The Formal Definition of the Programming Language Apple[1]

Richard B. Kieburtz
Oregon Graduate Center

Bengt Nordström
Chalmers Technical University
and
University of Gothenburg

Abstract

This report contains a complete, self-contained formal definition of the experimental programming language Apple. The definition is given in the style of denotational semantics. However, the language design allows a *direct* semantics, that is, without the use of a domain of continuations. It also provides static environments in which variables cannot have alias names, and the semantics therefore can be stated without the use of a domain of locations.

Apple is a typed language, and allows the declaration of programmer-defined type constructors called **class**es, which afford the means to define abstract data types in the language. A type constructor may be parameterized by types and constants, and therefore can define a set of polymorphic operators. The description of types and of the notion of type-correctness adds considerably to the volume of text in the formal definition. However, typing helps to make the definition more precise by making explicit the class of syntactically well-formed sentences that are intended to be given meanings in the language.

Keywords: abstract data types, static environments, denotational semantics, type polymorphism

Oregon Graduate Center Technical Report No. CS/E-82-02, May, 1982
Revised September, 1983

[1]The research reported here has been supported in part by the National Science Foundation, under grant MCS 790417 and by Naturvetenskapliga Forskningsrådet under grant F 3766.

1. Introduction

Apple has been designed to be a simple, yet powerful programming language. It is basically an applicatitive language. Although its syntax allows statements, its semantics allows a straightforward translation of every Apple program that uses statements into an equivalent Apple program that is purely applicative. Statements are provided in Apple as a syntactic alternative to purely applicative programming.

A criterion for simplicity applied during the design of Apple is that its formal semantic definition must be concise, direct, and understandable. This definition, given in the notation pioneered by Scott and Strachey [SS], is presented here. A companion paper [KN81] presents the language informally and gives some motivation for its design and use. *The Apple Programmer's Manual* has yet to be written.

Customarily, a programming language is designed before its formal semantics is given. This practice has implied that the description of the semantics of a traditional language such as Pascal, PL/1, Algol 68 or Ada has been understandable to only a few persons other than those who have taken an active part in producing the description. The design of Apple has been guided by the requirement that it should have a simply described semantics.

A first decision in the language design process was to do without semantic "features" that would require the use of locations and continuations for their description. These domains are often used to explain programming language concepts that derive from a machine-oriented view of computation. Locations are needed when a language contains features such as explicit pointers and parameters passed by reference. Continuations have been used to explain the *goto* statement and its relatives, *escape*, *exit*, *return* etc.

Some other important design decisions were:

- that the language should exhibit a high degree of orthogonality, as advocated by van Wijngaarden [vW].
- that functions are first-class objects, as are numbers and characters. It is possible to declare either a constant or a variable of a function type. The application of a function to an argument cannot produce a "side effect".
- that types, as well as constants of a function type, may be declared recursively. This means that it is possible to declare types whose values are lists or trees.
- Apple has assignable variables, however its scope rules do not allow global variables. Since there are no global variables, assignment cannot cause a side effect. Assignment to variables is allowed as a "safe" convenience to a programmer, yet the semantics of expression evaluation remains functional.
- there is no restriction on the type of a value that can be assigned to a variable or used for the range of a function.
- polymorphic operators can be defined, using the **class** declaration construct. Polymorphism is explicitly resolved in Apple when a class is instantiated to a type, by specifying its type parameters.

Apple is a typed language. Each constant or variable has associated with it, by declaration, a denotation called a type signature. Type signatures are intended to denote sets of objects, but they have not been given formal interpretations. The use of types enables a type-checking algorithm to separate those syntactically correct texts which may be meaningful from those that have no meaning in Apple.

Type-correctness is determined by direct inspection of the type signatures associated with named objects, and represented in an abstract syntax. It does not depend upon an interpretation of types as sets of values or as algebras. Accordingly, we have not attempted to give any deeper meaning to types in the formal definition of Apple. (For a different point of view concerning types, see [Mi78, McS82].)

Type signatures are constructed from:

— predefined types which include:

integer, denoting the integer numbers (without a maximal or a minimal element);

rational, denoting the rational numbers;

ascii, denoting the ascii character set;

bool, whose values are *true* and *false*.

— enumeration types, which include any finitely enumerated set of literal constants;

— type constructors, which are templates for the synthesis of type expressions.

There are constructors for:

— restricted types, which denote a subset of the values of a containing type, as specified by a characteristic predicate;

— recursive types, for defining structures such as lists and trees;

— structured types, patterned after the record types of Pascal;

— product types, denoting tuples of values;

— mapping types, whose values are functions and for which arrays may provide possible representations;

— powerset types, whose domain is restricted to the types integer, or an enumeration type, or a restriction of one of these;

— discriminated unions of types, such as are found in Algol 68.

Another primary objective of Apple is to present a notation suitable for composing large programs out of modular units. There are two ways to define these units.

An **environment** is comprised of declarations of classes, types, constants, and nested environments. It provides a semantic context for the interpretation of constant definitions or of program statements. The contents of an environment may be explicitly imported into another environment.

A **class**, or abstract data type, abstracts a set of objects characterized by a behavior, rather than by the programmed representation of these objects. Only the constants (including functions) that define this behavior are exported from a class definition. Classes are integrated into the language through the type mechanism. A class becomes a new, programmer-defined type constructor. It may have formal parameters that designate types or typed constants. A class must be instantiated by providing actual parameters when it is used in a type declaration to define a new type.

1.1. A brief description of environments

An environment definition consists of declarations of constants, types, classes and environments. An environment may be named in an environment declaration,

```
env display
        [const erase : ascii == '
         class screen ==
                        :
        ]
```

A named environment is a self-contained unit. The declarations it contains use only those definitions given within it, or which have been explicitly imported, or which are contained in the universal environment. The universal environment contains definitions of arithmetic, ascii, and boolean types and their operators. Its contents are imlicitly imported into every declared environment.

An unnamed environment occurs in the body of a function definition, and in the representation part of a class definition. An unnamed environment always imports the entire environment that surrounds it.

In order to import some definitions from another environment, a **ref**erence declaration can be given, referring to the name of an environment and to the names of types, constants, classes or environments that are to be imported.

Example: **ref** foo (**type** t_1, t_2 **const** c_1, c_2, c_3 **env** E, foo)

Names imported from other environments can be redefined, so that collisions between imported names and those defined in a local environment can be avoided.

1.2. A brief description of classes

An Apple program can be partitioned into modules called **class**es (following the notation of Simula [Da]) which serve as programmer-defined constructors for types. A class defines a family of constants, including functions. An instance of this family will be defined for each type that is derived from the class.

A constant may have as its type (or as a component of its type signature) the class-derived type itself. A function constant having the class-derived type as its range type is called a *constructor* for the type. If the class-derived type does not occur as a component of the type signature of the range, but does occur in the domain type of the function, we call the function an *extractor* for the type. There is no syntactic distinction between constructors and extractors, however.

A class is seen as a template for constructing new types in Apple. A class definition consists of the following parts:

- a heading which names the class and lists its formal parameters. These may be types and constants to be used in the class definition.
- a **def**ines list which gives names and type signatures to the constants that are to be exported by the class definition. For each type derived from the class (by a type declaration) a set of these constants will be included in the environment in which the class-derived type is defined.
- a representation part, which declares a local environment in which meanings are given to the constants named in the defines list. This local environment must include:

.. declaration of a type named *carrier*, in terms of which each class-derived type is represented;

.. declarations of the constants named and typed in the **def**ines list.

The representation part may also include declarations of other constants and types needed by its mandatory declarations.

There is a predefined class in Apple, called *stream*, whose values are sequences. Input and output is achieved by binding stream-typed objects in Apple to sequential files of the computer environment in which an Apple program is to be run.

1.2.1. Function definitions

A value of a mapping type can be denoted by a function constructor, which is an expression that has the following parts:

— a heading, beginning with the keyword **function**, and naming the formal parameter. If the domain type of the mapping is a product type, then the components of the formal parameter may be named individually.

— any declarations to be included in the local environment of the function definition;

— an optional statement part, which consists of a list of statements prefaced by the declarations of program variables;

— a **return** expression, which may contain occurrences of variables that have been declared in the statement part, as well as constants and formal parameters.

Only the heading and the **return** expression are required parts of a function constructor.

Variables declared in the statement part of a function constructor are not imported into a nested environment, and therefore are not inherited by any nested function definition. There is no concept of a global variable. On the other hand, the formal parameter of a function constructor has the status of a constant in its local environment, and will be imported into the local environments of functions whose declarations are nested within the declaration part of the function constructor.

Since variables are not imported, a function defined by a constructor can never produce a "side effect" when it is applied to an argument. The value of a variable can be changed in no way other than by an explicit assignment statement in which the variable appears on the left side. This is a profound simplification over the uses made of variables in most statement-oriented languages.

1.2.2. An example of a class declaration

```
class Stack of type T        // T is a parameter of the class
   == def
         Newstack : carrier,
         Push  : carrier * T -> carrier,
         Pop   : carrier -> carrier,
         Top   : carrier -> NonEmpty#T +
                            EmptyStack#Failure

      rep
         type carrier :record
                          Store : integer -> NonEmpty#T +
                                             EmptyStack#Failure,
                          Index : integer
                       end

         const
            Newstack == [Store == function (i)
                                     return EmptyStack# failure
                                     end,
                         Index == -1],

            Push ==  function (S,x)
                        return
                           let P : integer == succ S.Index in
                              [Index == P,
                              Store == S.Store|P: NonEmpty# x]
                           endlet
                        end,

            Pop ==   function (S)
                        return
                           let P : integer == S.Index in
                              if P >= 0 then
                                 [Store == S.Store, Index == pred P]
                              else
                                 S
                              endif
                           endlet
                        end,

            Top ==   function (S)
                        return S.Store(S.Index)
                        end
endclass
```

In an environment in which this class has its declaration, or into which its declaration has been imported, new types may be declared to be instances of this class, and variables of these types can be declared.

2. Notational conventions

— $\perp$ denotes the undefined element (called "bottom") in any domain.

— "?" denotes a distinguished value (in any domain) called the *error element*. This value differs from $\perp$, the undefined element, in that "?" is well defined and may be tested by an equality predicate, for instance. Almost every one of the semantic functions is strict with respect to "?". Exceptions are the updating functions $\Leftarrow$ and $\oplus$ defined below.

— **if** b **then** e_1 **else** e_2 **fi** will denote the biconditional which maps
$(true, e_1, e_2)$ to e_1, $(false, e_1, e_2)$ to e_2, $(?, e_1, e_2)$ to ?, and $(\perp, e_1, e_2)$ to $\perp$

— **if** b_1 **then** e_1 **elseif** b_2 **then** e_2 **else** e **fi** will mean
if b_1 **then** e_1 **else (if** b_2 **then** e_2 **else** e **fi) fi**

— **if** b **then** e **fi** will mean **if** b **then** e **else ? fi**

— An identifier having the initial letter in upper case will stand for a domain. If the same identifier is used with the initial letter in lower case, then it will stand for an element within that domain. For instance, $val \in$ **Val**, $id \in$ **Id**.

— Instead of writing $f = \lambda x.e$ we will write $f(x) = e$
Therefore $g(x)(y) = x * y$ means $g = \lambda x.\ \lambda y.\ x * y$

If **A** and **B** are domains then

— $\mathbf{A} \to \mathbf{B}$ is the domain of all continuous functions from **A** to **B**

— $\mathbf{A} \times \mathbf{B}$ is the cartesian product

— $[a_1{:}b_1, \ldots\ a_n{:}b_n]$ is the element in $\mathbf{A} \to \mathbf{B}$ which maps a_i to b_i, $1 \le i \le n$.
All other elements of **A** are mapped to "?" in **B**.

— If x is a tuple, for instance $x \in \mathbf{A} \times \mathbf{B}$, then $x{\downarrow}i$ will denote the ith projection of x. Alternatively, we may use the names of component domains to denote projections when the notation is unambiguous, i.e. $x.A$ and $x.B$ may be used in place of $x{\downarrow}1$ and $x{\downarrow}2$

— $x \Leftarrow y$ will denote
if $x, y \in \mathbf{A} \to \mathbf{B}$ **then** $\lambda a.$ **(if** $y(a) = ?$ **then** $x(a)$ **else ? fi) fi**

— $x \oplus y$ will denote
if $x, y \in \mathbf{A} \to \mathbf{B}$ **then**

$$\lambda a.\ \textbf{(if}\ x(a) = ?\ \textbf{then}\ y(a)\ \textbf{elseif}\ y(a) = ?\ \textbf{then}\ x(a)\ \textbf{else ?}\ \textbf{fi)}\ \textbf{fi}$$

— $x \Leftarrow_A y$ will denote
if $x \in A \times B$ **and** $y \in A$ **then** $(x.A \Leftarrow y, x.B)$ **fi**

— $x \oplus_A y$ will denote
if $x \in A \times B$ **and** $y \in A$ **then** $(x.A \oplus y, x.B)$ **fi**

In giving the meaning functions we shall use an informal pattern-matching mechanism in order to identify parameters. We use double brackets, $[\![\ ,\ ]\!]$, to enclose syntactical expressions. A syntactical expression consists of terminal symbols and variables that denote syntactical expressions.

When an expression has a finite number of components, we shall often use ellipses, "...", to denote the components not explicitly written.

We also adopt the **let** and **where** notations to factor components of definitions. When an expression to be defined consists of a pair, we may write "**where** $(x, y) = \cdots$" to give the names "x" to its first part and "y" to its second part.

The reader familiar with denotational semantics will recognize the operators $\Leftarrow$ and $\oplus$ as operators which extend a mapping. The $\oplus$ operator extends the mapping which is given as its first argument by the mapping which is its second argument, when the extended mapping is applied to any value on which one or the other of the argument mappings was undefined (i.e. mapped an argument to "?"). The $\Leftarrow$ operator also extends its first argument by the second, but differs from $\oplus$ in that the second argument supersedes the first, on values on which both are defined.

