* $\sqsubseteq_{\mathrm{U}}$*, such that for all* k*,* $1 \leq k \leq n$*,* $\bot_{\mathrm{U}} \sqsubseteq_{\mathrm{U}} (\bot_{\mathrm{Q}_k}, k) \wedge (x, k) \sqsubseteq_{\mathrm{U}} (w, k)$*, if and only if* $x \sqsubseteq_{\mathrm{Q}_k} w$*.*

This definition assures that each member of the union corresponds to exactly one element of a union member. The tag associated with each element of the union allows inferring from which union member it comes.

Theorem 8.5 *The separate union* $U = Q_1|Q_2|\cdots|Q_n$ *of the domains* Q_i*, for* $1 \leq i \leq n$*, is also a domain.*

Proof:

1. By definition, the union U has a bottom element $\bot_{\mathrm{U}}$.

2. The relation $\sqsubseteq_{\mathrm{U}}$ is reflexive, antisymmetric and transitive, because $\sqsubseteq_{\mathrm{Q}_i}$, for $1 \leq i \leq n$, are partial order relations.

3. All chains $(t_0, k) \sqsubseteq (t_1, k) \sqsubseteq \cdots$ in U, for $t_0, t_1, \cdots \in Q_k$, and $1 \leq k \leq n$, have a **lub** (t, k) in U, such that $t = \underset{i \geq 0}{\mathbf{lub}}\, t_i$ in Q_k.

4. Therefore $U = Q_1|Q_2|\cdots|Q_n$ is a domain. $\square$

8.8 Domain of sequences

Definition 8.16 *$Q*$ is a poset defined as the union*
$Q* = \{\bot_{Q*}\} \cup Q \cup Q \times Q \cup Q \times Q \times Q \cup \cdots$, *together with the partial order $\sqsubseteq_{Q*}$, such that $\forall k$, $1 \leq k \leq n$,*

$$(\bot_{Q*} \sqsubseteq_{Q*} \bot_{Q}) \wedge (\bot_{Q*} \sqsubseteq_{Q*} \bot_{Q \times Q}) \wedge \cdots \wedge (\bot_{Q*} \sqsubseteq_{Q*} \bot_{Q \times Q \times \cdots Q}),$$

and that $x \sqsubseteq_{Q*} y$, if and only if $x \sqsubseteq_{R} y$, where R is a member of the union.

Theorem 8.6 *If Q is a domain, then $Q*$ is also a domain.*

Proof: $Q*$ is defined as a union of domains of different structure, and thus it is similar to a separate union, which, according to Theorem 8.5, is a domain. Therefore, the same reasoning steps used to show that separate unions are domains lead to the desired proof. □

8.9 Domain of functions

In addition to the cartesian product, union of domains and domain sequencing, new domains can be constructed by defining mapping between domains. In order to introduce this operation to construct more elaborated domains, the notion of monotonic and continuous functions is required.

8.9.1 Monotonic functions

Definition 8.17 *The function $f : P \rightarrow Q$ is monotonic, if, for all $p, p' \in P$, $p \sqsubseteq_{P} p' \Rightarrow f(p) \sqsubseteq_{Q} f(p')$.*

Proposition 8.1 *Let $p_0 \sqsubseteq_{\mathrm{P}} p_1 \sqsubseteq_{\mathrm{P}} p_2 \sqsubseteq_{\mathrm{P}} \cdots$ be a chain in P, and f, a monotonic function, then $f(p_0) \sqsubseteq_{\mathrm{Q}} f(p_1) \sqsubseteq_{\mathrm{Q}} \cdots$ is a chain in Q.*

Proof:

1. f is monotonic, so $p_i \sqsubseteq_{\mathrm{P}} p_{i+1} \Rightarrow f(p_i) \sqsubseteq_{\mathrm{Q}} f(p_{i+1})$, for $i \geq 0$.

2. $p_0 \sqsubseteq_{\mathrm{P}} p_1 \sqsubseteq_{\mathrm{P}} p_2 \sqsubseteq_{\mathrm{P}} \cdots$ is a chain in P, and the partial orders $\sqsubseteq_{\mathrm{P}}$ and $\sqsubseteq_{\mathrm{Q}}$ are transitive relations, then $f(p_0) \sqsubseteq_{\mathrm{Q}} f(p_1) \sqsubseteq_{\mathrm{Q}} \cdots$ is a chain in Q. □

8.9.2 Continuous functions

The notion of continuous functions is also essential to provide a method to solve recursive equations.

Definition 8.18 *The one-argument function $f : P \to Q$ is continuous with respect to its argument, if, for all nonempty chain $p_0 \sqsubseteq p_1 \sqsubseteq \cdots$ in P, the following conditions hold:*

1. *$f(p_0) \sqsubseteq f(p_1) \sqsubseteq \cdots$ is a chain in Q*
2. *$f(\underset{i \geq 0}{\mathbf{lub}}\, p_i) = \underset{i \geq 0}{\mathbf{lub}}\, f(p_i)$*

Henceforth, $P \mapsto Q$ will denote the domain of continuous functions from P to Q.

Proposition 8.2 *All continuous functions are monotonic.*

Proof:
Let $p,\ p' \in \mathrm{P}$, such that $p \sqsubseteq p'$, and let $f : P \mapsto Q$, then, by definition 8.18, $f(p) \sqsubseteq f(p')$ is a chain in Q. Therefore, f is monotonic. □

Proposition 8.3 *Let P be a domain of finite height, and Q be domain of any height. If function $f : P \to Q$ is monotonic, then f is continuous.*

Proof:

1. Let $p_0 \sqsubseteq p_1 \sqsubseteq \cdots \sqsubseteq p_n$ be a chain in P.
2. Since f is monotonic, $f(p_0) \sqsubseteq f(p_1) \sqsubseteq \cdots \sqsubseteq f(p_n)$ is, according to preposition 8.1, a chain in Q.
3. $f(\mathop{\mathbf{lub}}_{0 \leq i \leq n} p_i) = f(p_n) = \mathop{\mathbf{lub}}_{0 \leq i \leq n} f(p_i)$ □

Proposition 8.4 *Let $f : P \mapsto Q$ and $g : Q \mapsto R$. Then function composition $g \ o \ f : P \to R$ is continuous, i.e., $g \ o \ f : P \mapsto R$.*

Proof:

