UNIVERSITETET I OSLO
Institutt for informatikk

Preliminary Report on the Specification and Programming Language Abel

Ole-Johan Dahl,
Dag F. Langmyhr
and Olaf Owe

Research Report
no.106

December 1987

Contents

Chapter 1

Introduction.

Abel (Abstraction Building Experimental Language) is a language for specification and programming which has been developed at the University of Oslo, primarily as an aid for teaching techniques of specification and machine aided reasoning, with imperative programming and program verification as special cases. It is also a vehicle for, and a result of, local research activity in these areas. The language contains mechanisms for constructive and non-constructive specification as well as applicative and imperative programming. Object orientation is reflected in a class-like construct for imperative programming and comparable ones for non-imperative texts.

Interactive aids to reasoning about *Abel* texts are under development, centered on a term rewriting system for formula simplification and proof. Also a special "weak" logic has been designed for dealing efficiently with partial functions [Owe 84].

The *Abel* project is several years old [FS 76, Dahl 77, Lang 78, Dahl 84], and *Abel* is a thoroughly revised version of a language designed in the late 1970's. In addition to traditional programming languages like *Simula* and *Pascal* the specification language *Larch* [Gutt 84] has provided a source of inspiration. The overall goal of the project is to design a language and supporting software as well as underlying theory, that

- encourage high quality specifications and programs,
- facilitate secure reasoning about specifications and programs,
- enable long term accumulation of reusable concepts,
- are well suited for teaching purposes.

The *Abel* project has recently spawned some interesting descendant language activities: BABEL which is a further development of applicative features for language design and implementation [Kirk 86], and CABEL in which concurrent processes are introduced [Meld 86].

1.1 The *Abel* Language.

The main structuring mechanism of *Abel* is a *module* construct for encapsulating groups of functions and possibly types. There are the following specialized kinds of module:

1. **type** module, which defines one or more types and associated functions,

2. **class** module, a similar construct for imperative programming, roughly comparable to a *Simula* class,

3. **group** module, which defines a group of functions on given types, and

4. **property** module, used to specify minimal requirements on types and functions.

Types are defined by algebraic means or by explicit type expressions referring to types and type families previously defined. There is a subtype mechanism in the language. A subtype inherits the value set and the set of functions associated to its supertype, and it may constrain the former, add new functions, and redefine old ones. Functions may be defined non-constructively by sets of arbitrary first order axioms, or constructively by equational definitions possibly using generator induction. Definitions are usable for interpretative evaluation and, if certain requirements are satisfied, also as rules for a term rewriting system.

We define the semantics of all applicative definition mechanisms in terms of adding axioms and inference rules to an underlying formal system of many-sorted first order logic (as well as adding type and function symbols to the first order language). The intended way of using the language and the support system under construction is to provide mainly constructive definitions in **type**, **class**, and **group** modules. Thereby logical consistency and completeness in a certain sense are easily ensured, and a term rewriting system with some embellishments can be fully exploited for proving theorems (e.g. the contents of **property** modules).

The language is object oriented in the sense that textual nesting is very limited. In particular there is no nesting of modules or function definitions. Large structures are formed by putting modules together. *Abel* is strongly typed, and since most functions will be associated with an "owner" type or class, the overloading principle of object orientation applies, but slightly generalized, see below.

Imperative language mechanisms have been introduced for a variety of pragmatic reasons. We believe that an imperative program text, when written at a suitable level of abstraction, may sometimes be the best possible system specification. Also, when using *Abel* to program computers, imperative mechanisms in general provide better tools for controlling resource usage in time and space.

There are two main imperative constructs in the language, which may be used separately or in conjunction.

1. **prog** sections may be used as the right hand side of a function definition. Within a **prog** section assignment and I/O operations are available, as well as conventional constructs for the sequencing of actions in time.

2. **class** modules, already mentioned, are like types with associated functions, but give rise to "values" which are objects in the conventional imperative sense. The typical life history of an object is to become initialized at the time of generation as the result of the declaration of an object variable, and then, during the remainder of its life, to become incrementally updated from time to time. This is the recommended way of dealing with high volume data objects since it provides explicit control of storage usage.

Functions are introduced by declarations of the form

$$\textbf{func } f(D) =: E$$

where D, the domain, and E, the codomain, are types or classes. Let

$$\textbf{func } f(T, U) =: T$$

(where the domain of f is a Cartesian product). If t and u are expressions of types T and U respectively, then $f(t, u)$ is an expression of type T. For a variable v of type T an assignment operation

$$v := f(v, u)$$

may alternatively be written as a *call statement*

$$\textbf{call } f(@v, u)$$

where the @-character indicates that the variable v is used as an *update parameter*.

For a function f associated with a type (or class) the first argument, if any, of that type is defined as the dominant argument; therefore, given that all functions named f in the working context have the same dominant argument position, the type of that argument determines the identity of an occurrence of f. In certain cases the traditional dot notation is used, putting the dominant argument in front of the dot, writing e.g. $v.f(u)$ rather than $f(v, u)$. Then one may think of $v.f$ as a function "owned" by the object v and local to it in the sense of traditional object oriented languages. If v is an object of a class and f declared to be an updating procedure, then the notation $@v.f(y)$ is available for incrementally changing the state of v.

For partial functions certain error "values" may be declared as part of the **func** declaration. For most purposes of reasoning an expression denoting an error value is *illdefined*, but it is also possible, to a limited extent, to reason explicitly about error values. From an operational point of view the "evaluation" of an error behaves as an abnormal termination, possibly leading to an "exception handler".

Pointers are useful tools for building dynamically defined data structures and for dealing efficiently with high volume data. For the purpose of efficiency class objects are manipulated through pointers, in particular parameter transmission, as well as assignment of objects, are implemented by pointer copying. Unfortunately, in an imperative environment the unrestricted use of pointers may give rise to data sharing which makes reasoning about programs quite difficult. Since ease of reasoning about *Abel* texts is a prime concern the use of object expressions and object assignment in executable code is restricted so as to prevent alias problems. The restrictions on objects occurring in a statement consist of certain disjointness or inclusion requirements which can be enforced by textual checks (except in certain cases involving subscripted object variables). They are sufficiently liberal to permit recursive data structures to be built and processed dynamically.

Explicit object assignment is sometimes necessary when dealing with recursive structures; otherwise the use of call statements with update parameters is recommended as the standard way of updating objects incrementally. Some of the restrictions applied to explicit assignments are then automatically satisfied.

1.2 Meta Syntax.

1.2.1 Basic Grammar Language.

The syntax of *Abel* will be given in an extended form of BNF.

1. All terminal symbols are written as they appear, like +, *a*, or **a**.
2. Non-terminals are represented by text set in roman font and enclosed in diamond brackets, like the nonterminal ⟨ident⟩. The roman text of a nonterminal may be preceded by a *qualifier*, which is a text set in *slanted* font, as in ⟨*type* ident⟩. For the significance of qualifiers see section 1.2.2.
3. Meta-brackets (⟦ and ⟧) may be used for grouping together right hand side elements without having to introduce auxiliary nonterminals.
4. A raised question mark is used to indicate an optional element. Thus $\mathrm{A}^{?}$ means

$$⟦\mathrm{A} \mid ⟨\text{empty}⟩⟧$$

5. A raised plus sign indicates an element iterated to occur one or more times, and an accompanying subscript indicates a separating symbol. Thus $\mathrm{A}^{+}_{\mathrm{B}}$ means

$$⟦\mathrm{A} \mid \mathrm{ABA} \mid \mathrm{ABABA} \mid \ \ldots \ ⟧$$

6. A raised asterisk indicates iteration zero or more times. Thus $\mathrm{A}^{*}_{\mathrm{B}}$ means $⟦\mathrm{A}^{+}_{\mathrm{B}}⟧^{?}$.

1.2.2 Meta meta syntax

The *Abel* defining language is a two-level grammar, in the sense that qualifiers of nonterminals are themselves described by a meta-grammar. The latter is a very simple version of BNF where the alternatives of a right hand side (as well as the left hand side) are single qualifiers. The meta-rules are set in slanted font throughout. Example:

module ::= *group-module* | *property-module* | *type-module*

Any ordinary BNF rule whose left hand side is a qualified nonterminal, represents the set of rules obtained by systematically replacing that qualifier by its terminal (meta-)derivations. (Any differently qualified nonterminal occurring in the right hand side stands for the disjunction of its terminally qualified versions.) Thus, in the context of the above meta-rule the following ordinary rule:

⟨*module* ⟩ ::= ⟨*module* heading⟩ == ⟨*module* rhs⟩

represents the following three rules:

⟨*group-module* ⟩ ::= ⟨*group-module* heading⟩ == ⟨*group-module* rhs⟩
⟨*property-module* ⟩ ::= ⟨*property-module* heading⟩ == ⟨*property-module* rhs⟩
⟨*type-module* ⟩ ::= ⟨*type-module* heading⟩ == ⟨*type-module* rhs⟩

For convenience we add the following "meta-meta-rule":

- Any qualifier is derivable from the empty one.

Thus a BNF rule defining an unqualified nonterminal is understood as an en-bloc definition of all qualified versions of that nonterminal, and the qualifiers in that case can be freely used to convey informal semantic information. Example: In the context of the rules of section 1.3.2 each of the nonterminals ⟨*function* ident⟩, ⟨*type* ident⟩, etc., represent the same syntactic category as ⟨ident⟩.

1.3 Lexicographic Conventions.

An *Abel* program consists of a sequence of symbols, each of which is either a single character or a sequence of characters. The symbols are divided into the following groups. (In the following subsections some of the nonterminals occurring in the grammatical definitions are informally defined.)

1.3.1 Separators.

⟨separator⟩ ::= (|) | [|] | { | } | . | , | : | ; | " | ‘ | ’ | ˆ | ⟨spacing⟩
⟨spacing⟩ ::= ⟨spacing character⟩$^+$

The spacing characters are the single space character and all non-printable characters, such as the tabulation character and the end-of-line character. Spacing characters may be used to separate symbols if they would otherwise be regarded as a single symbol. For example, in

$A * -2$

the characters $*$ and $-$ must be separated by spacing, or they would otherwise be taken as the single symbol $*-$.

Apart from separating adjacent symbols, spacing characters have influence on neither the syntactic nor the semantic properties of an *Abel* text. They may therefore be freely inserted between symbols to improve readability.

1.3.2 Identifiers.

⟨ident⟩ ::= ⟨letter⟩ [[⟨letter⟩ | ⟨digit⟩ | ⟨underscore⟩]]* | ⟨boldface numeral⟩
⟨letter⟩ ::= a | b | ... | z | A | B | ... | Z
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨underscore⟩ ::= _
⟨boldface numeral⟩ ::= ⟨boldface digit⟩$^+$
⟨boldface digit⟩ ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Identifiers are predefined or user defined names on entities of the following categories: contexts, modules including types and classes, local types, functions including procedures, errors, branches, formal parameters, and variables of different categories including program constants. A naming mixfix, see section 1.4, counts as a function identifier. The identifier of each entity has one *defining occurrence*, and all *applied occurrences* must be located within the *scope* of the former. Scopes are individually defined for the different language constructs, but for entities defined within a module instance (section 7.5) each scope is located textually after the defining identifier occurrence. These entities must be named by distinct identifiers, except for function overloading and entities with disjoint scopes. Quantifiers (section 4.12) and case discriminators (section 4.10) may reintroduce identifiers naming local variables, in which case those of textually enclosing scopes become temporally inaccessible.For convenience, identifiers will be set in *italic* font in this document.

Boldface numerals are default function identifiers for tuple component selection, not usable for other purposes, see section 8.3.

1.3.3 Special symbols.

⟨special symbol⟩ ::= [[⟨special character⟩ | ⟨boldface letter⟩ | ⟨boldface digit⟩]]$^+$
⟨special character⟩ ::= ⟨*any ASCII character except escape, letter, digit, underscore, separator or spacing char*⟩
⟨escape⟩ ::= ~
⟨boldface letter⟩ ::= **a | b |** ... **| z | A | B |** ... **| Z**

Special symbols are sequences of *special characters* and *boldface* letters and digits. Boldface script is set apart from ordinary letters and digits in some predefined fashion. In this document the boldface script will be set in a **boldface** font. On a user terminal the boldface script could be underlined or set in inverse video mode. If a program text is stored in some medium having no possibility of distinguishing letters and digits, the boldface script should be prefixed and terminated by the escape character tilde (~). A tilde followed by a letter or a digit turns on "boldface mode", and the next one turns it off. Thus, the symbols **loop** and **1-1** should be represented as `~loop~` and `~1-1~`, respectively; notice that boldface mode only affects letters and digits. This is the way boldface script is represented within string literals (section 1.3.4).

Special symbols may be used to construct infix and other "mixfix" functional notations, see section 1.4. However, some symbols are part of fixed language constructs and are not allowed as part of user defined mixfix notations. These "reserved" symbols are the following:

all	**any**	**array**	**as**	**assert**	**assoc**
assuming	**axioms**	**binrel**	**branch**	**call**	**case**
class	**coercion**	**const**	**context**	**def**	**default**
do	**else**	**elsif**	**end**	**endcase**	**endconst**
endcontext	**endif**	**endloop**	**endon**	**endprog**	**endmodule**
endvar	**error**	**ex**	**fi**	**fo**	**for**
fork	**func**	**genbas**	**group**	**if**	**implements**
import	**in**	**init**	**include**	**including**	**invoke**
join	**lemmas**	**let**	**loop**	**module**	**ni**
no	**obs**	**obsbas**	**of**	**on**	**oper**
others	**partially**	**prog**	**property**	**qua**	**proc**
repeat	**return**	**rev**	**skip**	**some**	**subclass**
subtype	**then**	**lemmas**	**total**	**type**	**var**
where	**while**	**with**	**one-one**		

In addition several functions are predefined in mixfix notation, like arithmetic and Boolean operators. These may be overloaded according to standard *Abel* rules, but the (leading) special symbols must not be used in other mixfix notations. This is a consequence of a general restriction stated in section 1.4.

1.3.4 Literals.

⟨literal⟩ ::= ⟨numeric literal⟩ | ⟨string literal⟩
⟨numeric literal⟩ ::= ⟨digit⟩$^+$
⟨string literal⟩ ::= ' [[⟨normal string char⟩ | ⟨special string char⟩]]* '
⟨normal string char⟩ ::= ⟨any printable ASCII char except apostrophe and escape⟩
⟨special string char⟩ ::= ~ ' | ~~ | ~(⟨numeric literal⟩)

A numeric literal is a natural number in conventional decimal notaltion, except that leading zeros are not necessarily suppressed. There is an implementation-defined upper limit to the size of numeric literals. Those within the limit represent values of type Num, see section 7.7.2.

A string literal is a representation of a sequence of ASCII characters surrounded by apostrophes ('). All printable characters except apostrophes and escape characters are represented directly and the latter two by the pairs ~' and ~~, respectively. Arbitrary characters may be represented as ~(n), where n is the numeric code equivalent. There is an implementation defined upper limit to the length of string literals. Those within the limit represent values of type String, see section 7.7.4. A string of length one may be seen as (is coercible to) a value of type Char.

Examples of string literals:

" '+' 'a string' 'O~'Neill' '~~boldface~~'

These literals represent string values of lenghts 0, 1, 8, 7, and 10. The last value in turn represents the text **boldface**.

1.4 Application Patterns.

appl-ptn	::=	*domain* | *formal* | *actual* | *naming*
domain	::=	*appldomain* | *procdomain*
appldomain	::=	*funcdomain* | *errordomain* | *branchdomain*
formal	::=	*formalappl* | *formalcall*
formalappl	::=	*funchead* | *errorhead* | *branchhead* | *discriminator*
formalcall	::=	*prochead* | *discrimcall*
actual	::=	*application* | *call*
application	::=	*funcappl* | *domainpred* | *errorappl* | *branchappl*
⟨*appl-ptn* standard⟩	::=	⟨*appl-ptn* operator⟩⟨*appl-ptn* tuple⟩$^{?}$
⟨*appl-ptn* tuple⟩	::=	(⟨*appl-ptn* operand⟩$^{+}_{,}$)
⟨*appl-ptn* mixfix⟩	::=	⟨*appl-ptn* operand⟩$^{?}$⟨special symbol⟩ ⟦⟨*appl-ptn* operand⟩⟨special symbol⟩⟧* ⟨*appl-ptn* operand⟩$^{?}$
⟨function name⟩	::=	⟨*function* ident⟩ | ⟨*naming* mixfix⟩
⟨*naming* operand⟩	::=	^

Many constructs of *Abel* are patterned after function applications of which there are two syntactic categories:

- Standard notation: f or $f(x_1, x_2, \ldots, x_n)$ $(n > 0)$,
  where the main operator f is usually a function identifier and $x_1, \ldots, x_n$ are operands.

- Mixfix notation, consisting of a strictly alternating sequence of operands and special symbols containing at least one special symbol. A function defined in mixfix notation is named by the corresponding *naming mixfix*, which is the sequence of special symbols, in which each operand position is marked by the character ^.

A function application in mixfix notation may alternatively be expressed in standard notation using the naming mixfix as the main operator. For instance, the expression $x + y$ would be $\hat{} + \hat{}(x, y)$ in standard notation.

The following constructs are application-like: function, procedure, error, and branch domain definitions (chapters 2, 5), function, domain predicate, error, and branch applications (chapters 4, 5), procedure calls (chapter 5), function, procedure, error, and branch headings (chapters 6, 4, 5), and case discriminators (chapters 4, 5). In addition stand-alone tuple-like constructs are used in some contexts.

Some kinds of mixfix notations may be nested. Ambiguity of parsing is then partly prevented by the following restriction on user defined mixfix constructs.

- No mixfix function name may contain the leading special symbol of another (regarding every special symbol as atomic).

1.5 Mixfix Operators.

Parsing ambiguity of nested mixfix notations is possible only if a text contains

s1⟨operand⟩**s2**,

where **s1** and **s2** are special symbols and there are mixfix function names of the forms ...**s1**$\hat{}$ and $\hat{}$**s2**... . The ambiguity is resolved by fixed symbol priorities and a left association rule for the case of equal priorities. The built-in unary and binary operators have the following priorities (high first). For operator symbols used in this document which are outside the ASCII character set the ASCII representations are given in parantheses.

1. . (dot notation)
2. **S**, **P**
3. $\uparrow$ ($**$)
4. $*$, /, **mod**
5. $+$, $-$ (both unary and binary), **max**, **min**
6. $\vdash(!-)$, $\dashv(-!)$, $\vdash\!\dashv(!-!)$, $\#$, //, \\
7. $<$, $>$, $\leq(<=)$, $\geq(>=)$, $=$, $\neq(<>)$
8. **not**
9. **and**, **andthen**, $\wedge$(& or /\)
10. **or**, **orelse**, $\vee$(\/)
11. **implies**, **impthen**, $\Rightarrow(=>)$
12. ==

All user defined mifix notations are given the same priority as =. Notice that so called *closed* notations of the form **s1**^...^**sn** are parsed unambiguously independent of priorities.

Since all the above operator symbols are leading special symbols of mixfix notations, none of them is available for user defined notations. The same is true for the following "mixfix" constant functions: **f**,**t** (section 7.7.1), **z** (7.7.2), and **e** (7.7.4).

The signs → and ▯ used in this document are ASCII represented as −> and [] respectively. The former is a special symbol, but is not allowed as part of mixfix notations.

Some built-in operator notations like assignment operators (section 5.2.3), default selector functions (section 8.3), and several sequence type operations (section 7.7.4), as well as standard functional notation and type expressions contain separator characters and therefore are not mixfix notations in the strict sense of section 1.4. Parsing ambiguity is prevented by assigning maximal priority towards the left to all opening parantheses (, [, and {. The reserved special symbols **as**, **qua**, and **on** are operators with left priority immediately above that of =. Both case- and if-constructs (though not mixfix notations in the strict sense) are closed notations parsed unambiguously.

Chapter 2

Function Signatures.

function	::= *ordinary* **/** *special*
⟨*function* signatures⟩	::= ⟨*function* header⟩⟦⟨*function* signature⟩⟨error-part⟩*⟧$^{+}_{,}$
⟨*function* signature⟩	::= ⟨*function* domain-part⟩⟨*function* codomain-part⟩
⟨*ordinary* header⟩	::= **func**
⟨*ordinary* domain-part⟩	::= ⟨*funcdomain* standard⟩ **|** ⟨*funcdomain* mixfix⟩
⟨*funcdomain* operator⟩	::= ⟨*function* ident⟩
⟨*appldomain* operand⟩	::= ⟦⟨type label⟩:⟧$^{?}$⟨type expr⟩
⟨type label⟩	::= ⟨ident⟩
⟨*ordinary* codomain-part⟩	::= =: ⟨codomain⟩
⟨codomain⟩	::= ⟨type expr⟩

The above definitions should be read in the context of those of section 1.4. A function signature names a function and expresses its syntactic properties: type association, domain, codomain, and associated errors. This information is used for the purpose of expression parsing and type control, as well as function overloading. The constituent function name is a defining occurrence, whose scope is the remainder of the containing module instance (section 7.5), not including the signature itself. The domain of the function is the Cartesian product of the types occurring in the domain part, whether the latter is in standard or mixfix notation (section 1.4). Notice that the domain may be an empty product, in which case the function is a constant. Type labels may be used to convey informal semantic information about the arguments of the function; technically they are user defined selector functions on the argument tuple (see section 8.3), and may be used as implicit formal parameter names in the function definition, if any, or as implicit formal variables of case discriminators if the function is a basic generator.

A function is said to be *associated with* a type T if its signature has the *home type* T. The home type, if any, of a signature may be implicitly defined, or it may be explicitly defined by prefixing the signature with the type expression T, see sections 7.1 and 7.4. It may be referred to by the special symbol $$. A function associated with T is also called a T-function, and T is called the *owner type* of the function. If f is the name of a T-function, then its so called *prefixed name* is 'T'f.

If a function signature has no home type the declared function is said to be *free* (of type associations). A free function must be declared by an ordinary signature. A free function named f has the prefixed name "f.

All functions in the working module must have distinct prefixed names. The latter therefore are unique function identifications. The binding rules of applied function occur-

rences are, however, such that it normally suffices to refer to a function by its unprefixed name. In that sense the language provides for function "overloading". The owner type of a function occurrence may be determined by the *dominant argument*, if any, or by the textual environment of the application, see section 7.5.

The first occurrence of the home type in the domain part of a function signature identifies the dominant argument position of the declared function. All synonymous functions in the working module must have the same or no dominant argument position. (This implies that a free function may not have the same name as an associated function with a dominant argument.) Arguments in non-dominant positions are (automatically) coerced to the expected type, if possible.

