May 1982

cse-82-002

Richard B. Kieburtz

Bengt Nordstram

Follow this and additional works at: http://digitalcommons.ohsu.edu/csetech

Recommended Citation
http://digitalcommons.ohsu.edu/csetech/121

This Article is brought to you for free and open access by OHSU Digital Commons. It has been accepted for inclusion in CSETech by an authorized administrator of OHSU Digital Commons. For more information, please contact champieu@ohsu.edu.
The Formal Definition
of the Programming Language Apple

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 classes, 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

1The 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.

Copyright (C) 1982 R. B. Kieburtz and B. Nordström
1. Introduction

Apple has been designed to be a simple, yet powerful programming language. It is basically an applicative 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 [KNB1] 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.

\[
\text{env display}
\begin{array}{l}
\text{const} \ erase \ : \ \text{ascii} = \ ' \\
\text{class} \ screen = : \\
\end{array}
\]

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 implicitly 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 reference 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 classes (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 defines 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 defines 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 pro-
gram 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 con-
structor.

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 environ-
ments 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 state-
ment 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

- \( \bot \) denotes the undefined element (called "bottom") in any domain.
- "?' denotes a distinguished value (in any domain) called the error element. This value differs from \( \bot \), 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 "?' except the updating functions and \( @3 \) defined below.

- if \( b \) then \( e_1 \) else \( e_2 \) fi will denote the biconditional which maps \((\text{true}, e_1, e_2)\) to \( e_1 \), \((\text{false}, e_1, e_2)\) to \( e_2 \), \((\text{true}, e_1, ?)\) to \( e_1 \), and \((\text{false}, e_1, ?)\) to \?

- if \( b_1 \) then \( e_1 \) elseif \( b_2 \) then \( e_2 \) else \( e \) fi will mean
  
  \[
  \text{if } b_1 \text{ then } e_1 \text{ elseif } b_2 \text{ then } e_2 \text{ else } e \text{ fi}
  \]

- 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, \( \text{val} \in \text{Val}, \text{id} \in \text{id} \).

- Instead of writing \( f = b.e \) we will write \( f(z) = e \)

Therefore \( g(x)(y) = x*y \) means \( g = \lambda x. \lambda y. x*y \)

If \( A \) and \( B \) are domains then

- \( A \rightarrow B \) is the domain of all continuous functions from \( A \) to \( B \)
- \( A \times B \) is the cartesian product
- \([a_1, b_1, \ldots; a_n, b_n]\) is the element in \( A \rightarrow B \) which maps \( a_i \) to \( b_i \), \( 1 \leq i \leq n \).

All other elements of \( A \) are mapped to '?' in \( B \)

- If \( z \) is a tuple, for instance \( z \in A \times B \), then \( z^i \) will denote the \( i \)th projection of \( z \). Alternatively, we may use the names of component domains to denote projections when the notation is unambiguous, i.e. \( z.A \) and \( z.B \) may be used in place of \( z^1 \) and \( z^2 \)

- \( x \oplus y \) will denote

  \[
  \text{if } x.y \in A \rightarrow B \text{ then } \lambda a. (\text{if } y(a) = ? \text{ then } x(a) \text{ else } ?) \text{ fi}
  \]

- \( x \otimes y \) will denote

  \[
  \text{if } x.y \in A \rightarrow B \text{ then } \lambda a. (\text{if } x(a) = ? \text{ then } y(a) \text{ elseif } y(a) = ? \text{ then } x(a) \text{ else } ?) \text{ fi}
  \]

- \( x \oplus_A y \) will denote

  \[
  \text{if } x \in A \times B \text{ and } y \in A \text{ then } (x.A \oplus y, x.B) \text{ fi}
  \]

- \( x \otimes_A y \) will denote

  \[
  \text{if } x \in A \times B \text{ and } y \in A \text{ then } (x.A \otimes y, x.B) \text{ fi}
  \]

In giving the meaning functions we shall use an informal pattern-matching mechanism in order to identify parameters. We use double brackets, \([ \ldots ] \), 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, "\( \ldots \)\", 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 "\( \text{where}(z, y) = \ldots \)" to give the names "\( z \)" to its first part and "\( y \)" to its second part.
The reader familiar with denotational semantics will recognize the operators $\oplus$ and $\otimes$ 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 $\otimes$ 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,

$$\sum_{j=1}^{n} ["i_{j}":"m_{j}]$$

to mean

$$["i_{1}":"m_{1}] \oplus ["i_{2}":"m_{2}] \oplus \ldots ["i_{n}":"m_{n}]$$

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 $\times$ is the coalesced product:

$$0 = R + \text{Id} + (0 \times 0) + (0 \rightarrow 0)$$

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

$$T = \text{Id} + \text{Enumeration} + \text{Restriction} + \text{Set} + \text{Record} + \text{Product} + \text{Union} + \text{Map}$$

where

- $\text{Enumeration} = \text{Id}^*$
- $\text{Restriction} = T \times \text{Expr}$
- $\text{Set} = T$
- $\text{Record} = (\text{Id} \times T)^*$
- $\text{Product} = T^*$
- $\text{Union} = (\text{Id} \times T)^*$
- $\text{Map} = T + T$

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.)

$$\text{Env} = \text{Const} \times \text{Type} \times \text{Var} \times \text{Op} \times \text{E.name} \times \text{Class}$$

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

$$\text{Const} = \text{Id} \rightarrow (T \times 0)$$

--- An element of $\text{Const}$ maps identifiers to pairs consisting of a type and a value.
\[ \text{Type} = \text{Id} \rightarrow (T \times 0) \]

--- An element of \text{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.

\[ \text{Var} = \text{Id} \rightarrow T \]

--- An element of \text{Var} associates a type with each variable.

\[ \text{Op} = \text{Id} \rightarrow T \]

--- An element of \text{Op} maps identifiers which are the names of class-derived types to elements of \text{Const} which define the meanings of the class constants for these types.

\[ \text{E.name} = \text{Id} \rightarrow \text{Env} \]

--- An element of \text{E.name} associates environments with environment names.

\[ \text{Class} = \text{Id} \rightarrow \text{Env} \rightarrow (C \times \text{Const}) \]

where \( C = P(\text{Id}) \times (\text{Id} \rightarrow T) \)

--- An element of \text{Class} maps class identifiers to their meanings. The class signature domain, \( C \), contains characterizations of the parameters of classes.

\[ \text{Val} = \text{Id} \rightarrow 0 \]

--- An element of \text{Val} maps names of program variables to values, and represents a state of a computation.