In the above definitions, and throughout the rest of this document we shall use summation notation,

$$\sum_{j=1}^{n} [\text{"}i_j\text{"} : m_j]$$

to mean

$$[\text{"}i_1\text{"} : m_1] \oplus [\text{"}i_2\text{"} : m_2] \oplus \ldots [\text{"}i_n\text{"} : m_n]$$

3. Description of the semantic definition

We use denotational semantics to define the meaning of a program. This method uses a set of meaning functions to associate mathematical objects with the syntactical expressions of the programming language. It is necessary to give such a definition when the programming language is not itself a mathematical language. The meaning functions are defined on domains. We shall use the following domains, where + is the separated sum and × is the coalesced product:

O = R + Id + (O × O) + (O → O)

An object (or value) is either a rational number, or an identifier, or an ordered pair of values, or a function.

T = Id + Enumeration + Restriction + Set + Record + Product + Union + Map

where

Enumeration	**= Id***
Restriction	**= T × Expr**
Set	**= T**
Record	**= (Id × T)***
Product	**= T***
Union	**= (Id × T)***
Map	**= T → T**

The domain of types is composed from the primitive domains of identifiers and expressions, where expressions is a class of syntactical objects. (We have made use of the Kleene star to denote the repeated product of a domain. This should cause no confusion.)

Env = Const × Type × Var × Op × E_name × Class

--- The domain of environments is composed of the domains:

Const = Id → (T × O)

--- An element of **Const** maps identifiers to pairs consisting of a type and a value.

Type = Id → (T × O)

— An element of **Type** maps identifiers to pairs consisting of a type and a value. The value is the default initial value given to variables of that type.

Var = Id → T

— An element of **Var** associates a type with each variable.

Op = Id → T

— An element of **Op** maps identifiers which are the names of class-derived types to elements of **Const** which define the meanings of the class constants for these types.

E_name = Id → Env

— An element of **E_name** associates environments with environment names.

Class = Id → Env → (C × Const)

where **C = P(Id) × (Id → T)**

— An element of **Class** maps class identifiers to their meanings. The class signature domain, **C**, contains characterizations of the parameters of classes.

Val = Id → O

— An element of **Val** maps names of program variables to values, and represents a state of a computation.

We shall use the following meaning functions:

eval : **Stmt → Env → Val → Val**

— where **Stmt** is a primitive domain of syntactical objects. *eval* yields the meaning of a program statement which changes the mapping that associates values with variables in a given environment.

value_of : **Expr → Env → Val → O**

— yields the meaning of an expression.

e_type : **Type_expr → Env → T**

— where **Type_expr** is a primitive domain of syntactical objects. *e_type* yields the meaning of an expression denoting a type.

m_decl : **Decl → Env → Env**

— where **Decl** is a primitive domain of syntactical objects. *m_decl* yields the meaning of a declaration of a constant, type, class, or environment.

m_var : **Var_decl → Env → (Env × Val)**

— where **Var_decl** is a primitive domain of syntactical objects. *m_var* yields the meaning of a declaration that gives types and initial values to variables.

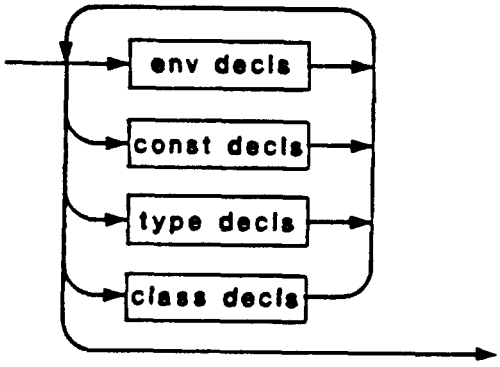
m_param : **Param_decls → Env → Env**

— where **Param_decls** is a declaration of actual parameters of a class instance. *m_param* yields the meanings of the parameters.

4. Environments

An environment is a collection of declarations of constants, types, classes, and (either nested or imported) environments. These declarations provide a semantic context in which to give meanings to variables, expressions, and statements. Environment declarations may be nested within one another or within the body of a function definition or the representation of a class. An environment may include definitions made in surrounding environments. When the environment is that of a function body or a class representation, it imports the immediately surrounding environment by default. The universal environment, which contains definitions of the basic types and their constants is also imported by default into every other environment. In all other cases, the names which are to be imported into an environment from other known environments must be explicitly declared in a reference list.

Syntax



Semantics

m_decl ⟦<environment>⟧(env) =

Y_{Env} $\lambda e.env \Leftarrow m\_decl$ ⟦<class decl>⟧

(m_decl ⟦<type decl>⟧

(m_decl ⟦<const decl>⟧

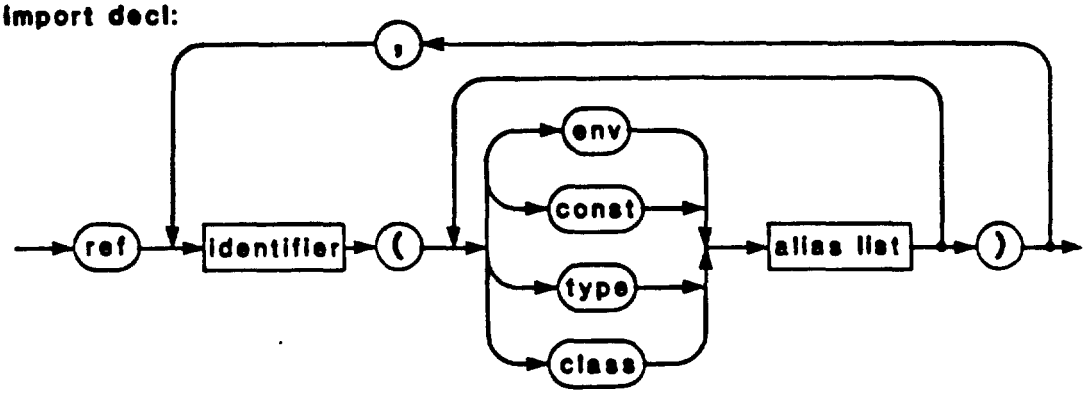
(m_decl ⟦<env decl>⟧(e)))))

The significance of the fixed-point operator is that the meanings associated with declared names can depend upon one another, regardless of the order in which declarations are listed. In particular, constants of a mapping type (i.e. functions) may have recursive or mutually recursive definitions.

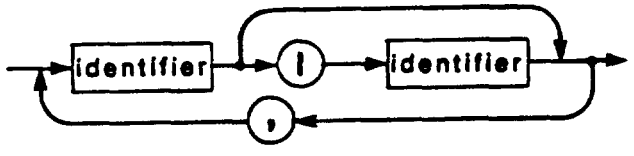
4.1. Environment importation

Syntax

import decl:



alias list:



The notation $id_1 \mid id_2$ signifies that id_2 is to be an alias name in the local environment for the name id_1 from an imported environment. id_2 has all the same attributes (type, value, etc.) associated with id_1 in the environment in which is declared. When an alias is the same as the originally declared name, it need not be denoted, i.e.

ref A (**const** C)

is an abbreviation for

ref A (**const** C|C)

Semantics

$$m\_decl\ [\![\ \textbf{ref}\ id\ (\textbf{env}\ id_1 \mid a_1, \ldots\ \textbf{const}\ j_1 \mid b_1, \ldots\ \textbf{type}\ k_1 \mid c_1, \ldots\ \textbf{class}\ l_1 \mid d_1, \ldots)\ ]\!](env) =$$

$$[\,] \oplus_{E\_name} \sum_i [\text{"}a_i\text{"} : e'.E\_name\ [\![id_i]\!]]$$

$$\oplus_{Const} \sum_i [\text{"}b_i\text{"} : e'.Const\ [\![j_i]\!]]$$

$$\oplus_{Type} \sum_i [\text{"}c_i\text{"} : e'.Type\ [\![k_i]\!]]$$

$$\oplus_{Op} \sum_i [\text{"}c_i\text{"} : e'.Op\ [\![k_i]\!]]$$

$$\oplus_{Class} \sum_i [\text{"}d_i\text{"} : e'.Class\ [\![l_i]\!]]$$

where $e' = env.E\_name\ [\![id]\!]$

4.2. Environment declarations

Syntax

Semantics:

m_decl ⟦ **env** *id* [<import decl><environment>] ⟧(*env*) =

$$env \oplus_{E\_name}["id": Y_{Env}\ \lambda e'.\ m\_decl\ ⟦\text{<import decl>}⟧(env) \oplus m\_decl\ ⟦\text{<environment>}⟧(e') \oplus_{E\_name}["id": e']]$$

A named environment contains the meaning of its own name.

4.3. Constant declarations

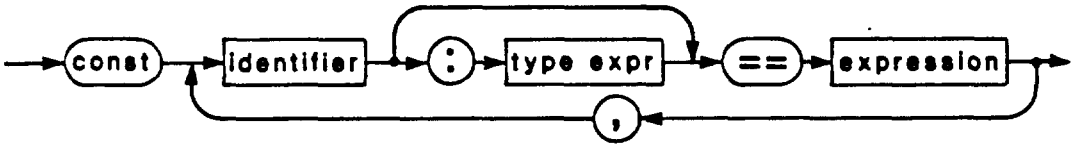
The declaration of a constant defines a name which must be unique among identifiers used for constants (or variables) in the current environment. The declaration gives it a type and a value.

A small example:

const *int* 15 : *integer* == 32767,

complex_zero : *float* * *float* == (0,0)

Syntax



Semantics:

m_decl ⟦ **const** *id* : *t* == *expr*, <const decls>⟧(*env*) =

m_decl ⟦ **const** <const decls>⟧

(*env* $\oplus_{Const}$["*id*" : (*e_type* ⟦ *t* ⟧, *value_of* ⟦ *expr* ⟧(*env*)([ ]))])

4.4. Type declarations

A type declaration defines a name and associates with it a type and an initial value to be given to variables declared of the type. If the initial value declaration is omitted, an initial value is supplied by a language-defined function, **init**, applied to the type expression, enumeration, or class instance that appears as the right-hand side of the type declaration. The function **init** is defined in Section 5.5.

In case a type is specified by explicit enumeration of its values, the names that are listed to denote values are implicitly declared as constants of the type. In case a type is specified to be an instance of a class, operators defined by the class are implicitly declared for the type.

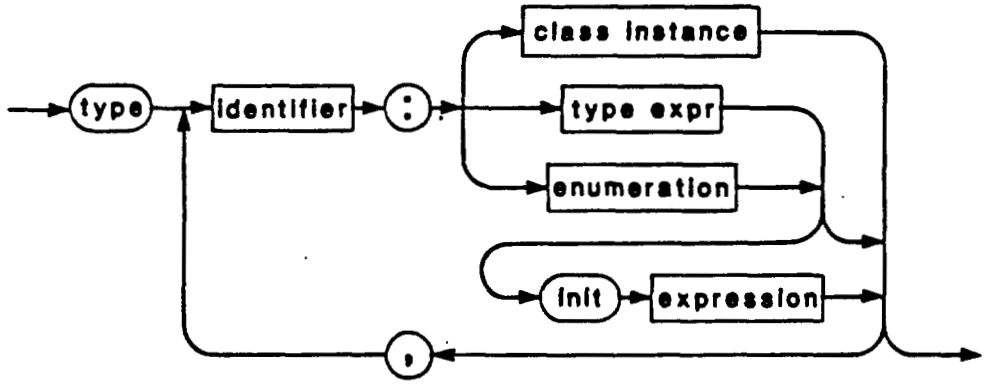
Enumerations and instances of classes may be used only to define types. That is, these forms may appear only as the right side of a type declaration. Type expressions other than these two forms may occur in any context in which a type is to be specified. The reason that enumerations and class instances are restricted to explicit type declarations is that they contribute new names to components of the environment in which they occur.

A small example:

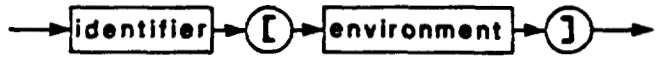
type *complex* : *float* * *float*,
color : {*red*, *green*, *blue*},
ascii_stream : *stream* [**type** *T* : *ascii*]

Syntax

type decls:



class instance:



Semantics:

m_decl ⟦ **type** $id : t$, <type decls> ⟧(env) =

m_decl ⟦**type** <type decls>⟧(m_decl ⟦ **type** $id : t$ ⟧(env))

m_decl ⟦ **type** $id : t$ ⟧(env) =

cases t **is**

"$\{id_1, \ldots id_n\}$" ==> $env \oplus_{Type}$ ["id" : t']

$\oplus_{Const} \sum_{j=1}^{n}$ ["id_j" : (t', j)]

where t' = enumeration("id_1",... "id_n")

"C_id [**type** $i_1{:}t_1, \ldots$ **const** $k_1{:}s_1 == e_1, \ldots$]" ==>

$env \oplus_{Type}$ ["id" : class]

$\oplus_{Op}$ ["id" : $C\_def.Const$]

where C_def = $env.Class$ ⟦C_id ⟧

(m_param ⟦ **type** $i_1{:}t_1, \ldots$ **const** $k_1{:}s_1 == e_1, \ldots$ ⟧(env))

"<type expr>" ==> $env \oplus_{Type}$ ["id" : e_type ⟦t ⟧(env)]

esac

where m_param is a function that defines the local environment contributed by the actual parameters in a class instance:

m_param ⟦**type** $i_1{:}t_1, \ldots$ **const** $k_1{:}s_1 == e_1, \ldots$ ⟧(env) =

[] $\oplus_{Type}$ $type\_args$

$\oplus_{Const} \sum_{j}$ ["k_j" : (e_type ⟦s_j ⟧(env'), $value\_of$ ⟦e_j ⟧(env')([]))]

where env' = $env \oplus_{Type}$ $type\_args$

and $type\_args$ = $\sum_{j}$ ["i_j" : e_type ⟦t_j ⟧(env)]

Parameters to an instance of a class are evaluated in the environment in which the class instance is declared.

5. Types

Types are used to partition the constants and variables of an Apple program into disjoint sets. Rules for checking type-correctness enable a language translator to reject a program that is patently without meaning.

Types occur in declarations of constants, variables, classes and named types. A type can be specified by an explicit enumeration of values, by instantiation of a class, or by a type expression. A type expression is composed from the names of types and from applications of predefined functions yielding types (so-called type constructors). The type constructors of Apple are: restrictions, records, products, mappings, powersets, unions and recursive type constructors. The classes that a programmer can declare are also analogous to type constructors, in that existing types can be given as parameters to construct a new type. Class instances and enumerations may not be used directly as types in type expressions, however.