2.1 Domain predicates.

For each function f with domain D, a function named **D_**f, called the *domain (membership) predicate of f*, is implicitly introduced:

$$\textbf{func D\_}f(D) =: \mathit{Boolean}$$

where the symbol f represents an ordinary identifier or a "naming mixfix" notation (section 1.4). Thus, a signature for f is really an abbreviation for two signatures, one for f and one for **D_**f. The **D_**f function has the same domain, domain constraint if any, dominance, and owner type as f. It is strict and total (on the possibly constrained domain).

Let e be a welldefined expression of the domain type whose value satisfies the domain constraint, if any, of f. Then the expression **D_**f(e) is **t** if $f(e)$ is welldefined, otherwise **f**. **D_**f is in general a non-executable function, but a non-recursive definition of f yields an executable definition of **D_**f (see section 3.1.1).

2.2 Special Signatures.

special	::= *initiator* | *attribute* | *nonstandard*
attribute	::= *observer* | *operator*
nonstandard	::= *binop* | *coercion* | *procedure*
⟨*initiator* header⟩	::= **init**
⟨*initiator* domain-part⟩	::= ⟨*ordinary* domain-part⟩
⟨*initiator* codomain-part⟩	::= ⟨empty⟩
⟨*attribute* domain-part⟩	::= ⟨*funcdomain* standard⟩
⟨*observer* header⟩	::= **obs**
⟨*observer* codomain-part⟩	::= ⟨*ordinary* codomain-part⟩
⟨*operator* header⟩	::= **oper**
⟨*operator* codomain-part⟩	::= ⟨*ordinary* codomain-part⟩?
⟨*binop* header⟩	::= **binrel** | **assoc**
⟨*binop* domain-part⟩	::= ⟨special symbol⟩
⟨*binop* codomain-part⟩	::= ⟨empty⟩
⟨*coercion* header⟩	::= **coercion**
⟨*coercion* domain-part⟩	::= ⟨type expr⟩
⟨*coercion* codomain-part⟩	::= ⟨empty⟩
⟨*procedure* header⟩	::= **proc**

⟨*procedure* domain-part⟩ ::= ⟨*procdomain* standard⟩
⟨*procdomain* operator⟩ ::= ⟨*procedure* ident⟩
⟨*procdomain* operand⟩ ::= @? [[⟨type label⟩:]]? ⟨type expr⟩
⟨*procedure* codomain-part⟩ ::= ⟨*ordinary* codomain-part⟩?

Special signatures are only meaningful if the home type is defined, say as T, and represent signatures where T is implicitly part of the domain and/or the codomain.

A T-function may be classified as a T-*initiator* (**init**) if its codomain is T and T does not occur in the domain. A corresponding special signature is an abbreviation of an ordinary function signature. (In the following signatures parentheses should be deleted as necessary if U or V or both are type products or U is empty.)

init $f(U)$	stands for	**func** $f(U) =: T$

A T-function whose domain contains T may be classified as a T-*operator* (**oper**) or a T-*observer* (**obs**), according to whether T occurs in the codomain or not. The corresponding special signatures also stand for ordinary signatures (in which mixfix notation has been abused to denote standard dot notation).

obs $f(U) =: V$	stands for	**func** $\${:}T.f(U) =: V$
oper $f(U)$	stands for	**func** $\${:}T.f(U) =: T$
oper $f(U) =: V$	stands for	**func** $\${:}T.f(U) =: (T, V)$

Standard functional notation is required here, and the function name must be an ordinary identifier. The implicit domain component of a T-observer or T-operator corresponds to the dominant parameter, which is provided by dot notation. (The implicit formal parameter is named \$.) For that reason functions declared **obs** or **oper** are called *attribute functions*. The dominant parameter position in this case has the ordinal number zero (cf. section 8.3).

Binary T-relations (**binrel**) and *associative binary T-operators* (**assoc**) may be introduced by special signatures:

binrel o	stands for	**func** T**o**$T =:$ *Boolean*
assoc o	stands for	**func** T**o**$T =: T$

where **o** is a special symbol to serve as an infix operator. Relational operators introduced as **binrel** are treated in a special way syntactically, in that $e1$ **o1** $e2$ **o2** $e3$ ... abbreviates

$$(e1 \textbf{ o1 } e2) \textbf{ andthen } (e2 \textbf{ o2 } e3) \textbf{ andthen } ...$$

(the **andthen** operator is defined in section 4.11). For every operator introduced as **assoc**, a lemma of the form

$$x \textbf{ o } (y \textbf{ o } z) \;=\; (x \textbf{ o } y \textbf{ o } z)$$

is added automatically, flagged "unproven".

The purpose of a T-associated *coercion function* is to provide a mapping from T-values to values of some other type, which preserves meaning in some suitable sense. The special signature

coercion U	stands for	**func** $\${:}T$ **as** $U =: U$

where mixfix notation has been abused to denote a unary postfix function named "ˆ**as** U", U a type expression. When a coercion function has been introduced in this way the type T is said to be *coercible* to U. The function will implicitly be applied to every expression of type T occurring where a type U expression is required. See also section 4.7.

A *procedure* is a special kind of operator which must be associated to a class-like type (section 7.4). A procedure signature

$$\textbf{proc}\ f(@U, V) =: W \quad \text{stands for} \quad \textbf{func}\ \${:}T.f(U, V) =: (T, U, W)$$

A procedure is an attribute function in the sense that the implicit parameter is the dominant one, and is provided by dot notation. A function application of the procedure f, say $t.f(u, v)$ where t, u, and v are expressions of the appropriate types, is non-executable but legal and meaningful. An executable procedure invocation must take the form of a **call** statement (section 5.?), say

$$\textbf{call}\ @tt.f(@uu, v) =: ww$$

where tt, uu, and ww are *updatable variables* of the appropriate types satisfying certain disjointness requirements. An updatable variable is either a program variable, an updatable parameter, an updatable discriminator operand, or a component of an updatable variable identified through subscription and/or selection.

The parameter positions (zero or more) marked by the *update symbol* @, as well as the implicit type T position, correspond to *update parameters* whose values may be changed as the result of a call to f. They must occur first in the list, and their types must be class-like. The actual update parameters of a call must be program variables marked by the update symbol. Axiomatically the above call statement is equivalent to the following non-executable assignment operation.

$$(tt, uu, ww) := tt.f(uu, v)$$

2.3 Errors.

⟨error-part⟩ ::= **error** ⟨*errordomain* standard⟩
⟨*errordomain* operator⟩ ::= ⟨*error* ident⟩

A partial function f may have a number of associated errors which correspond to distinct situations of abnormal termination of f, caused by explicit **error**-expressions in the function body. Errors may be parameterized in order to bring information from an error application in the function body to an "error handler". See section 4.3.

Chapter 3

Logical Foundation

3.1 Axiomatic Basis

Since we allow partial functions, an expression may not always denote a value. When an expression does not denote a value, it is said to be *illdefined* (and its value is an error). Conversely, when an expression denotes a value, it is said to be *welldefined*. An executable expression is welldefined if, and only if, its evaluation terminates normally. For each value of its free variables, an expression is either welldefined or illdefined. (If the expression involves non-constructively characterized functions, its value may be "undefined" in the sense that its value is not known.)

Since the *Abel* expression language allows partial functions and error-values, traditional first order logic is unsatisfactory. The *Abel* semantics is based on a modified version of first order logic (with equality) which caters for illdefined expressions as well. This logic, called *Weak Logic* [Owe 84], may be thought of as a variant of tree-valued many-sorted logic with specialized concepts of validity and welldefinedness. Weak Logic was designed to allow simple and natural specification and reasoning about partial functions. For instance, in order to automate expression simplification and theorem proving, it is essential that the theory of term rewriting systems (based on total functions) is easily adapted to Weak Logic. In order to allow simple specifications, it was desired that error conditions should not have to be explicitly present in every axiom and lemma, but should be defined once and for all by definitions or axioms of a special kind (about domain-predicates, see below). In particular, implementation errors reflecting capacity limits should be dealt with in such a way that they can be ignored at more abstract levels of specification.

The main characteristics of Weak Logic is its concept of validity and the welldefinedness properties of its predefined constructs, in particular universal and existential quantification. In order to explain and motivate these concepts, the following terminology is used. A construct is said to be *strict* if an illdefined argument yields an illdefined result. A strict construct is said to be *total* if it is welldefined for all welldefined arguments. A construct is said to be *monotonic* if, by replacing a welldefined argument by an illdefined one, the result becomes illdefined or remains unchanged. A formula (Boolean expression) is said to be *monotonic* if, whenever it is valid, it remains valid when replacing a subexpression by an illdefined expression. As a consequence, a formula is monotonic if it satisfies certain syntactic requirements restricting the use of non-monotonic constructs.

In Weak Logic all non-strict constructs are predefined, and all user defined functions are strict as a consequence of the parameter passing rules. In particular, equality (=), the

executable Boolean operators **not**, **and**, **or**, **implies**, and the domain-predicates are strict and total. The **if**-construct, as well as the predefined non-executable Boolean operators $\wedge, \vee, \Rightarrow$, are non-strict and monotonic. The quantifiers are non-monotonic.

Welldefinedness. In order to formalize the welldefinedness of an expression, we introduce a non-monotonic syntactic operator Δ, defined by structural induction over the expression language, see chapter 4. The Δ-operator takes an expression e to a Boolean expression, denoted $\Delta[e]$, such that $\Delta[e]$ is true if, and only if, e is welldefined; and such that $\Delta[\Delta[e]]$ is a propositional tautology. For instance, $\Delta[\textbf{if } e \textbf{ then } e1 \textbf{ else } e2 \textbf{ fi}]$ is defined as

$$\Delta[e] \wedge \textbf{if } e \textbf{ then } \Delta[e1] \textbf{ else } \Delta[e2] \textbf{ fi}$$

The welldefinedness of a function application is defined in terms of the domain-predicate (which is strict and total). For instance the formula

$$\Delta[\mathbf{P}\ x]$$

where $x : Nat$ and **P** is the predecessor operation, is reduced to **D_P** which may be simplified to $x \neq 0$ (given the *Abel* definition of Nat). Since an expression with free variables may be welldefined for some values of the free variables and illdefined for others, $\Delta[e]$ will in general be a boolean expression over the free variables of e.

By means of the Δ-operator it is possible within Weak Logic to express welldefinedness properties of expressions.

Validity. The Weak Logic validity of a welldefined formula corresponds to that of traditional first order logic. The validity of a formula P which is not always welldefined is equivalent to the validity of the welldefined formula

$$\textbf{if } \Delta[P] \textbf{ then } P \textbf{ else t fi}$$

(where the **if**-construct has the obvious semantics) expressing that P is trivially valid whenever P is illdefined. This "weak" interpretation of validity has given Weak Logic its name. Notice that a propositional tautology is valid, such as $P or \textbf{not} P$, even if it contains partial functions or predicates.

Due to this concept of validity, axioms and lemmas, involving partial functions, can be written in a short and natural form where conditions guarding against errors are implicit rather than explicit. For instance, the axiom

$$a/b * b = a$$

where $a, b : Nat$, is valid even though division by zero is defined as an error. The formula

$$s.push(x).pop = s$$

where $s : Stack\{T\}$, $x : T$, expresses a property about abstract stacks; however, it is valid also for bounded stacks for which the push-operation may result in the error overflow. This formula is an example of a monotonic formula. In general a monotonic formula expressing some property about abstract objects remains valid in the sense of Weak Logic for any partial implementation. The fact that capacity constraints in this way may be ignored at abstract levels of specification is essential in order to obtain at all manageable formal statements about most real systems. The validity concept of Weak Logic can be seen as a generalization of the idea of *partial correctness* of programs.

Universal and existential quantification. The free variables in the formulas given above should be understood as universally quantified over their respective types. Consequently, the meaning of the universal quantification **all** $x : T \mid P$ where P may be illdefined, is equivalent to

$$\textbf{all } x : T \mid \textbf{if } \Delta[P] \textbf{ then } P \textbf{ else t fi}$$

which is welldefined and has the obvious meaning. Thus, the formula

$$\textbf{all } x : Integer \mid x/x = 1$$

expresses that for all integers x except 0, x/x is 1.

The meaning of an existential quantification, say **ex** $x : T \mid P$, is defined as

$$\textbf{not all } x : T \mid \textbf{not } P$$

which is welldefined and is equivalent to

$$\textbf{ex } x : T \mid \textbf{if } \Delta[P] \textbf{ then } P \textbf{ else f fi}$$

In a monotonic formula written on prenex form, there may not be any existential quantifiers.

The Boolean operators $\wedge$, $\vee$, $\Rightarrow$. The $\wedge$ operator is symmetric and yields **f** if one (or both) argument is **f** even if the other argument is illdefined.

$p \vee q$ is equivalent to $\textbf{not}(\textbf{not } p \wedge \textbf{not } q)$

$p \Rightarrow q$ is equivalent to $\textbf{not } p \vee q$

Notice that both $\wedge$ and $\vee$ are associative. These non-strict Boolean operators are defined as non-executable (however, it would be possible to evaluate them by means of parallel processes with exits).

By means of the welldefinedness-operator (Δ) it is possible to reason about errors and absence of errors. Although an illdefined formula such as $x/0 = 1$ is trivially valid in Weak Logic, it is possible within Weak Logic to express and to prove that a formula P is welldefined and true; this is expressed by the formula

$$\Delta[P] \wedge P$$

Presence of errors can be expressed as in the formula

$$lth(s) = max \Rightarrow \textbf{not } \Delta[s.push(x)]$$

which states a condition which makes $s.push(x)$ result in error.

3.1.1 Formalization of error handling

The formalism introduced so far is sufficient for reasoning about errors in a language without error (or exception) handling. However, it is not suited as the axiomatic basis for a language with error handling, because there is no means of reasoning about named errors. For instance, in order to reason about the postscripted expression

$$f(5) \textbf{ on } X \rightarrow e \textbf{ no}$$

we may need to prove something about when **error** X is propagated from f. For this purpose we introduce strong (non-strict and non-monotonic) equality $==$. Intuitively, $e1 == e2$ is true if either both $e1$ and $e2$ are welldefined and $e1 = e2$, or both $e1$ and $e2$ are illdefined and result in identical errors, and is false otherwise. For instance, the formula

$$f(5) == f \textbf{ error } X$$

expresses that the f associated **error** X is propagated from $f(5)$.

The meaning of $f(5)$ **on** $X \rightarrow e$ **no** can then be explained in terms of strong equality as

$$\textbf{if } f(5) == f \textbf{ error } X \textbf{ then } e \textbf{ else } f(5) \textbf{ fi}$$

provided X is the only error declared in the signature of f.

Strong equality will also serve as the formal basis for rewrite rule systems because it allows the following substitution rule: [1]

$$\frac{\vdash e1 == e2 \ , \ \vdash \ ...e1...}{\vdash ...e2...}$$

With strong equality the formula $\Delta[P] \wedge P$ can be expressed as $P == \mathbf{t}$. Similarly, $P == \mathbf{f}$ expresses that P is welldefined and false.

Formally, strong equality is introduced as an equivalence relation which satisfies the substitution rule above, as well as the rules and axioms below, and $\Delta[e1 == e2]$ is defined as $\mathbf{t}$. The relationship between equality and strong equality is expressed by the rule

$$\frac{\vdash e1 = e2 \ , \ \vdash \ \Delta[(e1, e2)]}{\vdash e1 == e2}$$

The rules below formalizes how strong equality treats error values.

Error distinctness axioms

The following axioms state that strong equality over errors is the identity relation.

not error $==$ f **error** ..
not f **error** .. $==$ g **error** .. *where f and g are distinct function names*
not f **error** X.. $==$ f **error** Y.. *where X and Y are distinct*
not $a = b \Rightarrow$ **not** f **error** $X(a)$ $==$ f **error** $X(b)$

where an error X associated to f is written as f **error** X, and where f **error** .. represents any error associated with f.

Propagation rules

For the purpose of language clarity and security, the set of errors that may be propagated from an expression e is severely limited and is determined by simple textual analysis. Specifically, e may only result in a g-error if either this g-error or a (non-formal) application

[1] In a language without exception handling a corresponding equivalence relation $e1 == e2$ satisfying substitutivity could be defined as

$$\textbf{if } \Delta[e1] \wedge \Delta[e2] \textbf{ then } e1 = e2 \textbf{ else not } \Delta[e1] \wedge \textbf{not } \Delta[e2] \textbf{ fi}$$

of g is textually occurring in e (including implicit applications of coercions). For the same reasons an error caused by a function application may only be handled in a postscript to that application.

As a consequence of the strict parameter passing rules, the parameters to a function are evaluated before the the function is evaluated; if the evaluation of the parameters yields an error, that error is propagated. The same holds for error parameters as well as if-tests and case-discriminands, which are also strict: This is formalized by the axioms:

$f(\Omega) == \Omega$
f **error** $E(\Omega) == \Omega$
case Ω **of** ... **fo** $== \Omega$
if Ω **then** ... **fi** $== \Omega$
let $x = \Omega$ **in** ... **ni** $== \Omega$

where Ω denotes any associated error (possibly parameterized), or the anonymous error.

In the case where a function has several parameters and one or more of them yield error, then the leftmost error is propagated. This is formalized by the following rule for propagation from tuples:

$$\frac{\vdash \Delta[(e_1, e_2, .., e_i)]}{\vdash (e_1, e_2, .., e_i, \Omega, ..) == \Omega}$$

where the premise reduces to **t** in the case where i is 0. This rule also defines propagation from the boolean operators $\wedge$, $\vee$, $\Rightarrow$, in the cases where they are illdefined; for instance,

$$(\Omega 1 \ \wedge \ \Omega 2) == \Omega 1$$

An illdefined function application, $f(e)$, results in an error propagated from e, if e is illdefined, as already explained. Otherwise, it results in an f-error, or the anonymous error, as formalized by the axiom

all $x : T \mid \mathbf{D\_}f(x) \vee (f(x) ==$ **error**$) \vee (f(x) == f$ **error** $X1) \vee (f(x) == f$ **error** $X2) \vee ...$

where T is the domain of f and $X1$, $X2$, ... are the errors declared in the signature of f. For a parameterized f-error say $Xi(Ti)$ the corresponding disjunct should be:

$$\textbf{ex } y : Ti \mid f(x) == f \textbf{ error } Xi(y)$$

The semantics of postscripts is formalized as follows:

$f(a)$ **on**..[] $Xi(y) \rightarrow Hi$ []..**no** $==$ **if** $\mathbf{D\_}f(a)$ **then** $f(a)$
...
elsif any $y : Ti \mid f(a) == f$ **error** $Xi(y)$ **then** Hi
...
else error fi

where the else-part corresponds to the implicit or explicit others-part of the postscript, and where Ti is the type of the parameters to f-error Xi, if any. In the case where Xi has no value-parameters, the any-expression reduces to $f(a) == f$ **error** Xi. Notice that a postscript without an else-part will turn an unhandled error into the anonymous error.

3.1.2 Weak Logic Theory

A *Weak Logic theory* is a triple ⟨L,R,A⟩ where L is a formula language, R is a set of proof rules, and A is a set of axioms. A theory may contain any number of user-defined signatures, proof rules and axioms in addition to the predefined ones. For convenience we shall assume that in axioms and proof rules all type checking, coercion, and binding of function identifiers have been performed. Consequently, type declarations need not be part of the theory, except for variable declarations involving subtypes. The semantics of subtypes may be explained by rewriting expressions of form

$$\mathbf{Q}\ x : ST \mid p$$

where **Q** is a quantifier and ST is a subtype of T, as

- $\mathbf{Q}\ x : T \mid x\ \mathbf{isin}\ ST \Rightarrow p$, if **Q** is **all**
- $\mathbf{Q}\ x : T \mid x\ \mathbf{isin}\ ST \wedge p$, if **Q** is **ex**, **some**, or **any**.

The semantics of an *Abel* module instance M (see 7.5) is defined as the Weak Logic theory ⟨L,R,A⟩ where the user-defined parts of L,R, and A are determined by the expression language of the module instance, the basis statements in M, and the axioms and function definitions in M.

A function definition, say

$$\mathbf{def}\ f(x) == e$$

may be seen as an abbreviation for the following axiom and proof rule:

- $$\mathbf{all}\ x : T \mid f(x) ==< e >$$

  where T is the domain of f, and where $< e >$ is e with every (non-formal) function application without a postscript augmented by the empty postscript **on no** (for a function without associated errors this augmentation may be omitted), and in addition, with all function, error, and coercion bindings performed, as described in section 7.5 steps 4 and 5.

  Notice that this axiom does explain the identity of error-values. Also notice that the axiom implies the two axioms $f(x) =< e >$ and $\mathbf{D\_}f(x) = \Delta[< e >]$ and implication the other way holds if f has no associated errors.

- $$\frac{(\mathbf{all}\ x : T \mid \mathbf{D\_}f(x) \Rightarrow P\ ) \vdash \Delta[e] \Rightarrow P}{\vdash \mathbf{D\_}f(x) \Rightarrow P}$$

  which expresses least fix point semantics for recursive function definitions. It follows that non-terminating recursion results in the anonymous error.

For a function f characterized non-constructively, the user may explicitly state desired error conditions by means of axioms characterizing $\mathbf{D\_}f$, and also specify error value identity by means of strong equality.

3.2 Basis Statements

⟨basis stmt⟩ ::= ⟨genbas stmt⟩ | ⟨obsbas stmt⟩
⟨genbas stmt⟩ ::= **one-one**$^?$ **genbas** ⟨function name⟩$^+_,$
⟨obsbas stmt⟩ ::= **obsbas** ⟨function name⟩$^+_,$

There must be an identified home type (section 7.1) for any basis statement, and the functions listed must be associated with that type. Basis statements are used to define the value set of the home type. It is required that the value set of any non-formal type be completely defined. That is achieved by providing either a **one-one genbas** statement or a **genbas** statement and an **obsbas** statement, see below.

3.2.1 Generator Basis

A *generator basis* for a type T is a list of functions called it (basic) T-generators. The generators must be *T-producers*, i. e. functions whose codomain is T. The first function listed must be a it relative T-constant, i. e. a function with no argument of type T). For home type T the statement

$$\textbf{genbas}\ g_1, g_2, ..., g_n$$

identifies the generator basis of T. It is thereby expressed that all T-values may be finitely generated by means of the functions listed. The set of *T-value denotations* is defined as the set of variable-free expressions of type T consisting of basic T-generator applications and possibly value denotations of other types. These facts are expressed formally by adding the following proof rule to the logic:

$$\frac{P^t_{t_1},\ P^t_{t_2}, ..,\ P^t_{t_{k_i}} \vdash P^t_{g_i(x_i)}\ ;\ i = 1, 2, ..., n}{\vdash \textbf{all}\ t : T \mid P}$$

where x_i is a list of distinct variables not free in P, containing $t_1, ..., t_{k_i} : T,\ k_i \geq 0;\ i = 1, 2, .., n$. The rule expresses the principle of *generator induction* introduced by the **genbas**-statement.