1. Let $p_0 \sqsubseteq p_1 \sqsubseteq \cdots$ be a chain in P.
2. f is continuous $\Rightarrow f(p_0) \sqsubseteq f(p_1) \sqsubseteq \cdots$ is a chain in Q.
3. g is continuous $\Rightarrow g(f(p_0)) \sqsubseteq g(f(p_1)) \sqsubseteq \cdots$ is a chain in R.
4. Hence, $g(f(\mathop{\mathbf{lub}}_{i \geq 0} p_i)) = g(\mathop{\mathbf{lub}}_{i \geq 0} f(p_i))$, because f is continuous.
5. Hence, $g(\mathop{\mathbf{lub}}_{i \geq 0} f(p_i)) = \mathop{\mathbf{lub}}_{i \geq 0} g(f(p_i))$, because g is continuous.
6. Hence, $g(f(\mathop{\mathbf{lub}}_{i \geq 0} p_i)) = \mathop{\mathbf{lub}}_{i \geq 0} g(f(p_i))$
7. Therefore, $g \ o \ f$ is a continuous function. □

Definition 8.19 *A function $f : P \times Q \to R$ is continuous in its first argument if and only if, $\forall x \in Q$:*

1. *for each chain $p_0 \sqsubseteq p_1 \sqsubseteq \cdots$ in P,*
$$f(p_0, x) \sqsubseteq f(p_1, x) \sqsubseteq \cdots \text{ is a chain in } R$$
2. $f(\mathop{\mathbf{lub}}_{i \geq 0} p_i, x) = \mathop{\mathbf{lub}}_{i \geq 0} f(p_i, x)$

Theorem 8.7 *A multi-argument function is continuous if and only if it is continuous in each of its individual arguments.*

Proof: The definition 8.19 can be extended to cover continuity of a function in its second argument, or to other arguments, so as to establish this theorem [23]. □

8.9.3 Domain mapping

Definition 8.20 *Let $P \mapsto Q$ be a set of functions together with the partial order relation $\sqsubseteq_{\mathrm{P}\mapsto\mathrm{Q}}$, such that, $\forall f, g \in P \mapsto Q$, $f \sqsubseteq_{\mathrm{P}\mapsto\mathrm{Q}} g$, when $f(x) \sqsubseteq_{\mathrm{Q}} g(x), \forall x \in P$.*

Lemma 8.1 *Let $f_0 \sqsubseteq f_1 \sqsubseteq \cdots$ be a chain in $P \mapsto Q$. Then, for all $p \in P$, $(\mathop{\mathbf{lub}}_{i \geq 0} f_i)(p) = \mathop{\mathbf{lub}}_{i \geq 0} f_i(p)$.*

Proof:

1. Since $P \mapsto Q$ is a cpo, and $f_0 \sqsubseteq f_1 \sqsubseteq \cdots$ is a chain in $P \mapsto Q$, then there exists $f = \mathop{\mathbf{lub}}_{i \geq 0} f_i$, $f \in P \mapsto Q$.
2. From the definition of $\sqsubseteq_{\mathrm{P}\mapsto\mathrm{Q}}$, and that $f_0 \sqsubseteq f_1 \sqsubseteq \cdots$ is a chain in $P \mapsto Q$, $f_0(p) \sqsubseteq f_1(p) \sqsubseteq \cdots$, for any $p \in P$, is a chain in Q.
3. Since Q is a cpo, all of its chain has a least upper bound, so, there exists a $q \in Q$, such that $q = \mathop{\mathbf{lub}}_{i \geq 0} f_i(p)$.

4. Since $f(p) = (\mathop{\mathbf{lub}}_{i \geq 0} f_i)(p)$, $p \in P$, is an upper bound for $f_0(p) \sqsubseteq f_1(p) \sqsubseteq \cdots$, then $q = \mathop{\mathbf{lub}}_{i \geq 0} f_i(p) \sqsubseteq (\mathop{\mathbf{lub}}_{i \geq 0} f_i)(p)$.
5. On the other hand, considering the function $g : P \mapsto Q$, defined as $g(p) = \mathop{\mathbf{lub}}_{i \geq 0} f_i(p)$, then $f_i(p) \sqsubseteq g(p)$, for all $p \in P$, implies that $f_i \sqsubseteq g$, for $i \geq 0$, and therefore g is an upper bound for the chain $f_0 \sqsubseteq f_1 \sqsubseteq \cdots$.
6. From items (1), (2) and (3), $f(p) = (\mathop{\mathbf{lub}}_{i \geq 0} f_i)(p) \sqsubseteq \mathop{\mathbf{lub}}_{i \geq 0} f_i(p)$.
7. Item (4) establishes that $\mathop{\mathbf{lub}}_{i \geq 0} f_i(p) \sqsubseteq (\mathop{\mathbf{lub}}_{i \geq 0} f_i)(p)$, item (6) shows that $(\mathop{\mathbf{lub}}_{i \geq 0} f_i)(p) \sqsubseteq \mathop{\mathbf{lub}}_{i \geq 0} f_i(p)$, and relation $\sqsubseteq$ is reflexive, therefore: $(\mathop{\mathbf{lub}}_{i \geq 0} f_i)(p) = \mathop{\mathbf{lub}}_{i \geq 0} f_i(p)$ □

Lemma 8.2 *Let $f_0 \sqsubseteq f_1 \sqsubseteq \cdots$ be a chain in $P \mapsto Q$, where P and Q are domains, and $p_0 \sqsubseteq p_1 \sqsubseteq \cdots$ be a chain in P. Hence, $\mathop{\mathbf{lub}}_{i \geq 0} (\mathop{\mathbf{lub}}_{j \geq 0} f_i(p_j)) = \mathop{\mathbf{lub}}_{j \geq 0} (\mathop{\mathbf{lub}}_{i \geq 0} f_i(p_j))$.*

Proof:

1. Let $f = \mathop{\mathbf{lub}}_{i \geq 0} f_i$ and $p = \mathop{\mathbf{lub}}_{j \geq 0} p_j$ be the least upper bounds of their respective chains.
2. Functions f_i, for $i \geq 0$, are continuous by hypothesis, then $\mathop{\mathbf{lub}}_{i \geq 0} (\mathop{\mathbf{lub}}_{j \geq 0} f_i(p_j)) = \mathop{\mathbf{lub}}_{i \geq 0} (f_i(\mathop{\mathbf{lub}}_{j \geq 0} p_j))$
3. Since $p = \mathop{\mathbf{lub}}_{j \geq 0} p_j$, then $\mathop{\mathbf{lub}}_{i \geq 0} (f_i(\mathop{\mathbf{lub}}_{j \geq 0} p_j) = \mathop{\mathbf{lub}}_{i \geq 0} (f_i(p))$,
4. Removing the unnecessary parenthesis,
   $\mathop{\mathbf{lub}}_{i \geq 0} (f_i(p)) = \mathop{\mathbf{lub}}_{i \geq 0} f_i(p)$
5. According to Lemma 8.1, $\mathop{\mathbf{lub}}_{i \geq 0} f_i(p) = (\mathop{\mathbf{lub}}_{i \geq 0} f_i)(p)$
6. Hence, $(\mathop{\mathbf{lub}}_{i \geq 0} f_i)(p) = f(p)$