5.1. Type identifiers

Semantics:

e_type ⟦*id*⟧(*env*) = *env*.*Type* ["*id*"]↓1

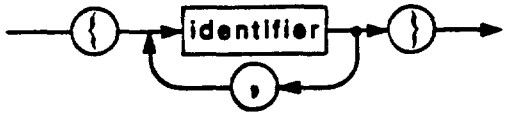
5.2. Enumeration types

A small example:

{red, green, blue}

Syntax

enumeration:



Semantics:

e_type ⟦{id_1,... id_n}⟧(*env*) = enumeration("id_1",... "id_n")

There is a predefined enumeration type called *ascii* which consists of the symbols defined by the international standard character set.

The predefined enumeration type *Bool* consists of two values, *false*, *true*, and defines the operators *and*, *or*, *not* with their conventional meanings.

5.3. Arithmetic types

Semantics:

e_type ⟦float⟧(*env*) = "float"

e_type ⟦integer⟧(*env*) = "integer"

Informally, the type corresponds to the set of integer numbers. Note that there is no restriction on the size of an integer. The type corresponds to the rational numbers, also without restriction as to size or precision of representation.

5.4. Predefined type constructors

type expr:

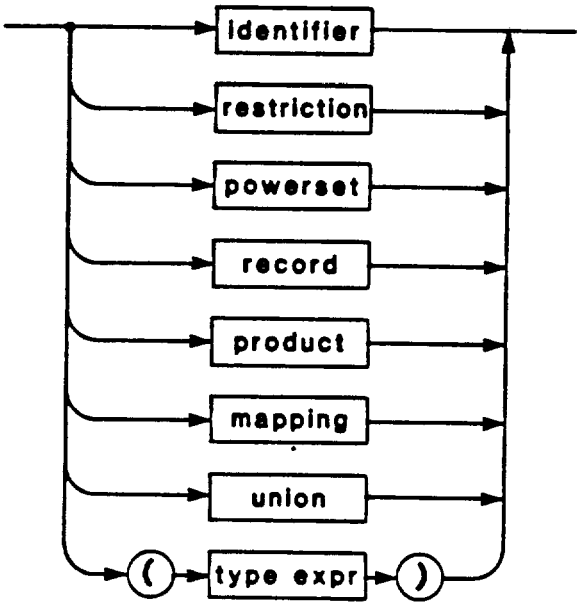


Table 5.4.1 – Operator precedence in type expressions

The following table gives the precedence of the type-formation operators that may occur in type expressions. Operators of lower precedence level bind their arguments before those of higher precedence level. Left association means that in an expression involving multiple occurrences of the same operator, occurrences to the left bind their operands first.

Precedence of type operators			
level	**operator**	**name**	**association**
1	*	cartesian product	associative
2	->	mapping	right
3	+	discriminated union	associative
4	rec	recursive type	

5.4.1. Restrictions

A new type may be defined by specifying a characteristic predicate which restricts the set of values of some base type. The predicate is given as a Boolean expression, functional upon a variable bound in the type specification. Abbreviations of this syntax are allowed to specify intervals of values from totally ordered base types.

An expression of the base type may be used in a context in which a value of a restricted type is expected. In order to convert the type from that of the base type to that of the restriction, a predefined function *cnv* is applied to the expression. *cnv* evaluates the characteristic predicate of the restriction,

applied to the value of the expression, in order to determine whether the value is allowed in the context of its use. This provides a powerful facility for checking that the values generated during execution of a program satisfy the expectations of its author.

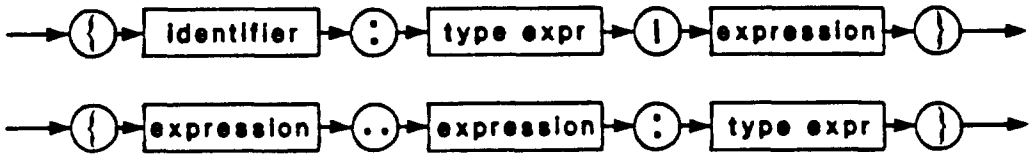
Some small examples:

{1..10 : integer}

{x:float | abs(x - sqrt 2) <= 0.5}

{x:vector | sorted x}

Syntax



Semantics:

$e\_type\ [\![\{id:t \mid expr\}]\!](env) =$

restriction($e\_type\ [\![t]\!](env)$, ($ground\ [\![\lambda id\ .\ expr]\!](env)$))

The use of a λ-expression in a type expression expresses only the fact that the identifier is bound locally. The only conversion rule of the λ-calculus that is used in comparing types is the rule of α-conversion (renaming of a bound variable).

The function *ground* : **Expr** → **Env** → **Expr** substitutes for all identifiers occurring in the expression given as its first argument, their definitions in the environment given as its second argument. Thus the term $ground\ [\![expr]\!](env)$ contains no occurrence of any identifier bound in *env*. This allows the definition of a restricted type to be imported from a named environment into a current environment without requiring additional constant identifiers to be imported.

To denote an interval of values of a totally ordered type, the type expression

$\{e_1..e_2 : t\}$

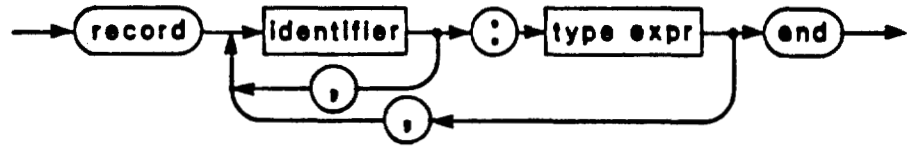
is allowed as an abbreviation for the type expression

$\{x:t \mid e_1 <= x$ **and** $x <= e_2\}$

where e_1 and e_2 are expressions.

5.4.2. Records

Syntax



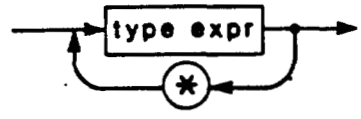
Semantics:

$$e\_type\ [\![\ \textbf{record}\ id_1{:}t_1,\ldots id_n{:}t_n\ \textbf{end}\ ]\!](env)$$
$$= \text{record}((id_{\pi_1}, e\_type\ [\![\ t_1\ ]\!](env)),\ldots\ (id_{\pi_n}, e\_type\ [\![\ t_n\ ]\!](env)))$$

where π is a permutation of $1..n$ such that $i<j\ \Rightarrow id_i<id_j$ in the lexical order on identifiers that is induced by the collating sequence on ASCII characters.

5.4.3. Cartesian products

Syntax



Semantics:

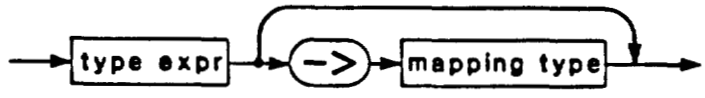
$$e\_type\ [\![\ t_1{*}\ldots{*}t_n\ ]\!](env) = \text{product}(e\_type\ [\![\ t_1\ ]\!](env),\ldots\ e\_type\ [\![\ t_n\ ]\!](env))$$

5.4.4. Mappings

A small example

type vector : {1..n: integer} -> float,
dictionary : string -> string

Syntax



Semantics:

$$e\_type\ [\![\ t_1 \text{ -> } t_2\ ]\!](env) = \text{map}(e\_type\ [\![\ t_1\ ]\!](env),\ e\_type\ [\![\ t_2\ ]\!](env))$$

The mapping constructor corresponds to the function space in mathematics.

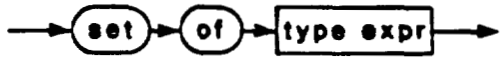
Note that Apple imposes no restriction upon the domain of a mapping type.

5.4.5. Powersets

A small example

set of {0..maxval: integer}

Syntax



Semantics:

e_type ⟦ **set of** *t* ⟧(*env*) = set(*e_type* ⟦ *t* ⟧(*env*))

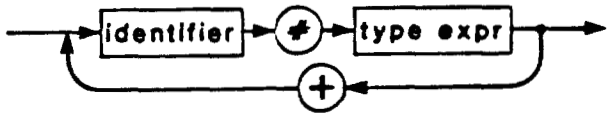
Composition of powerset types is restricted in Apple. The base type of a powerset must be or an enumeration type, or a restriction of one of these.

5.4.6. Discriminated unions

A small example

type sparse_matrix : integer*integer -> (defined#float + undefined#nulltype)

Syntax



Semantics:

e_type ⟦ $id_1\#t_1+\ldots\ id_n\#t_n$ ⟧(*env*) =

union($(id_{\pi_1}$, *e_type* ⟦ t_1 ⟧(*env*)),... (id_{π_n}, *e_type* ⟦ t_n ⟧(*env*)))

where π is a permutation of $1..n$ such that $i<j \Rightarrow id_i<id_j$ in the lexical order on identifiers that is induced by the collating sequence on ASCII characters.

5.4.7. Recursive type expressions

A small example:

```
type nulltype : {null},
                intlist : list rec empty#nulltype + cons#(integer*list),
                inttree : tree rec empty#nulltype +
                                  maketree#(integer*tree*tree)
```

Syntax

Semantics:

e_type ⟦*id* **rec** *t_expr* ⟧(*env*) = recursive(*e_type* ⟦*t_expr* ⟧(*env*)[rec/*id*])

where $e[\xi/x]$ denotes the expression gotten from e by replacing every free occurrence of x by ξ.

In order to simplify the semantics of recursive type expressions, the following restrictions are imposed:

1) The type expression following the keyword **rec** must be a union whose first term does not contain the identifier which precedes the keyword **rec**.
2) The identifier which precedes the keyword **rec** must not occur in the domain type of a mapping in the type expression that follows it.

Mutually recursive definitions of types have no meanings in Apple.

6. Classes

A class is a program unit in which a programmer can characterize a set of objects by their behaviors. This is done by defining a set of functions which produce values of these objects or which map these objects into values of other types. In order to define these functions by program units, it is necessary to give a specific representation for the values of objects in the class. This representation is given as a type declared as the *carrier* of the class.

The carrier and the programmed definitions of the functions are not germane to the programs that use objects of the class. Therefore these representational details are not exported into a surrounding environment.

A class can also be parameterized by types and constants. Parameters are particularly useful when classes are used to generalize data structures, as in the familiar example of a stack. Actual parameters are bound to the formal parameter names when an instance of the class is declared as a type.

Since a parameterized class may be instantiated in many different ways in the same program, its operators (exported functions) may be polymorphic. In order that polymorphism can be resolved, Apple requires each occurrence of one of these operators in an expression to be annotated with the name of a specific type which has been declared as an instance of the class. This is the solution that has been adopted in the data abstraction language CLU [Lis]. For example, if a class has been defined by

```
class Stack of type T ==
    def Push : carrier*T -> carrier,
        . . .
endclass
```

and if a type derived from this class has been declared by

```
type Intstack : Stack [type T : integer]
```

then a variable declared by

```
var S : Intstack
```

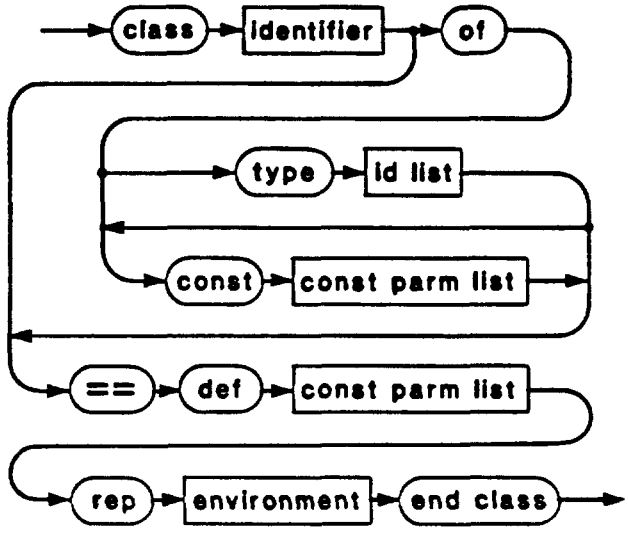
could be given a new value by execution of a statement such as

```
S := Intstack's Push (S,3)
```

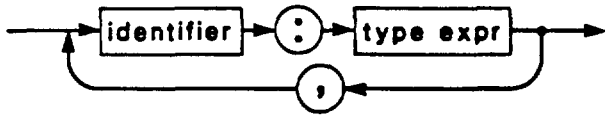
Because we have chosen to disambiguate the polymorphism of class-derived operators in this way, class instances are not regarded as first-class types. A class instance cannot be used in all contexts that require a type expression, but can only occur as the right-hand side of a type declaration.

6.1. Class declarations

Syntax



const parm list:



The environment declared as the representation of a class must contain the declaration of a type whose name is *carrier*, and must also contain declarations of all constants named in the *def* list.

A small example

```
class Bounded_stack of type T const bound : integer ==
    def Push  : carrier*T -> carrier,
        Pop   : carrier -> carrier,
        Top   : carrier -> T,
        Is_empty: carrier -> Bool
    rep
        type carrier : record
                          Inx: {0..bound: integer},
                          Store : {1..bound: integer} -> T
                       end
        const
            Push == function (S,x) ...
                 :
                 :
endclass
```

Semantics:

m_decl [**class** C **of type** $id_1,\ldots id_m$, **const** $c_1{:}s_1,\ldots c_n{:}s_n$
== **def** $f_1{:}t_1,\ldots f_k{:}t_k$ **rep** <environment> **endclass**] (env)

$= env \oplus_{Class}$ ["C" : $\lambda\rho$. // where ρ is in the domain Env
[ ] $\oplus_C$ t // type signature
$\oplus_{Const}$ $sift$ [**def** $f_1{:}t_1,\ldots f_k{:}t_k$] (r)]
// meanings of exported constants

where $sift$ [**def** $f_1{:}t_1,\ldots f_k{:}t_k$] (r) =

$$\sum_{j=1}^{k} [\text{"}f_j\text{"} : (e\_type\ [\![t_j]\!](r), r.Const\ [\![f_j]\!], 0)]$$

and t = ({"id_1",...}, ["c_1" : e_type [s_1](env'),...])

where env' = $env \oplus_{Type} \sum_{j=1}^{n} [\text{"}id_j\text{"} : \text{"}id_j\text{"}]$

and r = m_decl [<environment>] $(env \oplus \rho)$

Informal comments

Recall that the domain **Class** is a product of two domains, **(C × Const)**. An element of **C** describes the formal parameters of a class. An element of **Const** gives the meanings of the constants defined by a class-derived type. These meanings are obtained from the environment declaration that provides the class representation. They are extracted from the meaning of the representation by the function *sift*, defined above, which uses the **def**ines list of the class declaration.