We shall use the following meaning functions:

\[ \text{eval} : \text{Stmt} \rightarrow \text{Env} \rightarrow \text{Val} \rightarrow \text{Val} \]

--- where \( \text{Stmt} \) is a primitive domain of syntactical objects. \text{eval} yields the meaning of a program statement which changes the mapping that associates values with variables in a given environment.

\[ \text{value.of} : \text{Expr} \rightarrow \text{Env} \rightarrow \text{Val} \rightarrow 0 \]

--- yields the meaning of an expression.

\[ \text{e.type} : \text{Type.expr} \rightarrow \text{Env} \rightarrow T \]

--- where \( \text{Type.expr} \) is a primitive domain of syntactical objects. \text{e.type} yields the meaning of an expression denoting a type.

\[ \text{m.decl} : \text{Decl} \rightarrow \text{Env} \rightarrow \text{Env} \]

--- where \( \text{Decl} \) is a primitive domain of syntactical objects. \text{m.decl} yields the meaning of a declaration of a constant, type, class, or environment.

\[ \text{m.var} : \text{Var.decl} \rightarrow \text{Env} \rightarrow (\text{Env} \times \text{Val}) \]

--- where \( \text{Var.decl} \) is a primitive domain of syntactical objects. \text{m.var} yields the meaning of a declaration that gives types and initial values to variables.

\[ \text{m.param} : \text{Param.decls} \rightarrow \text{Env} \rightarrow \text{Env} \]

--- where \( \text{Param.decls} \) is a declaration of actual parameters of a class instance. \text{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

\[
\text{\texttt{m\_decl [<environment>]}(env) = Y_{ev} \lambda \text{.} \text{ev} \otimes m\_decl [<\text{class\ decl}>] (m\_decl [<\text{type\ decl}>] (m\_decl [<\text{const\ decl}>] (m\_decl [<env\ decl>] (e)))))}
\]

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:
```

The notation $id_l | id_r$ signifies that $id_l$ is to be an alias name in the local environment for the name $id_r$ from an imported environment. $id_l$ has all the same attributes (type, value, etc.) associated with $id_r$ 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 [ ref id ( env id [ a_1, ..., const j_1|b_1, ... type k_1|c_1, ..., class l_1|d_1, ... ] ) ] ( env ) =
```

```
[ ] \_E\_name \sum_i ["a_i": e'.E\_name [id_i ]]
```

```
\_Const \sum_i ["b_i": e'.Const [ j_i ]]
```

```
\_Type \sum_i ["c_i": e'.Type [ k_i ]]
```

```
\_Op \sum_i ["c_i": e'.Op [ k_i ]]
```

```
\_Class \sum_i ["d_i": e'.Class [ l_i ]]
```

where e' = env.E\_name [id ]
4.2. Environment declarations

Syntax

Semantics:
\[
m_{\text{decl}} \left[ \text{env id } \left[ \langle \text{import decl} \rangle \langle \text{environment} \rangle \right] \right](\text{env}) = \\
\text{env} \oplus E_{\text{name}}["id" : \lambda e'. m_{\text{decl}} \left[ \langle \text{import decl} \rangle \right](\text{env}) \\
\oplus m_{\text{decl}} \left[ \langle \text{environment} \rangle \right](e') \\
\oplus E_{\text{name}}["id" : e']]
\]

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:
\[
\text{const int 15 : integer } = 32767,
\text{complex_zero : float } \times \text{float } = (0,0)
\]

Syntax

Semantics:
\[
m_{\text{decl}} \left[ \text{const id } : t = expr, <\text{const decls}> \right](\text{env}) = \\
m_{\text{decl}} \left[ \text{const } <\text{const decls}> \right] \\
(\text{env} \oplus \text{Const}["id" : (e_{\text{type}} [t], \text{value_of } [expr ])(\text{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:

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

Syntax

type decls:

```
class instance:
```

12
Semantics:
\[ m_{\text{decl}} \left[ \text{type id : } t \ , \text{<type decls>} \right](env) = \]
\[ m_{\text{decl}} \left[ \text{type <type decls>} \right](m_{\text{decl}} \left[ \text{type id : } t \right](env)) \]

\[ m_{\text{decl}} \left[ \text{type id : } t \right](env) = \]
cases t is
"\{id_1, ..., id_n\}" \implies \( env \otimes \text{type} \left[ \text{"id" : } t' \right] \)
\[ \otimes_{\text{Const}} \sum_{j=1}^{n} \left[ \text{"id}_j" : (t',j) \right] \]
where \( t' = \text{enumeration("id_1", ..., "id_n")} \)

"C.id \left[ \text{type i_1 : } t_1,...\text{ const k_1 : } s_1 = e_1,...\right]" \implies
\( env \otimes \text{type} \left[ \text{"id" : } \text{class} \right] \)
\[ \otimes_{\text{Def}} \left[ \text{"id" : } C\cdot\text{def}\cdot\text{Const} \right] \]
where \( C\cdot\text{def} = env\cdot Class \left[ C\cdot\text{id} \right] \)
\( (m_{\text{param}} \left[ \text{type i_1 : } t_1,...\text{ const k_1 : } s_1 = e_1,...\right](env)) \)

"<\text{type expr}>" \implies \( env \otimes \text{type} \left[ \text{"id" : e\cdot\text{type} \ [t]} \right](env) \)
esac

where \( m\cdot\text{param} \) is a function that defines the local environment contributed by the actual parameters in a class instance:
\[ m_{\text{param}} \left[ \text{type i_1 : } t_1,...\text{ const k_1 : } s_1 = e_1,...\right](env) = \]
\[ [] \otimes_{\text{type}} \text{type\_args} \]
\[ \otimes_{\text{Const}} \sum_{j} \left[ \text{"k}_j" : (\text{e\cdot\text{type} \ [s_j]} \ (env'), \text{value\_of} \ [e_j] \ (env') \ ([])]) \right] \]

where \( env' = env \otimes_{\text{type}} \text{type\_args} \)
and \( \text{type\_args} = \sum_{j} \left[ \text{"i}_j" : \text{e\cdot\text{type} \ [t_j]} \ (env) \right] \)

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, power sets, 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:
\[ e_{\text{type}}[\text{id}](\text{env}) = \text{env}.\text{Type}["\text{id}"]+1 \]

5.2. Enumeration types

A small example:

{red, green, blue}

Syntax

\[
\text{enumeration:}
\]

Semantics:
\[ e_{\text{type}}[\{\text{id}_1, \ldots, \text{id}_n\}](\text{env}) = \text{enumeration}("\text{id}_1", \ldots, "\text{id}_n") \]

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:
\[ e_{\text{type}}[\text{float}](\text{env}) = "\text{float}" \]
\[ e_{\text{type}}[\text{integer}](\text{env}) = "\text{integer}" \]

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.

<table>
<thead>
<tr>
<th>Precedence of type operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

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 cvn is applied to the expression. cvn 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 expecta-
tions of its author.

Some small examples:

{1..10 : integer}
{x:float | abs(x - sqrt 2) <= 0.5}
{x:vector | sorted x}

Syntax

Semantics:

\[ e_{\text{type}} \{ \{\text{id}: t \mid \text{expr} \} \}(\text{env}) = \]
\[ \text{restriction}\{e_{\text{type}} \{ t \} \}(\text{env}), \text{ground} \{\text{\lambda id . expr}\}(\text{env})) \]

The use of a \( \lambda\)-expression in a type expression expresses only the fact that the
identifier is bound locally. The only conversion rule of the \( \lambda\)-calculus that is
used in comparing types is the rule of \( \alpha\)-conversion (renaming of a bound vari-
able).