7. Therefore, $\underset{i \geq 0}{\mathbf{lub}}(\underset{j \geq 0}{\mathbf{lub}} f_i(p_j)) = f(p)$
8. On the other hand, $\underset{j \geq 0}{\mathbf{lub}}(\underset{i \geq 0}{\mathbf{lub}} f_i(p_j)) = \underset{j \geq 0}{\mathbf{lub}}(f(p_j))$
9. From the continuity of f, $\underset{j \geq 0}{\mathbf{lub}}(f(p_j)) = f(\underset{j \geq 0}{\mathbf{lub}}(p_j))$
10. Therefore, $\underset{j \geq 0}{\mathbf{lub}}(\underset{i \geq 0}{\mathbf{lub}} f_i(p_j)) = f(p)$ □

Theorem 8.8 *The set $P \mapsto Q$, together with a partial order relation $\sqsubseteq_{P \mapsto Q}$ defined as*

$$f \sqsubseteq_{P \mapsto Q} g \stackrel{\triangle}{=} (\forall p \in P)\ (f(p) \sqsubseteq_{\mathrm{Q}} g(p))$$

and a bottom element $\bot_{P \mapsto Q} \in P \mapsto Q$ defined as

$$(\forall p \in P)(\bot_{P \mapsto Q}(p) = \bot_{\mathrm{Q}}),$$

is also a domain.

Proof:

1. This theorem requires that the order relation $\sqsubseteq_{\mathrm{P} \mapsto \mathrm{Q}}$ must be partial, that the value $\bot_{\mathrm{P} \mapsto \mathrm{Q}}$ be indeed the bottom element, and all chains in $P \mapsto Q$ have a least upper bound in $P \mapsto Q$.
2. The relation $\sqsubseteq_{\mathrm{P} \mapsto \mathrm{Q}}$ is reflexive, transitive and antisymmetric because $\sqsubseteq_{\mathrm{Q}}$ is.
3. Note that function $\bot_{\mathrm{P} \mapsto \mathrm{Q}}$ is continuous, and $\forall f \in P \mapsto Q$, $\bot_{\mathrm{P} \mapsto \mathrm{Q}} \sqsubseteq f$.
4. If $P \mapsto Q$ is a domain, all chains in $P \mapsto Q$ must have a least upper bound in $P \mapsto Q$.
5. Let $f_0 \sqsubseteq f_1 \sqsubseteq \cdots$ be a chain in $P \mapsto Q$. From the definition of $\sqsubseteq_{\mathrm{P} \mapsto \mathrm{Q}}$ (Theorem 8.8), $f_0(p) \sqsubseteq f_1(p) \sqsubseteq \cdots$ is a chain in Q, for each $p \in P$.

6. Q is a domain, therefore $\underset{i \geq 0}{\mathbf{lub}}\, f_i(p)$ exists.
7. From Lemma 8.1, $(\underset{i \geq 0}{\mathbf{lub}}\, f_i)(p) = \underset{i \geq 0}{\mathbf{lub}}\, f_i(p)$, and as $\underset{i \geq 0}{\mathbf{lub}}\, f_i(p)$ exists, then $\underset{i \geq 0}{\mathbf{lub}}\, f_i$ also exists, i.e., all chains have a least upper bound.
8. The next step is to show that the least upper bound of each chain in $P \mapsto Q$ is in $P \mapsto Q$.
9. Let $p_0 \sqsubseteq p_1 \sqsubseteq \cdots$ be a chain in P, then, from Lemma 8.1, $(\underset{i \geq 0}{\mathbf{lub}}\, f_i)(\underset{j \geq 0}{\mathbf{lub}} p_j) = \underset{i \geq 0}{\mathbf{lub}}\, (f_i(\underset{j \geq 0}{\mathbf{lub}} p_j))$.
10. As f_i is a continuous function,
    $\underset{i \geq 0}{\mathbf{lub}}\, (f_i(\underset{j \geq 0}{\mathbf{lub}} p_j)) = \underset{i \geq 0}{\mathbf{lub}}\, (\underset{j \geq 0}{\mathbf{lub}}\, f_i(p_j))$.
11. So far: $(\underset{i \geq 0}{\mathbf{lub}}\, f_i)(\underset{j \geq 0}{\mathbf{lub}} p_j) = \underset{i \geq 0}{\mathbf{lub}}\, (\underset{j \geq 0}{\mathbf{lub}}\, f_i(p_j))$
12. $p_j \sqsubseteq p_{j+1}$ holds and f_i is monotonic, then $f_i(p_j) \sqsubseteq f_i(p_{j+1})$.
13. As $f_i \sqsubseteq_{\mathrm{P} \mapsto \mathrm{Q}} f_{i+1}$, then $f_i(p_j) \sqsubseteq_{\mathrm{Q}} f_{i+1}(p_j)$.
14. From Lemma 8.2, $\underset{i \geq 0}{\mathbf{lub}}\, (\underset{j \geq 0}{\mathbf{lub}}\, f_i(p_j)) = \underset{j \geq 0}{\mathbf{lub}}\, (\underset{i \geq 0}{\mathbf{lub}}\, f_i(p_j))$.
15. From Lemma 8.1, $\underset{j \geq 0}{\mathbf{lub}}\, (\underset{i \geq 0}{\mathbf{lub}}\, f_i(p_j)) = \underset{j \geq 0}{\mathbf{lub}}\, ((\underset{i \geq 0}{\mathbf{lub}}\, f_i)(p_j))$.
16. Hence, $(\underset{i \geq 0}{\mathbf{lub}}\, f_i)(\underset{j \geq 0}{\mathbf{lub}} p_j) = \underset{j \geq 0}{\mathbf{lub}}\, ((\underset{i \geq 0}{\mathbf{lub}}\, f_i)(p_j))$
17. Therefore, $\underset{i \geq 0}{\mathbf{lub}}\, f_i$ is a continuous function.
18. And hence, $\underset{i \geq 0}{\mathbf{lub}}\, f_i \in P \mapsto Q$.
19. And $P \mapsto Q$ is a poset in which all chains have a least upperbound in $P \mapsto Q$.
20. Therefore, $P \mapsto Q$ is a domain, according to its definition. $\square$

8.9.4 Composition of functions

Theorem 8.9 *If $f : P \mapsto Q$ and $g : Q \mapsto R$, for domains P, Q and R, then $g\ o\ f : Q \to R$ is a continuous function.*