Basic generators are by definition total functions.

The *default value of T* is defined as g_1 if that is a constant function, otherwise as $g_1(d)$ where d is the default value of the domain of g_1.

A generator basis defines a partial complexity-ordering over the set of T-value denotations. A denotation of form $g(.., \alpha, ..)$, where α is a T-value denotations, is said to be more *complex* than α.

A *one-one* generator basis expresses that T-values generated differently are different or, in other words, every T-value has a unique denotation (given that constituent value denotations of other types are unique). Then T-equality amounts to syntactic equality on value denotations. Formally, a one-one basis defines T-equality by axioms of the form:

not $g(w) = g'(w')$
$g(w) = g(v) \Rightarrow w = v$

where there is one axiom of the first kind for each pair of distinct basic generators, g and g', and one of the second kind for each basic generator g, and where w, w', and v are appropriate lists of distinct variables. These axioms together with the standard congruence axioms define the equality relation on T completely. The following constructive definition is consistent with these axioms.

$$\begin{array}{lll} \textbf{def } x = y \;==\; \textbf{case } (x,y) \textbf{ of} & & \\ \quad \ldots \;[] & & \\ \quad (g(w), g'(w')) & \rightarrow \textbf{ f } [] & \\ \quad (g(w), g(v)) & \rightarrow (w) = (v) \;[] & \\ \quad \ldots \;\textbf{ fo} & & \end{array}$$

It follows that equality defined by a one-one generator basis is executable, provided that all other types occurring in the generator domains have executable equality relations.

3.2.2 Observation Basis

For a given home type T the statement

$$\textbf{obsbas } h_1, h_2, , ..., h_m$$

where $m \geq 1$, defines an *observation basis* for the type T. The functions listed are called *basic T-observers*. The statement defines any two T-value denotations x and y as equal if all observations by means of basic observers yield the same result for x and y. The following proof rule is introduced:

$$\frac{\ldots;\ \vdash \textbf{all } v : V, w : W \mid h(v,x,w) == h(v,y,w);\ \ldots}{\vdash x = y}$$

where there is a premise for each T-argument of each basic observer function h, with domain of form (V, T, W), where $v : V$ and $w : W$ denote appropriate variable lists, and $x, y : T$. This rule, together with the standard congruence axioms, defines the equality relation on T completely and implies an analogous explicit definition.

$$\textbf{def } x = y \;==\; \ldots \wedge (\textbf{all } v : V, w : W \mid h(v,x,w) == h(v,y,w)) \wedge \ldots$$

It follows that equality defined through an observation basis is executable only if each basic observer is unary.

There must be at least one T-observer in an observation basis and no T-initiator. If it contains any T-operator, then it is required that its definition reduces the complexity of T-value denotations. This implies that repeated application of them to a T-value must eventually stop. It also implies that an observation basis must be disjoint from the generator basis.

3.3 Axioms

$$\begin{array}{ll} \langle \text{axioms} \rangle & ::= \textbf{axioms } \langle \text{free variable part} \rangle^{?} \langle \text{axiom} \rangle^{+}_{,} \\ \langle \text{free variable part} \rangle & ::= (\langle \text{free variable list} \rangle) \\ \langle \text{free variable list} \rangle & ::= [\![\langle \textit{free variable} \text{ ident} \rangle^{+}_{,} : \langle \text{type expr} \rangle]\!]^{+}_{,} \\ \langle \text{axiom} \rangle & ::= \langle \textit{Boolean} \text{ expr} \rangle \end{array}$$

A list of axioms of arbitrary form may be used to characterize semantically one or more functions. They need not provide a complete characterization, but they should be logically consistent, or more precisely: they should preserve the consistency of the underlying logical system. Although non-constructive axioms provide a more flexible specification tool than do constructive definitions, they are also more dangerous; it is entirely up to the user to

prove that logical consistency is maintained, and the mechanized tools for formal reasoning with free axioms are less powerful.

All free variables used in the axioms must be declared in the free variable part. They are understood as being universally quantified over their respective types. A variable list $x, y : T$ is an abbreviation for $x : T,\ y : T$.

Example

func $max(Seq\{T\})\ =:\ T$;
$\vdots$
axioms $(s : Seq\{T\})$
ex i :*Nat1* | $s[i] = max(s)$,
all j :*Nat1* | $s[j] \leq max(s)$

These axioms characterize the function max completely in a non-constructive way, relative to the other functions occurring.

3.4 Lemmas

⟨lemmas⟩ ::= **lemmas** ⟨typed param list⟩$^?$⟨lemma⟩$^+_,$
⟨lemma⟩ ::= ⟨*Boolean* expr⟩ |
⟨function def⟩ |
⟨basis stmt⟩ |
⟨implements stmt⟩ |
⟨*property-module* expr⟩

A lemma states a formula or property which should follow from the given axioms and proof rules. There is an obligation to prove every lemma within the Weak Logic theory of the working module. Lemmas that are proved are tagged "proven"; lemmas that are not yet proved are tagged "not proven". For each proven lemma, it is recorded which other lemmas were actually used in the proof. Since axioms and basis statements of a property may be turned into lemmas (see section 7.5), dependencies of such items are also recorded.

3.4.1 Implements statement

⟨implements stmt⟩ ::= **implements partially**$^?$
⟨type expr⟩$^?$⟨equivalence clause⟩$^?$
⟨equivalence clause⟩ ::= **equiv** *(* ⟨function name⟩$^+_,$ *)*

Only constants and formal parameters (of home type) may occur as value parameters in ⟨type expr⟩. If the keyword **partially** is not present, the implementation is said to be total.

An implements statement expressing that HT implements (partially or totally) T is necessary in order to use a type expression of form T **by** HT (see section 7..). The implements statement **implements** T states that the home type HT is an implementation of T, in the following sense:

- Functions in T may or may not be implemented in HT; however, all basic T-generators must be constructively implemented in HT. For each implemented T-function f there is a HT-function f' with the same signature (replacing T by HT), except as follows:

- The names of the two functions may differ, in that case the correspondence must be given by a renaming clause in the type expression for T (as explained in section 7.1).
- The name of the i-th error in the signature of f may differ from the name of the i-th error in the signature of f'; if so, the first error is renamed by the latter.
- f' may have more associated errors than f. These additional f'-errors are called *implementation errors*.

Equality over T in general corresponds to an equivalence relation over HT, denoted **equiv** (which is total). The equivalence relation may be given explicitly by an equivalence clause, specifying an "observation basis" for the HT-equivalence (defined in a manner analogous to that of section 3.2.2), otherwise, a definition of **equiv** is derived from that of T-equality.

- Every T-value can be represented as a HT-value. For this reason, it is required that each basic T-generator is constructively defined in HT, or is a basic HT-generator.

  In the presence of the keyword **partially**, a T-function may be partially implemented in HT, reflecting capacity constraints on the representation. This means that some T-values cannot be represented (except as implementation errors).

- As explained below, the statement **implements** T implies that for every provable T-lemma, the corresponding HT-lemma is also valid (at least in a subtype of HT, as explained below). The weaker statement **implements partially** T ensures that this is true for every provable T-lemma which is monotonic and without occurrences of domain-predicates of T-functions. In particular, an observation of a T-value is equal to the corresponding observation of the corresponding HT-value. This includes observation of error values, except in the case of an **implements partially** statement, where a T-expression may correspond to an implementation error.

Proof requirements

The correspondence between T and HT is formalized by an abstraction mapping **A** from T' to T (in the case of a partial implementation, to a subset of T), where T' is the subtype of HT whose value set is the set of HT-values spanned by the list $prod'$ of all implemented T-producers. Thus the subtype T' satisfies the lemma **genbas** $prod'$ in the sense that the corresponding induction rule is valid.

The signature **binrel equiv** and the axiom **total equiv** is included in T' (as the T'-function corresponding to T-equality), and its semantics may be specified by an equivalence-clause in the implements statement. Also appropriate signatures matching those of T are added, unless provided by the user.

Formally, an implements T statement means that the theory consisting of T, T', and the axioms 3.1-3 below regarding **A** is a consistent theory. (This formalization requires that the basic T-generators are implemented constructively in T'.) The abstraction mapping from T' to T must satisfy the following two axiom schemas:

$$\mathbf{A}(f'(w)) == f(w') \qquad \text{provided}\mathbf{D}\_f'(w), \quad \text{for producers} \tag{3.1}$$

$$f'(w) == f(w') \qquad \text{provided}\mathbf{D}\_f'(w), \quad \text{for non} - \text{producers} \tag{3.2}$$

where w is a variable list appropriate for f', and w' is equal to w except that each T'-variable, say x, is replaced by $\mathbf{A}(x)$, and f' denotes the HT-function that corresponds to f. It follows that each HT-function f' must satisfy

$$\mathbf{D\_}f'(w) \Rightarrow \mathbf{D\_}f(w').$$

If the stronger requirement

$$\mathbf{D\_}f'(w) = \mathbf{D\_}f(w')$$

holds, the function f is said to be *totally implemented*; otherwise it is said to be *partially implemented*. If the basic generators of T are totally implemented then every T-value has a representation in T', and if all T-functions are totally implemented then T itself is said to be totally implemented, otherwise partially implemented.

For a partial implementation of T the following requirement must be satisfied in addition to (3.1) and (3.2),

$$\mathbf{all}\ x : T' | \bigvee_i ((\mathbf{ex}\ w : W_i | \mathbf{A}(x) == \mathbf{A}(g_i'(w))) \wedge w << x) \tag{3.3}$$

where each g_i' corresponds to the T-generator g_i, W_i is its domain, and $w << x$ means that any T'-component of w, say y, is such that $\mathbf{A}(y)$ is less complex than $\mathbf{A}(x)$. The purpose of (3.3) is to ensure that T-induction carries over to the set $T'/\mathbf{equiv}$. If the generator basis of T has the one-one property the conjuncts $w << x$ are redundant, and so is (3.3) itself if the implementation of each T-generator is total.

The abstraction mapping is strict; and the following propagation axioms explain the correspondence between HT-errors and T-errors:

$\mathbf{A}(\mathbf{error}) == \mathbf{error}$
$\mathbf{A}(<\text{implementation error}>) == \mathbf{error}$
$\mathbf{A}(f'\ \mathbf{error}\ X') == f\ \mathbf{error}\ X$

where error X' has the same position in the signature of f' as X does in that of f.

As a consequence of an implements T-statement, for a provable T-lemma e, the T'-lemma e' is valid, where e' is obtained from e by first replacing any **case**-construct with a discriminand of type T by the corresponding **if any**-construct, as explained in section 4.11, and then replacing every occurrence of T by T', every occurrence of a T-function f (including T-equality) by the corresponding T'-function f', and renaming errors, as explained above. In the case of a partial implementation, e must be a monotonic formula and may not refer to domain predicates of partially implemented functions.

The consistency requirement above forms a simple formalization of an implements-statement, independent of a particular proof strategy. For proof purposes, it may be more practical to prove a set of stronger, but finitely provable, requirements. For instance, it is sufficient to prove as T'-lemmas the translation (as explained above) of all explicit T-axioms, as well as **A**-axiom (3.3), where the translation of T-definitions of non-implemented functions may be assumed. Rather than proving the whole set of translated congruence axioms for the **equiv**relation, it is usually more practical to prove an explicit definition of **equiv**using the **A**-axiom

$$(x\ \mathbf{equiv}\ y) == (\mathbf{A}(x) = \mathbf{A}(y))$$

and possibly induction with respect to the generator basis $prod'$. It may also be useful to establish a *representation invariant* I by proving $\mathbf{all}\ x : T' \mid I(x)$ using $prod'$-induction.

Whereas the proof of a theorem of the form **all** $x : T'|P$ usually requires induction with respect to $prod'$, it is often the case that

$$\textbf{all } x : HT|I(x)\Rightarrow P$$

is provable directly, for a sufficiently strong invariant I, cf. [Dahl 77].

Certain problems pertaining to the implements relation remain to be discussed.

Chapter 4

Expressions

⟨expr⟩ ::= ⟨literal⟩ | (⟨expr⟩) | ⟨variable⟩ |
⟨function application⟩⟨applicative postscript⟩* |
⟨error expr⟩ | ⟨expr tuple⟩ | ⟨special predicate⟩ | ⟨sequence related notation⟩ |
⟨qualified expr⟩ | ⟨let-expr⟩ | ⟨case-expr⟩ | ⟨if-expr⟩ | ⟨quantification⟩

The *Abel* expression language is a purely functional language, without side effects and with strong typing. There are expression constructs that enable efficient evaluation usable in executable code, as well as non-executable constructs to be used in formal reasoning. Constructs are executable unless otherwise is explicitly stated.

Literals are defined in section 1.3.4. Parentheses are used in the traditional way to indicate the nesting of expressions and, in addition, to form tuples.

4.1 Variables.

⟨variable⟩ ::= ⟨*variable* ident⟩ | ⟨boldface numeral⟩ | $

Variables are, or may be, introduced by (defining occurrences contained in) the following constructs:

- function and procedure headings (chapter 6), as well as error and branch headings (sections 4.3 and 5.2.4), introducing variables called formal arguments or formal parameters,
- discriminators (section 4.10), introducing formal variables,
- quantifications (section 4.12), axioms (section 3.3), and lemmas (section 3.4), introducing bound or quantified variables,
- let-expressions (section 4.9), introducing let-variables,
- subtype modules (section 7.4.2), introducing value parameters, and
- var- and const-declarations of an imperative statement list, introducing program variables and program constants, respectively.

Every variable has a unique type, either defined explicitly in a construct of the form

⟨*variable* ident⟩ : ⟨type expr⟩

or defined implicitly. Boldface numerals may be implicitly introduced in headings and discriminators, see section 4.10, and the special symbol $ denotes the implicit parameter of type constraints (section 7.4.2) and functions declared **obs**, **oper**, or **proc**(section 6.3)

Any applied variable occurrence may be considered a welldefined expression.

$$\Delta[x] == \mathbf{t}$$

That is mainly because all user defined functions are strict, so that operationally speaking the formal parameters represent welldefined values, and because in an imperative context program variables and program constants are initialized at the time of declaration.

4.2 Function Applications.

⟨function application⟩ ::= ⟨*funcappl* standard⟩ | ⟨*funcappl* mixfix⟩
⟨*funcappl* operator⟩ ::= ⟨function name⟩ | ⟨prefixed function⟩ | ⟨dotted function⟩
⟨prefixed function⟩ ::= ‘⟨function prefix⟩‘⟨function name⟩
⟨function prefix⟩ ::= ⟨type expr⟩ | ⟨empty⟩
⟨dotted function⟩ ::= ⟨expr⟩.[[⟨function name⟩ | ⟨boldface numeral⟩]]
⟨*funcappl* operand⟩ ::= ⟨expr⟩

A function introduced in mixfix notation may be applied in either mixfix or standard notation; in the latter case the function name is the “naming mixfix” (section 1.4) derived from the “domain mixfix” of the signature, rather than an ordinary function identifier.

There are two variants of standard notation in which the application operator is a function name specially embellished.

- Prefixed function, of the form ‘T‘f where T is a type or empty: A nonempty function prefix T is used to identify the owner type of the function (or a subtype not reintroducing f). Thereby the function is uniquely identified, independent of the standard binding rules of the language. The notation “f refers to the (one and only currently defined) free function f. This notation is not meanongful for the application of non-attribute functions.

- Dotted function, of the form $e.f$, where e is an expression: This notation is used for attribute functions (declared **obs**, **oper**, or **proc**). The prefixing expression is the dominant argument of f, and its value may be thought of as the “owner object” of the function $e.f$ in the sense of traditional object oriented programming.

All functions are strict, except the following built in ones.

$$==,\ \wedge,\ \vee,\ \Rightarrow,\ \textbf{andthen},\ \textbf{orelse},\ \textbf{impthen},$$

(see section 3.1 and 4.11), as well as some special notations described in the following sections. Thus, an ordinary function application is welldefined if and only if the arguments are welldefined and satisfy the domain predicate of the function.

$$\Delta[f(e)] == \Delta[e] \wedge \mathbf{D}\_f(e)$$

If e is welldefined the application $f(e)$, where f is defined by

$$\textbf{def}\ f(x) == ee,$$

is equivalent to

$$< ee >_e^x$$

even with respect to identity of error values (see section 3.1.2). If e is illdefined, the value of $f(e)$ is that of e. This is true also if e (really "(e)") is a tuple, in which case the value is that of the leftmost illdefined argument.

4.3 Postscripts

error	::= *applicative error* **/** *imperative error*
⟨*applicative error* postscript⟩	::= **on** ⟨*applicative error* handler⟩$^+$ **no**
⟨*error* handler⟩	::= ⟨*error* heading⟩$^+_{,}$ → ⟨*error* body⟩
⟨*error* heading⟩	::= ⟨*errorhead* standard⟩ **| others**
⟨*errorhead* operator⟩	::= ⟨*error* ident⟩
⟨*errorhead* operand⟩	::= ⟨*discriminator* operand⟩
⟨*applicative error* body⟩	::= ⟨expr⟩

Consider a postscript applied to an expression e with main operator f. The error headings of the error handlers are *formal error applications* patterned after the domains of errors associated with f. The operands are analogous to those of case discriminators, introducing explicit or implicit formal parameters, see section 4.10. The scope of the formal parameters of an error heading is the corresponding error body, and several headings may correspond to one body according to rules analogous to those of case discriminators. The heading **others** if present, must be the last one of the postscript, and represents the list of errors associated with f which are not listed explicitly (all operands implicit). The type of each error body must be coercible to that of e.

If the value of e is **error** $E(t)$ and the postscript contains a matching error handler $E(p) \rightarrow ee$, then the postscripted expression is equivalent to ee_t^p (or to ee if E has no parameters), coerced to the type of e if necessary, provided that the actual parameter tuple (t) is welldefined. If the parameters are illdefined the postscripted expression has the value **error** (anonymous). If e is welldefined or there is no matching error handler in the postscript, the value is that of the unpostscripted expression e itself.

4.4 Error Expressions

⟨error expr⟩	::= **error** **|** ⟨error prefix⟩$^?$ **error** ⟨error application⟩
⟨error prefix⟩	::= ⟨function name⟩ **|** ⟨prefixed function⟩
⟨error application⟩	::= ⟨*errorappl* standard⟩
⟨*errorappl* operator⟩	::= ⟨*error* ident⟩
⟨*errorappl* operand⟩	::= ⟨expr⟩

The value of an error expression is either anonymous error, or it is an error named by an error application. Any named error textually occurring within the body of a function f must be unprefixed, and must be named by an application of an error associated with f. Operationally its evaluation terminates the current invocation of f and may invoke an error handler, viz. if the current application (call) of f has a postscript containing a matching handler. If not, the error is treated as an anonymous error which causes abnormal termination of the whole program execution.

Error expressions occurring outside definition bodies must be prefixed by the function to which it is associated.

$$\Delta[...\textbf{error}\ ...] == \mathbf{f}$$

4.5 Tuples.

⟨expr tuple⟩ ::= ⟨*application* tuple⟩

An expression tuple may be regarded as an application of a polymorphic tupling function whose domain and codomain both are equal to the product of the types of the operands. The function has no associated errors. There is, however, no unary tupling function, and the parentheses of a singleton tuple are redundant, unless required for correct parsing or because the tuple is part of an application pattern. Non-trivial tuples are not associative.

A tuple $(e_1, e_2, ..., e_n)$, where $n > 1$ is said to be of *order* n. Since a type product of order n has the default selector functions **1**, **2**,...,**n** (see section 8.3), each of them may be used to select a component of a tuple, explicit or not, of order n.

A tuple is welldefined if all its components are welldefined; otherwise its value is that of the leftmost illdefined component.

$$\Delta[(e_1, e_2, ..., e_n)] == \Delta[e_1] \wedge \Delta[e_2] \wedge ... \wedge \Delta[e_n]$$

4.6 Special Predicates.

⟨special predicate⟩ ::= ⟨subtype test⟩ | ⟨domain-predicate⟩
⟨subtype test⟩ ::= ⟨expr⟩**isin**⟨type expr⟩
⟨domain-predicate⟩ ::= ⟨*domainpred* standard⟩
⟨*domainpred* operator⟩ ::= **D_**⟨function name⟩
⟨*domainpred* operand⟩ ::= ⟨expr⟩

Let $T_0, T_1, ..., T_n,\ n \geq 0$ be a sequence of types such that T_0 is a basetype, and T_k is a direct subtype of $T_k - 1$ with constraint C_k (possibly empty), $0 < k \leq n$. (See section 7.4.) Let e be an expression whose basetype is T_0. Then the expression e **isin** T_n means

$$(C_1)^{\$}_e \textbf{ andthen } ... \textbf{ andthen } (C_n)^{\$}_e$$

provided e and T are welldefined. Otherwise the subtype test is an anonymous error.

A domain-predicate tests whether or not a function application is welldefined. Thus, **D_**$f(x)$ and x**isin** T are equivalent if T is the domain of f. If the domain of f is constrained the user has the proof obligation to prove that the operand tuple of the domain-predicate strongly satisfies the constraint.

4.7 Sequence Related Notations.

⟨sequence related notation⟩ ::= [⟨expr⟩$^+_,$] | [⟨expr⟩;⟨expr⟩] | [⟨expr⟩..⟨expr⟩] |
⟨*sequence* expr⟩[⟨expr⟩] |
⟨*sequence* expr⟩[⟨expr⟩;⟨expr⟩] |
⟨*sequence* expr⟩[⟨expr⟩..⟨expr⟩] |
⟨*sequence* expr⟩[⟨expr⟩ → ⟨expr⟩]

A number of special sequence related notations using square brackets are built in by ad-hoc syntax. They are defined semantically as if they were definable mixfix notations (section 7.7.4). The sequence associated ones are redefined as array associated functions as well (section 8.5).