The function \text{ground} : \text{Expr} → \text{Env} → \text{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 \text{ground} \{\text{expr}\}(\text{env}) contains no
occurrence of any identifier bound in \text{env}. This allows the definition of a re-
stricted type to be imported from a named environment into a current environ-
ment 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 \text{ and } x <= e_2\}

where \( e_1 \) and \( e_2 \) are expressions.
5.4.2. Records

Syntax

```
record identifier => type expr end
```

Semantics:

\[
e_{type} \{ \text{record id}_1 \colon t_1, \ldots, id_n \colon t_n \ \text{end} \} (\text{env}) = \text{record}(\langle id_{\pi_1}, e_{type} \[ t_1 \](\text{env}) \rangle, \ldots, \langle id_{\pi_n}, e_{type} \[ t_n \](\text{env}) \rangle)
\]

where \(\pi\) 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

```
\text{type expr} \times \text{type expr}
```

Semantics:

\[
e_{type} \[ t_1 \ldots \times t_n \] (\text{env}) = \text{product}(e_{type} \[ t_1 \](\text{env}), \ldots, e_{type} \[ t_n \](\text{env}))
\]

5.4.4. Mappings

A small example

\[
\text{type vector} : \{1..n: \text{integer}\} \rightarrow \text{float},
\text{dictionary} : \text{string} \rightarrow \text{string}
\]

Syntax

```
type expr => mapping type
```

Semantics:

\[
e_{type} \[ t_1 \rightarrow t_2 \] (\text{env}) = \text{map}(e_{type} \[ t_1 \](\text{env}), e_{type} \[ t_2 \](\text{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:

\[ e_{\text{type}} \{ \text{set of } t \}(env) = \text{set}(e_{\text{type}} \{ t \}(env)) \]

Composition of powerset types is restricted in Apple. The base type of a powerset must be 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:

\[ e_{\text{type}} \{ id_1#t_1 + ... + id_n#t_n \}(env) = \]
\[ \text{union}(e_{\text{type}} \{ t_1 \}(env),..., e_{\text{type}} \{ t_n \}(env)) \]

where \( \pi \) 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:

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

Syntax

```
\text{identifier} \rightarrow \text{rec} \rightarrow \text{type expr}
```

Semantics:

\[
\text{e\_type \{ id rec t\_expr \}(env)} = \text{recursive(e\_type \{ t\_expr \}(env)[rec/ id])}
\]

\text{where } e[x/z] \text{ denotes the expression gotten from } e \text{ by replacing every free occurrence of } z \text{ by } \xi.

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

1) The type expression following the keyword \text{rec} must be a union whose first term does not contain the identifier which precedes the keyword \text{rec}.

2) The identifier which precedes the keyword \text{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

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

```haskell
class BoundedStack 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,... id_n, const c_1:s_1,... c_n:s_n ===
    def f_{1:t_1},..., f_k:t_k rep <environment> endclass ] (env)

= env @class ["C" : λρ. [ ] Φ_C t ] @Const silt [def f_{1:t_1},..., f_k:t_k ](r)]
    // meanings of exported constants

where silt [def f_{1:t_1},..., f_k:t_k ](r) =
    \[ \sum_{j=1}^{k} ["f_j" : (e_type [t_j] (r), r.Const [f_j] O)] \]

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

where env' = env @Type Σ_{j=1}^{n} ["id_j", "id_j"]

and r = m_decl [<environment> ] (env @ ρ)
```

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 `silt`, defined above, which uses the `defines` 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.

<table>
<thead>
<tr>
<th>Operator Precedence Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
7.1. Identifiers

Semantics:

\[
\text{val}_{\text{ue}_f \mid \text{id}}(\text{en}_v)(\text{val}) = \\
\quad \text{if } \text{en}_v.\text{Const} [\ ] \neq ? \\
\quad \quad \quad \text{then } \text{en}_v.\text{Const} [\text{id}] : 2 \\
\quad \quad \quad \text{else } \text{val} [\text{id}] \\
\quad \fi
\]

7.2. Expressions for type \text{Boo1}

7.2.1. Conditional expressions

Syntax

\[
\text{Semantics:} \\
\quad \text{val}_{\text{ue}_f \mid \text{if } b \text{ then } e_1 \text{ elseif } e_2 \ldots \text{ else } e_{n+1} \text{ endif }}(\text{en}_v)(\text{val}) = \\
\quad \quad \text{if } \text{val}_{\text{ue}_f \mid b}(\text{en}_v)(\text{val}) \\
\quad \quad \quad \text{then } \text{val}_{\text{ue}_f \mid e_1}(\text{en}_v)(\text{val}) \\
\quad \quad \quad \text{else } \text{val}_{\text{ue}_f \mid e_2}(\text{en}_v)(\text{val}) \\
\quad \quad \fi
\]

The expression

\[
\text{if } b_1 \text{ then } e_1 \text{ elseif } e_2 \ldots \text{ else } e_{n+1} \text{ endif}
\]

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

\[
\text{if } b_1 \text{ then } e_1 \text{ else } \\
\text{if } b_2 \text{ then } e_2 \text{ else } \\
\quad \ldots \text{else } \\
\text{e}_{n+1} \text{ endif} \ldots \text{ endif} \text{ endif} \]
7.2.2. Boolean expressions

Syntax

**logical expr:**

```
logical expr  or  logical conjunct
```

**logical conjunct:**

```
logical conjunct  and  logical primary
```

**logical primary:**

```
not  relational expr
```

Semantics:

\[
\text{value\_of } \text{not } e \ (env)(val) = \\
\quad \text{if value\_of } [e] (env)(val) \quad \text{then false else true fi}
\]

\[
\text{value\_of } [e_1 \text{ and } e_2] (env)(val) = \\
\quad \text{if value\_of } [e_1] (env)(val) \\
\quad \quad \text{then value\_of } [e_2] (env)(val) \\
\quad \quad \text{else false fi}
\]

\[
\text{value\_of } [e_1 \text{ or } e_2] (env)(val) = \\
\quad \text{if value\_of } [e_1] (env)(val) \\
\quad \quad \text{then true} \\
\quad \quad \text{else value\_of } [e_2] (env)(val) \\
\quad \quad \text{fi}
\]
7.3. Arithmetic expressions

7.3.1. Denotation of values

Syntax

algebraic expr:

algebraic product:

algebraic primary:

selector expr:
simple expr:

- 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 \texttt{pred} and \texttt{succ} are defined on integer types and on enumeration types, for which the meanings of constants are defined as natural numbers.

Semantics:

\begin{align*}
\text{value}_\text{of} \ [ - e ](\text{env})(\text{val}) &= -\text{value}_\text{of} \ [ e ](\text{env})(\text{val}) \\
\text{value}_\text{of} \ [ \text{succ} \ e ](\text{env})(\text{val}) &= \text{value}_\text{of} \ [ e ](\text{env})(\text{val}) + 1 \\
\text{value}_\text{of} \ [ \text{pred} \ e ](\text{env})(\text{val}) &= \text{value}_\text{of} \ [ e ](\text{env})(\text{val}) - 1
\end{align*}

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 \texttt{div} and \texttt{mod} are defined only for type integer.

Semantics:

\begin{align*}
\text{value}_\text{of} \ [ e_1 \ \text{op} \ e_2 ](\text{env})(\text{val}) &= \ M \ [ \text{op} ](\text{value}_\text{of} \ [ e_1 ](\text{env})(\text{val}), \text{value}_\text{of} \ [ e_2 ](\text{env})(\text{val}))
\end{align*}

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

\begin{align*}
M \ [ \text{div} ] &= \lambda x. \lambda y. \max\{z : \text{integer} \mid z \leq x / y\} \\
M \ [ \text{mod} ] &= \lambda x. \lambda y. \ x - y * (M \ [ \text{div} ](x)(y))
\end{align*}
7.4. Relational expressions

The relational operators are a set of binary infix operators producing values of type \(\text{Bool}\). 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:**

\[
\text{value}_{of} \left[ e_1 = e_2 \right](\text{env})(\text{val}) = \\
(\text{value}_{of} \left[ e_1 \right](\text{env})(\text{val}) = \text{value}_{of} \left[ e_2 \right](\text{env})(\text{val}))
\]

The expression \(e_1 \neq e_2\) is equivalent to \(\text{not}(e_1 = e_2)\).