Proof:

1. Let $p_0 \sqsubseteq p_1 \sqsubseteq \cdots$ be a chain in domain P.
2. Since f and g are continuous, then
   $f(p_0) \sqsubseteq f(p_1) \sqsubseteq \cdots$ is a chain in Q, and
   $g(f(p_0)) \sqsubseteq g(f(p_1)) \sqsubseteq \cdots$ is a chain in R.
3. Since f is continuous, then $g(f(\mathop{\mathbf{lub}}_{i \geq 0} p_i)) = g(\mathop{\mathbf{lub}}_{i \geq 0} f(p_i))$
4. Since g is continuous, then $g(\mathop{\mathbf{lub}}_{i \geq 0} f(p_i)) = \mathop{\mathbf{lub}}_{i \geq 0} g(f(p_i))$
5. Therefore, $f\ o\ g$ is continuous, so,
   $g(f(\mathop{\mathbf{lub}}_{i \geq 0} f(p_i)) = \mathop{\mathbf{lub}}_{i \geq 0} g(f(p_i))$. □

8.10 LAMBDA functions

Definition 8.21 *Dana Scott's language* LAMBDA*[46] is an extension of Church's λ-calculus [11] that includes integers, conditionals and the operations* + *and* - *on integers, and whose syntax is:*

$$\begin{array}{rcl} L & ::= & L(L) \mid L\ L \mid \lambda x.L \mid L \to L, L \\ & \mid & x \quad \mid 0 \quad \mid x+1 \mid x-1 \end{array}$$

where x is a variable denoted by an identifier.

Theorem 8.10 *All* LAMBDA*-definable functions are continuous.*

Proof: The proof can be found in Dana Scott's paper [46]. □

8.11 $\mathcal{M}$ functions

$\mathcal{M}$ is a pure functional language of the λ-calculus family. It features a high-level module structure, all of its syntactic constructions can be mapped to basic λ-expressions, and all of its data types are defined as domains. Thus, the Scott's theorem 8.10 can be adapted for the meta-language $\mathcal{M}$ of Chapter 1 to become the following theorem:

Theorem 8.11 *All $\mathcal{M}$-definable functions are continuous.*

Proof:

1. The basic types of $\mathcal{M}$, i.e., `N`, `T`, `Q`, `Nonterminal`, `File`, `Token`, `Sart` and `?`, are flat domains by definition.
2. The domain operations of $\mathcal{M}$ for domain tupling, domain sequencing, domain union, and domain mapping, correspond to the cartesian product, sequencing, separate union and domain mapping operations of the Scott's theory, thus it is straightforward to show that these operations also preserve the domain structure of $\mathcal{M}$.
3. The rest of the proof consists in showing how $\mathcal{M}$-functions are translated into LAMBDA-expressions. This translation is a quite straightforward process, although long and tedious, and for this reason this proof has been omitted. □

This theorem establishes that: (i) all functions that can be written in $\mathcal{M}$ are naturally continuous; (ii) the fixpoints of $\mathcal{M}$-functions can always be computed; (iii) and all $\mathcal{M}$-equations are sound.

8.12 Fixpoints

The solution of recursively defined equations for domains and functions can be computed via the fixed point (or fixpoint) operator μ_{P}, which operates in the realm of **domains** and **continuous functions**.

Definition 8.22 *The function* $\mu_P : (P \mapsto P) \mapsto P$ *defined as*

$$\mu_P(f) \stackrel{\triangle}{=} \mathop{\mathbf{lub}}_{i \geq 0} f^i(\bot_{\mathrm{P}}),$$

such that

$$f \in P \mapsto P,\ f^0(\bot_{\mathrm{P}}) \stackrel{\triangle}{=} \bot_{\mathrm{P}}$$

and

$$f^{i+1}(\bot_{\mathrm{P}}) \stackrel{\triangle}{=} f(f^i(\bot_{\mathrm{P}})),\ \forall i \geq 0$$

has the following properties:

1. μ_P *is continuous.*
2. $f(\mu_P(f)) = \mu_P(f)$, *i.e.,* $\mu_P(f)$ *computes fixpoint of* f.
3. $\forall p \in P,\ f(p) = p$ *implies that* $\mu_P(f) \sqsubseteq_{\mathrm{P}} p$, *i.e.,* $\mu_P(f)$ *is the least fixed point of* f.

Proposition 8.5 *Let* $f \in P \mapsto P$. *Then*

$$f^0(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} f(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} f^2(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} f^3(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} \cdots$$

is a chain in P.

Proof:

1. From Theorem 8.11: $f^0(\bot_{\mathrm{P}}) = \bot_{\mathrm{P}}$.
2. From the definition of bottom: $\bot_{\mathrm{P}} \sqsubseteq_{\mathrm{P}} f(\bot_{\mathrm{P}})$.
3. From the fact that f is monotonic: $f(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} f(f(\bot_{\mathrm{P}}))$
4. Thus, $\bot_{\mathrm{P}} \sqsubseteq_{\mathrm{P}} f(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} f^2(\bot_{\mathrm{P}})$.

5. Repeating the argument:
   $\bot_{\mathrm{P}} \sqsubseteq_{\mathrm{P}} f(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} f^2(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} f^3(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} \cdots$ □

Proposition 8.6 *The operator $\mu_P : (P \mapsto P) \mapsto P$ computes a fixed point of $f \in P \mapsto P$, i.e., $f(\mu_P(f)) = \mu_P(f)$.*

Proof:

1. From the definition of fixpoint:
   $$f(\mu_{\mathrm{P}}(f)) = f(\mathbf{lub}_{i \geq 0} f^i(\bot_{\mathrm{P}}))$$

2. Since f is a continuous function:
   $$\begin{aligned} f(\mu_{\mathrm{P}}(f)) &= \mathbf{lub}_{i \geq 0} f(f^i(\bot_{\mathrm{P}})) \\ &= \mathbf{lub}_{i \geq 0} f^{i+1}(\bot_{\mathrm{P}}) \\ &= \mathbf{lub}_{i \geq 0} f^i(\bot_{\mathrm{P}}) \\ &= \mu_{\mathrm{P}}(f) \end{aligned}$$ □

Proposition 8.7 *The operator $\mu_P : (P \mapsto P) \mapsto P$ computes the least fixed point of $f \in P \mapsto P$, i.e., $\forall p \in P$, $f(p) = p$ implies $\mu_P(f) \sqsubseteq_{\mathrm{P}} p$.*

Proof:

1. Let $p = f(p)$, i.e., p is a fixed point of f.

2. Since P is a domain and $p \in \mathrm{P}$, $\bot_{\mathrm{P}} \sqsubseteq_{\mathrm{P}} p$

3. $f \in [\mathrm{P} \mapsto \mathrm{P}]$ is continuous, hence monotonic:
   $\Rightarrow \bot_{\mathrm{P}} \sqsubseteq_{\mathrm{P}} p \Rightarrow f(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} f(p) \Rightarrow f(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} p$
   $\Rightarrow f(f(\bot_{\mathrm{P}})) \sqsubseteq_{\mathrm{P}} f(p) \Rightarrow f^2(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} p$
   $\Rightarrow f(f^2(\bot_{\mathrm{P}})) \sqsubseteq_{\mathrm{P}} f(p) \Rightarrow f^3(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} p \cdots$
   $\Rightarrow f^i(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} p \cdots \Rightarrow \mathbf{lub}_{i \geq 0} f^i(\bot_{\mathrm{P}}) \sqsubseteq_{\mathrm{P}} p$

4. Therefore, $\mu_{\mathrm{P}}(f) \sqsubseteq_{\mathrm{P}} p$, $\forall p \in P$ □

Definition 8.23 *The solutions of the equation $x = f(x)$ are the fixpoints of function f.*

Definition 8.24 *If there two solutions x_1 and x_2 of the equation $x = f(x)$, $x_1 \sqsubseteq x_2$ means that x_1 is an approximation of x_2 or that solution x_1 has less information than x_2.*

Definition 8.25 *The least solution of the equation $x = f(x)$ is the least fixpoint of f.*

Definition 8.26 *If P is a domain and $f \in P \mapsto P$, then*
$$x = \mu_P(f) = \mathop{\mathbf{lub}}_{i \geq 0} f^i(\perp_{\mathrm{P}})$$
is the least fixpoint of the equation $x = f(x)$.

8.12.1 Calculation of fixpoints

Let $\mathbb{N}$ be the domain of natural numbers, Bool = {**true**, **false**}, the domain of boolean values, and let $\mathbb{N}_\perp$ and $\mathrm{Bool}_\perp$ be the corresponding flat domains.

Fixpoint of the factorial equation

Consider, as an example, $f \in \mathbb{N}_\perp \mapsto \mathbb{N}_\perp$ defined by
$$f = \lambda n \,.\, n = 0 \to 1, n * f(n-1), \text{ for } n \geq 0,$$
which can be rewritten as
$$f = H(f), \text{ for } H \in (\mathbb{N}_\perp \mapsto \mathbb{N}_\perp) \mapsto (\mathbb{N}_\perp \mapsto \mathbb{N}_\perp), \text{ and}$$
$$H(f) = \lambda n \,.\, n = 0 \to 1, n * f(n-1), \text{ for } n \geq 0,$$
The domain theory establishes that the least fixpoint of H is given by equation $f = \mathop{\mathbf{lub}}_{i \geq 0} H^i(\perp)$.

In order to provide the necessary basis for computing the solution of H, the following primitive operations, in which $x, y, z \in \mathrm{N}_\perp$ and $b \in \mathrm{Bool}_\perp$, are assumed available:

- $x = 0 \stackrel{\triangle}{=} \begin{cases} \bot_{\text{Bool}} & \text{if } x \; is \; \bot_{\text{N}} \\ \textbf{true} & \text{if } x \; is \; 0 \\ \textbf{false} & \text{otherwise} \end{cases}$

- $x * y \stackrel{\triangle}{=} \begin{cases} \bot_{\mathbb{N}} & \text{if } x \text{ or } y \text{ is } \bot_{\mathbb{N}} \\ x * y & \text{as usual with } \mathbb{N} \end{cases}$

- $x - y \stackrel{\triangle}{=} \begin{cases} \bot_{\mathbb{N}} & \text{if } x \text{ or } y \text{ are } \bot_{\mathbb{N}} \\ 0 & \text{if } x \leq y \\ x - y & \text{as usual with } \mathbb{N} \end{cases}$

- $b \rightarrow \; y, z \stackrel{\triangle}{=} \begin{cases} \bot_{\text{Bool}} & \text{if } b = \bot_{\text{Bool}} \\ y & \text{if } b = \textbf{true} \\ z & \text{if } b = \textbf{false} \end{cases}$

A step by step computation of the fixpoint of the recursive factorial equation is given by the formula:

$f = \underset{i \geq 0}{\textbf{lub}} \, H^i(\bot)$, where $H(f) = \lambda n \,.\, n = 0 \rightarrow 1, n * f(n-1)$

To find f the limit of the chain $\bot \sqsubseteq H^1(\bot) \sqsubseteq H^2(\bot) \sqsubseteq \cdots$ may be computed as follows:

1. $H^0(\bot) = \bot$

2. $\begin{aligned} H^1(\bot) &= \lambda n \,.\, n = 0 \rightarrow 1, n * \bot(n-1) \\ &= \lambda n \,.\, n = 0 \rightarrow 1, \bot \end{aligned}$

3. $\begin{aligned} H^2(\bot) &= \lambda n \,.\, n = 0 \rightarrow 1, n * H(\bot)(n-1) \\ &= \lambda n \,.\, n = 0 \rightarrow 1, n * (n - 1 = 0 \rightarrow 1, \bot) \\ &= \lambda n \,.\, n = 0 \rightarrow 1, n * (n = 1 \rightarrow 1, \bot) \\ &= \lambda n \,.\, n = 0 \rightarrow 1, n = 1 \rightarrow n, \bot \end{aligned}$

4. $H(f) = \lambda n \,.\, n = 0 \rightarrow 1, n * f(n-1)$
   $H^2(\bot) = \lambda n \,.\, n = 0 \rightarrow 1, n = 1 \rightarrow n, \bot$

5. $H^3(\bot) = H(H^2(\bot))$
$= \lambda n \,.\, n = 0 \to 1, n * H^2(\bot)(n-1)$
$= \lambda n \,.\, n = 0 \to 1,$
$\quad n * (n - 1 = 0 \to 1, n - 1 = 1 \to n - 1, \bot)$
$= \lambda n \,.\, n = 0 \to 1,$
$\quad n * (n = 1 \to 1, n = 2 \to n - 1, \bot)$
$= \lambda n \,.\, n = 0 \to 1,$
$\quad n = 1 \to n, n = 2 \to n * (n - 1), \bot$

6. $H(f) = \lambda n \,.\, n = 0 \to 1, n * f(n-1)$
$H^3(\bot) = \lambda n \,.\, n = 0 \to 1,$
$\quad n = 1 \to n, n = 2 \to n * (n - 1), \bot$