4.8 Qualified Expressions

⟨qualified expr⟩ ::= ⟨expr⟩ ⟦**qua**|**as**⟧ ⟨type expr⟩

In the qualified expression e **qua** T, T must be a subtype of (or equal to) the basetype of the expression e. The type of the qualified expression is T, and operationally e is accepted as a T-value without any runtime checking. There is, however, an obligation to prove that e satisfies the constraints of T; the lemma

$$e \textbf{ isin } T == \textbf{t}$$

is added marked "not proven".

$$\Delta[e \textbf{ qua } T] == \Delta[e] \wedge \Delta[T]$$

In the expression e **as** T the expression e must be *coercible* to the type T. Let the type of e be U. Then e is coercible to T if and only if $U \subseteq T$ (no checking necessary), or $T \subseteq U$, or there is an explicitly defined coercion function from U' to T', where U' is the basetype of e and $T \subseteq T'$.

$$\Delta[e \textbf{ as } T] == \Delta[e] \wedge \Delta[T] \wedge \begin{cases} \textbf{t} & \text{if } U \subseteq T \\ \Delta[e \textbf{ isin } T] \wedge e \textbf{ isin } T & \text{if } T \subseteq U \\ \textbf{D\_\^{}as } T(e) & \text{otherwise} \end{cases}$$

The qualifiers **qua** and **as** both play a role in the binding of the main operator of e if the function is unprefixed and has no dominant argument. See section 7.5, step 4. It is assumed that all occurrences of the expression e above are bound accordingly.

4.9 Let-expressions

⟨let-expr⟩ ::= **let** ⟨let-equation⟩ **in** ⟨expr⟩ **ni**
⟨let-equation⟩ ::= ⟨*discriminator* operand⟩ = ⟨expr⟩

The let-expression

$$\textbf{let } x = e \textbf{ in } ee \textbf{ ni}$$

means ee^x_e, provided the expression e is welldefined, otherwise its value is that of e. The left hand side of a let-equation may in general be any discriminator operand of the type of the right hand side (see the next section); and the scope of the formal variables introduced is the expression enclosed by the symbols **in** and **ni**.

$$\Delta[\textbf{let } x = e \textbf{ in } ee \textbf{ ni}] == \Delta[e] \wedge \Delta[ee^x_e]$$

4.10 Case-Expressions

Abel ::= *applicative* | *imperative*
⟨case-expr⟩ ::= ⟨*applicative* case costruct⟩
⟨*Abel* case construct⟩ ::= **case** ⟨discriminand⟩ **of**
⟦⟨discriminator⟩$^{+}_{,}$ → ⟨*Abel* alternative⟩⟧$^{+}_{▯}$ **fo**
⟨discriminand⟩ ::= ⟨expr⟩
⟨discriminator⟩ ::= ⟨*discriminator* standard⟩ | ⟨*discriminator* operator⟩() |
⟨*discriminator* mixfix⟩ | **others**
⟨*discriminator* operator⟩ ::= ⟨*basic function* ident⟩ | ⟨empty⟩
⟨*discriminator* operand⟩ ::= ^ | ⟨*formal variable* ident⟩ | ⟨discriminator⟩
⟨*applicative* alternative⟩ ::= ⟨*alternative* expr⟩

A case construct is used to choose between alternatives on the basis of the value of an expression called the *discriminand*. A generator basis must be defined for the type of the discriminand, say T. (This is true for any non-formal type, but may not be for formal ones.) Let the generator basis be $g_1, g_2, ..., g_n,\ n \geq 0$. As explained in section 3.2, this implies that any value of type T is of one of the forms $g_i(...),\ 1 \leq i \leq n$, and the objective is to discriminate between these n possibilities.

A *discriminator* therefore looks like an application of a basic generator for the type T and is called a *formal application* of that function. (If T is a product type, or a subtype of a product, the only discriminator takes the form of a tuple.) The operands, if any, of a discriminator are typed as specified in the signature of the basic generator. A type U operand is either:

1. the defining occurrence of a *formal variable* of type U whose scope is the textually following alternative, or

2. a type U discriminator, but only if the generator basis of U consists of a single function, or

3. the separator ^, which stands for the *default identifier* for that argument position, interpreted as in case 1. If the corresponding type component U of the discriminator domain is labelled (see section 8.3) the default identifier is that label, otherwise it is the special identifier consisting of a boldface numeral specifying the ordinal number of the argument position. If in standard notation all operands are default identifiers the whole operand tuple may be replaced by an empty pair of parentheses.

The formal variables (effectively) occurring in a discriminator must be mutually distinct.

An applicative case construct, or *case-expression*, for a discriminand e of type T would be of the form

case e **of** $g_1(x_1, x_2, ...) \rightarrow e_1$ ▯ $g_2(x_1, x_2, ...) \rightarrow e_2$ ... ▯ $g_n(x_1, x_2, ...) \rightarrow e_n$ **fo**

where the discriminators, one for each basic generator, may be listed in an arbitrary order. If the value of e is $g_i(a, b, ...)$ the case-expression is equal to the alternative e_i with all free occurrences of $x_1, x_2, ...$ replaced by $a, b, ...$, respectively. All alternative expressions must be of related types; the type of the case-expression is the closest common ancestor type.

Operationally the case construct is a pattern matching mechanism which selects an alternative according to the main operator of a T-value and assigns its arguments, if any,

to the formal variables of the corresponding discriminator, and then evaluates the selected alternative.

If an observation basis is defined for the type T of the discriminand, logical inconsistency may arise from the use of the case construct. The user is obliged to prove that this will not happen, and to that end the following lemmas are added marked "not proven":

$$g_i(x_1, x_2, ...) \textbf{ isin } T \ \wedge \ g_j(y_1, y_2, ...) \textbf{ isin } T \ \wedge \ g_i(x_1, x_2, ...) = g_j(y_1, y_2, ...) \ \Rightarrow \ e_i = e_j$$

for $i, j = 1, 2, ..., n, \ i \neq j$.

The case construct is non-strict; in order that a case-expression be welldefined it is sufficient that the discriminand and the selected alternative expression are welldefined. If the latter is illdefined its error value is propagated, see section 3.1. If the discriminand is illdefined the case-expression has that same error value.

$$\Delta[\textbf{case } e \textbf{ of } .. \ [] \ g_i(x_i) \rightarrow e_i \ [] \ .. \textbf{ fo}] == \Delta[e] \wedge \textbf{case } e \textbf{ of } .. \ [] \ g_i(x_i) \rightarrow \Delta[e_i] \ [] \ .. \textbf{ fo}]$$

Examples

Let e be an expression of the union type $(u : U + v : V)$ (section 8.4). Then, in the case-expression

$$\textbf{case } e \textbf{ of } u(x) \rightarrow e_1(x) \ [] \ v(x) \rightarrow e_2(x) \textbf{ fo}$$

the formal variable x of the first branch is of type U, and the x of the second branch is of type V.

For an expression e of the product type $(u : U, \ v : V)$ (section 8.3) the case expression

$$\textbf{case } e \textbf{ of } (x, y) \rightarrow ee \textbf{ fo}$$

means the same as the expression ee with all free occurrences of x and y replaced by $e.u$ and $e.v$, respectively (provided e is welldefined). And in the expression

$$\textbf{case } e \textbf{ of } \rightarrow ee \textbf{ fo}$$

the empty discriminator stands for $(\hat{}, \hat{})$, which in turn stands for (u, v).

4.10.1 Abbreviated case-expressions.

Several discriminators may lead to the same alternative expression, as in $f(...), g(...) \rightarrow e$, which abbreviates $f(...) \rightarrow e \ [] \ g(...) \rightarrow e$. It is required that only those formal variables which are (effectively) defined and have related types in all discriminators of the list, occur free in e. The type of each variable as occurring in e is the closest common ancestor of its defined types.

The special symbol **others** may occur as the last discriminator of a case construct. It is an abbreviation for the list of those basic generators of the discriminand type which have not occurred explicitly, all with implicit formal arguments.

Example: The following expression computes the suffix of the English rendering of a day of month ordinal:

$$\textbf{case } \textit{day} \textbf{ of } \textit{1,21,31} \rightarrow \text{'st'} \ [] \ \textit{2,22} \rightarrow \text{'nd'} \ [] \ \textit{3,23} \rightarrow \text{'rd'} \ [] \textbf{ others} \rightarrow \text{'th'} \textbf{ fo}$$

where *day* has the type {1..31}.

Any formal variable of a type T whose generator basis consists of a single function may be replaced by a type T discriminator, throughout the scope of the variable and including the defining occurrence. In this way it is possible to save a case construct which might otherwise be needed on that variable. For instance, an expression of the form

$$\textbf{case } e \textbf{ of } f(..., x, ...) \rightarrow \textbf{case } x \textbf{ of } g(...) \rightarrow ee \textbf{ fo } [] \ ... \textbf{ fo}$$

where the type of x has the single basic generator g, may be simplified as follows.

$$\textbf{case } e \textbf{ of } f(..., g(...), ...) \rightarrow ee^{x}_{g(...)} \ [] \ ... \textbf{ fo}$$

Also formal variables introduced in the heading of a function (or procedure) definition or an error handler may be thus replaced. For instance, the following two function definitions are equivalent.

$$\textbf{def } f(..., x, ...) == ... \textbf{ case } x \textbf{ of } g(...) \rightarrow e \textbf{ fo } ...$$

$$\textbf{def } f(..., g(...), ...) == (...e...)^{x}_{g(...)}$$

Nested case-expressions on "disjoint" expressions may be combined into one case discriminating on a tuple. In that case each discriminator should take the form of a tuple of discriminators. The full Cartesian product of the individual generator bases must be listed, unless an **others** clause is provided. Thus, an expression of the form

$$\textbf{case } (e, ee) \textbf{ of } ... \ [] \ (g(...), gg(...)) \rightarrow a \ [] \ ... \textbf{ fo}$$

is an abbreviation for

$$\textbf{case } e \textbf{ of } ... \ [] \ g(...) \ \rightarrow \ \textbf{case } ee \textbf{ of } ... \ [] gg(...) \rightarrow a \ [] \ ... \textbf{ fo } [] \ ... \textbf{ fo}$$

where g and gg are basic generators for the types of e and ee, respectively.

4.11 If-expressions

⟨*Abel* if-construct⟩ ::= **if** ⟦⟦⟨*Boolean* expr⟩ **|** ⟨any-test⟩⟧**then** ⟨*Abel* alternative⟩⟧$^{+}_{\textbf{elsif}}$
⟦**else** ⟨*Abel* alternative⟩⟧$^{?}$ **fi**
⟨if-expr⟩ ::= ⟨*applicative* if-construct⟩

An if-construct is a shorter and more traditional notation for a case-construct on a discriminand of type *Boolean*. Thus, the case-expression

$$\textbf{case } b \textbf{ of t } \rightarrow e_1 \ [] \ \textbf{f} \rightarrow e_2 \textbf{ fo}$$

where b is a *Boolean* expression, may be written as

$$\textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}$$

and **if** b **then** e **fi** means **if** b **then** e **else error fi**. There is also a shorthand for the case that the else-alternative is itself an if-construct, so that **else if** ... **fi fi** may be replaced by **elsif** ... **fi**. Thus,

$$\textbf{if } b_1 \textbf{ then } e_1 \textbf{ elsif } b_2 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ fi}$$

means

$$\textbf{if } b_1 \textbf{ then } e_1 \textbf{ else if } b_2 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ fi fi}$$

where b_1 and b_2 are *Boolean* expressions. Notice that the alternatives of an if-expression must be expressions of related types.

The non-strict but executable *Boolean* operators **andthen**, **orelse**, and **impthen** are abbreviations for if-constructs. The expressions

$$b_1 \textbf{ andthen } b_2 \ , \qquad b_1 \textbf{ orelse } b_2 \ , \qquad b_1 \textbf{ impthen } b_2$$

abbreviate

$$\textbf{if } b_1 \textbf{ then } b_2 \textbf{ else f fi} \ , \quad \textbf{if } b_1 \textbf{ then t else } b_2 \textbf{ fi} \ , \quad \textbf{if } b_1 \textbf{ then } b_2 \textbf{ else t fi}$$

respectively.

An *any-test* is a quantification (see the next section) which may only occur as the discriminand of an if-construct. It represents an abstract, non-excutable "searching" mechanism. If the search is successful there is access to the searched-for value through the bound variable, whose scope (for this particular kind of quantification only) extends through the following alternative. The expression

$$\textbf{if } (\textbf{any } x : T \mid b) \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}$$

is equivalent to

$$\textbf{if } (\textbf{ex } x : T \mid b) \textbf{ then let } x = \textbf{some } x : T \mid b \textbf{ in } e_1 \textbf{ ni else } e_2 \textbf{ fi}$$

where the occurring quantifiers are explained in the next section.

Using any-tests an arbitrary case-construct

$$\textbf{case } e \textbf{ of } \ldots \ [] \ g_i(x_i) \rightarrow e_i \ [] \ \ldots \textbf{ fo}$$

may be rewritten as an if-construct:

$$\textbf{if } \ldots \textbf{ elsif any } x_i : T_i \mid e = g_i(x_i) \textbf{ then } e_i \textbf{ elsif } \ldots \textbf{ fi}$$

where $x_i : T_i$ stands for an appropriate list of fresh variable declarations. Notice that the implicit final alternative **else error** takes care of the case that the case-discriminand e is illdefined. The two constructs are usually equivalent, however, in the case where an observation basis is defined for the type of e, equivalence is guaranteed only if the former expression satisfies the consistency requirement of section 4.10 (or an analogous one for imperative constructs).

4.12 Quantifications

⟨quantification⟩	::= ⟨universal quantification⟩ **|** ⟨existential quantification⟩ **|** ⟨some-expr⟩ **|** ⟨any-test⟩
⟨universal quantification⟩	::= **all** ⟨q-body⟩ **|** **total**⟨function prefix⟩$^?$⟨function name⟩
⟨existential quantification⟩	::= **ex** ⟨q-body⟩
⟨some-expr⟩	::= **some** ⟨type expr⟩ **|** **some** ⟨q-body⟩
⟨any-test⟩	::= **any** ⟨q-body⟩
⟨q-body⟩	::= ⟦⟨*bound variable* ident⟩$,^+$: ⟨type expr⟩⟧$,^+$ **|** ⟨*Boolean* expr⟩

Quantifications are non-executable.

4.12.1 Universal quantification

A universal quantification

$$\textbf{all } x : T \mid b$$

is a *boolean* expression which has the value **t** if the expression b is true for those values of x for which it is defined, and is **f** otherwise. It can be seen as an abbreviation of the quantification

$$\textbf{all } x : T \mid \textbf{not } (b == \textbf{f})$$

and therefore a universal quantification is by definition welldefined.

$$\Delta[\textbf{all } x : T \mid b] == \textbf{t}$$

The quantification **all** $x, y : T$ ... is an abbreviation for **all** $x : T \mid$ (**all** $y : T$ ...) Similarly, the quantification **all** $x : T1, y : T2$ ... is an abbreviation for **all** $x : T1 \mid$ (**all** $y : T2$ ...) These abbreviations apply to all quantifiers.

The notation

$$\textbf{total } f$$

where f is the name of a function, abbreviates

$$\textbf{all } x : T \mid \textbf{D\_}f(x)$$

where T is the domain of f. And **total** f, g abbreviates **total** $f \wedge$ **total** g.

Example: **all** $y : Integer \mid \ y/y = 1$
This formula is welldefined and true.

4.12.2 Existential quantification

The existential quantification

$$\textbf{ex } x : T \mid b$$

is defined as **not all** $x : T \mid$ **not** b, and is equivalent to **ex** $x : T \mid b == \textbf{t}$, i.e. there is a value of x in T such that b is welldefined and true.

$$\Delta[\textbf{ex } x : T \mid p] == \textbf{t}$$

4.12.3 Some-expressions

The some-construct **some** $x : T \mid b$ means: Take, non-deterministically, some value in T that satisfies b; and if not possible, take the value **error** *None*.

Semantically, each occurrence of the some-expression **some** $x : T \mid b$ is an abbreviation for an application of a fresh T-initializer, say $some_i(z)$ where z is the list of variables declared at the place of occurrence (including $ if it is within the body of an attribute function or a procedure), and where the function $some_i$ is characterized by the following axioms.

$$\textbf{let } x = some_i(z) \textbf{ in } b \textbf{ ni}$$

$$\textbf{D\_}some_i(z) == \textbf{ex } x : T \mid b$$

$$\textbf{not D\_}some_i(z) \ \Rightarrow \ some_i(z) == \textbf{error } None$$

The trick of replacing each textual occurrence of a some-expression by a different Skolem-function models the non-determinacy introduced (to a certain extent, see below).

$$\Delta[\textbf{some}\ x : T \mid b] == \textbf{ex}\ x : T \mid b$$

The some-construct **some** T means **some** $x : T \mid \mathbf{t}$.

Example: The expression

$$\textbf{ex}\ x : Nat1 \mid x = \textbf{some}\ Nat1$$

is interpreted as

$$\textbf{ex}\ x : Nat1 \mid\ x = some_1$$

where $some_1$ is a fresh Skolem constant of type *Nat1*. This interpretation obviously yields a valid formula.

Notice that a some-expression occurring in the body of a function f will be represented by the same Skolem-function for each application of f. Still the fact that the Skolem function depends on all the arguments to f may provide an illusion of "real" non-determinism. For instance, the input procedure *get* of an input stream object (section 9.1) changes the "input history", $\$.p$, upon every invocation. Thereby successive calls to the same *get* procedure will deliver inputs selected independently.

4.13 Disjointness restrictions

The fact that values of class-like types should be passed and assigned by means of pointer copying implies that problems of aliasing might occur in an imperative context. In order to prevent such problems the use of variables of class-like types in expressions of executable code is subject to the following "disjointness" requirement:

- Any two such variable occurrences both in the same expression, and neither contained in any expression of non-class type, must be disjoint, unless they are contained in textually disjoint case alternatives. Two variable occurrences are disjoint unless they are occurrences of the same variable, or if the one is an applied occurrence of a formal variable of a case discriminator, the other is outside the corresponding discriminand, but the discriminand contains a variable occurrence not disjoint with the other occurrence.

Examples: The expression $(v, v.a)$, not itself contained in a larger one, is illegal if v is a variable of a product type and a is a selector function selecting a value of a class-like type. If a selects a non-class value the expression is legal, even if the type of v is class-like, because the second occurrence of v is contained in an expression of non-class type.

The rightmost occurrences of x and z in

$$\textbf{case}\ x\ \textbf{of}\ (.., y, ..) \rightarrow\ \textbf{case}\ y\ \textbf{of}\ (.., z, ..) \rightarrow\ (x, z)\ \textbf{fo fo}$$

are not disjoint, because the occurrence of x is not disjoint with the y of the innermost case discriminand, which in turn is because the outermost discriminand contains x.

The following exceptions apply to the disjointness requirement:

1. For a variable x of a product type the expressions $x.a$ and $x.b$, where a and b are distinct selector functions, may co-occur, even if a and b both select components of class-like types. (In an imperative context and for a program variable x, $x.a$ and $x.b$ may be seen as disjoint program variables.)

2. For a variable x of a (class-like) sequence type the expressions $x[i]$ and $x[j]$ may co-occur, but the semantical requirement $i \neq j$ is a proof obligation. Similar exceptions apply to the co-occurrence of expressions of the form $x[i]$ and/or $x[i; n]$ and/or $x[i..j]$.

It follows from the form of the disjointness restriction that it is only effective for function body expressions, **return** expressions, and assignment right hand sides, whose types are class-like, as well as **error** expressions associated with functions of class-like codomains. Assignments of values of class-like types must satisfy additional requirements, see section 5.2 It is reasonable to classify code not satisfying these requirements as non-executable, because execution adhering to the axiomatic properties of the code would in that case have to be based on the technique of class object copying rather than pointer copying.

Chapter 5

Program Bodies.

This chapter is part of an old version of the ABEL report.
It needs readjustment in order to fit within the final version.
Also a section on the concept of effect functions is missing.

⟨Program body⟩ ::= **prog** ⟨Statement list⟩ **endprog**
⟨Statement list⟩ ::= ⟨Statement⟩$^{+}_{;}$
⟨Statement⟩ ::= ⟨Declarative statement⟩ | ⟨Imperative statement⟩ | ⟨Mythical statement⟩

In *Abel*, a program body consists of a statement list, which is a non-empty sequence of statements. Statements may be either Declarative, Imperative or Mythical.

5.1 Declarative Statements.

⟨Declarative statement⟩ ::= ⟨Var-declaration⟩ | ⟨Const-declaration⟩ | ⟨Branch-declaration⟩ | ⟨Scope terminator⟩
⟨Scope terminator⟩ ::= ⟨Variable terminator⟩ | ⟨Constant terminator⟩

5.1.1 Declaration of Variables and Constants.

⟨Var-declaration⟩ ::= **var** $[\![$⟨*Var* ident⟩ $^{+}_{,}$ $[\![$: ⟨Type expr⟩$]\!]^{?}[\![$= ⟨initialization⟩$]\!]^{?}$ $]\!]^{+}_{,}$
⟨Const-declaration⟩ ::= **const** $[\![$⟨*Const* ident⟩ $^{+}_{,}$ $[\![$: ⟨Type expr⟩$]\!]^{?}[\![$= ⟨initialization⟩$]\!]^{?}$ $]\!]^{+}_{,}$
⟨initialization⟩ ::= ⟨expr⟩
⟨Variable terminator⟩ ::= **endvar** ⟨*Var* ident⟩
⟨Constant terminator⟩ ::= **endconst** ⟨*Const* ident⟩

Local program variables and constants are introduced and declared by declarative statements called *variable declarations* and *constant declarations*. Their scopes are terminated by *variable terminators* and *constant terminators*.

Example 1 Variable and constant declarations.

const X: *Nat* = 5;
var *I, J*: *Nat1* = $X + 2$;
⋮
endconst X;

const *Temp* := *X*;
X := *Y*; *Y* := *Temp*
endconst *Temp*

(The last example, which swaps the contents of the two variables *X* and *Y*, could more elegantly be written as the swap statement "*X* :=: *Y*"; that statement is described in section 5.2.3.2.)

The semantic effect of the variable declaration statement

var *X*: *T* = *E*

is mainly the same as the assignment

$$X := E$$

The difference is that the declaration introduces a new variable (*X*), the scope of which is the rest of statement list in which the declaration occurred. It is, however, possible to terminate a scope explicitly by giving a variable terminator, as in

endvar *X*

This will terminate the scope of *X* as well as the scopes of all variables and constants declared between the declaration of *X* and its termination, and which have not already had their scopes terminated. An example of this is the declaration of *I* and *J* in the first part of example 1.

It is possible to omit the ⟨type expr⟩ or the ⟨initialization⟩, but not both. In the latter case the variable or constant is initialized to the default value of the given type; in the former case its type is declared to be that of the initial value.

A declaration naming several new variables (or constants), as in

var *A*, *B*, *C*, ... : *T* = *X*

is equivalent to a sequence of declarations:

var *A*: *T* = *X;*
var *B* := *A;*
var *C* := *B;*
⋮

The only difference between a variable and a constant is that program constants cannot be changed. This is possible to check at compile-time, since a ⟨General variable⟩ (see section 5.2.3.1) may not contain any program constants.