7.4.2. Inequalities

**Semantics:**

\[
\text{value}_{of} \left[ e_1 \leq e_2 \right](\text{env})(\text{val}) = \\
M \leq \left[ \text{value}_{of} \left[ e_1 \right](\text{env})(\text{val}), \text{value}_{of} \left[ e_2 \right](\text{env})(\text{val}) \right]
\]

where \(M \leq \left[ = \right] = Y \lambda x. \lambda y. \text{if } x=y \text{ or } r(\text{succ } x, y) \text{ then } \text{true}
\]

\[\text{elseif } r(\text{succ } y, x) \text{ then } \text{false} \text{ else } \bot\]

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

"\(e_1 < e_2\)" is equivalent to "\(e_1 \leq e_2 \text{ and not } (e_1 = e_2)\)"

"\(e_1 \geq e_2\)" is equivalent to "\(\text{not}(e_1 < e_2)\)"

"\(e_1 > e_2\)" is equivalent to "\(\text{not}(e_1 \leq e_2)\)"
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 \( A_1 \times \cdots \times A_n \).

\[
value_{of} [(a_1, \ldots, a_n)](env)(val) = (a_1', \ldots, a_n')
\]

where \( a_i' = value_{of} [a_i](env)(val) \) for \( 1 \leq i \leq n \)

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 \( A_1 \times \cdots \times A_n \) and \( n > 1 \) and \( i \) is a numeral satisfying \( 1 \leq i \leq n \), is defined by

\[
value_{of} [e!i](env)(val) = (value_{of} [e](env)(val))_i
\]
7.6. Expressions involving record types

7.6.1. Value constructor

Syntax

record constructor:

Semantics:

\[
value_{of} \left[ [id_1=e_1, \ldots, id_n=e_n] \right](env)(val) = \sum_{j=1}^{n} \left[ "id_j" : value_{of} \left[ e_j \right](env)(val) \right]
\]

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:

\[
value_{of} \left[ e.id \right](env)(val) = value_{of} \left[ e \right](env)(val) [ id ]
\]

Syntax

expression:

Semantics:

\[
value_{of} \left[ r | id : e \right](env)(val) = value_{of} \left[ r \right](env)(val) \oplus \left[ "id" : value_{of} \left[ e \right](env)(val) \right]
\]
7.7. Expressions involving mapping types

7.7.1. Value constructor

Syntax

function constructor:

\[
\text{function } \text{identifier } : \text{type expr } \text{end}
\]

var decls:

\[
\text{var } \text{identifier } : \text{type expr } == \text{expression}
\]

Semantics:

\[
\text{value_of } [\text{function } (i : t) e ; v \text{ do stmt return expr end }](env)(val) =
\]

\[
\lambda x. \text{value_of } [\text{expr } ([\text{env'}, \text{env}["i": (e \text{ type } t)(\text{env}), x])])(val')
\]

where \((\text{env'}, \text{val'}) = \text{m_var } [v ](\text{m_decl } [e ](\text{env})))

7.7.1.1. The meaning of a declaration of variables

\[
\text{m_var } [\text{var } id : t == expr, <\text{var decls}>](\text{env}) =
\]

\[
(\text{e } \oplus_{\text{var}} ["id": \text{e_type } t](\text{env})),
\]

\[
\text{v } \oplus ["id": \text{value_of } [\text{expr } ](\text{env})][[]]
\]

where \((\text{e }, \text{v}) = \text{if } <\text{var decls}> \text{ is an empty string}
\]

\[
\text{then } (\text{env}, [])
\]

\[
\text{else } \text{m_var } [<\text{var decls}>](\text{env})
\]

f

The declaration

\[
\text{var } id_1, \ldots, id_n : t == expr
\]

is equivalent to

\[
\text{var } id_1 : t == expr, \ldots, id_n : t == expr
\]
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

\[ \text{value_of \, } f \, a \, (\text{env})(\text{val}) = \]
\[ \text{value_of \, } f \, (\text{env})(\text{val})(\text{value_of \, } a \, (\text{env})(\text{val})) \]

If \( f \) and \( g \) are expressions typed as \( f : A \rightarrow B \) and \( g : B \rightarrow C \), then the composition of \( g \) with \( f \) is defined by

\( g \circ f \) is equivalent to \( \text{function (} z : A \text{) return } g(f(z)) \text{ end} \)

If \( f \), \( a \) and \( b \) are expressions typed as \( f : A \rightarrow B \), \( a : A \), and \( b : B \), then the following defines the expression read as "\( f \) supersedes at \( a \) by \( b \)."

\( f \mid a:b \) is equivalent to \( \text{function (} z : A \text{) return if } z = a \text{ then } b \text{ else } f(z) \text{ end} \)
7.8. Expressions involving powerset types

7.8.1. Value constructor

**Syntax**

**powerset constructor:**

![Diagram of powerset constructor]

**range element:**

![Diagram of range element]

If \( e(z) \) is an expression of type \( \text{Bool} \) with only \( z \) free, then
\[
\{z:A | e(z)\}
\]
is an expression of type \( \text{set of } A \).

**Semantics:**

\[
\text{value of } \{z:A | e\}(env)(val) = \\
\lambda y. \text{value of } e\{e, y\}(\text{env} \oplus \text{const} \{z: \text{type } A\}(env, y))(val)
\]
The meaning of a powerset expression is a function from values of the base type to Boolean values.

If \( e_1, ..., e_n \) are expressions of type \( A \), then the expression
\[
\{e_1, ..., e_n\}
\]
is equivalent to \( \{z:A | z = e_1 \text{ or } ... \text{ or } z = e_n\} \)
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 \( \text{min} \) finds the least element in \( e \) which is not less than \( a \).