7. Expressions

The various syntactic forms of expressions are strongly connected with types. Semantics are given on the supposition that expressions are type-correct, for otherwise an expression may have no meaning. Association of binary infix operators is generally to the left, subject to the precedence rules expressed in Table 7.1.

Syntax

Table 7.1 – Precedence of operators in Apple

Operators of lower precedence level bind their arguments before operators of higher precedence. An operator is said to associate to the left (right) if it binds its arguments before operators of equal precedence that occur to its right (left) in the same expression.

Operator Precedence Rules			
level	**operator**	**name**	**association**
1	#	injection	right
	$	type resolution	right
2	@	function composition	associative
	!	projection	left
	.	record field selection	left
3		function application	left
4	* / div mod	product operators	left
5	+ -	additive operators	left
6	= <> < <= > >=	relational operators	
	member	set membership	
7	not	negation	
8	and	conjunction	associative
9	or	disjunction	associative
10	| :	supercedes	left

7.1. Identifiers

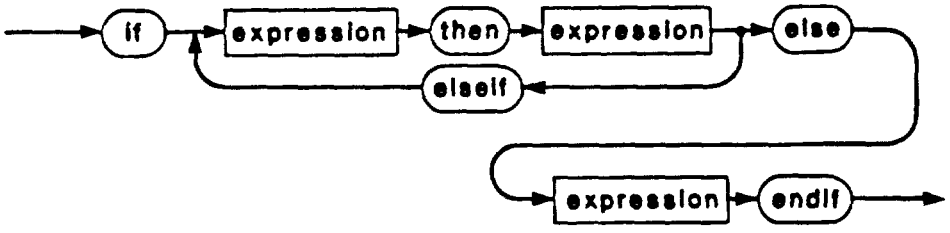
Semantics:

value_of ⟦*id*⟧(*env*)(*val*) =
    if *env.Const* ⟦ ⟧ ≠ ?
        then *env.Const* ⟦*id*⟧↓2
        else *val* ⟦*id*⟧
    fi

7.2. Expressions for type Bool

7.2.1. Conditional expressions

Syntax



Semantics:

value_of ⟦**if** b **then** e_1 **else** e_2 **endif**⟧(*env*)(*val*) =
    if *value_of* ⟦b⟧(*env*)(*val*)
        then *value_of* ⟦e_1⟧(*env*)(*val*)
        else *value_of* ⟦e_2⟧(*env*)(*val*)
    fi

The expression

if b_1 **then** e_1 **elseif** e_2 · · · **else** e_{n+1} **endif**

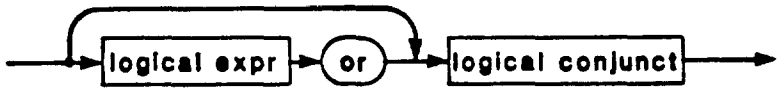
may be written as a more convenient notation for the nested conditional

if b_1 **then** e_1 **else**
if b_2 **then** e_2 **else**
...**else**
e_{n+1} **endif** · · · **endif endif**

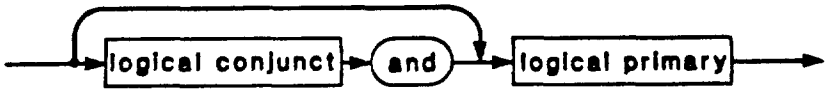
7.2.2. Boolean expressions

Syntax

logical expr:



logical conjunct:



logical primary:

Semantics:

value_of ⟦ **not** *e* ⟧ (*env*)(*val*) =
    if *value_of* ⟦ *e* ⟧ (*env*)(*val*) **then** *false* **else** *true* **fi**

value_of ⟦ e_1 **and** e_2 ⟧ (*env*)(*val*) =
    if *value_of* ⟦ e_1 ⟧ (*env*)(*val*)
        then *value_of* ⟦ e_2 ⟧ (*env*)(*val*)
        else *false*
    fi

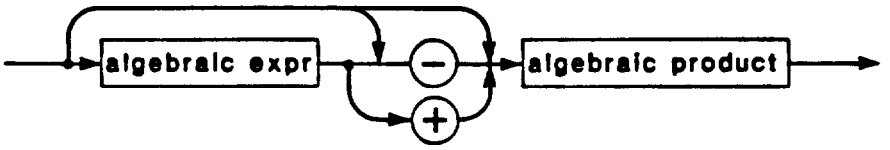
value_of ⟦ e_1 **or** e_2 ⟧ (*env*)(*val*) =
    if *value_of* ⟦ e_1 ⟧ (*env*)(*val*)
        then *true*
        else *value_of* ⟦ e_2 ⟧ (*env*)(*val*)
    fi

7.3. Arithmetic expressions

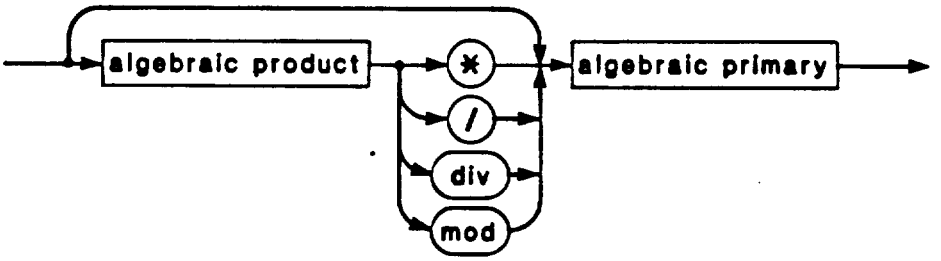
7.3.1. Denotation of values

Syntax

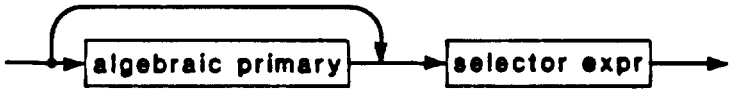
algebraic expr:



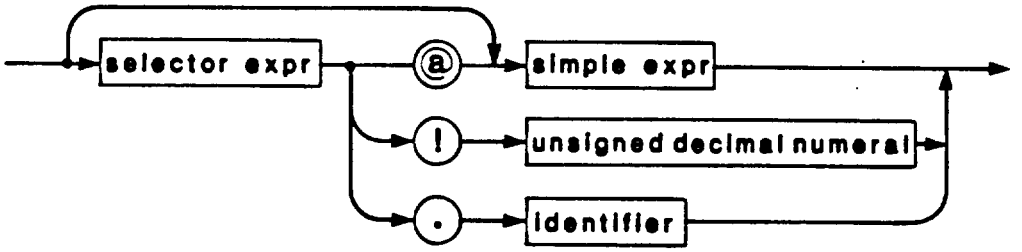
algebraic product:



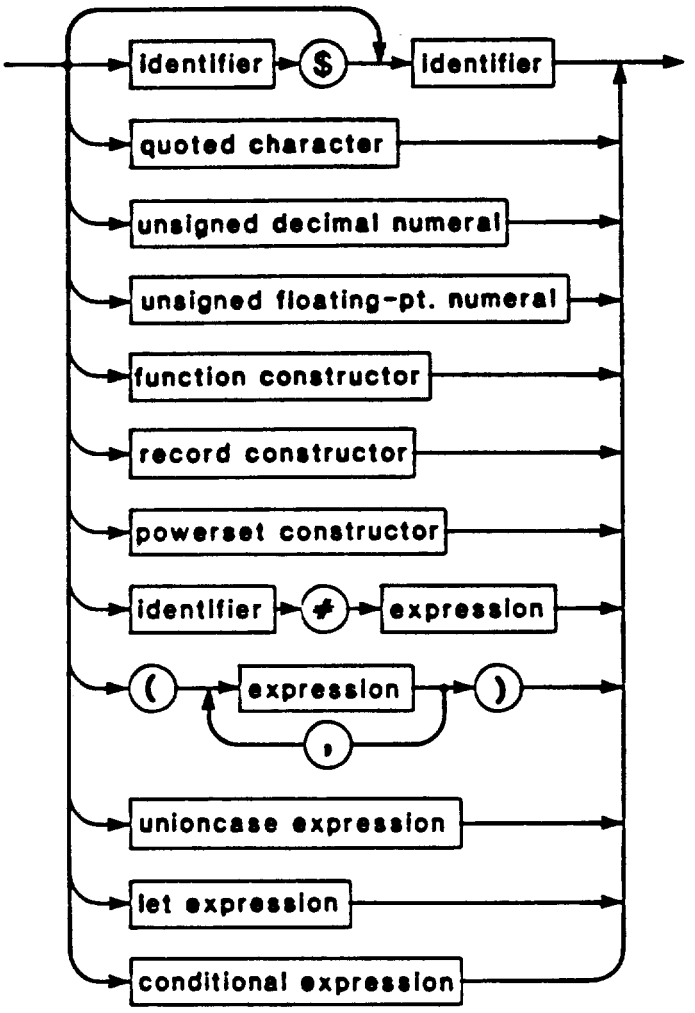
algebraic primary:



selector expr:



simple expr:


identifier
$
identifier
quoted character
unsigned decimal numeral
unsigned floating-pt. numeral
function constructor
record constructor
powerset constructor
identifier
#
expression
(
expression
,
)
unioncase expression
let expression
conditional expression

7.3.2. Unary operators

The unary minus "-" is defined on both *float* and *integer* types. The functions *pred* and *succ* are defined on integer types and on enumeration types, for which the meanings of constants are defined as natural numbers.

Semantics:

$$value\_of\ [\![-e]\!](env)(val) = -value\_of\ [\![e]\!](env)(val)$$

$$value\_of\ [\![succ\ e]\!](env)(val) = value\_of\ [\![e]\!](env)(val) + 1$$

$$value\_of\ [\![pred\ e]\!](env)(val) = value\_of\ [\![e]\!](env)(val) - 1$$

7.3.3. Binary operators

The operators "+", "-", "*" are defined for both *float* and *integer* types. The operator "/" is defined only for type *float*, and the operators *div* and *mod* are defined only for type *integer*.

Semantics:

$$value\_of\ [\![e_1\ \mathrm{op}\ e_2]\!](env)(val) =$$

$$M\ [\![\mathrm{op}]\!](value\_of\ [\![e_1]\!](env)(val), value\_of\ [\![e_2]\!](env)(val))$$

where "op" is one of the binary arithmetic operators. For the operators "+", "-", "*", "/", $M\ [\![op]\!]$ gives the customary operations of addition, subtraction, multiplication, or division.

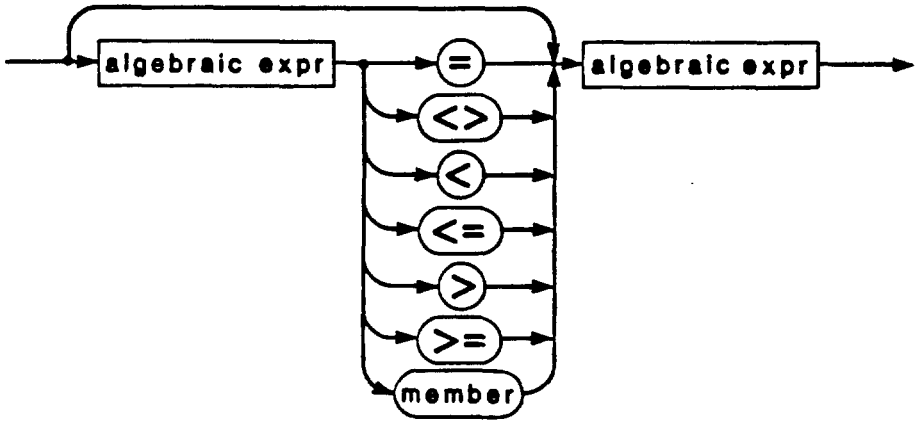
$$M\ [\![div]\!] = \lambda x.\ \lambda y.\ \max\{z : integer \mid z \le x / y\}$$

$$M\ [\![mod]\!] = \lambda x.\ \lambda y.\ x - y * (M\ [\![div]\!](x)(y))$$

7.4. Relational expressions

The relational operators are a set of binary infix operators producing values of type *Bool*. They include equality and disequality, which expect arguments of a compatible type (but not of a mapping type, nor of a type derived from a class), plus several inequalities which expect arguments of an arithmetic or an enumeration type. Another relational operator, the test for membership in a set, is defined in Section 7.9.

Syntax



7.4.1. Equality and disequality

Semantics:

$$value\_of \llbracket e_1 = e_2 \rrbracket(env)(val) =$$
$$(value\_of \llbracket e_1 \rrbracket(env)(val) = value\_of \llbracket e_2 \rrbracket(env)(val))$$

The expression e_1 <> e_2 is equivalent to **not** $(e_1 = e_2)$.

7.4.2. Inequalities

Semantics:

$$value\_of \llbracket e_1 <= e_2 \rrbracket(env)(val) =$$
$$M \llbracket <= \rrbracket (value\_of \llbracket e_1 \rrbracket(env)(val), value\_of \llbracket e_2 \rrbracket(env)(val))$$

where $M \llbracket <= \rrbracket = Y\, \lambda r.\, \lambda x.\, \lambda y.$ **if** $x = y$ **or** $r(succ\ x, y)$ **then** *true*
elseif $r(succ\ y, x)$ **then** *false* **else** $\perp$
fi

Other relational operators are defined in terms of "=", "<=", and the Boolean operators:

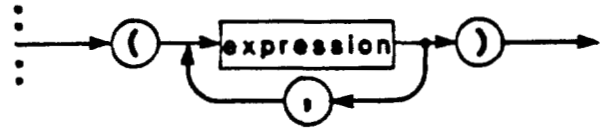
"$e_1 < e_2$" is equivalent to "$e_1 <= e_2$ **and not** $(e_1 = e_2)$"
"$e_1 >= e_2$" is equivalent to "**not** $(e_1 < e_2)$"
"$e_1 > e_2$" is equivalent to "**not** $(e_1 <= e_2)$"

7.5. Expressions involving product types

7.5.1. Value constructor

Syntax

simple expression:



Semantics:

If $a_1,\ldots a_n$ are expressions of types $A_1,\ldots A_n$ then $(a_1,\ldots a_n)$ is an expression of type $A_1 * \ldots * A_n$.