5.1.2 Declaration of Branches.

⟨Branch declaration⟩ ::= **branch** $[\![$⟨*Branch* ident⟩ ⟨Type name list⟩$^{?}]\!]^{+}_{;}$
in ⟨Statement list⟩ ⟨Branches⟩$^{?}$ **join**
⟨Branches⟩ ::= **fork** ⟨*Branch* handler⟩$^{+}_{[]}$
⟨Handler⟩ ::= ⟨ident⟩ ⟨Argument list⟩$^{?}$ → ⟨Statement list⟩

Branches are in many respects similar to exceptions in function definitions. Declared branches are invoked by means of the Invoke-statement, which may occur at any level textually inside the ⟨Statement list⟩. Invoking a branch will transfer the program control to the corresponding ⟨Branch handler⟩, which will be executed. After the completion of the branch handler, program execution will continue after the **join**. Not defining a handler for a branch B_i is equivalent to defining

$B_i \rightarrow Skip$

A branch name is known inside the ⟨Statement list⟩ constituting the body of the branch declaration, and also in all its branch handlers. It is thus possible from one branch handler to invoke other branch handlers declared in the same branch declaration. Branches may be parameterized. An actual parameter list containing classes must satisfy the disjointness requirements.

Example 2 Use of branches.

```
(* Binary search for Key in array A. *);
var Lo := A.Min, Hi := A.Max;
branch Found(Index), NotThere in
  loop
    invar A.Min≤Lo and Hi≤A.Max and
          (all I: [A.Min..Lo−1] | A[I] < Key) and
          (all J: [Hi+1..A.Max] | A[J] > Key);
    if Lo>Hi → invoke NotThere endif;
    const Inx := (Lo + Hi) / 2;
    assert Lo ≤ Inx ≤ Hi;
    if Key < A[Inx] → Hi := Inx − 1
     ▯Key > A[Inx] → Lo := Inx + 1
    else             invoke Found(Inx)
    endif
  endloop
fork Found(X) → Write('Found at index ' ⊢ X)
 ▯   NotThere → Write('No such element!')
join
```

5.2 Imperative Statements.

⟨Imperative statement⟩ ::= *Skip* | ⟨Invoke-statement⟩ | ⟨Return-statement⟩ | ⟨Update⟩
⟨If-statement⟩ | ⟨Case-statement⟩ | ⟨Loop-statement⟩

The statement *Skip* has the obvious meaning.

5.2.1 Return-Statement.

⟨Return-statement⟩ ::= **return** ⟨expr⟩$^?$

A return-statement in a function body, effectuates an immediate termination of the function and produce the result value, if there is any. The ⟨expr⟩ must be compatible with the output type of the function heading, see section 5.2.3.3.

Example 3 Return-statements.

return
return (12, **t**)
return error *Overflow*

5.2.2 Invoke-Statement.

⟨Invoke-statement⟩ ::= **invoke** ⟨*Branch* ident⟩ ⟨Argument list⟩$^?$

This statement is used to invoke a specific branch handler. Branches and branch handlers are described in section 5.1.2. For an example of the use of the Invoke-statement, see example 2 above.

5.2.3 Updates.

⟨Update⟩ ::= ⟨Assignment⟩ | ⟨Swap⟩ | ⟨Function call⟩

There are three main forms of updates:

- *Assignments*, where a value is computed and assigned.
- *Swaps*, where the contents of two variables are swapped.
- *Function calls*, in which a function is called, updating update variables and assigning values to output variables.

5.2.3.1 Assignments.

⟨Assignment⟩ ::= ⟨General variable⟩ := ⟨expr⟩
⟨General variable⟩ ::= ⟨Single variable⟩ | ⟨Compound variable⟩
⟨Compound variable⟩ ::= (⟨Single variable⟩$^+_,$)
⟨Single variable⟩ ::= ⟨Simple variable⟩ [[⟨Subscript⟩|⟨Component selection⟩]]*
⟨Subscript⟩ ::= [⟨expr⟩]
⟨Component selection⟩ ::= . ⟨*Function* ident⟩

The ⟨expr⟩ must be compatible with the type of the ⟨General variable⟩. The effect of an assignment is that the ⟨expr⟩ is first evaluated, and then the value is assigned to the ⟨General variable⟩ on the left side of the := . When necessary, the value of the ⟨expr⟩ is coerced to the type of the ⟨General variable⟩.

Example 4 Assignments.

$$x := y \qquad a[i+1] := b+1 \qquad (A[i], A[j]) := (A[j], A[i])$$
$$a := F(b) \qquad x.cmp1 := 5 \qquad (i, A[i]) := (1, 1)$$

An assignment with a compound left hand side is simultaneous; so the $A[i]$ in the last example refers to the "old" $A[i]$, i.e. before i has been assigned a new value.

The assignment statement has the proof rule

$$\{Q_e^x\}\ \ x := e\ \{Q\}$$

or, if the assignment involves coercion of e to the type x, say T,

$$\left\{Q^x_{[T](e)}\right\}\ \ x := e\ \{Q\}$$

when x is compound, occurrences of the components are replaced simultaneously by the corresponding components of e; and where substitution for indexed variables is handled the ususal way, i.e., the assignment $A[i] := e$ is semantically equivalent to $A := A[i \to e]$. In the case where the left hand side contains two or more subscripted variables referencing the same array A, the simultaneous assignments to A are semantically equivalent to nested updates on A, for instance, the assignment $(A[i], A[j]) := (x, y)$ is semantically equivalent to $A := A[i \to x][j \to y]$ (provided $i \neq j$ holds, which would be a proof obligation).

Disjointness restrictions.

In order to be legal, an assignment statement must satisfy the following disjointness requirements:

- The right hand side expression must obey the disjointness requirements of section 4.13.
- Each variable of a class-like type, occurring in the right hand side outside all (sub-)expressions of non-class types, must also occur as a component of the left hand side, or it must be a formal variable whose scope is dynamically terminated by the assignment statement.

5.2.3.2 Swaps.

⟨Swap⟩ ::= ⟨Single variable⟩ :=: ⟨Single variable⟩

Swaps are used to interchange values between variables. The two ⟨Single variable⟩s must be of the same type. The swap $x :=: y$ where both x and y may be indexed variables, is equivalent to $(x, y) := (y, x)$, or to *skip* if x and y are identical.

Example 5 Swap.

$a[i] :=: a[j]$

The last swap is legal irrespective of whether $i = j$ or not, whereas the assignment

$(a[i], a[j]) := (a[j], a[i])$

is legal only if $i \neq j$ can be verified.

5.2.3.3 Function Calls.

⟨Function call⟩ ::= [[⟨Std function call⟩ | ⟨Mixfix function call⟩]]⟨Imperative postscript⟩?
⟨Std function call⟩ ::= [[**call** | @⟨Single variable⟩.]]⟨function name⟩⟨call parameter list⟩?
[[=: ⟨General variable⟩]]?
⟨Mixfix function call⟩ ::= **call** ⟨call parameter⟩? ⟨Special symbol⟩
[[⟨call parameter⟩ ⟨Special symbol⟩]]* ⟨call parameter⟩?
[[=: ⟨General variable⟩]]?
⟨call parameter list⟩ ::= (⟨call parameter⟩$^{+}_{,}$) | ⟨call parameter⟩
⟨call parameter⟩ ::= ⟨expr⟩ | @⟨Single variable⟩

Update parameters are identified in each call by means of the *update symbol* @.

The semantics of the function call

call $f(t) =: x$

is equivalent to the assignment

$(y, x) := f(t')$

where t' denotes the parameter list t with all update-symbols removed and y denotes the list of update parameters (ordered as in the call). This means that the codomain of f must have the form (Ry, Rx) where Ry and Rx are lists of types such that y is compatible with Ry and x is compatible with Rx. Ry is called the *update type*, and Rx is called the *output type* of the call.

A function call to f must satisfy the following requirements:

- Formal update parameters (including implicit parameters) in the definiton of f must be actualized by update parameters.
- The assignment corresponding to the call must be legal, disregarding the former requirement.

Notice that if f is defined without use of formal update parameters, the call and assignment above are equivalent also with respect to legality. Alias conflicts are avoided by the latter requirement; in particular, it implies that the class update parameters must be disjoint, and that they must also be disjoint from other class parameters as well as global variables accessible inside the function.

Example 6 Function calls.

func *Div*(*Int*,*Int*) =: (*Int*,*Int*);
Stack{*Item*} **oper** *Push*(*Item*),
Poptop =: *Item*;
var *a*, *b*, *x*, *y*: *Int*,
V: **array** *Int* **of** *Nat*;
⋮

call $Div(@x, b) =: y$	Equivalent to the assignment $(x, y) := Div(x, b)$
call $Div(a, @x) =: y$	Equivalent to $(x, y) := Div(a, y)$
call $Div(@x, @y)$	Equivalent to $(x, y) := Div(x, y)$
call $@V[e] + 1$	Equivalent to $V[e] := V[e] + 1$
$@S.Push(x)$	Equivalent to $S := S.Push(x)$
$@S.Poptop =: x$	Equivalent to $(S, x) := S.Poptop$

The function calls in the last two examples are legal even if S is of a class. The corresponding assignments will not, however, be legal in the case where *Push* and *Poptop* are defined with a formal update parameter.

5.2.3.4 Function Call Postscripts.

⟨Imperative postscript⟩ ::= **on** ⟨*imperative error* handler⟩$^{+}_{[]}$ **endon**

where ⟨handler⟩ is defined in section 5.1.2 (page 42). The exceptions mentioned in the postscript must all be declared in the function signature.

Example 7 Function call postscripts.

```
func Div(Num, Denom: Nat0) =: (Nat0, Nat0) error ZeroDiv ==
prog
  if Denom= 0 then return error ZeroDiv endif;
  var Quot := 0,
      Rem := Num;
  loop while Rem≥Denom;
    call @Quot+1;  call @Rem−Denom;
  repeat
  return (Quot, Rem)
endprog
  ⋮
call Div(1, @x) =: Remainder
on Zerodiv → Remainder := 0
endon
```

Note that if the call had been

(*Remainder*, x) := (*Div*(1, x) **on** *Zerodiv* → 0 **no**)

the postscript would be an applicative one bound to the expression (the function application) rather than the statement (the function call). The effect, however, would be the same.

5.2.4 Case-Statement.

⟨Case-statement⟩ ::= ⟨Simple case-statement⟩ **|** ⟨Updating case-statement⟩

5.2.4.1 Simple case-statement

⟨Simple case-statement⟩ ::= **case** ⟨expr⟩ **of** ⟨Case⟩$^{+}_{[]}$
⟦**others** ⟨Statement list⟩⟧$^{?}$ **endcase**
⟨Case⟩ ::= ⟨formal application⟩$^{+}_{,}$ ⟦ → **|then**⟧ ⟨Statement list⟩

This statement, which is the imperative counterpart of a Case-expression (see 4.10, page 34), will select one statement list from a set of alternatives according to use of generator base functions.

The semantics of a simple case-statement can be explained by the following proof rule:

$$\frac{..;\ \vdash \{P_i\}S_i\{Q\}\ ;\ ..}{\vdash \{\textbf{case}\ e\ \textbf{of}\ ..\ \square\ g_i(y) \rightarrow P_i\ \square\ ..\ \textbf{fo}\ \}\ \textbf{case}\ e\ \textbf{of}\ ..\ \square\ g_i(y) \rightarrow S_i\ \square\ ..\ \textbf{fo}\ \{Q\}}$$

where there is one premise for each case alternative.

One particular use of the case-statement is to select among different values of an enumeration type. In this case the different values form the generator base.

Example 8 Selection among values of an enumeration type.

var *V*: {*Red, Blue, Green*};
⋮
case *V* **of**
  Red → ...
□*Green* → ...
□*Blue* → ...
endcase

Another use of the Case-statement is to determine the actual type of a value of a disjoint union type. For a disjoint union the injection functions of the various constituent types form the generator basis.

Example 9 Selection among values of a disjoint union type.

var *X*: (*N:Nat* + *I:Int* + *C:Char* + *B:Bool*);
⋮
case *X* **of**
  N(*x*) → *Nvar* := *x*
□*I*(*x*) → *Ivar* := *x*
□*C*(*x*) → *Cvar* := *x*
□*B*(*x*) → *Bvar* := *x*
endcase

It is also possible to use the Case-statement to choose on the basis of a numeric value. Since all numeric literals are regarded as generators of the enumeration type *Num*, it is sufficient to convert the numeric value to a *Num* value, as in the following example:

Example 10 Selection among numeric literals.

case [*Num*](*Date*) **of**
  1, 21, 31 → *Suffix* := 'st'
□2, 22 → *Suffix* := 'nd'
□3, 23 → *Suffix* := 'rd'
others *Suffix* := 'th'
endcase

5.2.4.2 Updating case-statement

⟨Updating case-statement⟩ ::= **case** @⟨simple variable⟩ **of** ⟨Updating case⟩$^{+}_{\square}$
⟦**others** ⟨Statement list⟩⟧$^{?}$ **endcase**
⟨Updating case⟩ ::= ⟨formal call⟩$^{+}_{,}$ ⟦→ **|then**⟧ ⟨Statement list⟩

where a ⟨formal call⟩ is an formal application where some (or none) of the formal variables are preceded by the update symbol @. Such a parameter is called a *formal update variable*, and may be updated in the following statement list.

By an updating case it is possible to update the components of a variable. Components to be updated, must occur in the appropriate case alternative as update parameters. In order to avoid aliasing, the access to x in a **case** @x statement is restricted as follows:

- Write access to the variable x is lost in an inner ⟨Updating case⟩ with formal update variables, except in update statements dynamically terminating the case.

The semantics of an updating case can be explained by the following proof rule:

$$\frac{..;\ \vdash \{P_i\}S_i'\{Q\}\ ;\ ..}{\vdash \{\textbf{case}\ x\ \textbf{of}\ ..\ \square\ g_i(y') \rightarrow P_i\ \square\ ..\ \textbf{fo}\ \}\ \textbf{case}\ @x\ \textbf{of}\ ..\ \square\ g_i(y) \rightarrow S_i\ \square\ ..\ \textbf{fo}\ \{Q\}}$$

where y' is y with all update symbols removed, and where S_i' is obtained from S_i by inserting $;\ x := g_i(y')$ after each update to a formal update variable in y. It is assumed that the case statement satisfies the access restrictions to x stated above.

5.2.5 If-Statement.

⟨If-statement⟩ ::= **if** ⟨Guarded Statement List⟩$^{+}_{\square}$
[[**else** ⟨Statement List⟩]]$^{?}$ **endif**
⟨Guarded Statement List⟩ ::= ⟨Guard⟩ [[→ **|then**]] ⟨Statement List⟩
⟨Guard⟩ ::= ⟨*Boolean* expr⟩ **|** ⟨Any-test⟩

The If-statement is used to choose from a set of alternative statement lists. The different ⟨Guard⟩s will be evaluated in the order given, until one evaluates to **t**. The corresponding statement list will then be executed, and the other guarded statement lists will be ignored. If all guards evaluate to **f**, the statement list following the **else** will be executed. Omitting the **else** is equivalent to writing "**else** *Skip*".

The effect of a simple If-statement (i.e. one containing only one guard) is defined to be

$$\frac{\vdash \{P \wedge B\}S_1\{Q\}, \vdash \{P \wedge \neg B\}S_2\{Q\}}{\vdash \{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{endif}\ \{Q\}}$$

An If-statement containing more than one guard is equivalent to a nested If-statement, i.e.,

if ... **elsif** $G\ \rightarrow S\ \ldots$ **endif**

is equivalent to

if ... **else if** $G\ \rightarrow S\ \ldots$ **endif endif**

Example 11 If-statement.

if $X < 0\ \rightarrow$ **call** @*Neg* +1
$\square X = 0\ \rightarrow$ **call** @*Zero* +1
$\square X > 0\ \rightarrow$ **call** @*Pos* +1
endif

If a ⟨Guard⟩ is an ⟨Any-test⟩, the test will loop through all elements of the type until one is found that satisfies the predicate. In that case, the ⟨Guard⟩ is said to evaluate to **t**; otherwise it is said to evaluate to **f**. The scope of the constant being declared in the ⟨Any-test⟩, and which will contain the first value satisfying the predicate, is the ⟨Statement list⟩ following the ⟨Guard⟩.

Example 12

if any X: $[1..100] \,|\, A[X] = Val$ **then**
  $Answer := X$
else invoke *NotThere* **endif**

5.2.6 Loop-Statement.

⟨Loop-statement⟩ ::= ⟨Loop prefix⟩$^?$ **loop** ⟨Invariant⟩$^?$ ⟨Loop body⟩ ⟦**endloop**|**repeat**⟧
⟨Loop body⟩ ::= ⟨Statement list⟩ ⩊ ⟨While-clause⟩ ⩊ ⟨Statement list⟩
⟨Invariant⟩ ::= **invar** ⟨*Boolean* expr⟩ ;

5.2.6.1 Basic loops.

A Loop-statement without ⟨While-clause⟩, ⟨Loop prefix⟩ and Invoke-statements constitutes an infinite loop. The semantics for such a basic loop are given by the following rule:

$$\frac{\vdash \{I\}S\{I\}}{\vdash \{I\}\ \textbf{loop invar}\ I;\ S\ \textbf{repeat}\ \{\textbf{f}\}}$$

It is possible to exit a loop by invoking a *branch* (see section 5.1.2). All Loop-statements have a default branch named *Exit*. This means that the prescript "**branch** *Exit* **in**" and postscript "**fork** *Exit* → *Skip* **join**" are appended to every Loop-statement. The semantics of a Loop-statement containing one or more invocations of *Exit* are:

$$\frac{\{Q\}\ \textbf{invoke}\ Exit\ \{\textbf{f}\} \quad \vdash \{I\}S\{I\}}{\{I\}\ \textbf{loop invar}\ I;\ S\ \textbf{repeat}\ \{Q\}}$$

5.2.6.2 While-clause.

⟨While-clause⟩ ::= ⟨Invariant⟩$^?$ **while** ⟨*Boolean* Expression⟩ ;

The purpose of a While-clause is to terminate a loop if a given condition is not satisfied. If the expression following **while** evaluates to **f**, the loop terminates. Thus, writing "**while** B; " is equivalent to writing "**if not** B **then invoke** *Exit* **endif**; ".

Even though the Loop-statement sometimes consists of two separate statement lists (separated by the While-clause), they are regarded as belonging to the same scope. Thus a local declaration made before the while-clause will also be known both in the while-clause and after the while-clause.

Example 13 Loop-statement with While-clause.

loop
  call $@I + 1$;
while $I \leq Max$;
  call $@A[I] + 1$;
repeat

5.2.6.3 Loop Prefix.

⟨Loop prefix⟩ ::= **for** ⟨ident⟩ : ⟨Type expr⟩

Yet another way of controlling a loop is by using a counter (a *controlled variable*). The limits of the controlled variable (the ⟨ident⟩) are given by its ⟨Type Definition⟩. In the loop

for X: $[E_1 .. E_2]$ **loop**
  S
endloop

the controlled variable X will be given the values

$$E_1,\ \mathbf{S}\ E_1,\ \mathbf{S}\ (\mathbf{S}\ E_1),\ \ldots,\ E_2$$

The type given in the ⟨Type Definition⟩ must satisfy the property *LimLinOrd* (see section 7.6) which states that the type must have the functions **S**, **P**, $<$ and $\leq$. The effect of the statement above is equivalent to the following:[1]

const *%First* := E_1, *%Last* := E_2;
if *%First* $\leq$ *%Last* $\rightarrow$
  var *%Index* := *%First*;
  loop
    const X := *%Index*;
    S;
    endconst X
  while *%Index* $<$ *%Last*;
    call S @*%Index*
  endloop
endif
endconst *%First*

Notice that:

- For the user, X is accessible as a program constant.
- The expressions E_1 and E_2 are evaluated only once. This is done before the first execution of the loop.

It is possible to define loops counting downwards by using the **rev** operator on the type of the controlled variable.

Example 14 For-loop counting downwards.

for Nx: **rev** $[2..N]$ **loop**
  $A[Nx] := A[Nx - 1]$
endloop

[1] identifiers starting with a % are introduced to indicate that the identifier is different from all user-defined identifiers.

5.3 Mythical statements.

⟨Mythical statement⟩ ::= "[[⟨Declarative statement⟩|⟨Imperative statement⟩]]" |
⟨Assertion⟩ | ⟨Invariant⟩
⟨Assertion⟩ ::= **assert** ⟨*Boolean* expr⟩

Mythical statements are statements whose purpose is the documentation of programs rather than their implementation. Thus mythical statements will never produce any executable code.

Other mythical statements occur as parts of various statements, like the Invariant-clause of a loop statement. These mythical statements are discussed together with the statement of which they are a part.

A mythical statement can be constructed by placing an "ordinary" statement in quotes, like

"**const** *C0* := *C*"

This mythical constant declaration makes it possible to refer to the value of C in mythical statements. Declarations made in mythical parts are invisible in non-mythical parts of the program. Mythical statements may not alter the value of non-mythical variables.

A mythical statement may also specify some condition which must hold at a particular place in the program. Assertions are used for this purpose.

assert $X > 0$
assert all $I : [Lo\,..\,Hi]\,|\,A[I] < Limit$

Chapter 6

Function Definitions

non-coercion	::= *ordinary* **/** *initiator* **/** *attribute* **/** *binop* **/** *procedure*
⟨definition item⟩	::= **def** ⟦⟨*function* definition⟩ **|** ⟨combined definition⟩⟧
⟨*non-coercion* definition⟩	::= ⟨*non-coercion* heading⟩⟨domain constraint⟩$^?$== ⟨*non-coercion* body⟩
⟨*coercion* definition⟩	::= ⟨*coercion* heading⟩==⟨*coercion* body⟩

A function definition is a semantic characterization of a function previously introduced, which takes the form of a defining equation. All aspects of the function are defined, including the associated domain predicate, as well as the resulting error values when illdefined, see sections 3.1.1 and 3.1.2.

It is possible to combine the definition of a function with its signature to form a function declaration as in traditional programming languages, see section 6.4, but it is often more convenient and sometimes necessary to separate the two, because *Abel* requires names to be introduced before they can be applied.