**Semantics:**

\[
\text{value}_{of} \{ \text{min}(e, a) \}(\text{env})(\text{val}) = \\
(Y \lambda m. \lambda e'. \lambda a'. \text{if } e'(a') \text{ then } a' \text{ else } m(e')(a' + 1) \alpha) \\
(value_{of} [e](\text{env})(\text{val}))(value_{of} [a](\text{env})(\text{val}))
\]

7.8.2.2. The operator \( \text{max} \), 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

\[
\text{value}_{of} \{ e_1 \text{ op } e_2 \}(\text{env})(\text{val}) = \lambda x. e_1'(x) \text{ op}' e_2'(x)
\]

where \( e_i' = \text{value}_{of} [e_i](\text{env})(\text{val}) \)

and where if \( \text{op} \) is "union" then \( \text{op}' \) is or,

if \( \text{op} \) is "intersection" then \( \text{op}' \) is and,

if \( \text{op} \) is "difference" then \( \text{op}' \) is and not

7.8.2.4. set membership

\[
\text{value}_{of} \{ e_1 \text{ member } e_2 \}(\text{env})(\text{val}) = \\
\text{value}_{of} [e_2](\text{env})(\text{val}))(\text{value}_{of} [e_1](\text{env})(\text{val}))
\]

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.

\[
\text{value}_{of} \{ \{ e(x) \mid x \text{ in } S \text{ and } b(x) \} \}(\text{env})(\text{val}) = \\
\{ y \mid (\exists x) x \in \text{value}_{of} [S](\text{env})(\text{val}) \text{ and } \\
\text{value}_{of} [b](\text{env})(\text{val}) x = \text{true} \text{ and } \\
y = \text{value}_{of} [e](\text{env})(\text{val}) x \}
\]

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

\[
\text{value}_{of} [id\#e](\text{env})(\text{val}) = ("id", \text{value}_{of} [e](\text{env})(\text{val}))
\]
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 be used in the unioncase expression which discriminates on the case tag. The meaning of this expression is defined by

\[
\text{value_of } \mid \text{unioncase } e \mid = \\begin{cases}
    e_1 & \text{if } e' = i_1 \\
    e_2 & \text{if } e' = i_2 \\
    \vdots \\
    e_n & \text{if } e' = i_n \\
    \text{otherwise} & \text{in } e_{n+1}
\end{cases}
\]

where \( e' = \text{value_of } e \mid (\text{env})(\text{val}) \), and

\( e_i' = \text{value_of } e_i \mid (\text{env})(\text{val}) \) for \( 1 \leq i \leq n \).
7.10. Expressions involving locally defined constants

Syntax

let expression:

Semantics:

\[
\text{value}_\text{of} \begin{array}{l}
\text{let} \; \mathbf{id}_1 : t_1 \equiv e_1, \ldots, \mathbf{id}_n : t_n \equiv e_n \; \text{in} \; \text{expr} \; \text{endlet} \end{array} (\text{env})(\text{val}) = \\
\text{value}_\text{of} \begin{array}{l}
\text{expr} \end{array} (Y_{\mathbf{Env}} \lambda e. \text{env} \oplus \text{Const} \sum_{i=1}^{n} ["\mathbf{id}" : (t_i', v_i)])(\text{val})
\]

where \( t_i' = \text{e\_type} \begin{array}{l}
\mathbf{id}_i \end{array} (\text{env}) \), and

\( v_i = \text{value}_\text{of} \begin{array}{l}
e_i \end{array} (e)(\text{val}) \)
8. Statements

Syntax

Semantics:
\[ \text{eval } \{ \text{skip} \} (\text{env})(\text{val}) = \text{val} \]
8.2. Assignment

Syntax

Examples:

- \( a := 5 \times x \);
- \((\text{day}, \text{month}, \text{year}) := (31, 12, 79)\);
- \((x, y) := (y, x)\)

Semantics:

\[
\text{eval } [id := e](env)(val) = \text{val} \triangleleft ["id" : \text{value_of } [e](env)(val)]
\]

\[
\text{eval } [(id_1, \ldots, id_n) := e](env)(val) = \\
\text{val} \triangleleft \sum_{i=1}^{n} ["id_i" : \text{value_of } [e](env)(val) \times i]
\]

8.3. Sequential composition

Syntax

A small example

- \( a := 5; \ b := a+b \)

Semantics:

\[
\text{eval } [s_1; s_2](env)(val) = \text{eval } [s_2](env)(\text{eval } [s_1](env)(val))
\]
8.4. The conditional statement

**Syntax**

\[
\text{if } \text{expression} \text{ then } \text{statement} \text{ elseif } \text{statement} \text{ else } \text{statement} \text{ endif}
\]

*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:**

\[
eval \left( \text{if } e \text{ then } s_1 \text{ elseif } s_2 \text{ endif} \right)(env)(val) = \\
\text{if value_of } [e](env)(val) \text{ then eval } [s_1](env)(val) \text{ else eval } [s_2](env)(val)
\]

The repeated conditional statement

\[
\text{if } e_1 \text{ then } s_1 \text{ elseif } e_2 \text{ then } s_2 \ldots \text{ else } s_{n+1} \text{ endif}
\]

can be written as an abbreviation for the nested conditional statement

\[
\text{if } e_1 \text{ then } s_1 \text{ else } \\
\text{if } e_2 \text{ then } s_2 \text{ else } \\
\ldots \\
\text{else } \\
s_{n+1} \text{ endif endif}
\]
8.5. The `unioncase` statement

**Syntax**

```
unioncase expression of
  identifier # identifier : type expr
do statement
endcase
```

**Small example:**

```
unioncase S_list of
  Null # 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' + 1 = id_i then eval [s_i](env @ Const["a_1": (Bool, e' + 2)])(val)
  ...
elseif e' + n = id_i then eval [s_n](env @ Const["a_n": (Bool, e' + 2)])(val)
e else eval [s_{n+1}](env)(val)
fi
```

where e' = value_of [e](env)(val)
Note: the definition of \texttt{eval} does not depend upon the type associated with a constant. Therefore the type \texttt{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

\textit{Syntax}

\[
\rightarrow \text{identifier} \rightarrow \vdots \rightarrow \text{statement} \rightarrow \vdots \rightarrow \text{identifier} \rightarrow
\]

\textit{An example:}

\[
(x,y,z) := (A,N,1);
\]

Exponent::

\[
\begin{align*}
\text{if } y & \leq 0 \text{ then skip} \\
\text{else if } \text{even } y & \text{ then} \\
\quad (x,y) & := (x^x, \ y \ div \ 2) \\
\text{else} \\
\quad (y,z) & := (y-1, x*z) \\
\text{endif}; \\
\text{repeat Exponent} \\
\text{endif}
\end{align*}
\]

\textit{Semantics:}

\[
\text{eval \ [id::stmt::id \ (env\ (val)) = } \\
\ Y_{\text{val}}^{-1_{\text{val}}} \ \lambda s. \ \text{eval \ [stmt\ (env\ (val \ \oplus \ \text{"id": s}) )] }
\]

8.7. The repeat statement

\textit{Syntax}

\[
\rightarrow \text{repeat} \rightarrow \text{identifier} \rightarrow
\]

\textit{Semantics:}

\[
\text{eval \ [repeat \ id \ (env\ (val)) = \ val \ [id \ (val) ]}
\]
8.8. The let statement — local constant definition

Syntax

An example:

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

Semantics:

```plaintext
eval [let id_1 | t_1 == e_1, ..., id_n | t_n == e_n in stmt endlet] (env)(val)
    = eval [stmt] (Y_{env} \lambda e . env \otimes Const \sum_{i=1}^{n} \{"id_i": (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**

```plaintext
for statement:
```

```
for <identifier> : <type expr> incr <expression> from <p> in <stmt> endfor
```

**Semantics:**

The statement

```plaintext
for <x> : <t> incr <set expr> from <p> in <stmt> endfor
```

is equivalent to

```plaintext
let <set const> : set of <t> == <set expr> in
  <p>' := <p>;
  for ::
    if \{<y> : <t> | <y> >= <p>' and <y> member <set const>\} = {} then
      skip
    else
      let <z> : <t> = min(<set const>, <p>') in
      <stmt> : <p>' := succ <z> ; repeat for
    endif
  endfor
endlet
::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

\[
e_{\text{type}} : T_{\text{expr}} \to \text{Env} \to T
\]