$value\_of\ [\![(a_1,\ldots a_n)]\!](env)(val) = (a_1',\ldots a_n')$

where $a_i' = value\_of\ [\![a_i]\!](env)(val)$ for $1 \le i \le n$

7.5.2. Operators

There is one operator on expressions of a product type, called projection. It takes an argument of a product type and a second argument which is a natural number.

Semantics:

The projection $e\,!i$, where e is an expression of type $A_1 * \ldots * A_n$ and $n > 1$ and i is a numeral satisfying $1 \le i \le n$, is defined by

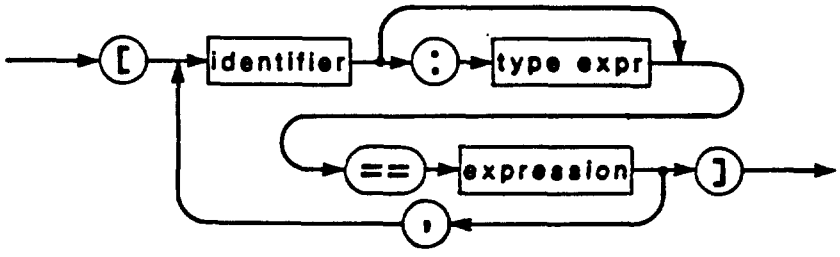
$value\_of\ [\![e\,!i]\!](env)(val) = (value\_of\ [\![e]\!](env)(val)) \downarrow i$

7.6. Expressions involving record types

7.6.1. Value constructor

Syntax

record constructor:



Semantics:

$$value\_of\ [\![[id_1 == e_1, \ldots\ id_n == e_n]]\!](env)(val) =$$

$$\sum_{j=1}^{n} [\text{"}id_i\text{"} : value\_of\ [\![e_j]\!](env)(val)]$$

7.6.2. Operators

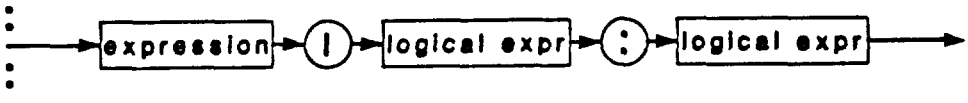
The operators defined on an expression of a record type are *field selection*, which yields the value of a field named by an identifier, and *supersedes*, which replaces the value of one of the fields.

Semantics:

$$value\_of\ [\![e.id]\!](env)(val) = value\_of\ [\![e]\!](env)(val)\ [\![id]\!]$$

Syntax

expression:



Semantics:

$$value\_of\ [\![r\ |id:e\ ]\!](env)(val) =$$

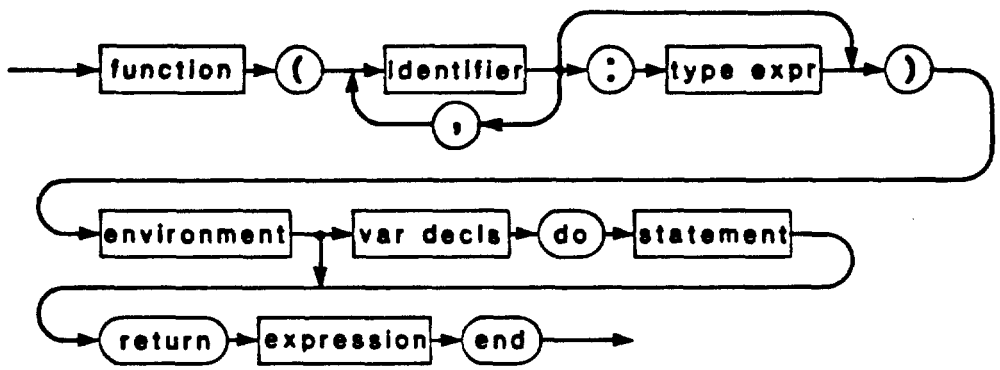
$$value\_of\ [\![r]\!](env)(val) \oplus [\text{"}id\text{"} : value\_of\ [\![e]\!](env)(val)]$$

7.7. Expressions involving mapping types

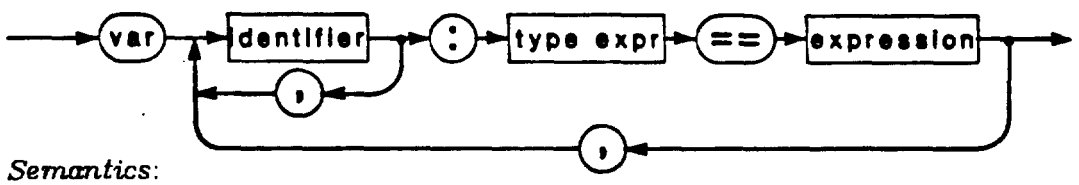
7.7.1. Value constructor

Syntax

function constructor:



var decls:



Semantics:

$value\_of$ ⟦ **function** $(i{:}t)\ e$; v **do** $stmt$ **return** $expr$ **end** ⟧$(env)(val)$ =

$$\lambda x.\ value\_of\ ⟦\ expr\ ⟧(env' \oplus_{Const} ["i" : (e\_type\ ⟦\ t\ ⟧(env),\ x)])(val')$$

where (env', val') = m_var ⟦ v ⟧$(m\_decl$ ⟦ e ⟧$(env))$

7.7.1.1. The meaning of a declaration of variables

m_var [**var** id : t == $expr$, <var decls>](env) =

$(e \oplus_{Var} ["id" : e\_type\ ⟦\ t\ ⟧(env)],$

$v \oplus ["id" : value\_of\ ⟦\ expr\ ⟧(env)([\,])])$

where (e, v) = **if** <var decls> is an empty string

then $(env, [\,])$

else m_var [<var decls>](env)

fi

The declaration

var $id_1, \ldots\ id_n$: t == $expr$

is equivalent to

var id_1 : t == $expr, \ldots\ id_n$: t == $expr$

7.7.2. Operators

Function application, function composition, and supersedes are the operations defined for mapping types. Function application is the only operation of the language that is not denoted by an explicit operator symbol. Although the argument in a function application may be set off by parentheses if desired, it is not necessary to do so.

Semantics:

Function application is defined by

$$value\_of\ [\![f\ \ a]\!](env)(val) =$$

$$value\_of\ [\![f]\!](env)(val)(value\_of\ [\![a]\!](env)(val))$$

If f and g are expressions typed as $f : A \text{->} B$ and $g : B \text{->} C$, then the composition of g with f is defined by

g @ f is equivalent to **function** $(x{:}A)$ **return** $g\,(f\,(x))$ **end**

If f, a and b are expressions typed as $f : A \text{->} B$, $a : A$, and $b : B$, then the following defines the expression read as "f superseded at a by b",

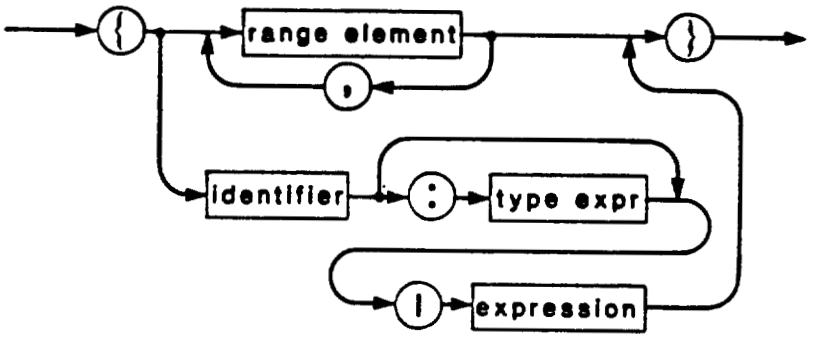
$f\,|a{:}b$ is equivalent to **function** $(x{:}A)$ **return if** $x = a$ **then** b **else** $f\ x$ **end**

7.8. Expressions involving powerset types

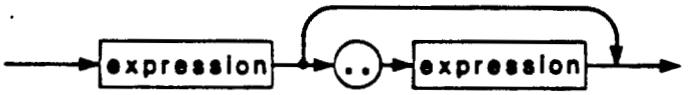
7.8.1. Value constructor

Syntax

powerset constructor:



range element:



If $e(x)$ is an expression of type *Bool* with only x free, then

$\{x:A \mid e(x)\}$

is an expression of type **set of** A.

Semantics:

$$value\_of \llbracket \{x:A \mid e\} \rrbracket(env)(val) =$$

$$\lambda y.\ value\_of \llbracket e \rrbracket(env \oplus_{Const} ["x" : (e\_type \llbracket A \rrbracket(env),\ y)])(val)$$

The meaning of a powerset expression is a function from values of the base type to Boolean values.

If $e_1, \ldots e_n$ are expressions of type A, then the expression

$\{e_1, \ldots e_n\}$ is equivalent to $\{x:A \mid x = e_1$ **or** $\ldots$ **or** $x = e_n\}$

7.8.2. Operators

if e_1 and e_2 are expressions of type **set of** A, and a is an expression of type A, then the following expressions are defined:

7.8.2.1. The operator *min* finds the least element in e which is not less than a.

Semantics:

$$value\_of \llbracket min(e, a) \rrbracket(env)(val) =$$

$$(Y\, \lambda m.\ \lambda e'.\ \lambda a'.\ \textbf{if } e'(a') \textbf{ then } a' \textbf{ else } m(e')(a'+1) \textbf{ fi})$$

$$(value\_of \llbracket e \rrbracket(env)(val))(value\_of \llbracket a \rrbracket(env)(val))$$

7.8.2.2. The operator *max*, which finds the greatest element in e which is not greater than a, is defined similarly.

7.8.2.3. union, intersection, and set difference

$$value\_of \llbracket e_1\ op\ e_2 \rrbracket(env)(val) = \lambda x.\ e_1'(x)\ op'\ e_2'(x)$$

where $e_i' = value\_of \llbracket e_i \rrbracket(env)(val)$

and where if *op is* "union" **then** *op'* is **or**,
if *op* is "intersection" **then** *op'* is **and**,
if *op* is "difference" **then** *op'* is **and not**

7.8.2.4. set membership

$$value\_of \llbracket e_1 \text{ member } e_2 \rrbracket(env)(val) =$$

$$value\_of \llbracket e_2 \rrbracket(env)(val)(value\_of \llbracket e_1 \rrbracket(env)(val))$$

7.8.2.5. The set map function The set map applies a function to each member of a set that satisfies a given predicate. The result is the set of values in the range of the function.

$$value\_of \llbracket \{e(x) \mid x \textbf{ in } S \textbf{ and } b(x)\} \rrbracket(env)(val) =$$
$$\{y \mid (\exists x)\ x \in value\_of \llbracket S \rrbracket(env)(val) \textbf{ and}$$
$$value\_of \llbracket b \rrbracket(env)(val)\, x = true \textbf{ and}$$
$$y = value\_of \llbracket e \rrbracket(env)(val)\, x\}$$

7.9. Expressions involving union types

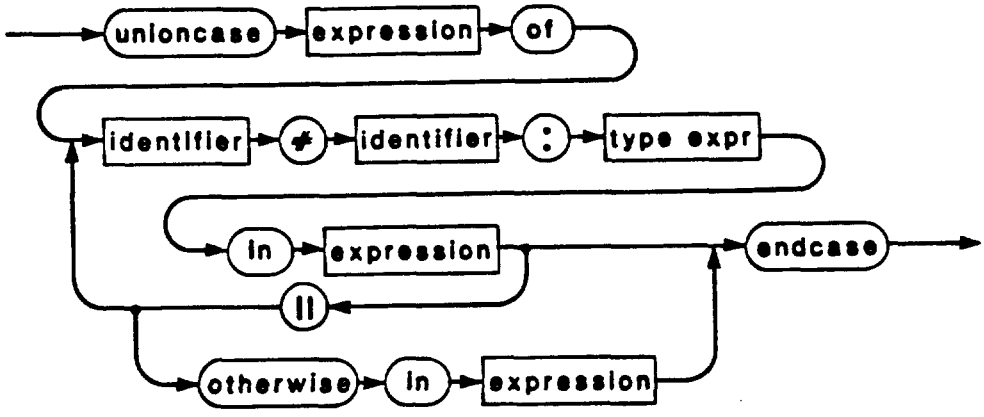
7.9.1. Value constructor

If e is an expression of type t, then the injection $id\#e$ is an expression of a union type $id_1\#t_1 + \ldots\ id_n\#t_n$ such that $id = id_i$ and $t = t_i$ for some $i \leq n$. The injection is defined by

$$value\_of \llbracket id\#e \rrbracket(env)(val) = (\text{"}id\text{"}, value\_of \llbracket e \rrbracket(env)(val))$$