6.1 The Definition Heading

ordorinit	::= *ordinary* **/** *initiator*
⟨*ordorinit* heading⟩	::= ⟨standard heading⟩ **|** ⟨*funchead* mixfix⟩
⟨standard heading⟩	::= ⟨*funchead* standard⟩ **|** ⟨*funchead* operator⟩()
⟨*funchead* operator⟩	::= ⟨*function* ident⟩
⟨*funchead* operand⟩	::= ⟨*discriminator* operand⟩
⟨*attribute* heading⟩	::= ⟨standard heading⟩
⟨*binop* heading⟩	::= ⟨*funchead* operand⟩⟨special symbol⟩⟨*funchead* operand⟩
⟨*procedure* heading⟩	::= ⟨*prochead* standard⟩ **|** ⟨*prochead* operator⟩()
⟨*prochead* operator⟩	::= ⟨*procedure* ident⟩
⟨*prochead* operand⟩	::= @$^?$⟨*discriminator* operand⟩
⟨*coercion* heading⟩	::= ⟨*funchead* operand⟩ **as** ⟨type expr⟩

A definition heading should be in mixfix or standard notation as indicated by the corresponding function signature. In particular that of a binary operator declared **binrel** or **assoc** should be in infix notation.

The heading of an attribute function or a procedure should be analogous to the signature in leaving the dominant home type parameter implicit. The formal parameter is named by the special symbol \$. Thus, the heading of an attribute function, $f(...)$, or a procedure, $p(...)$, is interpreted as an abbreviation for $\$.f(...)$ or $@\$.p(...)$, respectively.

Otherwise the operands of any function heading are analogous to those of a case discriminator, therefore the heading is called a *formal application* of the function being defined. The formal variables explicitly or implicitly introduced, including a possible \$, are called *formal parameters*, and their scope consists of the domain constraint, if any, as well as the function body. If in standard notation all formal parameters are implicit the whole tuple may be replaced by an empty pair of parentheses. In that case the parameters inherit any update symbols occurring in the domain.

Similarly a procedure heading is called a *formal call* of the procedure being defined. For each explicit domain component of the signature marked by the update symbol, @, the following rules apply to the corresponding operand of the heading: The operand is considered similarly marked, whether explicitly marked or not, and if it is a discriminator any constituent operand may be marked. The significance of a marked formal parameter is that it represents a program variable which may be assigned to or updated in other ways.

When executing executable code, parameter transmission is by value copy for unmarked formal parameters and by value copy-in copy-out for update parameters. For parameters of class-like types the value copied is a pointer to an object, not the object itself.

Only program variables and formal update parameters, including updatable discriminator operands, may have marked applied occurrences (possibly subscripted or dotted). Only update parameters may have :::applied occurrences marked with the update symbol. It follows that procedures are only accessible through update parameters, and that unmarked parameters only provide read access to class objects.

6.2 The Domain Constraint

⟨domain constraint⟩ ::= **where** ⟨*Boolean* expr⟩

A domain constraint expresses a condition to be satisfied by the parameters. In the definition body a constraint B may be assumed to hold strongly for the formal parameters, i.e. the formula $\Delta B \wedge B$ may be assumed. There is a corresponding obligation to prove, for every application of the function, that the constraint is strongly satisfied by the actual parameters whenever welldefined. No coercion function may have a domain constraint.

6.3 The Definition Body

non-procedure ::= *ordinary* | *initiator* | *attribute* | *binop* | *coercion*
⟨*non-procedure* body⟩ ::= ⟨applicative body⟩ | ⟨imperative body⟩
⟨*procedure* body⟩ ::= ⟨imperative body⟩
⟨applicative body⟩ ::= ⟨expr⟩

A function definition is said to be *applicative* if the function body is applicative, otherwise *imperative*. The semantics of an applicative function definition is explained in chapter 3, and that of an imperative one is obtained by replacing the imperative body by the corresponding effect function, see chapter 5. An applicative body must be of a type coercible to the codomain of the function, and any execution of an imperative one must end in a

return statement whose operand satisfies the same requirement. If necessary, coercion will be automatically applied. The body of a coercion function is not coercible by self application.

Error values occurring in a definition body must be unprefixed applications of errors associated with the function being defined. They are interpreted as if prefixed by that function.

A function definition is said to be *constructive* if the body contains no quantifier, and *executable* if in addition all function applications occurring in the body are executable. A set of applicative definitions is said to be *convergent* if they define a convergent term rewriting system when seen as rewrite rules. A function definition is called *recursive* if an execution of its body may lead to self application, and is called *inductive* if the body contains a case-construct discriminating on a formal parameter.

An occurrence of "$." or "@$." anywhere in an *Abel* text may be omitted. Thus, from within the body of an attribute function, coercion function, or procedure any other attribute function or procedure associated with the same object is "directly accessible", as in traditional object oriented languages. See also sections 3.3 and 3.4.

Notice that, from within any non-procedure function body, even an imperative one, procedures associated with objects other than locally declared variables will only be accessible as non-executable functions. That is because any parameter, including a possible $, or any parameter component, is in this case non-updatable and therefore can not be used as the dominant actual parameter of a call. Since assignments to non-updatable parameters or parameter components are also illegal, side effects from non-procedure functions are effectively prevented. Notice also that the effects of a procedure call can only concern variables occurring explicitly in the call, or possibly an implicit $. The former are either marked by the update symbol or identified as receivers of the function value, if any.

Examples

We illustrate some of the above concepts by providing alternative definitions of a function which divides two natural numbers and returns the quotient and the rest.

func $Div(num : Nat,\ denom : Nat) =: (quot : Nat,\ rest : Nat)$

- Imperative, nonrecursive:

```
def Div() ==
prog
  var q : Nat = 0, r : Nat = num;
  loop assert num = q * denom + r;
  while r >= denom;
    @r - denom; @q + 1
  endloop;
  return (q, r)
endprog
```

- Applicative, recursive:

```
def Div() ==
    if num < denom then (0, num)
    else let (q, r) = Div(num - denom, denom) in (q + 1, r) ni fi
```

This definition is executable, but not convergent because the termination property of the rewriting system would be lost.

- Applicative, inductive:

$$\begin{array}{l}\textbf{def } Div() \ == \\ \quad \textbf{case } num \textbf{ of } \mathbf{z} \ \rightarrow \ (0,0) \\ \qquad\qquad [\!] \ \mathbf{S}x \ \rightarrow \ \textbf{let } (q,r) = Div(x, denom) \textbf{ in} \\ \qquad\qquad\qquad \textbf{if } \mathbf{S}r = denom \textbf{ then } (\mathbf{S}q, 0) \textbf{ else } (q, \mathbf{S}r) \textbf{ fi ni fo}\end{array}$$

using the induction on natural numbers defined by **genbas z,S**, see section 7.7.2. This definition is inductive with respect to the the first parameter *num*. It is also convergent, because the recursive application $Div(x, denom)$ is "protected" by the discriminator $\mathbf{S}x$. A rewrite rule whose right hand side contains a case-expression (other than an if-expression) will only be applied if the discrimination is decided by the rewriting machine, in which case the case-expression is replaced by the appropriate alternative. Therefore the above definition can be applied only a finite number of times to any given expression.

For an application $Div(e_1, e_2)$, where e_1 and e_2 are welldefined expressions of type *Nat*, and $e_2 = 0$, the two first definitions would lead to a non-terminating evaluation (axiomatically equivalent to the anonymous error value), and the last definition would produce the unreasonable value $(0, e_1)$. A better idea might be to declare an associated named error which could lead to appropriate corrective actions by postscripting the application.

$$\begin{array}{l}\textbf{func } Div(num : Nat,\ denom : Nat) =: (quot : Nat,\ rest : Nat) \textbf{ error } ZeroDiv \\ \textbf{def } Div() == \textbf{if } denom = 0 \textbf{ then error } ZeroDiv \textbf{ else } ...\end{array}$$

Alternatively the possibility of the denominator being equal to zero might be ruled out by a domain constraint,

$$\begin{array}{l}\textbf{func } Div(num : Nat,\ denom : Nat) =: (quot : Nat,\ rest : Nat) \\ \textbf{def } Div() \textbf{ where } denom \neq 0 == \ ...\end{array}$$

in which case the user would be obliged to prove for the above application that e_2 is non-zero if welldefined. Division by zero may also be ruled out by declaring the parameter *denom* to be of the subtype *Nat1*.

$$\textbf{func } Div(num : Nat,\ denom : Nat1\,) =: (quot : Nat,\ rest : Nat)$$

In this case the actual parameter e_2 would be subjected to coercion leading to a named error if the value is zero.

6.4 Function Declarations

⟨combined definition⟩ ::= ⟨*function* header⟩⟨*function* declaration⟩
⟨*non-coercion* declaration⟩ ::= ⟨*non-coercion* domain-part⟩⟨domain constraint⟩$^?$
⟨*non-coercion* codomain-part⟩⟨error-part⟩== ⟨*non-coercion* body⟩
⟨*coercion* declaration⟩ ::= ⟨*coercion* definition⟩

A function declaration is obtained by textually merging together a function signature and the corresponding function definition. Notice that any domain constraint shall occur immediately after the domain part of the signature. Notice also that any labels occurring in the codomain part do not count as parameters accessible within the function body. However, if the codomain is a type product with labelled components, the labels are selector functions identifiers as usual (section 8.3), which may be applied to any application of the function being defined.

Chapter 7

Modules

module	::= *group* **|** *property* **|** *type*
⟨*module* decl⟩	::= ⟨*module* heading⟩==⟨*module* rhs⟩
⟨*module* heading⟩	::= ⟨*module* header⟩⟨*module* name clause⟩
	⟨formal type params⟩⟨*module* formal value params⟩
	⟨assumption clause⟩$^{?}$⟨inclusion clause⟩$^{?}$
⟨formal type params⟩	::= {⟨formal type⟩$^{+}_{,}$}
⟨formal type⟩	::= ⟨ident⟩
⟨*module* ident⟩	::= ⟨ident⟩

The main structuring mechanism of ABEL is the **module** construct, which is used to define and encapsulate a group of related functions and/or types. A module may be parameterized by formal type parameters and, to a limited extent, by typed value parameters. The parameters may be assumed to satisfy certain minimum requirements. A module represents a theory of formal logic as explained in chapter 5. See also section 7.5. An identifier occurring in the name clause of a module identifies that module. All module identifiers of a given working context must be distinct. The main text of a module is (usually) the module body introducing functions, axioms, etc.

We distinguish between three different main kinds of module:

- A **group** module introduces a group of related functions over actual or formal types. An *instance* of the module may later be *included* in other modules.

- A **property** module is like a group module but may be understood as a predicate over its formal type parameters. The purpose of a property is to state the minimum requirements to the type parameters of other modules. This is done by *assuming* an instance of the property.

- A **type** module introduces one or more types together with associated functions. Type modules are instantiated by *type expressions* occurring in other modules.

By *assumptions, inclusions*, and *type expressions* one may use old modules to build new ones, as explained below. In sections 7.1-7.4 the syntax of the various language constructs are given together with informal semantic explanations. Section 7.5 provides a more precise definition of the key concept of *module instance*.

7.1 Module Composition

⟨assumption clause⟩ ::= **assuming** ⟨*property* module-expr⟩$^{+}_{,}$
⟨inclusion clause⟩ ::= **including** ⟨*group* module-expr⟩$^{+}_{,}$
⟨include stmt⟩ ::= **include** ⟨*group* module-expr⟩$^{+}_{,}$
⟨*module* module-body⟩::= **module** $[\![$⟨item prefix⟩$^{?}$⟨*module* item⟩$]\!]^{+}$ **endmodule**
⟨item prefix⟩ ::= ⟨type expr⟩
⟨*module* module-expr⟩ ::= ⟨*module* ident⟩ ⟨actual type params⟩$^{?}$
⟨actual value params⟩$^{?}$⟨renaming clause⟩$^{?}$
⟨actual type params⟩ ::= {⟨type expr⟩$^{+}_{,}$}
⟨type expr⟩ ::= ⟨*type* module-expr⟩ | ⟨local type⟩ | ⟨special type⟩ | $$
⟨implemented type⟩
⟨actual value params⟩ ::= ⟨*application* tuple⟩
⟨renaming clause⟩ ::= **with** (⟨substitution⟩$^{+}_{,}$)
⟨substitution⟩ ::= $[\![$⟨function name⟩ | ⟨prefixed function⟩$]\!]$**as** ⟨function name⟩

A module expression with module identifier M represents an instance of the module M essentially obtained by replacing each occurrence of a formal parameter in M by the corresponding actual one. The module expression must contain an actual parameter (argument) for each formal parameter of M. The module instance in turn represents a formal theory obtained from the theory of M by similar textual substitutions.

Assumptions about the formal type parameters of a module are indicated by an assumption clause in the module heading. As the result the module body is extended by the contents of each property module instance identified in the assumption clause. In the extensions each axiom, function definition and basis statement is marked as "assumed", and certain syntactic consistency checks are performed (section 7.5, steps 3.3 and 3.6).

Inclusions are indicated by an inclusion clause in the module heading and/or by include statements in the module body. As the result the module body is extended by the contents of each group module instance identified to become included. In the extensions each module item marked "assumed" is replaced by a corresponding lemma marked "not proven". Thereby the user has an obligation to prove that assumptions made by a group module are in fact satisfied by the actual type parameters of an included instance. Syntactic consistency checks are performed as for assumptions.

Each type expression occurring in a module leads to the extension of the module body by the corresponding type or class module instance in a manner analogous to the inclusion of group module instances. Thus, no explicit inclusion statement is required in order to incorporate a type or class module instance into another module. See also section 7.1.1 below.

An item prefix serves to define the *home type* for that module item. The home type in turn defines the type association of functions introduced and plays a role in the binding of applied occurrences of function identifiers (see section 7.5, step 4). The home type, if defined, is identified by the symbol $$, or by its initial type identifier (occurrences of local types replaced by their defining expressions). The home type of an item prefix is the main type, if any, see section 7.4.

An item sequence of the form T⟨*module* item⟩$^{+}$, where T is a type expression, is an abbreviation of $[\![T$⟨*module* item⟩$]\!]^{+}$.

The use of value parameters to types leads to certain semantical complications, because type expressions may occur within the scope of local variables and may thus depend on

them. In order to simplify the language semantics the use of value parameters to modules is heavily restricted:

1. Only subtype modules are allowed to have formal value parameters.
2. Item prefixes containing value parameters are illegal (local types transparent).
3. A formal type parameter occurring as, or as part of, an item prefix must not be instantiated to a type expression containing value parameters, unless the item is part of a property module body. (Any value parameter of the effective prefix of a property module item will be ignored semantically, see section 7.5, step 1.1.)

One consequence of the above rules and restrictions, together with the fact that item prefixing is illegal in subtype modules, is that no function body can depend on value parameters of types other than its owner type. For a function associated to a value dependent type, say $`T(e)`f$, the value parameters e of the owner type can be seen as additional arguments to a function f associated to a type "skeleton" $T(.)$ in which the value parameters are ignored. By this trick the theory $TT(e)$ corresponding to a "local" type $T(e)$ can be defined in terms of the theory $\forall n|T(n)$ corresponding to the type skeleton, which is globally defined.

Identifiers of functions introduced in a module instance (including all extensions) may be systematically changed by adding a renaming clause to the module expression. The substitutions are carried out as explained in section 7.5, step 5. The purpose of function renaming may be to avoid name conflicts, or to identify one function with a differently named one introduced in another module. Example:

$$PriorityQueue\{Nat\}\textbf{with}(\hat{}\leq\hat{}\ \textbf{as}\ \hat{}\geq\hat{})$$

where the function $\leq$ of the *PriorityQueue* instance is the one introduced in the property *TotOrd* (section 7.6) assumed by the former.

Let TM be a type module expression with no renaming clause. Then the expression

$$TM\ \textbf{with}\ (f\ \textbf{as}\ g)$$

represents the same type as TM, although textual renaming takes place in the type module. Thus, if both type expressions occur in a module then f and g will be alternative names on the same function. The name g must be unprefixed and is interpreted as if prefixed by the prefix of f. If the latter is unprefixed it is interpreted as if prefixed by TM. The renaming of a free function f must in this case be indicated by an explicit "empty" prefix, $`f$. It is illegal to rename basic generators.

7.1.1 Implemented types

⟨implemented type⟩ ::= ⟨simple type expr⟩ **by** ⟨simple type expression⟩

This is so far a loose end of the language. (Cf. section 3.4.1).

7.2 Group Modules

⟨*group* header⟩	::= **group**
⟨*group* name clause⟩	::= ⟨*group* ident⟩

⟨*group* formal value params⟩ ::= ⟨empty⟩
⟨*group* rhs⟩ ::= ⟨*group* module-body⟩
⟨*group* item⟩ ::= ⟨include stmt⟩ | ⟨local types⟩ | ⟨*function* signatures⟩ |
⟨definition item⟩ | ⟨combined definition⟩ |
⟨axioms⟩ | ⟨lemmas⟩
⟨local types⟩ ::= **def type** $[\![$⟨*local type* ident⟩==⟨*type* module-expr⟩$]\!]^{+}_{,}$

The purpose of a group module is to define a set of possibly related functions on types expressible in terms of existing type modules and formal parameters. The functions will typically be defined constructively, but arbitrary axioms are allowed as well. The function identifiers introduced must be different from the formal parameters of the module (for all kinds of module). A local type declaration introduces a name to serve as an abbreviation for a type expression. The name is strictly local to the module body containing the declaration, see section 7.5, step 2. Basis statements are not allowed in a group module body.

Unprefixed signatures are abbreviations for signatures with an "empty" prefix (introducing free functions).

7.3 Property Modules

⟨*property* header⟩ ::= **property**
⟨*property* name clause⟩ ::= ⟨*property* ident⟩
⟨*property* formal value params⟩ ::= ⟨empty⟩
⟨*property* rhs⟩ ::= ⟨*property* module-body⟩
⟨*property* item⟩ ::= ⟨*group* item⟩ | ⟨basis stmt⟩

The purpose of a property module is to express minimal requirements on type parameters to other modules. This is typically achieved by introducing functions on formal types characterized (weakly) by nonconstructive axioms. Notice that renaming of these functions is possible, and may be necessary in many cases of property module instantiation, see the example of section 7.1. Basis statements occurring in a property module must apply to (be prefixed by) formal types.

7.4 Type Modules

type ::= *basetype* / *subtype*
⟨local type⟩ ::= ⟨formal type⟩ | ⟨local type ident⟩ | ⟨type under definition⟩
⟨type under definition⟩ ::= ⟨*type module* ident⟩

Type module identifiers serve a double purpose: they identify type modules, and they stand for types, possibly parameterized, defined by the type module right hand side. Within a type module body type module identifiers listed in the module heading are selfcontained type expressions denoting the *types under definition*, the first one in the list is called the *main type*. Unprefixed type module items are interpreted as if prefixed by the main type identifier.

7.4.1 Basetype modules

⟨*basetype* header⟩ ::= **type** | **class**

⟨*basetype* name clause⟩ ::= ⟨*basetype module* ident⟩$^{+}_{,}$
⟨*basetype* formal value params⟩ ::= ⟨empty⟩
⟨*basetype* rhs⟩ ::= ⟨*basetype* module-body⟩
⟨*basetype* item⟩ ::= ⟨*group* item⟩ | ⟨basis stmt⟩

The type module indentifiers of a basetype module heading are alternative identifiers for the same module. The types, all parameterized as indicated in the module heading, are defined algebraically in the module body by mutual recursion. Any basis statement should be (effectively) prefixed by a type under definition. Each such type must be completely defined through sufficiently strong basis statements: either **one-one genbas**, or **genbas** plus **obsbas**.

If the module header is **type** the types under definition are said to be *non-class types*; if it is **class** they are said to be *class-like types* or *classes*. The only semantic difference is the way the respective values are manipulated in the execution of executable programs (value vs. pointer copying). There are, however, restrictions on the use of expressions of class-like types as stated in sections 4.4.2 and 5.2.3.1; also the use of formal update parameters is limited to those of class-like types. An arbitrary type expression represents a non-class type if all types occurring in the expression are non-class. If one or more class-like types occur the expression represents a class-like type. (See also the next subsection.)

For each type under definition in a basetype-module the binary relations ^=^ and ^≠^ are introduced automatically as associated functions. See section 3.2 and section 7.5, step 1.3.

7.4.2 Subtype modules

⟨*subtype* header⟩ ::= **subtype** | **subclass**
⟨*subtype* name clause⟩ ::= ⟨*subtype module* ident⟩
⟨*subtype* formal value params⟩ ::= ⟨*appldomain* tuple⟩
⟨*subtype* rhs⟩ ::= ⟨module prefix⟩⟨*subtype* module-body⟩$^{?}$
⟨module prefix⟩ ::= ⟨type expr⟩⟨constraint⟩$^{?}$⟨default indication⟩$^{?}$
⟨default indication⟩ ::= **default** ⟨expr⟩
⟨*subtype* item⟩ ::= ⟨*group* item⟩ | ⟨genbas stmt⟩

The heading of a subtype-module identifies one type under definition, say ST (possibly parameterized). A subtype relation is established between ST and the type identified by the type expression of the module prefix for every occurring instantiation of the formal parameters. If the module header is **subtype** the type expression of the module prefix must be a non-class type and so is the subtype under definition. If the header is **subclass** the subtype under definition is class-like, and there is no restriction on the module prefix.

The value set of the subtype is (usually) identical to that of the prefix type, unless a constraint is given. For a module prefix of the form T **where** $B(\$)$ the value set of the subtype is equal to $\{\ t : T \mid B(t) == \mathbf{t}\}$. The restricting Boolean expression is interpreted as if being the body of a T-associated function with implicit dominant argument ($). It also has access to the contents of module instances assumed or included in subtype heading. A coercion function from T to ST is defined automatically as explained in section 7.5 step 1.2. No coercion from ST to T is needed since a ST-value belongs to T by definition.

A default indication redefines the default value associated with the type ST. The default expression is interpreted with ST as the home type, and its value must satisfy the preceding constraint, if any.

The subtype module ST inherits all functions declared in the module instance T of the module prefix (with domains and codomains unchanged), except those which are redefined. See section 7.5, in particular steps 3 and 4. Also the generator- and observation bases are inherited. However, the subtype module body may contain a **genbas** statement identifying a subset of the original generator basis. This is another way of reducing the value set of the prefix type. As a consequence the case construct with respect to the subtype is reduced accordingly, but this reduction is not reflected in the coercion function from T to ST. In order to have full type protection for arguments of type ST one should in this case also provide a constraint only satisfied by values generated by the reduced basis.

All items of a subtype-module body must be unprefixed. This implies that the home type is defined to be the type under definition throughout the module body. It also implies that coercion between subtypes can not be defined explicitly, but must be indirect through a base type. See also section 7.1.

7.5 Module Instances.

A module expression ME with the module identifier M identifies a *module instance*

$$MI = (MB, TP, RP)$$

where:

- MB, the *specification part*, is a specification text obtained from the module body of the module M.
- TP, the *type part*, is the set of "skeletons" of the types occurring in M. The skeleton of a type is obtained by replacing the value parameters at all paranthesis levels of the type expression by distinct formal parameters.
- RP, the *relation part* representing the subtype relation, is a set of ordered pairs (T, U), one for each subtype skeleton T in TP, and where U is the corresponding instance of the type expression of the module prefix of the subtype module in question (occurrences of value parameters replaced by those of the skeleton T).