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 \( \mathcal{D} \), which is a distinguished element in the domain \( T \). That is, \( \mathcal{D} \) 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 \( \text{subtype} \). This partial order is not absolute, but is relative to an environment in which types can be named. When the element \( \mathcal{D} \) 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,

\[
\text{type.of} : \text{Expr} \to \text{Env} \to T
\]

\[
\text{subtype} : T \times T \to \text{Env} \to \text{Bool}
\]

\[
\text{lub.type} : T \times T \to \text{Env} \to T
\]

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

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

\[
\text{pcheck} : \text{Stmt} \to \text{Env} \to 2^{\{\text{Id}\}} \to \text{Bool}
\]

\[
\text{dcheck} : \text{Decl} \to \text{Env} \to \text{Bool}
\]
9.2. A partial ordering of types

The function \( \text{subtype} : T \times T \rightarrow \text{Env} \rightarrow \text{Bool} \), 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:

1. \( \text{subtype}(t, \_)(\text{env}) = \text{true} \)
2. \( \text{subtype}(t, t)(\text{env}) = \text{true} \)
3. \( \text{subtype}((\text{restriction}(t, p), t))(\text{env}) = \text{true} \)
4. \( \text{subtype}(\text{id}, t)(\text{env}) = \text{subtype}(\text{env}.\text{Type}[\text{id}] \rightarrow 1, t)(\text{env}) \)
5. \( \text{subtype}((\text{map}(d_1, r_1), \text{map}(d_2, r_2)))(\text{env}) = \text{subtype}(r_1, r_2)(\text{env}) \land \text{subtype}(d_2, d_1)(\text{env}) \)
6. \( \text{subtype}((\text{record}((i_1, t_1), \ldots, (i_n, t_n)), \text{record}((i_1', t_1'), \ldots, (i_m, t_m')))(\text{env}) = \bigwedge_{k=1}^{n} \text{subtype}(i_k, i_k')(\text{env}) \)
7. If \( h : 1..m \rightarrow 1..n \) is injective and monotonic (with respect to the natural ordering of the natural numbers), then
   \( \text{subtype}((\text{union}((i_{h_1}, t_{h_1}), \ldots, (i_{h_m}, t_{h_m})), \text{union}((i_1, t_1'), \ldots, (i_m, t_m')))(\text{env}) = \bigwedge_{k=1}^{m} i_{h_k} = i_k \land \text{subtype}(t_{h_k}, t_k') \)
8. If \( \text{op}(t_1, \ldots, t_n) \) is an expression belonging to \( T \), and \( \text{op} \) is "set" or "product", then
   \( \text{subtype}((\text{op}(t_1, \ldots, t_n), \text{op}(t_1', \ldots, t_n'))(\text{env}) = \bigwedge_{t=1}^{n} \text{subtype}(t, t')(\text{env}) \)
9. If \( t_1 \) and \( t_2 \) are types and \( \text{subtype}(t_1, t_2)(\text{env}) \) is not defined by rules (0..8) then \( \text{subtype}(t_1, t_2)(\text{env}) = \text{false} \).
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

$$\text{lub}_\text{type} : T \times T \times \text{Env} \rightarrow T$$

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

$$\text{lub}_\text{type}(a, b)(\text{env}) =$$

if $a < b$ then $b$

else if $b < a$ then $a$

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

else if $a$ is an identifier then $\text{lub}_\text{type}(\text{env. Type}(a), b)(\text{env})$

else if $b$ is an identifier then $\text{lub}_\text{type}(\text{env. Type}(b), a)(\text{env})$

else if $a$ is restriction($t_a, p_a$) and $b$ is restriction($t_b, p_b$)

then $\text{lub}_\text{type}(t_a, t_b)(\text{env})$

else if $a$ 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 $\wedge_{k=1}^n s_k \neq ?$

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

else $?$

where $s_k = \text{lub}_\text{type}(t_k, t'_k)(\text{env})$

else if $a$ is union(($i_1, t_1), \ldots, (i_m, t_m)$) and $b$ is union(($i_1, t'_1), \ldots, (i_n, t'_n)$)

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 $z \in 1..p$ either

i) $i_g^{-1}(z) = k_z$ and $s_z = t_g^{-1}(z)$ and $j_h^{-1}(z)$ is undefined, or

ii) $j_h^{-1}(z) = k_z$ and $s_z = t_h^{-1}(z)$ and $i_g^{-1}(z)$ is undefined, or

iii) $i_g^{-1}(z) = k_z$ and $j_h^{-1}(z) = k_z$ and

$s_z = \text{lub}_\text{type}(t_g^{-1}(z), t_h^{-1}(z))(\text{env})$ and $s_z \neq ?$

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

else $?$

else if $a = \text{product}(t_1, \ldots, t_n)$ and $b = \text{product}(t'_1, \ldots, t'_n)$

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

then product($s_1, \ldots, s_n$)

else $?$

47
where $s_k = \text{lub_type}(t_k, t_k')(env)$

elseif $a = \text{set}(t_1, \ldots, t_n)$ and $b = \text{set}(t_1', \ldots, t_n')$

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

then $\text{set}(s_1, \ldots, s_n)$

else ?

fi

where $s_k = \text{lub_type}(t_k, t_k')(env)$

else ?

fi

The least upper bound determined by \text{lub_type} is used to define a type in which a pair of values can be compared by a relational operator. Note that \text{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 $\text{lub_type}(a, b)$ 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

\[ \text{type_of} : \text{Expr} \times \text{Env} \to \text{T} \]

by structural analysis of its first argument.

In the definition of \text{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

\[ \text{basetype} : \text{T} \to \text{Env} \to \text{T} = \]

\[ \lambda t. \lambda \text{env}. \ 	ext{cases} t \text{ is} \]

\[ "\text{integer}" , "\text{float}" , "\text{bool}" , "\text{ascii}" \Rightarrow t \]

\[ \text{identifier} \Rightarrow \text{basetype}(\text{env. Type}(t) \downarrow 1)(\text{env}) \]

\[ \text{restriction}(a, p) \Rightarrow \text{basetype}(a)(\text{env}) \]

\[ \text{otherwise} \Rightarrow t \]

\text{esac}

The type determination function is

\[ \text{type_of}(\text{expr})(\text{env}) = \]

\[ \text{cases} \ \text{expr} \ \text{is} \]

\[ \text{numeral} \Rightarrow "\text{integer}" \]

\[ \text{floating point numeral} \Rightarrow "\text{float}" \]

\[ \text{quoted character} \Rightarrow "\text{ascii}" \]

\[ "\text{true}" , "\text{false}" \Rightarrow "\text{bool}" \]
identifier \Rightarrow \text{if } \text{env.Const } [ expr ] \ne ? \text{ then } \text{env.Const } [ expr ] \downarrow 1 \text{ else } \text{env.Var } [ expr ] \downarrow 1 \text{ fi}

a \ op \ b \Rightarrow \text{let lub = lub.type (type.of } [\ a ] (\text{env}), \text{type.of } [\ b ] (\text{env}))(\text{env})\text{ in}
cases \ op \ is
  "+", "-", "." \Rightarrow \text{if lub }<\text{"integer" then "integer"}
  \text{elseif lub }<\text{"float" then "float"}
  \text{elseif lub }=\text{set(t) then set(t) }
  \text{else }
  !