7.9.2. The unioncase expression

Syntax



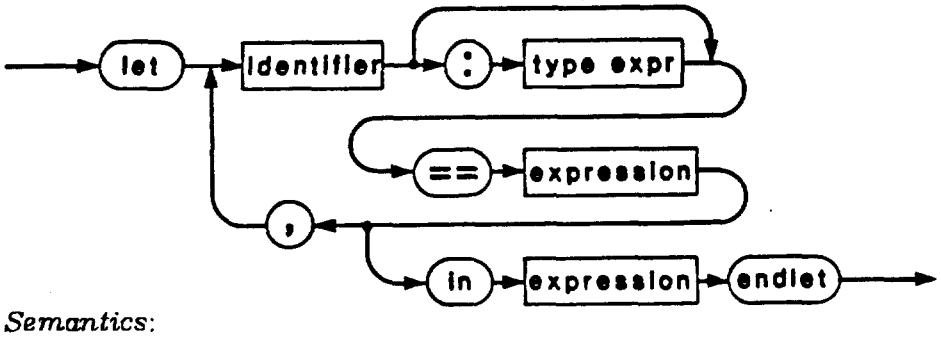
There is no operator (other than the value constructor) which yields a new value of a union type. Expressions of a union type can bve used in the unioncase expression which discriminates on the case tag. The meaning of this expression is defined by

$$
\begin{aligned}
&\mathit{value\_of}\ [\![\ \textbf{unioncase}\ e\ \textbf{of}\\
&\qquad\qquad i_1\#a_1\ \textbf{in}\ e_1\\
&\qquad |\,|\quad i_2\#a_2\ \textbf{in}\ e_2\\
&\qquad\qquad \vdots\\
&\qquad |\,|\quad i_n\#a_n\ \textbf{in}\ e_n\\
&\qquad |\,|\quad \textbf{others in}\ e_{n+1}\\
&\qquad \textbf{endcase}\ ]\!](\mathit{env})(\mathit{val}) =\\
&\textbf{if}\ e'\!\downarrow\!1 = i_1\ \textbf{then}\ e_1'\\
&\textbf{elseif}\ e'\!\downarrow\!1 = i_2\ \textbf{then}\ e_2'\\
&\qquad \vdots\\
&\textbf{elseif}\ e'\!\downarrow\!1 = i_n\ \textbf{then}\ e_n'\\
&\textbf{else}\ e_{n+1}'\ \textbf{fi}\\
&\textbf{where}\ e' = \mathit{value\_of}\ [\![\ e\ ]\!](\mathit{env})(\mathit{val}),\ \text{and}\\
&\qquad e_i' = \mathit{value\_of}\ [\![\ e_i\ ]\!](\mathit{env})(\mathit{val})\ \text{for}\ 1 \le i \le n.
\end{aligned}
$$

7.10. Expressions involving locally defined constants

Syntax

let expression:



Semantics:

$$value\_of\ [\![\ \textbf{let}\ id_1 : t_1 == e_1, \ldots\ id_n : t_n == e_n\ \textbf{in}\ expr\ \textbf{endlet}\ ]\!](env)(val) =$$

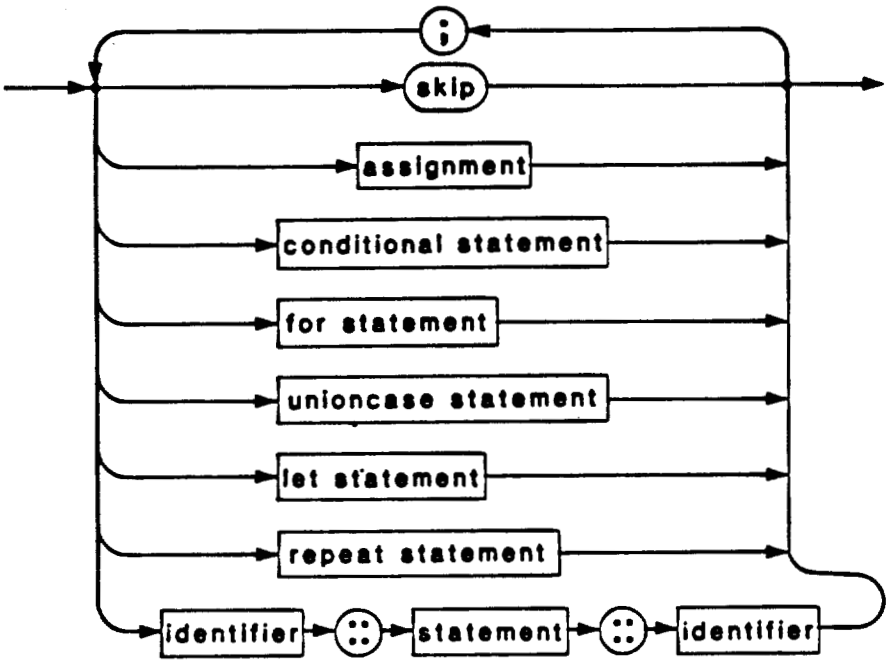
$$value\_of\ [\![\ expr\ ]\!](Y_{Env}\ \lambda e.\ env \oplus_{Const} \sum_{i=1}^{n} [\text{"}id\text{"} : (t_i', v_i)])(val)$$

where $t_i' = e\_type\ [\![\ t_i\ ]\!](env)$, and

$v_i = value\_of\ [\![\ e_i\ ]\!](e)(val)$

8. Statements

Syntax



8.1. The empty statement

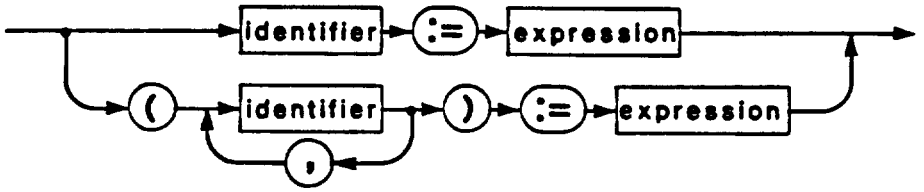
Syntax

Semantics:

$$eval\ [\![\ \textbf{skip}\ ]\!](env)(val) = val$$

8.2. Assignment

Syntax



Examples:

a := 5*x;

(day, month, year) := (31,12,79);

(x,y) := (y,x)

Semantics:

$$eval\ [\![id := e]\!](env)(val) = val \Leftarrow ["id" : value\_of\ [\![e]\!](env)(val)]$$

$$eval\ [\![(id_1, \ldots\ id_n) := e]\!](env)(val) =$$

$$val \Leftarrow \sum_{i=1}^{n} ["id_i" : value\_of\ [\![e]\!](env)(val))\downarrow i]$$

8.3. Sequential composition

Syntax

A small example

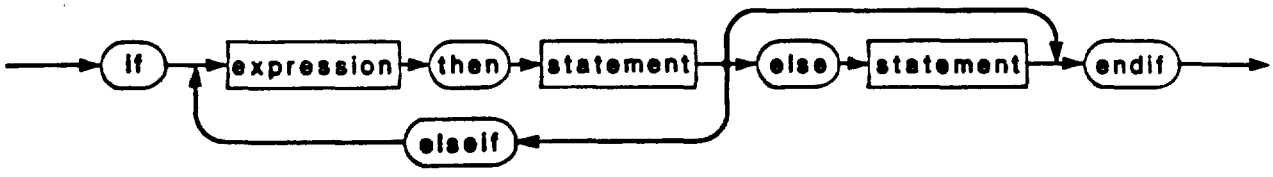
a := 5; b := a+b

Semantics:

$$eval\ [\![s_1;s_2]\!](env)(val) = eval\ [\![s_2]\!](env)(eval\ [\![s_1]\!](env)(val))$$

8.4. The conditional statement

Syntax



A small example

```
if a >= b and a >= c then m := a+b*c
elseif b >= c then m := b+a*c
else m := c+a*b
endif
```

Semantics:

$$eval\ [\![\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{endif}\ ]\!](env)(val) =$$
$$\textbf{if}\ value\_of\ [\![\ e\ ]\!](env)(val)\ \textbf{then}\ eval\ [\![\ s_1\ ]\!](env)(val)$$
$$\textbf{else}\ eval\ [\![\ s_2\ ]\!](env)(val)$$

The repeated conditional statement

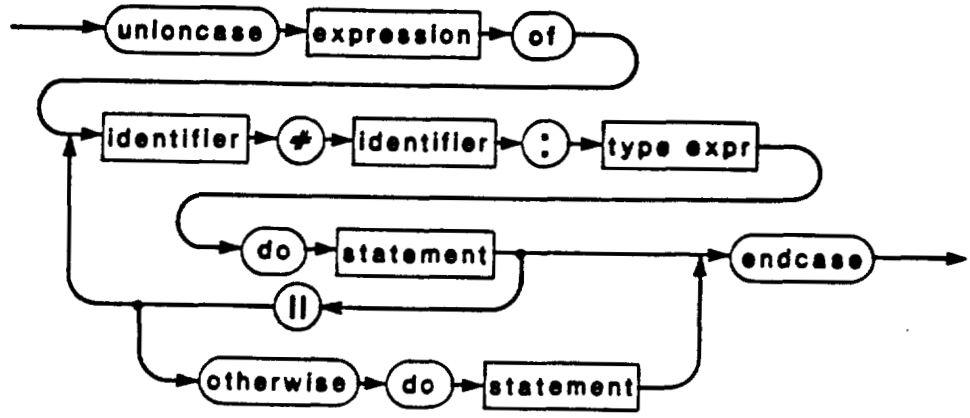
$$\textbf{if}\ e_1\ \textbf{then}\ s_1\ \textbf{elseif}\ e_2\ \textbf{then}\ s_2 \ldots \textbf{else}\ s_{n+1}\ \textbf{endif}$$

can be written as an abbreviation for the nested conditional statement

$$\textbf{if}\ e_1\ \textbf{then}\ s_1\ \textbf{else}$$
$$\textbf{if}\ e_2\ \textbf{then}\ s_2\ \textbf{else}$$
$$\vdots \qquad\qquad \textbf{lse}$$
$$s_{n+1}\ \textbf{endif}\ \cdots\ \textbf{endif endif}$$

8.5. The unioncase statement

Syntax



A small example:

```
unioncase S_list of
    Nullist#t   do Found_it := false
| | Atom#t      do Found_it := t = key
| | List#t      do if contains (key, front t)
                      then Found_it := true
                      else Found_it := contains (key, rest t)
                   endif
endcase
```

Semantics:

eval ⟦ **unioncase** *e* **of**

$id_1\#a_1$ **do** s_1

| | ...

| | $id_n\#a_n$ **do** s_n

| | **others do** s_{n+1}

endcase ⟧ (*env*)(*val*) =

if $e'\downarrow 1 = id_1$ **then** *eval* ⟦ s_1 ⟧ ($env \oplus_{Const}$ ["a_1": (*Bool*, $e'\downarrow 2$)])(*val*)

elseif $e'\downarrow n = id_n$ **then** *eval* ⟦ s_n ⟧ ($env \oplus_{Const}$ ["a_n": (*Bool*, $e'\downarrow 2$)])(*val*)

else *eval* ⟦ s_{n+1} ⟧ (*env*)(*val*)

fi

where e' = *value_of* ⟦ *e* ⟧ (*env*)(*val*)

Note: the definition of *eval* does not depend upon the type associated with a constant. Therefore the type *Bool* has arbitrarily been given to each constant a_i in providing an environment for evaluation of statement s_i.

8.6. The recursive statement

Syntax

An example:

```
(x,y,z) := (A,N,1);
Exponent::
    if y <= 0 then skip
    else if even y then
            (x,y) := (x*x, y div 2)
        else
            (y,z) := (y-1, x*z)
        endif;
        repeat Exponent
    endif
::Exponent
```

Semantics:

$$eval\ [\![id ::\text{stmt}:: id]\!](env)(val) =$$

$$Y_{Val \to Val}\ \lambda s.\ eval\ [\![\text{stmt}]\!](env)(val \oplus [\text{"}id\text{"} : s])$$

8.7. The repeat statement

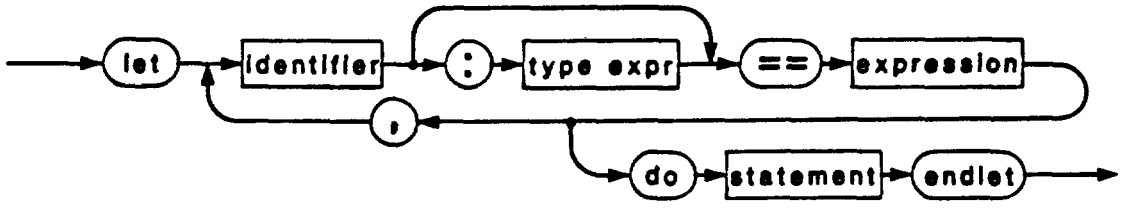
Syntax

Semantics:

$$eval\ [\![\textbf{repeat}\ id]\!](env)(val) = val\ [\![id]\!](val)$$

8.8. The let statement – local constant definition

Syntax



An example:

let x : float == position.NS, y : float == position.EW
in range := x*x + y*y;
azimuth := *arctan* y/x
endlet

Semantics:

$$eval\ [\![\ \textbf{let}\ id_1\,|\,t_1 == e_1, \ldots\ id_n\,|\,t_n == e_n\ \textbf{in}\ stmt\ \textbf{endlet}\ ]\!](env)(val)$$

$$eval\ [\![\ stmt\ ]\!]\,(Y_{Env}\ \lambda e.\ env \oplus_{Const} \sum_{i=1}^{n} [\text{"}id_i\text{"} : (t_i', e_i')])(val)$$

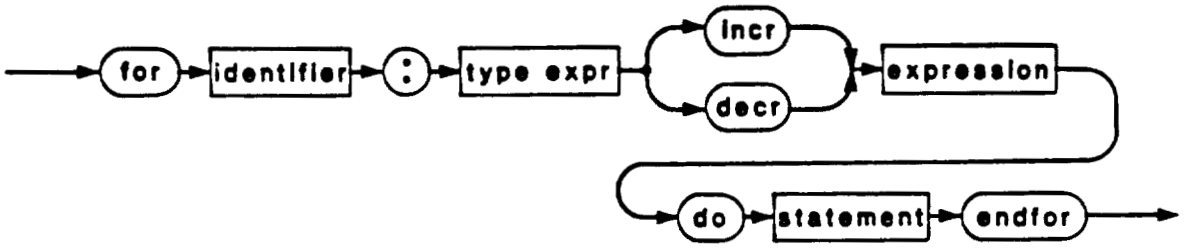
where $t_i' = e\_type\ [\![\ t_i\ ]\!]$, and

$e_i' = value\_of\ [\![\ e_i\ ]\!](e)(val)$

8.9. Bounded iteration

Syntax

for statement:



Semantics:

The statement

for x : t **incr** *setexpr* **from** p **in** *stmt* **endfor**

is equivalent to

let *setconst* : **set of** t == *setexpr* **in**
  p' :=p;
  for::
    if $\{y : t \mid y >= p'$ **and** y **member** *setconst*$\}$ = $\{\}$ **then**
      skip
    else
      let $x : t$ == *min*(*setconst*, p') **in**
        stmt: p' := *succ* x; **repeat** for
      endlet
    endif
  ::*for*
endlet

where p' is a new variable not previously declared in the environment.

The meaning of bounded iteration with the **decr** verb is similarly defined, using *max* in place of *min*, and <= in place of >=.