A type T is said to be a *direct subtype of* a type U if and only if there is a pair (T', U') in RP such that T can be obtained from T' by instantiating its formal value parameters, and U is the corresponding instance of U'. The subtype relation, $\subseteq$, is defined as the reflexive transitive closure of the direct subtype relation.

The module instance MI identified by ME is obtained in the following steps.

STEP 1. Initialization and syntactic sugar.

1. MB is initialized to a copy of the module body of the module M. If M is a property module, all axioms, definitions, and basis statements become flagged "assumed". If a **genbas** statement is so flagged, each occurrence of a **case** construct over the home type of the former is replaced by a corresponding **if-any** construct, as explained in section 4.11. All item prefixes are replaced by their skeletons.

2. If M is a subtype module then MB is extended by the declaration of a coercion function from the supertype to M. If the module prefix contains a constraint B the declaration is

TE **coercion** M **error** $M ==$ **if** B **then** \$ **else error** M **fi**

where TE is the type expression of the module prefix. If there is no semantic restriction coercion is by an identity function.

TE **coercion** $M ==$ \$

3. If M is a basetype module then for each type T under definition the following items are added to MB.

   T **binrel** $=, \neq$
   T **def** $x \neq y \ == \ \mathbf{not}(x = y)$

4. All abbreviations in the contents of MB are expanded.

5. Each occurrence in MB of the symbol \$\$ is replaced by the home type expression for that occurrence.

6. The type part TP is initialized to contain the formal type parameters of M, the identifiers of the types under definition, if any, as well as the skeletons of all (other) type expressions occurring in the whole of module M.

7. The relation part RP is initialized to the set $\{(M, TE)\}$ if module M has a module prefix and where TE is the type expression of the latter, otherwise to the empty set.

8. Each occurrence as a type expression in MI of the identifier of a type under definition is augmented by the formal parameter parts of module M.

9. Each occurrence of a formal parameter of M in MI is flagged so as to be distinguishable from all other identifiers.

10. Each occurrence of an unprefixed error application within a definition body becomes prefixed by the name of the function being defined.

STEP 2. Local types.

1. Each occurrence in MI of a local type is replaced by its defining expression. (This step is iterated as required.)

2. All local type declarations are deleted from MB.

STEP 3. Inclusions.

1. For each module expression ME' occurring in an assumption or inclusion clause in the heading of module M the item **include** ME' is added to MB. Also for each type T in TP, except formal type parameters and types under definition, the item **include** T is added.

2. Each item of the form **include** X is replaced by the specification part of the module instance identified by X, the delimiters **module** and **endmodule** deleted. The type part is added to TP, and the relation part is added to RP.

3. It is checked that any two function signatures for the same function identifier are either syntactically equal, or the prefixes are distinct type skeletons (or one is empty) and both signatures define the same or no dominant argument position.

4. For any two items T **genbas** S and T' **genbas** S' occurring in MB, it is checked that $T \subseteq T'$ implies S is a subset of S'.

5. If M is not a property module all axioms, definitions, and basis statements flagged "assumed" become lemmas flagged "not proven".

6. It is checked that no function has two syntactically distinct definitions, and that each axiom contains at least one function without a definition, other than basic generators and equalities.

STEP 4. Function binding.

Each applied occurrence in MI of a function identifier has a function prefix, FP, which may or may not be user defined. The occurrence is bound to some function signature. Each (sub-)expression and function call has two attributes: AT (the "actual" type) and ET (the "expected" type). Attribute values and bindings are defined according to the rules below. The syntactic function

$$\mathcal{B} : \langle \text{type expr} \rangle \times \langle \text{function ident} \rangle \to \langle \text{type expr} \rangle$$

plays a role in the rules defining FP. For type T and function identifier f $\mathcal{B}(T, f)$ is the smallest type U such that $T \subseteq U$ and U owns a function f. If no such type exists $\mathcal{B}(T, f)$ is undefined.

- AT of a variable occurrence is the type of the variable (which is either defined explictly or, if the variable is introduced in a formal application or formal call, by the ET of the defining occurrence).

- AT of a function application is the codomain of an instance of the function signature to which the main operator is bound, in which any formal value parameters of the signature prefix are instantiated by the actual value parameters of the function prefix FP.

- AT of a function call is like that of a function application, but reduced according to the list of update parameters, possibly resulting in an "empty" type product (see section ...).

- AT of an explicit tuple is the product of the component AT's.

- AT of an expression of the form e **as** T is T.

- AT of an expression of the form e **qua** T is T.

- AT of a **case**-expression with branch expressions $e_1, e_2, ..., e_n$ is the smallest type T such that $T_i \subseteq T$ can be verified textually, where T_i is the AT of e_i, i=1,2,...,n.

- AT of an expression of the form e **on** ... **no** is the AT of e.

- AT of an error expression is the (return part of) the codomain of the function (procedure) to which the error is associated.
- ET of the main operator of the left hand side of a function definition is the home type of that module item if defined, otherwise an imaginary type regarded as the "owner" of the free functions.
- ET of the main operator of a discriminator of a **case** construct is the base type of the AT of the discriminand.
- ET of the argument to a function occurrence (possibly an explicit tuple) is the domain of an instance of the function signature to which the occurrence is bound, in which any formal value parameters of the signature prefix are instantiated to the actual value parameters of the function prefix FP.
- ET of each component of an explicit tuple is the corresponding type component of the ET of the tuple. If the latter is not an explicit type product with the right number of components a syntactic error situation obtains.
- ET of the right hand side of a function definition is the codomain of the function signature to which the main operator of the left hand side is bound.
- ET of the right hand side of an assignment is the AT of the left hand side.
- ET of the left hand side of a function call is the AT of the right hand side if any, otherwise the empty type product.
- ET of the argument of a nonformal error is as defined in the error part of the signature to which the owner function is bound.
- ET of the argument of a formal error is as defined in the error part of the signature to which the owner function is bound, instantiated according to the value parameters, if any, of the FP of the postscripted function occurrence.
- ET of the argument of a formal or nonformal branch is as defined in the corresponding branch declaration.
- ET of e in the expression e **as** T is T.
- ET of e in the expression e **qua** T is the base type of T.
- ET of the expression of each branch of a **case**-expression is the ET of the latter.
- ET of e in an expression of the form e **on** ... **no** is the ET of the whole construct.
- ET of the expression of an applicative error handler is the AT of the postscripted expression.
- ET of a **return**-expression of a function- or procedure body is the codomain of the function signature to which the main operator of the left hand side is bound, respectively the return part of that codomain.
- If an occurrence of a function identifier f has a user defined prefix UP, its FP is $\mathcal{B}(UP, f)$ if defined, otherwise a syntactic error situation obtains. If there is no user defined prefix the following rules apply.

- If the occurrence is the operator of a formal application or formal call its FP is the ET of the latter (or the empty prefix if the ET is the imaginary owner of the free functions), otherwise:

- If f has a dominant argument position the FP is $\mathcal{B}(T, f)$, if defined, where T is the corresponding component of the AT of the argument (possibly an explicit tuple). If $\mathcal{B}(T, f)$ is undefined a syntactic error situation obtains.

- If f has no dominant argument position the FP is:

  1. $\mathcal{B}(ET, f)$ if defined, otherwise
  2. $\mathcal{B}(HT, f)$, where HT is the home type, if both types are defined, otherwise
  3. the empty prefix if there is a free function f, otherwise
  4. a type T in TP having an associated f if T is unique and the type expression T contains no value parameters on any paranthesis level.
  5. Otherwise a syntactic error situation obtains.

- An occurrence of f with prefix FP is bound to a function signature with identifier f and prefix coinciding with FP outside value parameters.

STEP 5. Coercion

Each expression for which $AT \subseteq ET$ does not hold is augmented by inserting a coercion from AT to ET. If such coercion is not defined a syntactic error situation obtains. Exceptions: The branch expressions of a **case**-expression, the components of a tuple, and the main part of a postscripted expression are ignored in this analysis. (Otherwise erroneous multiple coercion would result.)

Coercion from type T to type U is defined if there is type V such that $T \subseteq V$ and $U \subseteq V$, otherwise if there are types T' and U' such that $T \subseteq T'$, $U \subseteq U'$, and there is explicit coercion from T' to U'. In the latter case coercion is in two main steps, where the intermediate type U' is as small as possible. For product types T and U of the same number of components coercion is defined if there is coercion from each component of T to the corresponding component of U. Coercion from T to U is in this case expressed by a function applying the necessary coercions to the individual components. (Notice that the expression in question may or may not be an explicit tuple.)

STEP 6. Function identifier substitutions.

The substitution clause of ME, if any, is effected as follows.

1. For a substitution *'T'I* **as** *J*, (I, J unprefixed identifiers) it is checked that a function signature for I with prefix T exists. Its function identifier is replaced by J. Also each occurrence in MB of *'T'I*, is replaced by *'T'J*.

   For a substitution I **as** J all occurrences of I as function identifier in MB are replaced by J (prefixes retained).

2. Step 3.3 is repeated.

STEP 7. Parameter transmission.

All occurrences in MB of formal type parameters of M are simultaneously replaced by the corresponding actual parameters of ME. All occurrences in TP and RP of formal type parameters of M are simultaneously replaced by skeletons of the corresponding actual parameters of ME, in which the value parameters are fresh identifiers.

7.6 Examples of Properties and Function Groups

The following property defines the minimum requirements to a partially ordered type T:

property $PartOrd\{T\}$ ==
module
  T **binrel** $\leq$
  axioms $(x, y, z : T)$
    total $\leq$
    $x \leq x$
    $x \leq y \leq x \Rightarrow x = y$
    $x \leq y \leq z \Rightarrow x \leq z$
endmodule

The first axiom states that $\leq$ is a total relation; the intention is that if x and y are "unrelated" both $x \leq y$ and $y \leq x$ are false.

A given type S is (satisfies) $PartOrd$ if S has an associated relation $\leq$ which satisfies the four axioms. For instance, $PartOrd\{Integer\}$ is satisfied. The property $PartOrd$ may be used to specify an assumption about the formal type parameters of another module. For instance, the following module requires that the formal parameter T is partially ordered by $\leq$.

group $OrdAug\{T\}$ **assuming** $PartOrd\{T\}$ ==
module
  T **binrel** $<, >, \geq$
  T **def** $x < y == x \leq y$ **and not** $x = y$
    $x > y == y < x$
    $x \geq y == y \leq x$
endmodule

This module, "Augmented Order", defines new order relations in terms of $\leq$. When $OrdAug$ is included, the actual type parameter must satisfy $PartOrd$. (Such situations are illustrated below.) This is a convenient way of providing executable definitions of the relations $<, >, \geq$, given that an executable definition of $\leq$ exists at the point of use.

A total ordering is a partial ordering satisfying an additional axiom:

property $TotOrd\{T\}$ **assuming** $PartOrd\{T\}$ ==
module
  axioms $(x, y : T)$
    $x \leq y \vee y \leq x$
endmodule

We may now use *OrdAug* to define an augmented total ordering:

property $TotOrdAug\{T\}$ **assuming** $TotOrd\{T\}$ ==
module include $OrdAug\{T\}$ **endmodule**

Notice that the assumption of *TotOrd* implies the validity of *PartOrd* assumed in the *OrdAug* group. Using *TotOrdAug* in an assumption would have the effect of assuming *TotOrd* and including *OrdAug*.

Integers have the following property:

property $LinOrd\{T\}$ **assuming** $TotOrdAug\{T\}$ ==
module
T **func** $\mathbf{S}\,T$ =: T
$\mathbf{P}\,T$ =: T
axioms $(x, y : T)$
$\mathbf{P}x < x < \mathbf{S}x$
$\mathbf{PS}x = x = \mathbf{SP}x$
$(x < y) = (\mathbf{S}x \leq y) = (x \leq \mathbf{P}y)$
endmodule

As shown in the next example two types may be restricted simultaneously.

property $Tract\{S, T\}$ ==
module
T **coercion** S
S **coercion** T
axioms $(x : T)$
x **as** S **as** T == x
endmodule

Two types U and V satisfy *Tract* if coercion is defined both ways and if the coercion function from V to U is total and one-to-one (injective) and does not conflict with the coercion from U to V. For instance, *Tract{Integer,Nat}* is satisfied, but not *Tract{Nat,Integer}*.

The final example expresses a property satisfied by all enumeration and subrange types (sect. 8.1, 8.2). It is a necessary requirement for array index types (sect. 8.5).

property $Enum\{T\}$ **assuming** $LinOrd\{T\}$, $Tract\{Nat,T\}$ ==
module
T **init** *lo,hi*
T **axioms** $(x : T)$
$(lo \leq x \leq hi)$ == **t**
$\mathbf{P}lo$ == **error**
$\mathbf{S}hi$ == **error**
endmodule

7.7 Examples of Type Modules

Several of the following examples must or should be predefined types.

7.7.1 Boolean

type *Boolean* ==
module
  func f =: $$, t =: $$, **not**$$ =: $$
  assoc **and**, **or**
  binrel **implies**
  one-one genbas f, t
  def **not** x == **case** x **of** f → t ▯ t → f **fo**
    x **and** y == **case** x **of** f → f ▯ t → y **fo**
    x **or** y == **not**(**not** x **and not** y)
    x **implies** y == (**not** x) **or** y
endmodule

The type *Boolean* is predefined in ABEL. So are several additional associated mechanisms like quantifiers and non-strict operators, which are not definable within the language, see chapters 3 and 4.

7.7.2 Natural numbers

type *Nat* ==
module
  func z =: $$, **S**$$ =: $$
  one-one genbas z, **S**
  binrel ≤
  def $x \leq y$ == **case** x **of** z → t ▯ **S**x → $x \leq y$ **and** $x \neq y$
  include *OrdAug*{*Nat*}
  func **P**$$ =: $$ **error***Underflow* == **case** $ **of** z → **error** *Underflow* ▯ **S**x → x **fo**
  assoc +
  def $x + y$ == **case** y **of** z → x ▯ **S**y → **S**$(x + y)$ **fo**
  func x : $$ − y : $$ == **case** y **of** z → x ▯ **S**y → **P**$(x - y)$ **fo**
  .....
  lemmas *LinOrd*{$$}
endmodule

subtype *Nat1* == *Nat* **where** $ ≠ z
module genbas S endmodule

type *Num* == {0, 1, ..., *MAX*}
module ... *efficient implementations of arithmetic operators* ... **endmodule**

The types *Nat* and *Nat1* are executable, but not very efficient. The type *Num* is intended to be a built-in efficiently implemented type, where *MAX* stands for an implementation defined maximum decimal number. There is coercion both ways between *Nat* and *Num*, see section 8.1 on enumeration types. Since the generator basis of *Num* consists of numeric literals **case** constructs on expressions of this type have the conventional form.

7.7.3 The type *Char*

```
type Char ==
module
  func '~(0)' =: $$, '~(1)' =: $$, '~(2)' =: $$, ...
  genbas '~(0)', '~(1)', '~(2)', ...
  def func ' ' =: $$  ==  '~(32)';
  def func '!' =: $$  ==  '~(33)';
  def func '"' =: $$  ==  '~(34)';
      ⋮
  obs make_Nat =: Nat,
      is_space  =: Bool,
      is_digit  =: Bool,
      is_letter =: Bool,
      digit_val =: Nat error illegal_digit;
  Nat obs make_Char =: Char error illegal_Char,
          make_digit =: Char error illegal_digit;
  oper up_case;
  def make_Nat== case $ of
                   '~(0)' → 0
                 [] '~(1)' → 1
                   ⋮
                 fo;
  def is_space  == $ ≤ ' ';
  def is_digit  == '0' ≤ $ ≤ '9';
  def is_letter == 'A' ≤ $ ≤ 'Z' ∨ 'a' ≤ $ ≤ 'z';
  def digit_val == if is_digit then make_Nat($) − make_Nat('0')
                               else error illegal_digit;
  Nat def make_Char == case $ of
                         0 → '~(0)'
                       [] 1 → '~(1)'
                         ⋮
                       others error illegal_Char
                       fo;
  Nat def make_digit == if $ ≤ 9 then make_Char(make_Nat('0')+ $)
                                 else error illegal_digit fi
  def up_case == if is_letter ∧ $ ≥ 'a' then make_Char(make_Nat($)−32)
                                        else $ fi
endmodule
```

The *Char* values are '~(0)', '~(1)' etc. Additional names like '!', 'A' etc. are also defined, but these names and their definition are implementation-defined, as are the various functions operating on *Char* values; the definitions supplied here are based on the ASCII character set.

This section is not yet completed.

7.7.4 Sequences

```
type Seq{T} ==
module
        func e =: Seq                                   -- empty sequence
             Seq ⊢ T =: Seq                             -- append right
             T ⊣ Seq =: Seq                             -- append left
   T     func [T]  =: Seq                               -- singleton
             [T, T] =: Seq                              -- doubleton
             [T, T, T] =: Seq                           -- tripleton
             ⋮
   Seq assoc ⊢                                          -- concatenate
        func # Seq =: Nat                               -- length
             rterm(Seq) error rterm                     -- right term
             lrest(Seq) error lrest                     -- left rest
             lterm(Seq) error lterm                     -- left term
             rrest(Seq) error rrest                     -- right rest
             Seq[Nat1] =: T error subscript             -- subscript
             Seq[Nat1;Nat] =: Seq                       -- segment by length
             Seq[Nat1..Nat1] =: Seq                     -- segment by end point
             Seq[Nat1 → T] =: Seq                       -- point update
        one-one genbas    e, ^⊢^
        def  x ⊣ q == case q of e → e ⊢x  []  q ⊢y  → (x ⊣ q) ⊢y fo
   T    def  [x]  == e ⊢x
             [x, y] == e ⊢x ⊢y
             [x, y, z] == e ⊢x ⊢y ⊢z
             ⋮
   Seq def   q ⊢ r == case r of e → q  []  r ⊢x  → (q ⊢ r) ⊢x fo
             #q == case q of e → 0  []  q ⊢x  → #q + 1
             rterm(q) == case q of e → error rterm  []  q⊢x  → x fo
             lrest(q) == case q of e → error lrest  []  q⊢x  → q fo
             lterm(q) == case q of e → error lterm  []  q⊢x  →
                                 case q of e → x  []  ^⊢^  → lterm(q) fo
             rrest(q) == case q of e → error rrest  []  q⊢x  →
                                 case q of e → e  []  ^⊢^  → rrest(q)⊢x fo
             q[Si] == case q of e → error subscript  []  ^⊢^  →
                                 case i of z → lterm(q)  []  Sj  → rrest(q)[j] fo fo
             q[i; n] == case n of z  → e  []  Sm  → q[i; m] ⊢q[i + n]
             q[i..j] == q[i;  j + 1 − i]
             q[i → x] == q[1.. i − 1] ⊢[x] ⊢q[i + 1 .. #q]
        lemmas (q, r, s : Seq)
             e ⊢q = q
             q ⊢(r ⊢s) = (q ⊢r) ⊢s
             one-one genbas    e, ^⊣^
             genbas    e, [^], ^⊢^
endmodule
```

The concept of sequences is a generally useful one for reasoning about programs. Here we have only introduced a few of the most frequently used operators. The append operator symbols are intended to indicate asymmetric plus signs. A generator basis consisting of **e** and $\vdash$has been chosen because it has the one-one property, and because right extended sequences often occur when applying Hoare-type reasoning to imperative programs. Two of the lemmas assert the existence of alternative generator bases. In order to assist in proving this fact, the proof system should construct corresponding first order lemmas.

Some operations on sequences are only meaningful on sequences if the base type is ordered. We define the predicate **incr**, which holds for a sequence of strictly increasing values. The predicate is used in the example of sect. 8.6.

group *TotOrdSeq*$\{T\}$ **assuming** *TotOrd*$\{T\}$ ==
module
  def type *Seq* == *Seq*$\{T\}$
  Seq **func incr** *Seq* =: *Boolean* == **case** \$ **of e** $\rightarrow$ **t** $[]$ $q \vdash x$ $\rightarrow$
  case q **of e** $\rightarrow$ **t** $[]$ $r \vdash y$ $\rightarrow y < x \wedge$ **incr** q
  fo fo
endmodule

Some specially notated sequence constructors may be defined for types which are linearly ordered.

group *SeqRange*$\{T\}$ **assuming** *LinOrd*$\{T\}$ ==
module
  T **func** $[T; Nat]$ =: *Seq*$\{T\}$ – – *range by length*
  $[T..T]$ =: *Seq*$\{T\}$ – – *range by end point*
  def $[t; n]$ == **case** n **of z** $\rightarrow$ **e** $[]$ **S**$m \rightarrow [t; m] \vdash n$ **fo**
  $[t..u] == [t; 1 + u - t]$
endmodule

7.7.5 The type *String*

subtype *String* ==
Sequence$\{$*Char*$\}$ **module**
  coercion *Char* **error** *length_error*,
  Nat **error** *syntax_error*,
  Nat1 **error** *syntax_error*,
  Int **error** *syntax_error*;
  Char **coercion** *String*;
  Nat **coercion** *String*;
  Int **coercion** *String*;
  oper *strip*;

  def as *Char* == **if** $\#S = 1$ **then** $S[1]$ **else error** *length_error* **fi**;

  def as *Nat* ==
  prog
    var N: *Num*;
    call \$ **as** *Num* =: N
    on *syntax_error* $\rightarrow$ **return error** *syntax_error*
    $[]$ *overflow* $\rightarrow$ **return error** *syntax_error*

```
    endon;
    return N as Nat
  endprog;
  def as Nat1 ==
  prog
    var N: Num;
    call $ as Nat =: N
    on syntax_error → return error syntax_error endon;
    return if N = 0 then error syntax_error else N as Nat1 fi
  endprog;
  def as Int ==
  prog
    var S := $.strip,
        N: Nat;
    if S = e then return error syntax_error fi;
    if S[1] = '+' ∨ S[1] = '−' then
      call S[2..#S] as Nat =: N
      on syntax_error → return error syntax_error endon;
      return if S[1] = '+' then N else −N fi
    else
      call S as Nat =: N
      on syntax_error → return error syntax_error endon;
      return N
    endif
  endprog;
  Char def as String == < $ >;
  Nat  def as String == ($ as Num) as String;
  Nat1 def as String == ($ as Num) as String;
  Int  def as String == if sign = plus then abs as String
                        else '−' ⊣ (abs as String) fi;
  def strip == if     $ = e          → e
               elsif $[1].is_blank   → $[2..#$]
               elsif $[#$].is_blank  → $[1..#$ − 1]
               else                    $
               fi
endmodule
```

This section is not yet completed.