7. $H^4(\bot) = H(H^3(\bot))$
$= \lambda n \,.\, n = 0 \to 1, n * H^3(\bot)(n-1)$
$= \lambda n \,.\, n = 0 \to 1,$
$\quad n * (n - 1 = 0 \to 1, n - 1 = 1 \to n - 1,$
$\quad\quad n - 1 = 2 \to (n - 1) * (n - 1 - 1), \bot)$
$= \lambda n \,.\, n = 0 \to 1, n * (n = 1 \to 1, n = 2 \to n - 1,$
$\quad n = 3 \to (n - 1) * (n - 2), \bot)$
$= \lambda n \,.\, n = 0 \to 1, n = 1 \to n, n = 2 \to n * (n-1),$
$\quad n = 3 \to n * (n - 1) * (n - 2), \bot$
$H^4(\bot) = \lambda n \,.\, 0 \leq n < 4 \to n!, \bot$

8. $H(f) = \lambda n \,.\, n = 0 \to 1, n * f(n-1)$
$H^4(\bot) = \lambda n \,.\, 0 \leq n < 4 \to n!, \bot)$

9. $H^k(\bot) = \lambda n \,.\, 0 \leq n < k \to n!, \bot)$

10. $\bot \sqsubseteq H^1(\bot) \sqsubseteq H^2(\bot) \sqsubseteq \cdots$
is a chain of approximate solutions, whose limit is the solution of $f = H(f)$.

11. For $k \to \infty$: $H^k(\bot) \to \lambda n \,.\, n!$, which is the least fixpoint of $f = H(f)$.

Fixpoint of a circular equation

1. Consider the equation $f = H(f)$, such that
   $H(f) = \lambda x \,.\, (x = 0) \to 1,\ x * f(x)$, for $x \geq 0$,
2. The least solution of $f = H(f)$ computed by $f = \mathop{\mathbf{lub}}_{i \geq 0} H^i(\bot)$ is $f = \lambda x \,.\, (x = 0) \to 1, \bot_{\mathbb{N}}$, for $x \geq 0$
3. Note that:
   - $\bot_{\mathbb{N}}$ models the circularity of the equation.
   - f is a total function, although from the operational viewpoint it is partial.

Fixpoint of an unsolvable equation

1. Consider now the equation $f = H(f)$, such that
   $H(f) = \lambda x \,.\, (x = 0) \to 1,\ f(x + 1)$, for $x \geq 0$,
2. H has an infinite number of fixpoints $f_\bot$ and $f_k, \forall k \geq 0$, defined as:
   - $f_\bot = \lambda x \,.\, (x = 0) \to 1, \bot_{\mathbb{N}}$
   - $f_k = \lambda x \,.\, (x = 0) \to 1,\ k$
3. It is easy to show that $\forall k \geq 0, \quad f_\bot \sqsubseteq f_k$.
4. Therefore, $f_\bot$ is the least fixpoint of H, that is,
   $f_\bot = \mathop{\mathbf{lub}}_{i \geq 0} H^i(\bot_{\mathbb{N}_\bot \mapsto \mathbb{N}_\bot})$

The calculation of $f_\bot$ is as follows:

1. Let $f = H(f)$, where $H(f) = \lambda x \,.\, x = 0 \to 1, f(x + 1)$
2. $H^0(\bot) = \bot$

3. $H^1(\bot) = \lambda x \,.\, x = 0 \to 1, \bot(x+1)$
$= \lambda x \,.\, x = 0 \to 1, \bot$

4. $H^2(\bot) = \lambda x \,.\, x = 0 \to 1, H^1(\bot)(x+1)$
$= \lambda x \,.\, x = 0 \to 1, x+1 = 0 \to 1, \bot$
$= \lambda x \,.\, x = 0 \to 1, \bot$

5. $H^2(\bot) = \lambda x \,.\, x = 0 \to 1, \bot$

6. $H^3(\bot) = \lambda x \,.\, x = 0 \to 1, H^2(\bot)(x+1)$
$= \lambda x \,.\, x = 0 \to 1, x+1 = 0 \to 1, \bot$
$= \lambda x \,.\, x = 0 \to 1, \bot$

7. $H^k(\bot) = \lambda x \,.\, x = 0 \to 1, H^{k-1}(\bot)(x+1)$ $\qquad \forall k \geq 1$
$= \lambda x \,.\, x = 0 \to 1, x+1 = 0 \to 1, \bot$
$= \lambda x \,.\, x = 0 \to 1, \bot$

8. Therefore,
$$f_\bot = \mathop{\mathbf{lub}}_{i \geq 0} H^i(\bot_{\mathbb{N}_\bot \mapsto \mathbb{N}_\bot}) \text{ or } f_\bot = \lambda x \,.\, x = 0 \to 1, \bot$$

The functions $f_k = \lambda x \,.\, (x = 0) \to 1,\ k$, $\forall k \geq 0$, are also fixpoints of $f = H(f)$, where
$$H(f) = \lambda x \,.\, x = 0 \to 1, f(x+1),$$
because
$$\begin{aligned} H(f_k) &= \lambda x \,.\, x = 0 \to 1, f_k(x+1) \\ &= \lambda x \,.\, x = 0 \to 1, (x+1 = 0 \to 1, k) \\ &= \lambda x \,.\, x = 0 \to 1, k \\ &= f_k \end{aligned}$$
On the other hand, $f_\bot \sqsubseteq f_k, \forall k \geq 1$, because
$$f_\bot = \{\bot, (0,1)\} \sqsubseteq f_k = \{\bot, (0,1), (1,k), (2,k), \cdots\}$$

8.12.2 The paradoxal operator Y

The fixpoint of LAMBDA-expressions or $\mathcal{M}$-expressions can be computed by means of Curry's paradoxal combinator Y, which is defined by the following theorem:

Theorem 8.12 *All λ-expressions H have a fixpoint $Y\ H$, where Y is the paradoxal operator defined as*

$$Y = \lambda h\,.\,(\lambda x\,.\,h(x\ x))(\lambda x\,.\,h(x\ x))$$

Proof:

$$\begin{aligned} Y\ H &= (\lambda h\,.\,(\lambda x\,.\,h(x\ x))(\lambda x\,.\,h(x\ x)))H \\ &= (\lambda x\,.\,H(x\ x))(\lambda x\,.\,H(x\ x))) \\ &= H((\lambda x\,.\,H(x\ x))(\lambda x\,.\,H(x\ x))) \\ &= H(Y\ H) \end{aligned}$$

Hence, YH computes a fixpoint of H. □

8.13 Final comments

This chapter presents elementary aspects of Dana Scott's theory of domains and continuous functions, and how to use the theory of fixpoints to find solutions for recursive equations.

The concept of **domains** and **continuous functions** have been introduced in order to establish the basis of a sound framework for computing fixpoints of recursive equations, and ensure that all equations are consistent and sound, including the modeling of nonterminating programs.