9. Type checking

9.1. The notion of type-correctness in Apple

Types are used in Apple to distinguish potentially meaningful expressions from those which can have no meaning. For each operator in the language there are restrictions on the types of its operands. In Apple these restrictions have been defined such that it is decidable, by static analysis of a program text, whether or not the restrictions are met. This process is called type-checking.

Informally, a type A can be thought of as the set of all type A values. For a variety of reasons, it is infeasible to use this interpretation to give the mathematical meanings of types in Apple. The most important of these reasons is that we wish type-correctness of a program to be property decidable by a static analysis of the program text. Thus, we are willing to give a less precise meaning to a type than that it characterises a set of values that might be assumed by an expression having that type.

The function

$$e\_type : \mathbf{T\_expr} \rightarrow \mathbf{Env} \rightarrow \mathbf{T}$$

which was defined in Sec. 5, takes a well-formed type expression (or an enumeration, or a class instance) to its abstract syntactic representation, in an environment.

Every well-formed expression in Apple can be given a type, relative to an environment, provided that its component subexpressions (if any) are type-consistent with its embedded operators. Otherwise, it will be given the type ? which is a distinguished element in the domain **T**. That is, ? is the denotation for an ambiguous type.

A very useful notion for the types of Apple is that of a partial order among types, which we call *subtype*. This partial order is not absolute, but is relative to an environment in which types can be named. When the element ? in **T** is added to the partial order as a maximal element, it is possible to define a least upper bound for any pair of types within **T**. To determine the types of expressions, the partial order among types, and a least upper bound of a pair of types we shall define three functions,

$$type\_of : \mathbf{Expr} \rightarrow \mathbf{Env} \rightarrow \mathbf{T}$$

$$subtype : \mathbf{T} \times \mathbf{T} \rightarrow \mathbf{Env} \rightarrow \mathbf{Bool}$$

$$lub\_type : \mathbf{T} \times \mathbf{T} \rightarrow \mathbf{Env} \rightarrow \mathbf{T}$$

where $type\_of(e)(env)$ checks whether an expression e is internally type-consistent, and computes a type for it, with respect to a given environment; $subtype(t_1, t_2)(env)$ checks whether t_1 is a subtype of t_2 in the environment env, and $lub\_type(t_1, t_2)(env)$ computes a type which is a least upper bound of t_1 and t_2 in the environment env.

A function constructor expression may contain declarations and statements, in which expressions may occur. Declarations and statements will be inspected for type-correctness by the functions

$$pcheck : \mathbf{Stmt} \rightarrow \mathbf{Env} \rightarrow 2^{|\mathbf{Id}|} \rightarrow \mathbf{Bool}$$

$$dcheck : \mathbf{Decl} \rightarrow \mathbf{Env} \rightarrow \mathbf{Bool}$$

9.2. A partial ordering of types

The function *subtype* : **T** × **T** → **Env** → **Bool**, which compares pairs of types, induces a reflexive partial order upon the type domain **T** in any environment. This function is defined by the following rules:

0) $subtype(t, ?)(env)$ = true

1) $subtype(t, t)(env)$ = true

2) $subtype$(integer, float)(env) = true

3) $subtype$(restriction(t, p), t)(env) = true

4) $subtype(id, t)(env) = subtype(env.Type\ [\![id\ ]\!]\downarrow 1, t)(env)$

5) $subtype$(map(d_1, r_1), map(d_2, r_2))(env) =

$$subtype(r_1, r_2)(env)\ \textbf{and}\ subtype(d_2, d_1)(env)$$

6) $subtype$(record($(i_1, t_1), \ldots (i_n, t_n)$), record($(i_1, t_1'), \ldots (i_n, t_n')$))$(env)$ =

$$\bigwedge_{k=1}^{n} subtype(t_k, t_k')(env)$$

7) if $h : 1..m \to 1..n$ is injective and monotonic (with respect to the natural ordering of the natural numbers), then

$subtype$(union($(i_{h_1}, t_{h_1}), \ldots (i_{h_m}, t_{h_m})$), union($(i_1, t_1'), \ldots (i_n, t_n')$))$(env)$ =

$$\bigwedge_{k=1}^{m} i_{h_k} = i_k\ \textbf{and}\ subtype(t_{h_k}, t_k')$$

8) if $op(t_1, \ldots t_n)$ is an expression belonging to **T**, and *op* is "set" or "product", then

$$subtype(op(t_1, \ldots t_n), op(t_1', \ldots t_n'))(env) = \bigwedge_{i=1}^{n} subtype(t_i, t_i')(env)$$

9) if t_1 and t_2 are types and $subtype(t_1, t_2)(env)$ is not defined by rules (0..8) then $subtype(t_1, t_2)(env)$ = false.

9.2.1. Least upper bounds of types

In order that type-determination shall be as precise as possible, we want the domain **T** to contain a least upper bound of any two types, with respect to the partial order induced by the subtype function. To guarantee that an upper bound in **T** always exists, the element ? has been defined to be above any type in the subtype relation.

The function which yields a least upper bound of two types is

$$lub\_type : \mathbf{T} \times \mathbf{T} \rightarrow \mathbf{Env} \rightarrow \mathbf{T}$$

Here and in the following sections, we shall use the notation $a < b$ to denote $subtype(a,b)(env)$, where a and b are types.

$lub\_type\ (a,b)(env) =$

if $a < b$ **then** b

elseif $b < a$ **then** a

elseif either of a or b is "integer", "float", "bool", "ascii" or enumeration$(i_1,\ldots i_n)$ **then** ?

elseif a is an identifier **then** $lub\_type\ (env.Type\ (a), b)(env)$

elseif b is an identifier **then** $lub\_type\ (env.Type\ (b), a)(env)$

elseif a is restriction(t_a, p_a) **and** b is restriction(t_b, p_b)

then $lub\_type\ (t_a, t_b)(env)$

elseif a is record$((i_1, t_1),\ldots (i_n, t_n))$ **and** b is record$((i_1, t_1'),\ldots (i_n, t_n'))$

then if $\bigwedge_{k=1}^{n} s_k \neq ?$

then record$((i_1, s_1),\ldots(i_n, s_n))$

else ?

fi

where $s_k = lub\_type\ (t_k, t_k')(env)$

elseif a is union$((i_1,t_1),\ldots (i_m,t_m))$ **and** b is union$((j_1,t_1'),\ldots (j_n,t_n'))$

then if $m \leq p$ **and** $n \leq p$

and $g : 1..m \rightarrow 1..p$ and $h : 1..n \rightarrow 1..p$ are injective and monotonic

and for each $x \in 1..p$ either

i) $i_{g^{-1}(x)} = k_x$ **and** $s_x = t_{g^{-1}(x)}$ **and** $j_{h^{-1}(x)}$ is undefined, or

ii) $j_{h^{-1}(x)} = k_x$ **and** $s_x = t_{h^{-1}(x)}$ **and** $i_{g^{-1}(x)}$ is undefined, or

iii) $i_{g^{-1}(x)} = k_x$ **and** $j_{h^{-1}(x)} = k_x$ **and**

$s_x = lub\_type\ (t_{g^{-1}(x)}, t_{h^{-1}(x)}')(env)$ **and** $s_x \neq ?$

then union$((k_1,s_1),\ldots (k_p,s_p))$

else ?

elseif a = product$(t_1,\ldots t_n)$ **and** b = product$(t_1',\ldots t_n')$

then if $\bigwedge_{k=1}^{n} s_k \neq ?$

then product$(s_1,\ldots s_n)$

else ?

fi

where s_k = *lub_type* $(t_k, t_k')(env)$

elseif a = set$(t_1,\ldots t_n)$ **and** b = set$(t_1',\ldots t_n')$

then if $\bigwedge_{k=1}^{n} s_k \neq ?$

then set$(s_1,\ldots s_n)$

else ?

fi

where s_k = *lub_type* $(t_k, t_k')(env)$

else ?

fi

The least upper bound determined by *lub_type* is used to define a type in which a pair of values can be compared by a relational operator. Note that *lub_type* does not define upper bounds other than "?" for all kinds of types. In particular, if either a or b is a mapping or a class type, then *lub_type* (a, b) is "?". Relational operators are not defined for these kinds of types.

9.3. Type determination

The type of an expression is determined by the function

type_of : **Expr** × **Env** → **T**

by structural analysis of its first argument.

In the definition of *type_of*, it will be convenient to use a function which obtains the type expression underlying the definition of a named type or a restriction. This function is

basetype : **T** → **Env** → **T** =

$\lambda t.\ \lambda env.$ **cases** t **is**

"integer", "float", "bool", "ascii" ⇒ t

identifier ⇒ *basetype* $(env.Type(t)\downarrow 1)(env)$

restriction(a, p) ⇒ *basetype* $(a)(env)$

otherwise ⇒ t

esac

The type determination function is

type_of $(expr)(env)$ =

cases *expr* **is**

numeral ⇒ "integer"

floating point numeral ⇒ "float"

quoted character ⇒ "ascii"

"true", "false" ⇒ "bool"

```
identifier  ⇒ if env.Const [ expr ] ≠ ? then env.Const [ expr ]↓1
                                       else env.Var [ expr ]↓1 fi

a op b  ⇒ let lub = lub_type (type_of [ a ](env), type_of [ b ](env))(env) in
            cases op is
            "+","−","*"  ⇒ if lub < "integer" then "integer"
                           elseif lub < "float" then "float"
                           elseif lub = set(t) then set(t)
                           else ?
                           fi

            "/"          ⇒ if lub < "float" then "float"
                           else ? fi

            "div", "mod" ⇒ if lub < "integer" then "integer"
                           else ? fi

            "and", "or"  ⇒ if lub < "bool" then "bool"
                           else ? fi
            "member"     ⇒ if set(type_of [ a ](env)) < type_of [ b ](env)
                           then "bool" else ? fi

            "=", "<>"    ⇒ if lub = ? then ?
                           else "bool" fi
            "<", "<=", ">", ">="  ⇒
                           if basetype (lub) = "integer" or
                              basetype (lub)(env) = "ascii" or
                              basetype (lub)(env) = enumeration(i1,... in)
                           then "bool"
                           else ? fi

            "@"    ⇒ if basetype (type_of [ a ](env))(env) = map(d2,r2) and
                       basetype (type_of [ b ](env))(env) = map(d1,r1)
                       and r1 < d2
                       then map(d1,r2)
                     else ? fi
            esac
          endlet
```

op a ⇒ **cases** *op* **is**

"pred", "succ" ⇒ **let** *b* = *basetype* (*type_of* ⟦ *a* ⟧(*env*))(*env*) **in**
if *b* = "integer" **then** "integer"
elseif *b* = "ascii" **then** "ascii"
elseif *b* = enumeration(i_1,... i_n) **then** *b*
else ? **fi**
endlet

"–" ⇒ **if** *type_of* ⟦ *a* ⟧(*env*) < "integer" **then** "integer"
elseif *type_of* ⟦ *a* ⟧(*env*) < "float" **then** "float"
else ? **fi**

"not" ⇒ **if** *type_of* ⟦ *a* ⟧(*env*) < "bool" **then** "bool"
else ? **fi**

esac

e ! *m* (where *m* is a numeral) ⇒
if *basetype* (*type_of* ⟦ *e* ⟧(*env*))(*env*) = product(t_1,... t_n) **and** 0 < *m* < *n*
then t_m **else** ? **fi**

e.*id* (where *id* is an identifier) ⇒
if *basetype* (*type_of* ⟦ *e* ⟧(*env*))(*env*) = record((i_1,t_1),... (i_n,t_n)) **and**
id = i_m for some *m* in 1..*n*
then t_m **else** ? **fi**

f a ⇒ **if** *basetype* (*type_of* ⟦ *f* ⟧(*env*))(*env*) = map(*d*,*r*)
and *type_of* ⟦*a* ⟧(*env*) < *d*
then *r* **else** ? **fi**

f ⟦*a*:*b* ⇒ **if** *basetype* (*type_of* ⟦ *f* ⟧(*env*))(*env*) = map(*d*,*r*)
and *type_of* ⟦*a* ⟧(*env*) < *d*
and *type_of* ⟦ *b* ⟧(*env*) < *r*
hen *type_of* ⟦ *f* ⟧(*env*)
lseif *basetype* (*type_of* ⟦ *f* ⟧(*env*))(*env*) =
ecord((a_1:t_1),... (a_n:t_n))
and ∃ *j* in 1..*n* (*a* = a_j and *type_of* ⟦ *b* ⟧(*env*) < t_j)
then *type_of* ⟦ *f* ⟧(*env*) **else** ? **fi**

if b **then** e_1 **else** e_2 **endif** $\Rightarrow$

if *type_of* $[\![b]\!](env) <$ "bool"
then *lub_type* (*type_of* $[\![e_1]\!](env)$,*type_of* $[\![e_2]\!](env)$) (env)
else ? **fi**

let $x_1{:}t_1 == e_1, \ldots x_n{:}t_n == e_n$ **in** e **endlet** $\Rightarrow$

if *dcheck* $[\![$ **const** $x_1{:}t_1 == e_1, \ldots x_n{:}t_n \equiv e_n]\!](env)$
then *type_of* (e)(*m_decl* $[\![$ **const** $x_1{:}t_1 == e_1, \ldots x_n{:}t_n \equiv e_n]\!](env)$)
else ? **fi**

$x{:}t \mid p$ $\Rightarrow$ **if** *type_of* $[\![p]\!](env \oplus_{Const} [x{:}(e\_type [\![t]\!], ?)])$ < "bool"
then set(t) **else** ? fi

$(e_1, \ldots e_n)$ $\Rightarrow$ **if** $\bigwedge_{i=1}^{n}$ *type_of* $[\![e_1]\!](env) \neq ?$

then product(*type_of* $[\![e_1]\!](env), \ldots$ *type_of* $[\![e_n]\!](env)$)

else ? **fi**

$[a_1{:}e_1, \ldots a_n{:}e_n]$ where $a_1, \ldots a_n$ are identifiers $\Rightarrow$
if ($a_i \neq a_j$ when $i \neq j$ for i, j in $1..n$)
and $\bigwedge_{k=1}^{n}$ *type_of* $[\![e_k]\!](env) \neq ?$
then record($(a_1$,*type_of* $[\![e_1]\!](env)), \ldots (a_n$,*type_of* $[\![e_n]\!](env))$)
else ? **fi**