7.7.6 Finite Sets

```
type FinSet{T} ==
module
    func null =: FinSet                 -- empty set
         FinSet add T =: FinSet         -- add an element
```

FinSet **has** T =: *Boolean* – – *membership test*
FinSet **del** T =: *FinSet* – – *delete an element*
genbas *null*, ^**add**^
def s **has** x == **case** s **of** *null* → f [] s **add** y → $x = y \vee s$ **has** x **fo**
def s **del** x == **case** s **of** *null* → *null* [] t **add** y →
if $x = y$ **then** t **del** x **else** t **del** x **add** y **fi fo**
obsbas ^**has**^
lemmas (*s:FinSet, x,y:T*)
s **add** x **add** $x = s$ **add** x
s **add** x **add** $y = s$ **add** y **add** x
endmodule

The concept of finite sets is more difficult mathematically than the types introduced earlier, in the sense that no one-one generator basis exists. The **obsbas** statement implies the following explicit definition of the equality relation over *FinSet*

def $s = t$ == **all** *x:T* | (s **has** x) = (t **has** x)

which is non-constructive. The **lemmas** are direct consequences of the definitions of **has** .

Chapter 8

Special Types

⟨special type⟩ ::= ⟨enumeration⟩ | ⟨range⟩ | ⟨type product⟩ | ⟨type union⟩ | ⟨array type⟩

The special types are traditional ad hoc notations for certain useful type families. By stretching the syntax of module headings as well as the formal parameter mechanism the special types can be defined as type modules.

8.1 Enumerations

⟨enumeration⟩ ::= $\{\langle\text{ident}\rangle^{+}_{,}\}$

Define for arbitrary nonnegative integer n:

type $\{V_0, V_1, \ldots, V_n\}$ ==
module
  def type E == $$
  init $V_0, V_1, \ldots, V_n, lo, hi$
  one-one genbas $V_0, V_1, \ldots, V_n$
  def $lo == V_0$
    $hi == V_n$
  coercion Nat == **case** $ **of** V_0 → **z** ▯ V_1 → **Sz** ▯ ... ▯ V_n → **S**...**Sz fo**
  Nat **coercion** E **error** *overflow* == **case** $ **of z** → V_0 ▯ **S**i →
    case i **of z** → V_1 ▯ **S**i →
      ⋮
    case i **of z** → V_n ▯ **S**i →
    error *overflow* **fo** ... **fo fo**
  func S$x : E$ =: E **error** *overflow* == (‘*Nat*‘**S**x) **as**E **on** *overflow* →
    error *overflow* **no**
  func P$x : E$ =: E **error** *underflow* == (‘*Nat*‘**P**x) **on** *underflow* →
    error *underflow* **no**
  binrel ≤
  def $x \le y == x$ **as** $Nat \le y$
  include *OrdAug*{$$}
  lemmas *Enum*{$$}
endmodule

Enumeration types are (efficiently) executable. The default value is equal to V_0. There is coercion to and from natural numbers. The "actual parameters" of an enumeration type expression play the role of identifiers of constant functions associated to the type.

8.2 Range Types.

⟨range⟩ ::= {⟨expr⟩ . . ⟨expr⟩}

subtype $\{A..B\} == T$ **where** $A \leq \$ \leq B$ **default** A
module
  genbas *A,..,B* – – *listing all T-values between A and B, provided the actual parameters for A and B are constants*
  init $lo == A$
  init $hi == B$
endmodule

The actual value parameters for A and B must be coercible to one and the same enumeration type. T is an additional formal parameter identifying that type. Examples: {1..10}, $\{0..m-1\}$. In the latter case the generator basis can not be redefined. Range types are executable.

8.3 Product Types

⟨type product⟩ ::= ⟨*appldomain* tuple⟩

Define for arbitrary positive integer n:

type $(T_1, T_2, ..., T_n) ==$
module
  init $(T_1, T_2, ..., T_n)$
  genbas $(\hat{}, \hat{}...\hat{}, \hat{})$
  obs $\mathbf{1} =: T_1,\ \mathbf{2} =: T_2,\ ...,\ \mathbf{n} =: T_n$ – – **n** *stands for the numeral n in boldface*
  def $(x_1, x_2, ..., x_n).\mathbf{1} == x_1$
  def $(x_1, x_2, ..., x_n).\mathbf{2} == x_2$
    ⋮
  def $(x_1, x_2, ..., x_n).\mathbf{n} == x_n$
endmodule

The values of a type product are tuples. (Notice that we are stretching the rules for mixfix notation in declaring the generator function.) The components of a tuple value may be accessed by the predefined selector functions **1**, **2**,...,**n**, or by using a **case** construct. A type product with a single component, (T), is indistinguishable from the type T itself; in particular the generator function is redundant (since parentheses enclosing a single expression has no semantic significance).

If any of the components of a type product is a class-like type, then so is the product type. It is executable if all component types are executable.

Define for positive integers n and k such that $k \leq n$:

subtype $(T_1, ..., a : T_k, ..., T_n) == (T_1, ..., T_k, ..., T_n)$
module
obs $a := T_k$
def $(x_1, ..., x_k, ..., x_n).a == x_k$
endmodule

The label of a labelled component type identifies a user defined selector function. As the result of explicit or implicit **case** constructs the selector functions defined for a product type may play the role of formal variables, formal parameters of functions, or even program variables, see section 4.5 and chapter 6. (This also applies to the label, if any, of a singleton product.)

Examples:

subtype *Integer* $==$ $(sign : \{plus, minus\}, abs : Nat)$ **where** $abs = 0 \Rightarrow sign = plus$
module
coercion *Nat* $==$ **if** *sign=plus* **then** *abs* **else error fi**
Nat **coercion** *Integer* $==$ (*plus,*\$) **qua** *Integer*
func S\$\$ =: \$\$ $==$ **if** *sign=plus* **then** (*plus,***S***abs*)
elsif *abs*$= 1$ **then** $(plus, 0)$
else (*minus,***P***abs*) **fi qua** *Integer*
assoc $+, -$
...
func $gcd(x : \$\$, y : \$\$) =: \$\$ == ... ...$
...
endmodule

This type module could be an alternative to the definition of the Integer type given in the following section. The **qua** clauses express an obligation to prove that the function value of the coercion function from *Nat* and that of the function **S** satify the constraint of the module prefix. The default value is equal to (*plus,*0) **qua** *Integer.*

subtype *Rational* $==$ (*num*:*Integer,denom*:*Nat1*) **where** $gcd(num, denom) = 1$
module
coercion *Integer* $==$ **if** *denom*$= 1$ **then** *num* **else error fi**
Integer **coercion** *Rational* $==$ $(\$, 1)$ **qua** *Rational*
init *norm*(*Integer,Nat1*)
assoc $+$
def $norm(n, d) ==$ **let** $c == gcd(n, d)$ **in** $(n/c, d/c)$ **ni qua** *Rational*
$(a, b) + (c, d) == norm(a * d + b * c, b * c)$
...
endmodule

It is necessary to prove that the bodies of the function *norm* and the coercion function from *Integer* satisfy the constraint of the module prefix. The default value is equal to $(0, 1)$ **qua** *Rational.*

8.4 Union Types

⟨type union⟩ ::= (⟨type alternative⟩ [[+ ⟨type alternative⟩]]$^+$)
⟨type alternative⟩ ::= ⟨label⟩:⟨type expr⟩

Define for integer $n \geq 2$:

type $(a_1 : T_1 + a_2 : T_2 + ... + a_n : T_n) ==$
module
init $a_1(T_1), a_2(T_2), ..., a_n(T_n)$
one-one genbas $a_1, a_2, ..., a_n$
def type $U == \$\$$
T_1**coercion** $U == a_1(\$)$
endmodule

The labels of a union type play the role of injector functions associated to that particular union type. If any of the actual parameters for $T_1, T_2, ..., T_n$ is a class-like type, then so is the union type. It is executable if all constituents are executable. Notice that the first alternative type has coercion to the union. That fact is useful in the following example.

subtype *Integer* == (*pos*:*Nat* + *neg*:*Nat1*)
module
coercion *Nat* == **case** \$ **of** $pos(n) \rightarrow n$ **fo**
func S *i*:*Integer* =: *Integer* ==
case *i* **of** $pos(n) \rightarrow \mathbf{S}n \ [] \ neg(\mathbf{S}n) \rightarrow$ **if** $n = 0$ **then** 0 **else** $neg(n)$ **fi fo**
assoc $+, *$
...
endmodule

The module indicates the structure of the predefined type Integer. The built-in coercion from the first alternative to the union type causes natural numbers to be acceptable integer arguments. The default *Integer* value is $pos(0)$.

8.5 Array Types

⟨array type⟩ ::= **array** ⟨*index* type expr⟩ **of** ⟨*element* type expr⟩

subclass array I **of** E **assuming** $Enum\{I\} == Seq\{E\}$ **where** $\#\$ = 1 + `I`hi - `I`lo$
module
func $A : \$\$[i : I] =: E == (A \textbf{ as } Seq\{E\})[1 + i - `I`lo]$
func $A : \$\$[i : I; n : Nat] =: Seq\{E\} == (A \textbf{ as } Seq\{E\})[1 + i - `I`lo; n]$
func $A : \$\$[i : I..j : I] =: Seq\{E\} == A[i; 1 + j - i]$
func $A : \$\$[i : I \rightarrow x : E] =: \$\$ == A[`I`lo..\mathbf{P}i] \dashv[x] \dashv A[\mathbf{S}i..`I`hi]$
endmodule

The array class is predefined and efficiently implemented, provided that the index and element types are executable. The default (initial) value of an **array** I **of** E is an array of default E-values.

8.6 An Efficient Implementation of Finite Sets

subtype $NIL == \{Nil\}$

subclass $BinTree\{T\} == (nil : NIL + tree : (l : BinTree, v : T, r : BinTree))$

```
module
        obs infix =: Seq{T} == case $ of nil →e
                                        [] tree →l.infix ⊢[v] ⊢r.infix fo
endmodule

subclass FINSET{T} assuming TotOrd{T}
                   including TotOrdSeq{T} ==
BinTree{T} where incr infix
module
    obs has(x : T) =:Boolean ==
        case $ of nil → f [] tree →
        if x < v then l.has(x) elsif v < x then r.has(x) else t
        fi fo
    proc add(x : T) ==
        prog case @$ of nil → $ := tree(Nil, x, Nil) [] tree →
                if x < v then @l.add(x) elsif v < x then @r.add(x)
                fi fo
        endprog
    proc del(x : T) == prog case @$ of nil → skip [] tree →
                            if x < v then @l.del(x) elsif v < x then @r.del(x)
                            else case r of nil → $ := l
                                        [] tree(rl, rv, rr) → v := rv; @r.del(rv)
                            fo fi fo
                       endprog
    implements FinSet{T}
        with(null as Nil, ^has^ as has, ^add^ as add, ^del^ as del) eq infix
endmodule
```

This implementation of the finite set concept of section 7.7.4 is more efficient than the original definition by being a class. Also each operation takes logarithmic expected time, and each element of a set object occurs only once in the representation. Notice that two *FINSET* objects, *S1* and *S2*, represent the same *FinSet* value if and only if *S1.infix = S2.infix.*

Chapter 9

Communication and Input/Output.

This chapter describes the mechanisms provided by *Abel* for interprocess communication and input/output.

The two basic concepts provided are *streams* and *files*. Streams are used for input from and output to purely one-directional devices, such as the user's keyboard or a line printer. It will also be used for inter-process communication when *Abel* is extended with the concept of parallel processes. Files are used for writing to and reading from permanent storage devices, such as disk drives or magnetic tapes.

The data transmitted on streams and files are *bytes*, i.e. values in the range 0–255. Encoding and decoding of values to and from bytes is described in section 9.3 on page 87.

9.1 Streams.

In *Abel* a stream is represented by an object of class *Stream* or one of its subclasses *In_stream* and *Out_stream*. Since these three classes form a logical unit, they are combined into the context *STREAMS*.

The class *In_stream* defines the receiving end of a communication channel. It contains a single operation *get*, which is used to receive one element of data (a byte).

The class *Out_stream* defines the sending end of a communication channel. Its single operator *put* is used to send one data element.

The class *Stream* contains no operations and is never used to perform any communication. It is used to define the concept of a *history*, which is common to both *In_stream* and *Out_stream*. An attribute p (the "*past*") will collect a history of all elements transmitted on the stream. Thus each call on *put* or *get* will augment the history p with one element. An observer *last_byte* will give the last element added to p.

A non-executable version of *STREAMS* is given here; it is the responsibility of each *Abel* implementor to provide the corresponding executable one.

```
context STREAMS ==
  subtype Byte     == [0..255] ,
          Byte_seq == Sequence{Byte};

  subclass Stream ==
  (p: Byte_seq) module
```

```
    def obs last_byte =: Byte == p[#p]
  endmodule;

  subclass In_stream ==
  Stream module
    proc get =: Byte;
    def get ==
    prog
      const x := some Byte;
      @p ⊢ x;
      return x
    endprog
  endmodule;

  subclass Out_stream ==
  Stream module
    proc put(Byte);
    def put(x) ==
    prog
      @p ⊢ x
    endprog
  endmodule
endcontext
```

9.2 Files.

The context *FILES* contains mainly the definition of the class *File*, instances of which are used to represent files. The actual contents of the file are kept in two attributes: p and f. The former contains the part of the file which has been read (the "*past*") while the latter contains the part to be read (the "*future*").

```
context FILES ==
  import Byte, Byte_seq from STREAMS,
         File_name, Directory from DIRECTORIES;

  var dir: Directory;

  type File_state == {closed, readable, writable, updatable};

  subclass File ==
  (p: Byte_seq, f: Byte_seq, fn: File_name,
   state: File_state, altered: Bool)
  module
    init scratch,
         assign(File_name);
    proc open_read,
         open_write,
         open_append,
         open_update,
         close,
```

put(*Byte*) =: **error** *mode_error*,
get =: *Byte* **error** *mode_error* **error** *end_file*,
locate(*Nat1*) =: **error** *mode_error* **error** *locate_error*;
obs *more* =: *Boolean*,
loc =: *Nat1*,
last_byte =: *Byte*;

def *scratch* ==
prog
return(**e**, **e**, '', *closed*, **f**)
endprog;

def *assign*(*f_name*) ==
prog
return(**e**, (*dir.lookup*(*f_name*) **on** *no_such_file* → **e no**),
f_name, *closed*, **f**)
endprog;

def *open_read* ==
prog
(*p*, *f*, *state*) := (**e**, *p*⊢*f*, *readable*)
endprog;

def *open_write* ==
prog
(*p*, *f*, *state*) := (**e**, **e**, *writable*)
endprog;

def *open_append* ==
prog
(*p*, *f*, *state*) := (*p*⊢*f*, **e**, *writable*)
endprog;

def *open_update* ==
prog
(*p*, *f*, *state*) := (**e**, *p*⊢*f*, *updatable*)
endprog;

def *close* ==
prog
if *altered* ∧ *fn* ≠ '' **then**
@*dir.insert*(*fn*, *p*⊢*f*); *altered* := **f**
endif;
state := *closed*
endprog;

def *put*(*x*) ==
prog
if *state*=*closed* ∨ *state*=*readable* → **return error** *mode_error* **endif**;
@*p* ⊢ *x*; *altered* := **t**;
if *state* = *updatable* ∧ #*f* > 0 → *f* := *f*[2 .. #*f*] **endif**

```
      endprog;

      def get ==
      prog
        if state=closed ∨ state=writable → return error mode_error endif;
        if ¬more → return error end_file endif;
        const x := f[1];
        (p, f) := (p ⊢ x, f[2 .. #f]);
        return x
      endprog;

      def locate(n) ==
      prog
        if state ≠ updatable → return error mode_error endif;
        p := p⊢f;
        if n > #p + 1 → return error locate_error endif;
        (p, f) := (p[1 .. n − 1], p[n .. #p])
      endprog;

      def last_byte == p[#p];
      def more      == #f > 0;
      def loc       == #p + 1
    endmodule
  endcontext
```

This context contains a variable *dir* which represents the complete file directory. (The definition of the directory can be found in section 9.2.1, page 86.)

Example 15 A short routine to create a copy named 'B.INT' of a file named 'A.INT'.

```
def func copy_file ==
prog
  var a: File = assign('A.INT'),
      b: File = assign('B.INT');
  @a.open_read; @b.open_write;
  var x: Byte;
  loop while a.more;
    @a.get =: x; @b.put(x)
  endloop;
  @a.close; @b.close
endprog
```

9.2.1 The Directory.

The context *DIRECTORIES* consists mainly of the definition of the class *Directory*, an instance of which (in context *FILES*) is used to represent the file system available. All files are stored as byte sequences, and they are accessed by a file name. If the file system is structured, the file name is assumed to contain the necessary structure information.

```
context DIRECTORIES ==
  import Byte_seq from STREAMS;
  type File_name == String;
  class Directory ==
  module
    init empty_dir;
    proc insert(File_name, Byte_seq),
         remove(File_name);
    obs lookup(File_name) =: Byte_seq error no_such_file;
    genbas empty_dir, insert;
    def lookup(fn) ==
      case $ of empty_dir      → error no_such_file
              [] d.insert(x, y) → if x = fn then y
                                  else d.lookup(fn)
                                  fi
      fo;
    def remove(fn) ==
      case $ of empty_dir      → e
              [] d.insert(x, y) → if x = fn then d
                                  else d.remove(fn).insert(x, y)
                                  fi
      fo;
  endmodule
endcontext
```

9.3 Encoding and Decoding

This section describes an extension *TEXT_FILES* to the *FILES* context,[1] in which files are regarded as texts, i.e. sequences of characters (*Char* values) representing values of various types. The context defines a class *Text_file* containing routines to handle encoding and decoding of values of the predefined types.

```
context TEXT_FILES ==
  import File from FILES;
  const end_line == ...;
  subclass Text_file ==
  File module
    proc write(String),
         read_Char =: Char error mode_error error end_file,
         read =: String error mode_error error end_file;
    obs last_Char =: Char;
```

[1] A similar extension *TEXT_STREAMS* to the *STREAMS* context is assumed to exist; it contains an extension *In_text_stream* to class *In_stream* (containing the *read* and *read_Char* procedures) and an extension *Out_text_stream* to the class *Out_stream* (containing the *write* procedure). *TEXT_STREAMS* will not be described here.

```
    def write(s) ==
    prog
      for i in [1..#s] loop
        call put(make_Nat(s[i]))
      endloop
    endprog;

    def read_Char ==
    prog
      var b: Byte;

      call get =: b
      on mode_error → return error mode_error
      [] end_file   → return error end_file
      endon;

      return make_Char(b)
    endprog;

    def read ==
    prog
      var s: String = e,
          c: Char;

      loop
        call read_Char =: c
        on mode_error → return error mode_error
        [] end_file   → return error end_file
        endon

      while c.is_space;
      endloop;

      loop
        @s ⊢c;
        call read_Char =: c on end_file invoke exit endon
      while ¬c.is_space;
      endloop;

      return s
    endprog;

    def last_Char == make_Char(last_byte)
  endmodule
endcontext
```

9.3.1 The *end_line* constant.

The *TEXT_FILES* context defines a *Char* constant representing the line separator. (Even if the system uses more than one character to separate lines, *Abel* programs will use only the *end_line* character.) The actual character value is implementation-defined.

9.3.2 Output to text files.

The procedure *write* is defined for writing *String* values to text files. It will also handle values of other types if a coercion function from that type to *String* has been defined. This is the case for all predefined types.

Example 16 Writing to text files.

@*f.write*('x = ');
@*f.write*(*x*);
@*f.write*(*end_line*);

The three value output here are of types *String*, *Int* and *Char*, so the last two values will be coerced to *String* values. The three calls above could also be written as

@*f.write*('x = ' ⊢ *x* ⊢ *end_line*);

9.3.3 Input from text files.

The two basic input routines are *read_Char* and *read*; the former reads the next byte in the file and converts it to a *Char* value, the latter reads a *text field*, i.e. a non-empty sequence of non-blank characters surrounded by blanks. (For a definition of *blank characters*, see the defintion of *Char* in section 7.7.3 on page 72.) This field may subsequently be coerced to any type for which a coercion function from *String* has been defined.

Example 17 Reading from text files.

var *x*: *Int*, *c*: *Char*;
@*f.read* =: *x*;
@*f.read* =: *c*;
@*f.read_Char* =: *c*;

The difference between the last two calls is that the former will read the next non-blank character (which must be in a field of length 1) while the latter will read the next character irrespective of whether it is blank or not.

The *read* routine will always read the blank character following a field (and thereby terminating it). This character is accessible through the *last_Char* routine.

Bibliography

[Dahl 77] Ole-Johan Dahl. *Can Program Proving be made practical?* Research report no. 33, Institute for Informatics, University of Oslo.

[Dahl 79] Ole-Johan Dahl. *Time Sequences as a Tool for Describing Program Behaviour.* Research report no. 48, Institute for Informatics, University of Oslo.

[Dahl 84] Ole-Johan Dahl, Olaf Owe. *A Presentation of the Specification and Verification Project "Abel".* Research report no. 90, Institute for Informatics, University of Oslo. (Presented at VERkshop III, Watsonville, Ca., Feb. 1985.)

[Dahl 86] Ole-Johan Dahl. *Object Oriented Specification.* Presented at The Object Oriented Programming Workshop, IBM T. J. Watson Research Center, Yorktown Heights, June 9–13, 1986.

[FS 76] Hans Thomas Faye-Schjøll, Oddvar Hesjedal, Olaf Owe. *PROVER, utkast til et programverifikasjonssystem.* Cand.real. thesis, Institute for Informatics, University of Oslo.

[Gutt 84] J. V. Guttag. *Larch in Five Easy Pieces.* Digital Systems Research Center, Palo Alto, Ca., July 1985.

[Kirk 86] Bjørn Kirkerud. *BABEL, An Applicative, Strongly Typed Language.* Research Report, Institute for Informatics, University of Oslo. (In preparation.)

[Lang 78] Dag F. Langmyhr, Olaf Owe. *Revised Report on the Programming and Specification Language Abel.* Research Report no. 38, Institute for Informatics, University of Oslo.

[Meld 86] Sigurd Meldal. *Language Elements for Hierarchical Abstraction.* Part III of Ph.D.-thesis, Institute for Informatics, University of Oslo, May 1986.

[Owe 84] Olaf Owe. *An Approach to Program Reasoning Based on a First Order Logic for Partial Functions.* Technical Report no. CS-081, UCSD, La Jolla, Ca., June 1984. (Updated February 1985 as Research Report no. 89, Institute for Informatics, University of Oslo.)