  "/" \Rightarrow \text{if lub }<\text{"float" then "float"}
  \text{else }

  "div", "mod" \Rightarrow \text{if lub }<\text{"integer" then "integer"}
  \text{else }

  "and", "or" \Rightarrow \text{if lub }<\text{"bool" then "bool"}
  \text{else }

  "member" \Rightarrow \text{if set(type.of } [\ a ] (\text{env})) < \text{type.of } [\ b ] (\text{env})\text{ then "bool" else }

  "=", "<>" \Rightarrow \text{if lub }=\ ? \text{ then }
  \text{else "bool" fi}

  "<", "\le", ">", ">=" \Rightarrow
  \text{if basetype(lub)} = \text{"integer" or}
  \text{basetype(lub)(env)} = \text{"ascii" or}
  \text{basetype(lub)(env)} = \text{enumeration(t1,...,tn)}
  \text{then "bool"}
  \text{else }
  !

  "@" \Rightarrow \text{if basetype(type.of } [\ a ] (\text{env}))(\text{env}) = \text{map(d2,r2) and}
  \text{basetype(type.of } [\ b ] (\text{env}))(\text{env}) = \text{map(d1,r1)}
  \text{and } r_1 < d_2
  \text{then map(d1,r2)}
  \text{else }
  !
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
  then type_of [f] (env)
  elseif basetype (type_of [f] (env))(env) =
    record((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 

\[
\begin{align*}
\text{if } \text{typeOf}\ [\ b\ ](\text{env}) < "\text{bool}" \\
\text{then lub\ type} (\text{typeOf} [\ e_1\ ](\text{env}), \text{typeOf} [\ e_2\ ](\text{env})) (\text{env}) \\
\text{else } ? \ 
\end{align*}
\]

let \( x_1:t_1 == e_1, \ldots, x_n:t_n == e_n \) in \( e \) endlet 

\[
\begin{align*}
\text{if } \text{dcheck} (\text{const} \ [\ x_1:t_1 == e_1, \ldots, x_n:t_n == e_n\ ])(\text{env}) \\
\text{then type~f} (\text{env} @\text{const}[x:(\text{e~f}\ t\ ],?])) < "\text{bool}"
\text{then set}(t) \text{ else } ? \ 
\end{align*}
\]

\[
x:t | p \ 
\Rightarrow \text{if typeOf} [\ p\ ](\text{env} @\text{const}[x:(\text{e~f}\ t\ ],?])) < "\text{bool}"
\text{then set}(t) \text{ else } ? \ 
\]

\[
\begin{align*}
(e_1, \ldots, e_n) \Rightarrow \text{if } \bigwedge_{i=1}^{n} \text{typeOf} [\ e_i\ ](\text{env}) \neq ? \\
\text{then product}(\text{typeOf} [\ e_1\ ](\text{env}), \ldots, \text{typeOf} [\ e_n\ ](\text{env})) \\
\text{else } ? \ 
\end{align*}
\]

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

\[
a\#b \quad \text{where } a \text{ is an identifier } \Rightarrow \\
\text{if } \text{typeOf} [\ b\ ](\text{env}) \neq ? \text{ then union}((a, \text{typeOf} [\ b\ ](\text{env}))) \text{ else } ? \ 
\]

\[
\text{function } (x:t) \text{ <env decl> <var decls> do <stmt> return e end } \Rightarrow \\
\text{if } \text{dcheck} (\text{const} \ x==\text{initial}(t))(\text{env}) \text{ and } \\
\text{dcheck} [\text{<env decl> } ](e_1) \text{ and } \\
\text{dcheck} [\text{<var decls> } ](e_2) \text{ and } \\
\text{pcheck} [\text{<stmt> } ](e_3)\} \text{ and } \\
\text{typeOf} [\ e\ ](\text{env}) \neq ? \\
\text{then map}(\text{e~f}\ t\ ](\text{env}), \text{typeOf} [\ e\ ](e_3)) \\
\text{else } ? \ 
\]

\[
\text{where } e_1 = \text{env} @\text{const}[x:(\text{e~f}\ t\ ](\text{env}),?] \\
e_2 = m_{\text{decl}} [\text{<env decl> } ](e_1) \\
e_3 = m_{\text{var}} [\text{<var decls> } ](e_2)\} 
\]
unioncase e of $i_1 \# a_1 : t_1$ in $e_1$ | \cdots | $i_n \# a_n : t_n$ in $e_n$ endcase ⇒

if $type \_ of \ [e \ ] (env) < \text{union}((i_1, t_1), \ldots, (i_n, t_n))$

then if $\bigwedge_{j=1}^{n} \text{dcheck} | \text{const} a_j : t_j = \text{initial}(t_j) | (env)$

then $\bigvee_{j=1}^{n} (type \_ of \ [e_j \ ] (env \ @ \text{const}[^{"a_j"\ : \ (t_j, ?)}]))$

else ? fi

else ? fi

where $\bigvee_{j=1}^{n} (t_j) = \text{if } n = 1 \text{ then } t_1$

$\quad \text{else } \text{lub \_ type} (\bigvee_{j=1}^{n-1} (t_j), t_n) \text{ fi}$

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.

$$\text{value_of} [ t \text{'s } cnv \text{ expr }] (env) (val) =$$

$$\text{convert} (e \text{ type } [ t ]) (env) (\text{value_of} [ \text{expr}] (env) (val))$$

where $convert \in T \rightarrow \text{Env } \rightarrow O \rightarrow O$

$$= \lambda t. \lambda env. \lambda v. \text{cases } t \text{ is}$$

"float", "integer", "bool", "ascii", enumeration($id_1, ..., id_n$) $\Rightarrow v$

identifier $\Rightarrow \text{convert} (\text{env. Type}(t))(env) (v)$

restriction($A, p$) $\Rightarrow \text{if } \text{value_of} [ p \ v ] (e)[[ ]]

\quad \text{then } v \text{ else } ?$ fi

product($t_1, ..., t_n$) $\Rightarrow (\text{convert}(t_1)(env)(v_1), ..., \text{convert}(t_n)(env)(v_n))$

record({"$i_1$."$t_1$},...,{"$i_n$."$t_n$}) $\Rightarrow$

$[i_1: \text{convert}(t_1)(env)(v.i_1), ..., i_n: \text{convert}(t_n)(env)(v.i_n)]$

union({"$i_1$."$t_1$},...,{"$i_n$."$t_n$}) $\Rightarrow$

cases $v_1$ is

$i_1 \Rightarrow i_1 \# \text{convert}(t_1)(env)(v_2)$

$:$

$i_n \Rightarrow i_n \# \text{convert}(t_n)(env)(v_2)$

esac

map($d, r) \Rightarrow \text{convert}(r)(env) @ v @ (\text{convert}(d)(env))$
set $(t') \Rightarrow \lambda x. \text{if not } predicate\_of\ (t')(env)(x) \text{ then } ? \text{ else } v(x) \text{ fi}

where $predicate\_of\ (t')(env) =$

\[
\text{cases } t \text{ is} \\
\text{"integer", "bool", "ascii" } \Rightarrow \lambda x. x \\
\text{enumeration(id}_1,...,id_n) \Rightarrow \lambda x. x \\
\text{restriction}(s,p) \Rightarrow \lambda \xi. p \\
\text{identifier } \Rightarrow \text{ predicate\_of } (env.\text{Type}(t))(env)
\]

otherwise $\Rightarrow ?$

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.