Most definitions, theorems and propositions come from professor David F. Martin's class notes [23, 24].

For a deeper understanding of the domain theory, the reader may refer to the classical books on the subject, such as Carl

A. Gunter's [17], Robert Milne and Christopher Strachey's [26] and Joseph E. Stoy's [48] excellent books.

*****

Chapter 9

Epilogue

> *La lecture de tous les bons livres est comme une conversation avec les plus honnêtes gens des siècles passés, qui en ont été les auteurs, et même une conversation étudiée en laquelle ils ne nous découvrent que les meilleures de leurs pensées.* [1]
>
> René Descartes (1596-1650)

The design of $\mathcal{M}$ has been inspired by Peter Mosses' [4, 29, 30] and Michael Gordon's [15] works, which have been taken as $\mathcal{M}$'s starting point in the sense that most of their notations have been adopted, notably those for concrete grammars and abstract syntaxes, default declaration convention, cartesian products, lists, parse tree nodes, patterns, and `where` notations.

The concepts of standard denotational semantics and the formal definition of the language Small come from Michael Gordon's excellent book [15].

The component-based style for semantics presentation is

[1] The act of reading good books is like a conversation with the most qualified people from past centuries — their authors, and also an educated conversation in which they reveal to us their best thoughts.

inspired by Peter Mosses' concept of components [34, 35, 36].

The main ideas underlying the denotational definition of the Stack Machine $\mathcal{SC}$ come from the UCLA professor David F. Martin's lectures and class notes [23, 24].

The chapter on the foundations of the theory of domains is also heavily based on professor David F. Martin's class notes [23], which made this theory quite comprehensible for any engineering student.

As it is announced in this book's prologue, in fact, this is deliberately a very concise book on a vast and complex subject.

*****

Bibliography

[1] Ken Arnold, James Gosling and David Holmes. *The Java Programming Language*. Pearson Education, Inc, 2006.

[2] Lennart Augustsson and Thomas Johnsson. *Lazy ML: User's Manual*. Technical Report, Department of Computer Science, Chalmers University, Goteborg, Sweden, 1992.

[3] Daniel M. Berry and Richard L. Schwartz. *Type Equivalence in Strongly Typed Languages: one more look. ACM SIGPLAN NOTICES*, 14(9), 1979.

[4] Roberto S. Bigonha. *A Denotational Semantics Implementation System*. PhD thesis, University of California, Los Angeles, 1981.

[5] Roberto S. Bigonha. *Retractil Continuations*. Technical Report 01/96, DCC/UFMG, 1996

[6] Roberto S. Bigonha. *The Revised Report on the Language Script for Denotational Semantics*. Technical Report 002/98, DCC/UFMG, 1998.

[7] Roberto S. Bigonha, Fabio Tirelo and Guilherme H. S. Santos. *Separation of Concerns in Denotational Semantics Descriptions*. Technical Report LLP 01/2013,

DOI:10.131140/2.1.1897.7924, DCC/UFMG, 2013. Available in ResearchGate.net.

[8] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall International Series in Computer Science, 1988.

[9] William H. Burge. *Recursive Programming Techniques*. Reading, Mass, 1975.

[10] Luca Cardelli and Peter Wegner. *On Understanding Types, Data Abstraction, and Polymorphism*. ACM Computing Surveys, 17(4):471–522, 1985.

[11] Alonzo Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, vol 6, Princeton University Press, Princeton, N.J., 1951.

[12] Ole-Johan Dahl et al. *Structured Programming*. Academic Press, London and New York, 1972.

[13] Egon Börger and Roberto Stärk. *Abstract State Machine*. Springer, 1998.

[14] Fabíola Fonseca de Oliveira. *Compilação de uma Linguagem Funcional, Orientada por Objetos, para Definição de Semântica Denotacional*. Tese de Mestrado, UFMG, 1998.

[15] Michael J. C. Gordon. *The Denotational Description of Programming Languages - An Introduction*. Springer-Verlag, New York - Heiberg - Berlin, 1979.

[16] Carl A. Gunter, Peter D. Mosses, & Dana S. Scott. *Semantic Domains and Denotational Semantics*. Internal

Report DAIMI PB-276, Computer Science Department, Aarhus University, April 1989.

[17] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, Cambridge, Massachusetts, 1992.

[18] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science, Englewood Cliffs, 1987.

[19] Sheng Liang, Paul Hudak, and Mark Jones. *Monad Transformers and Modular Interpreters*. In POPL'95: Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995.

[20] Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell and Gerald J. Popek. *Report on the Programming Language EUCLID. ACM SIGPLAN NOTICES*, 2:1–79, 1977.

[21] Barbara Liskov and S. Zilles. *Programming with Abstract Data Types. ACM SIGPLAN NOTICES*, 9(4), 1974.

[22] Barbara Liskov. *An Introduction to CLU*. Memo 136, MIT. Computation Structure Group, 1976.

[23] David F. Martin. *Class Notes for CS225L*. University of California, Los Angeles, Fall 1977.

[24] David F. Martin. *Class Notes for CS282B*. University of California, Los Angeles, Fall 1981.

[25] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall International Series in Computer Science, New York, 1988.

[26] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics, Parts a and b.* Chapman and Hall, London, 1976.

[27] Eugenio Moggi. *Computational Lambda-Calculus and Monads.* In Proceedings of the Fourth Annual Symposium on Logic in Computer Science, pages 14, 23, Piscataway, NJ, USA. IEEE Press, 1989.

[28] Eugenio Moggi. *Notions of Computation and Monads.* Inf. Comput., 93(1):55-92, 1991.

[29] Peter D. Mosses. *Mathematical Semantics and Compiler Generation.* PhD thesis, Programming Research Group, Oxford University Computing Lab, 1975.

[30] Peter D. Mosses. *SIS - A Compiler-Generator System Using Denotational Semantics.* Technical Report, University of Aarhus, Denmark, 1978.

[31] Peter D. Mosses. *Denotational Semantics.* In Lectures Notes of the State of the Art Seminar on Formal Description of Programming Concepts – IFIP TC2 WG 2.2, Rio de Janeiro, Brazil, April 1989.

[32] Peter D. Mosses. *The Varieties of Programming Language Semantics.* In Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of

System Informatics: Akademgorodok, Novosibirsk, Russia, volume 2244, pages 165-190, London, UK. Springer-Verlag, 2001

[33] Peter D. Mosses. *What Use is Formal Semantics.* International Andrei Ershov Memorial Conference on Perspectives of System Informatics. Akademgorodok, Novosibirsk, Russia, 2001.