$a\#b$ where a is an identifier $\Rightarrow$
if *type_of* $[\![b]\!](env) \neq ?$ **then** union($(a$,*type_of* $[\![b]\!](env))$) **else** ? **fi**

function $(x{:}t)$ <env decl> <var decls> **do** <stmt> **return** e **end** $\Rightarrow$
if *dcheck* $[\![$ **const** $x ==$ *initial* (t) $]\!](env)$ **and**
dcheck $[\![$<env decl>$]\!](e_1)$ **and**
dcheck $[\![$<var decls>$]\!](e_2)$ **and**
pcheck $[\![$<stmt>$]\!](e_3)(\{\})$ **and**
type_of $[\![e]\!](env) \neq ?$
then map(*e_type* $[\![t]\!](env)$, *type_of* $[\![e]\!](e_3)$)
else ? **fi**
where $e_1 =$ *env* $\oplus_{Const} [x{:}(e\_type [\![t]\!](env), ?)]$
$e_2 =$ *m_decl* $[\![$<env decl>$]\!](e_1)$
$e_3 =$ *m_var* $[\![$<var decls>$]\!](e_2)\downarrow 1$

unioncase e **of** $i_1\#\ a_1 : t_1$ **in** e_1 || $\cdots$ || $i_n\#\ a_n : t_n$ **in** e_n **endcase** $\Rightarrow$

if *type_of* $[\![e]\!](env) <$ union$((i_1, t_1), \ldots (i_n, t_n))$

then if $\bigwedge_{j=1}^{n}$ *dcheck* $[\![$ **const** $a_j : t_j$ == *initial* $(t_j)]\!](env)$

then $\mathrm{Lub}_{j=1}^{n}$ $(type\_of\ [\![e_j]\!](env \oplus_{Const} [\text{"}a_j\text{"} : (t_j, ?)])$

else ? **fi**

else ? **fi**

where $\mathrm{Lub}_{j=1}^{n}$ (t_j) = **if** $n = 1$ **then** t_1

else *lub_type* $(\mathrm{Lub}_{j=1}^{n-1}(t_j), t_n)$ **fi**

esac

9.4. The type-conversion function

The meaning of the type-conversion function *cnv* depends upon the type expected in the context of its occurrence. It is defined by cases on the abstract syntax of an expected type. Thus there is a conversion function defined for every type; the function that attempts to convert it argument into a value of type t is called t's *cnv*. In practice, a programmer will be allowed to omit the prefix t's whenever an expected type can be inferred from the context.

value_of ⟦*t's cnv expr*⟧(*env*)(*val*) =

convert (*e_type* ⟦*t*⟧)(*env*)(*value_of* ⟦*expr*⟧(*env*)(*val*))

where *convert* ∈ **T** → **Env** → **O** → **O**

= λ*t*. λ*env*. λ*v*. **cases** *t* **is**

"float", "integer", "bool", "ascii", enumeration(id_1,... id_n) ⇒ v

identifier ⇒ *convert* (*env.Type*(*t*))(*env*)(*v*)

restriction(A,p) ⇒ **if** *value_of* ⟦p v⟧(e)([ ])
then v **else** ? **fi**

product(t_1,... t_n) ⇒ (*convert* (t_1)(*env*)(v↓1),... *convert* (t_n)(*env*)(v↓n))

record(("i_1", t_1),... ("i_n", t_n)) ⇒
[i_1 : *convert* (t_1)(*env*)(v.i_1),... i_n : *convert* (t_n)(*env*)(v.i_n)]

union(("i_1", t_1),...("i_n", t_n)) ⇒
cases v↓1 **is**
i_1 ⇒ i_1#*convert* (t_1)(*env*)(v↓2)
⋮
i_n ⇒ i_n#*convert* (t_n)(*env*)(v↓2)
esac

map(d,r) ⇒ *convert* (r)(*env*) @ v @ (*convert* (d)(*env*))

set (t') $\Rightarrow \lambda x.$ if not *predicate_of* $(t')(env)(x)$ **then** ? **else** $v(x)$ **fi**

where *predicate_of* $(t)(env)$ =

cases t **is**

"integer", "bool", "ascii" $\Rightarrow \lambda x.\ x$

enumeration$(id_1, \ldots id_n)$ $\Rightarrow \lambda x.\ x$

restriction(s, p) $\Rightarrow \lambda \xi.\ p$

identifier $\Rightarrow$ *predicate_of* $(env.Type(t))(env)$

otherwise $\Rightarrow$?

esac

esac

9.5. Type checking an environment

Type-checking the text of an environment consists in type-checking the declarations that constitute the environment. These declarations are checked relative to the meaning of the environment itself.

dcheck ⟦ **env** *id* [<import decl><environment>] ⟧(env) =

dcheck ⟦<environment>⟧

(m_decl ⟦ **env** *id* [<import decl><environment>] ⟧(env))

9.5.1. Type checking of declarations

dcheck ⟦ **const** $c_1 : t_1 == e_1, \ldots c_n : t_n == e_n$ ⟧(env) =

$\bigwedge_{i=1}^{n}$ *dcheck* ⟦ t_i ⟧(env) **and**

(*type_of* ⟦ e_i ⟧(env) < *e_type* ⟦ t_i ⟧(env))

dcheck ⟦ **var** $v_1 : t_1 == e_1, \ldots v_n : t_n == e_n$ ⟧(env) =

$\bigwedge_{i=1}^{n}$ (*dcheck* ⟦ t_i ⟧(env) **and** *type_of* ⟦ e_i ⟧(env) < *e_type* ⟦ t_i ⟧(env))

dcheck ⟦ **type** $t_1 : s_1, \ldots t_n : s_n$ ⟧(env) =

$\bigwedge_{i=1}^{n}$ (*dcheck* ⟦ s_i ⟧(env)

dcheck ⟦ **class** C **of type** $t_1, \ldots$ **const** $c_1 : s_1, \ldots$ ==

def $k_1 : r_1, \ldots k_n : r_n$ **rep** <environment> **endclass**⟧(env) =

dcheck ⟦ **const** $c_1 : s_1, \ldots$ ⟧(e_1) **and**

dcheck ⟦ **const** $k_1 : r_1, \ldots k_n : r_n$ ⟧(e_2) **and**

dcheck ⟦ <environment> ⟧(e_2) **and**

$$\bigwedge_{i=1}^{n} subtype\ (e_3[\![k_i]\!],\ e_4.Const\ [\![k_i]\!]\downarrow 1)(e_2)$$

where $e_1 = env \oplus_{Type} \sum_i [\text{"}t_i\text{"} : \text{"}t_i\text{"}]$

$e_2 = m\_decl\ [\![$ **const** $c_1 : s_1 == \text{"}c_1\text{"}, \ldots]\!](e_1)$

$e_3 = \sum_i [k_i : e\_type\ [\![r_i]\!](e_2)]$

$e_4 = m\_decl\ [\![$ <environment> $]\!](e_2)$

In the definition above, e_2 represents the environment with respect to which declarations in the body of the class are interpreted. For the purpose of type-checking the declarations, values of the constant parameters $c_1, \ldots c_m$ are taken to be the identifiers "c_1",..."c_m" themselves. e_3 gives the types of all exported operators, as bound in the **def** list.

Declaration checking applies also to type expressions and instances of classes:

$dcheck\ (t)(env) =$

cases t **is**

"**set of** s" $\Rightarrow$ $is\_scalar\ [\![s]\!](env)$

"C **of** [**type** $i_1 : t_1, \ldots i_m : t_m$ **const** $c_1 : s_1 == e_1, \ldots c_n : s_n == e_n$]" $\Rightarrow$

$dcheck\ [\![$ **type** $i_1 : t_1, \ldots i_m : t_m$

const $c_1 : s_1 == e_1, \ldots c_n : s_n == e_n\ ]\!](env)$ **and**

$\bigwedge_{j=1}^{n} i_j \in d.C\downarrow 1$ **and**

$\bigwedge_{j=1}^{n} subtype\ (e\_type\ [\![s_j]\!](env'),\ (d.C\downarrow 2)\ [\![c_j]\!])(env')$

where $d = env.Class\ [\![C]\!](env')$, and

$env' = m\_decl\ [\![$ **type** $i_1 : t_1, \ldots i_m : t_m$

const $c_1 : s_1 == e_1, \ldots c_n : s_n == e_n\ ]\!](env)$

otherwise $\Rightarrow$ *true*

esac

where $is\_scalar\ (t)(env) =$ **cases** t **is**

"*id*" $\Rightarrow$ $is\_scalar\ (env.Type\ [\![id]\!])(env)$

restriction(s, p) $\Rightarrow$ $is\_scalar\ (s)(env)$

enumeration("id_1",... "id_n") $\Rightarrow$ *true*

"integer", "bool", "ascii" $\Rightarrow$ *true*

otherwise $\Rightarrow$ *false*

esac

9.6. Type checking of statements

Here we use the notation e' to stand for $type\_of\,(e)(env)$, where e is an expression.

$$pcheck\ [\![\ \textbf{skip}\ ]\!]\,(env)(\Lambda) = true$$

$$pcheck\ [\![\ id := e\ ]\!]\,(env)(\Lambda) = e' < env.Var\ [\![\ id\ ]\!])$$

$$pcheck\ [\![\ (id_1,\ldots\ id_n) := e\ ]\!]\,(env)(\Lambda) =$$
$$e' < \text{product}(env.Var\ [\![\ id_1\ ]\!],\ldots\ env.Var\ [\![\ id_n\ ]\!])$$

$$pcheck\ [\![\ s_1;s_2\ ]\!]\,(env)(\Lambda) = pcheck\ [\![\ s_1\ ]\!]\,(env)(\Lambda)\ \textbf{and}\ pcheck\ [\![\ s_2\ ]\!]\,(env)(\Lambda)$$

$$pcheck\ [\![\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{endif}\ ]\!]\,(env)(\Lambda) =$$
$$e' < Bool\ \textbf{and}$$
$$pcheck\ [\![\ s_1\ ]\!]\,(env)(\Lambda)\ \textbf{and}\ pcheck\ [\![\ s_2\ ]\!]\,(env)(\Lambda)$$

$$pcheck\ [\![\ \textbf{unioncase}\ e\ \textbf{of}\ id_1\#c_1 : t_1\ \textbf{do}\ s_1\ ||\ \ldots$$
$$||\ \textbf{otherwise do}\ s_{n+1}\ \textbf{endcase}\ ]\!]\,(env)(\Lambda) =$$
$$e' < \text{union}((\text{"}id_1\text{"}, e\_type\ [\![\ t_1\ ]\!]),\ldots\ (\text{"}id_n\text{"}, e\_type\ [\![\ t_n\ ]\!]))\ \textbf{and}$$
$$\bigwedge_{j=1}^{n} pcheck\ [\![\ s_i\ ]\!]\,(env \oplus_{Const} [\text{"}c_j\text{"} : e\_type\ [\![\ t_n\ ]\!], ?)])(env)(\Lambda)$$
$$\textbf{and}\ pcheck\ [\![\ s_{n+1}\ ]\!]\,(env)(\Lambda)$$

$$pcheck\ [\![\ \textbf{for}\ id : t\ \textbf{incr}\ e_1\ \textbf{from}\ e_2\ \textbf{do}\ s\ \textbf{endfor}\ ]\!]\,(env)(\Lambda) =$$
$$type\_of\ [\![\ e_1'\ ]\!]\,(env) < \text{set}(e\_type\ [\![\ t\ ]\!])\ \textbf{and}$$
$$type\_of\ [\![\ e_2'\ ]\!]\,(env) < e\_type\ [\![\ t\ ]\!]\ \textbf{and}$$
$$pcheck\ [\![\ s\ ]\!]\,(env \oplus_{Const} [\text{"}id\text{"} : (e\_type\ [\![\ t\ ]\!], ?)])(\Lambda)$$

$$pcheck\ [\![\ id_1 :: s :: id_2\ ]\!]\,(env)(\Lambda) = (id_1 = id_2)\ \textbf{and}\ pcheck\ [\![\ s\ ]\!]\,(env)(\Lambda \cup \{id_1\})$$

$$pcheck\ [\![\ \textbf{repeat}\ id\ ]\!]\,(env)(\Lambda) = \text{"}id\text{"} \in \Lambda$$

$$pcheck\ [\![\ \textbf{let}\ id_1 : t_1 == e_1,\ldots\ id_n : t_n == e_n\ \textbf{do}\ s\ \textbf{endlet}\ ]\!]\,(env)(\Lambda) =$$
$$dcheck\ [\![\ \textbf{const}\ id_1 : t_1 == e_1,\ldots\ id_n : t_n == e_n\ ]\!]\,(env)\ \textbf{and}$$
$$pcheck\ [\![\ s\ ]\!]\,(env')(\Lambda)$$

$$\textbf{where}\ env' = m\_decl\ [\![\ \textbf{const}\ id_1 : t_1 == e_1,\ldots\ id_n : t_n == e_n\ ]\!]\,(env)$$

Acknowledgements

We wish to thank Jon Shultis and John Givler for reading and commenting on earlier versions of the formal definition. Particular thanks are due Gene Rollins for his detailed criticism of the definition as he made use of it in the first implementation of Apple.

References

[KN81]
Kieburtz, R.B. and Nordström, B., The design of a language for modular programs, Tech. Rept. CS/E-82-01, Dept. of Computer Science and Engr., Oregon Graduate Center, Beaverton, Oregon, March, 1982.

[McS82]
MacQueen, D.B. and Sethi, R., A semantic model of types for applicative languages, *Proc. of 1982 ACM Conf. on LISP and Functional Programming*, ACM, New York, August, 1982, 243-252.

[Mi78]
Milner, R., A theory of type polymorphism in programming, *Jour. Computer and System Science* **17**, 3 (December, 1978), 248-375.

[SS71]
Scott, D.S. and Strachey, C., Toward a mathematical semantics for computer languages, *Proc. of Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn Press, New York, 1971, 19-46.

[vW75]
van Wijngarten, A., *et al.*, Revised report on the algorithmic language Algol 68, *Acta Informatica* **5**, 1-3 (1975), 1-236.