\[
dcheck \ (\text{env id } [<\text{import decl}><\text{environment}>] ) (env) = \\
\text{dcheck } [\text{<environment>}] \\
\text{(m\_decl } \text{ env id } [<\text{import decl}><\text{environment}>] ) (env))
\]

9.5.1. Type checking of declarations

\[
dcheck \ (\text{const } c_1: t_1 == e_1,... c_n: t_n == e_n ) (env) = \\
\bigwedge_{i=1}^{n} \text{dcheck } [t_i] (env) \text{ and} \\
(type\_of \ [e_i](env) < e\_type \ [t_i](env))
\]

\[
dcheck \ (\text{var } v_1: t_1 == e_1,... v_n: t_n == e_n ) (env) = \\
\bigwedge_{i=1}^{n} \text{dcheck } [t_i] (env) \text{ and type\_of \ [e_i](env) < e\_type \ [t_i](env))}
\]

\[
dcheck \ (\text{type } t_1: s_1,... t_n: s_n ) (env) = \\
\bigwedge_{i=1}^{n} \text{dcheck } [s_i] (env)
\]

\[
dcheck \ (\text{class C of type } t_1,... \text{ const } c_1: s_1,... == \\
\text{def } k_1: r_1,... k_n: r_n \text{ rep } <\text{environment}> \text{ endclass} ) (env) = \\
dcheck \ (\text{const } c_1: s_1,... ] (e_1) \text{ and} \\
dcheck \ (\text{const } k_1: r_1,... k_n: r_n ] (e_2) \text{ and} \\
dcheck \ (\text{<environment> } ] (e_2) \text{ and}
\]
\[
\bigwedge_{i=1}^{n} \text{subtype}(e_3[k_i], e_4, \text{Const}[k_i \downarrow 1](e_2))
\]

where \(e_1 = \text{env} \oplus \text{Type} \sum_i ["t_i" : "t_i"]\)

\[e_2 = m_{\text{decl}} \{ \text{const } c_1 : s_1 = "c_1", \ldots \}(e_1)\]

\[e_3 = \sum_i [k_i : \text{e_type}[\tau_i](e_2)]\]

\[e_4 = m_{\text{decl}} \{ \langle \text{environment} \rangle \}(e_2)\]

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 \text{def} list.

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

\[
dcheck(t)(\text{env}) =
\]

\[
\text{cases } t \text{ is}
\]

"set of \(s\)" \(\Rightarrow \text{is_scalar}[s](\text{env})\)

"\(C\) of [\text{type} \(i_1 : t_1, \ldots, i_m : t_m\ \text{const } c_1 : s_1 = e_1, \ldots, c_n : s_n = e_n\)]" \(\Rightarrow\)

\[
dcheck[\text{type} \(i_1 : t_1, \ldots, i_m : t_m\]
\]

\[
\text{const } c_1 : s_1 = e_1, \ldots, c_n : s_n = e_n \}(\text{env}) \text{ and}
\]

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

\[
\bigwedge_{j=1}^{n} \text{subtype}(e_{\text{type}}[s_j](\text{env'}), (d.C \downarrow 2)[c_j])(\text{env'})
\]

where \(d = \text{env}.\text{Class}[C](\text{env'})\), and

\[
\text{env'} = m_{\text{decl}}[\text{type} \(i_1 : t_1, \ldots, i_m : t_m\]
\]

\[
\text{const } c_1 : s_1 = e_1, \ldots, c_n : s_n = e_n \}(\text{env})
\]

otherwise \(\Rightarrow \text{true}\)

esac

where \(\text{is_scalar}(t)(\text{env}) = \text{cases } t \text{ is}\)

"\(\text{id}\)" \(\Rightarrow \text{is_scalar}(\text{env}.\text{Type}[\text{id}])\)(\text{env})

\text{restriction}(s.p) \(\Rightarrow \text{is_scalar}(s)\)(\text{env})

\text{enumeration}("\text{id}_1", \ldots, "\text{id}_n") \(\Rightarrow \text{true}\)

"integer", "bool", "ascii" \(\Rightarrow \text{true}\)

otherwise \(\Rightarrow \text{false}\)

esac
9.6. Type checking of statements

Here we use the notation $e'$ to stand for $\text{type_of}(e)(\text{env})$, where $e$ is an expression.

$pcheck \{ \text{skip} \}(\text{env})(\Lambda) = \text{true}$

$pcheck \{ \text{id} := e \}(\text{env})(\Lambda) = e' < \text{env}.\text{Var}[\text{id}]$)

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

$pcheck \{ s_1; s_2 \}(\text{env})(\Lambda) = pcheck \{ s_1 \}(\text{env})(\Lambda) \text{ and } pcheck \{ s_2 \}(\text{env})(\Lambda)$

$pcheck \{ \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ endif} \}(\text{env})(\Lambda) =
\quad e' < \text{Bool} \text{ and } pcheck \{ s_1 \}(\text{env})(\Lambda) \text{ and } pcheck \{ s_2 \}(\text{env})(\Lambda)$

$pcheck \{ \text{unioncase } e \text{ of } id_1 : t_1 \text{ do } s_1 | \ldots | \text{otherwise do } s_{n+1} \text{ endcase} \}(\text{env})(\Lambda) =
\quad e' < \text{union}(("id_1", e\text{-type } t_1), \ldots, ("id_n", e\text{-type } t_n)) \text{ and } \bigwedge_{j=1}^{n} pcheck \{ s_i \}(\text{env} \oplus \text{onst} ["c_1": e\text{-type } t_1]) (\Lambda)$

and $pcheck \{ s_{n+1} \}(\text{env})(\Lambda)$

$pcheck \{ \text{for } id : t \text{ incr } e_1 \text{ from } e_2 \text{ do } s \text{ endfor} \}(\text{env})(\Lambda) =
\quad \text{type_of } [e_1'](\text{env}) < \text{set}(e\text{-type } t) \text{ and } \text{type_of } [e_2'](\text{env}) < e\text{-type } t \text{ and } pcheck \{ s \}(\text{env} \oplus \text{onst} ["e_1": e\text{-type } t]) (\Lambda)$

$pcheck \{ \text{id} : s :: \text{id}_2 \}(\text{env})(\Lambda) = (\text{id}_1 = \text{id}_2) \text{ and } pcheck \{ s \}(\text{env})(\Lambda \cup \{\text{id}_1\})$

$pcheck \{ \text{repeat } \text{id} \}(\text{env})(\Lambda) = "\text{id}' \in \Lambda$

$pcheck \{ \text{let } id_1 : t_1 ::= e_1, \ldots, id_n : t_n ::= e_n \text{ do } s \text{ endlet} \}(\text{env})(\Lambda) =
\quad pcheck \{ \text{const } id_1 : t_1 ::= e_1, \ldots, id_n : t_n ::= e_n \}(\text{env}) \text{ and } pcheck \{ s \}(\text{env'})(\Lambda)$

where $\text{env'} = \text{m-decl} [\text{const } id_1 : t_1 ::= e_1, \ldots, id_n : t_n ::= e_n ](\text{env})$

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]

[McS82]

[Mi78]

[SS71]

[vW75]