[34] Peter D. Mosses. *A Constructive Approach to Language Definition.* Journal of Universal Computer Science, 11(7):1117-1134, 2005.

[35] Peter D. Mosses. *Component-Based Description of Programming Languages.* In Visions of Computer Science. Electronic Proceedings, pages 275-286, 2008.

[36] Peter D. Mosses. *Component-Based Semantics.* In Proceedings of the 8th international workshop on Specification and verification of component-based systems, SAVCBS'09. ACM, pages 3-10, New York, NY, USA, 2009.

[37] Peter Naur (Editor). *Revised Report on the Algorithm Language ALGOL 60.* Communications of the ACM, 6(1), 1963.

[38] Hanne R. Nielson & Flemming Nielson, F. *Semantics with Applications - A Formal Introduction.* John Wiley & Sons, 1992.

[39] Frank Pagan. *Formal Specification of Programming Languages: A Panoramic Primer.* Prentice Hall, 1981.

[40] Paulo Rónai. *Não Perca seu Latim.* Editora Nova Fronteira, 1980.

[41] Guilherme H. S. Santos. *Semântica Denotacional Escalável de Linguagens Imperativas.* Dissertação de Mestrado, Departamento de Ciência da Computação, UFMG, March 2013.

[42] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn & Bacon, 1986.

[43] Dana S. Scott. *Outline of a Mathematical Theory of Computation.* In Proceedings of the 4th Princeton Conference on Information Sciences and Systems, 1970.

[44] Dana S. Scott and C. Strachey. *Toward a Mathematical Semantics for Computer Languages.* In Proceedings Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, 1971.

[45] Dana S. Scott and C. Strachey. *Toward a Mathematical Semantics for Computer Languages.* Technical Monograph PRG-6, Oxford University Computing Lab, Polytechnic Institute of Brooklyn, 1971.

[46] Dana S. Scott. *Data Type as Lattice.* SIAM Journal of Computing, 5:522-587, 1976.

[47] Joseph E. Stoy. *Foundations of Mathematical Semantics.* Lecture Notes in Computer Science, Springer-Verlag, 1979.

[48] Joseph E. Stoy. *Denotational Semantics: The Scott-Stratchey Approach to Programming Language Theory.* MIT Press, 1977.

[49] Robert D. Tennent. *The Denotational Semantics of Programming Languages.* Communications of the ACM, August 1976, 437-453.

[50] Robert D. Tennent. *Language Design Methods Based on Semantic Principles.* Acta Informatica, 8:97–112, 1977.

[51] Robert D. Tennent. *A Denotational Definition of the Programming Language PASCAL.* Technical Memo, Oxford University Computing Lab, Programming Research Group, 1978.

[52] Fabio Tirelo, Roberto S. Bigonha, and João Saraiva. *Disentangling Denotational Semantics Definitions.* Journal of Universal Computer Science, 14(21), pages 3592-3607, December 2008.

[53] David Turner. *An Overview of Miranda.* ACM SIGPLAN NOTICES, 21(12):158–166, 1986.

[54] Christopher P. Wadsworth and Christopher Strachey. *Continuations – A Mathematical Semantics for Handling Full Jumps.* Tech. Monograph PRG-11, Programming Research Group. Oxford University Computing Laboratory, 1974.

[55] David A. Watt. *Programming Languages Syntax and Semantics.* Prentice Hall International Series in Computer Science, New York, 1991.

[56] Adriaan Wijngaarden et al. *Revised Report on the Algorithm Language ALGOL 68. Acta Informatica*, 5(1):1–3, 1975.

[57] Glynn Winskel. *Semantics of Programming Languages.* MIT Press, 1993.

[58] Niklaus Wirth. *The Programming Language PASCAL. Acta Informatica*, 1(1), 1971.

[59] Niklaus Wirth. *What Can We Do about the Unnecessary Diversity of Notation for Syntatic Definition?* Communication of the ACM, 20(11):822-823, 1977.

[60] Yingzhou Zhang and Baowen Xu. *A Survey of Semantic Description Frameworks for Programming Languages.* SIGPLAN NOTICES, 39:14-30, March 2004.

Index

Approximation, 196

Bottom, 193

Chain, 194
Citations
  Albert Einstein, 117
  Antoine de Saint-Exupéry, 87
  Apelles, 157
  Leonardo da Vinci, 65, 185
  Michaelis, 125
  René Descartes, 95, 219
  William of Ockham, 1
Complete partial order, 194
Continuation
  definition, 106
  retractile, 117, 119, 123
  standard, 109
Cpo, 194
  bottom, 194
  least upper bound, 194

Domain, 185, 217
  built-in, 6
  cartesian product, 199
  compatibility, 17
  constant, 14
  cpo, 198
  equivalence, 17
  expression, 12
  flat, 198
  function, 15
  list, 14
  mapping, 204
  node, 15
  Scott, 195, 198
  standard, 105
  tuple, 14
  union, 16

Environment, 104
  denotable value, 104
  standard, 104
Expression
  abstraction, 19
  basic, 19, 24
  conditional, 19, 23
  integer, 27
  list, 31
  logical, 26

mapping, 34
node, 33
pattern, 22
quotation, 28
token, 45
tuple, 30
updating, 34

Files, 7
Fixpoint, 30, 210
circular equation, 215
computation, 213
factorial equation, 212
least, 210
operator, 210
paradoxal Y, 217
unsolvable equation, 215
Function
continuous, 202
definition, 191
Graph, 192
homomorphism, 97
monotonic, 201
partial, 191
recursive, 188
standard, 5
total, 191

LAMBDA, 208
Least upper bound, 193
Location, 104

Nontermination, 189
Noun
common, 4
indexed, 4
list, 4
proper, 4

Paradoxal operator, 217
Poset, 193
bottom, 193
least upper bound, 193

Relation, 192
antisymetric, 192
asymetric, 192
equivalence, 192
partial order, 192
reflexive, 192
symmetric, 192
total order, 194
transitive, 192

Semantics
continuation, 111
direct, 97
standard, 105
standard model, 103
Set
recursive, 189
Small, 129, 164
Stack Machine

architecture, 65
compiler, 87
continuations, 73
dump, 70
environment, 66
files, 71
stack, 65, 69
store, 68
Store, 104
standard, 104
storable value, 104

Upper bound, 193
least upper bound, 193

Value
denotable, 104
expressible, 105
left, 105
outputable, 105
right, 105
storable, 104
Variable
declaration, 9

The Author

Roberto S. Bigonha
Born 1948

PhD in Computer Science from University of California, Los Angeles, USA.
Professor Emeritus of Computer Science at Federal University of Minas Gerais, Brazil.
Member of the Brazilian Computer Society.

*****