Language Design for CGRA project. Design 11.

Theodore S Norvell
Electrical and Computer Engineering
Memorial University

January 2, 2019

Abstract

Abstract to be done.

Change History

- 2006 Sept 18 Version 3
- 2007 August. Version 4.
- 2007 November. Version 5. Added constants
- Changed syntax of elseif
- Allowed name at the end of a class or interface
- 2008 Nov 3. Version 6
  - Replaced atomic command with “with command”
  - Added section on parallel access to locations
- 2011. July. Added information on expressions
- 2013 Summer. Changes related to implementation. Including
  - More detail on expressions
  - Minor changes to syntax
  - Moved constructor parameters
- 2015
  - More information on expressions
- 2018
  - Adding “this” keyword.

Meta notation

$N \rightarrow E$	An N can be E
$\underline{(}E\underline{)}$	Grouping
$E\ F$	An E followed by an F
E^*	Zero or more Es
E^{*F}	Zero or more Es separated by Fs
E^+	One or more Es
E^{+F}	One or more Es separated by Fs
$E^?$	Zero or one Es
$\underline{[}E\underline{]}$	Zero or one Es
$E \mid F$	Either an E or a F

1 Classes and Objects

1.1 Programs

A program is a sequence of classes, interfaces, and objects.

$$\mathit{Program} \rightarrow \underline{(}\mathit{ClassDecl} \mid \mathit{IntfDecl} \mid \mathit{ObjectDecl} \mid ;\underline{)}^* \ \mathbf{eof}$$

If multiple files are compiled together the sequence is simply the concatenation of the sequences in the individual files.

1.2 Types

Types come in several categories.

- Primitive types: Primitive types represent sets of value. As such they have no mutators. However objects of primitive types may be assigned to to change their values. Primitive types represent such things as numbers. They include
  - Int8, Int16, Int32, Int64
  - Real16, Real32, Real64
  - Bool

The names of the types specify the minimum number of bits. However they do not specify the exact number of bits. E.g. int16 might be represented by 32 bit locations. The names of the primitive types are not reserved words.

- Classes: Classes represent sets of objects. As such they support methods that may change the object's state.
- Interfaces. Interfaces are like classes, but without the implementation.
- Arrays: Arrays may be arrays of primitives or arrays of objects.

- Generic types. Generic types are not really types at all, but rather functions from types to types. In order to be used, generic types must be instantiated.

Types are either names of classes, names of interfaces, array types or specializations of generic types

$$Type \rightarrow Name \mid Type\ GArgs \mid Type\ [\ Bounds\ ]$$

Arrays are 1 dimensional and indexed from 0, so the bounds are simply one number

$$Bounds \rightarrow ConstExp$$

1.2.1 Restrictions

The bounds of an array must be an integral expression greater than or equal to zero.

For a type of the form *Type GArgs*,the name must refer to a generic type and the generic arguments must be the same in number as the type's parameters. Each argument must not be generic.

TODO More to say?

1.3 Objects

Objects are named instances of types.

$$ObjectDecl \rightarrow (\textbf{const} \mid \textbf{obj})\ Name\ \underline{[}\ :\ Type\underline{]} := InitExp$$

1.3.1 Restrictions

If the type is missing it is considered to be the same as that of the *InitExp*. If the type is not missing, it must be a supertype of the type of the *InitExp*. It follows that the type may not be generic.

If the type (or inferred type) is primitive, the declaration creates a named location. If the type is an array type, the declaration names an array of locations. If the type is a class, the declaration names an object.

If the declaration is **const**, the type must be primitive and the named location is a constant location that may not be changed.

1.4 Initialization expressions

1.4.1 Constant initialization expression

$$InitExp \rightarrow ConstExp$$

The type of the *InitExp* is the same as the *ConstExp* in the first case.

1.4.2 Array Initialization Expressions

$$\mathit{InitExp} \to \Big(\textbf{for}\ \mathit{Name} : \mathit{Bounds}\ \textbf{do}\ \mathit{InitExp}\ \underline{[}\textbf{for}\underline{]}\Big)$$

The type of an *ArrayInit* is array k of t where k is the value of the bounds and t is the value of the *InitExp*.

1.4.3 New Initialization Expressions

$$\mathit{InitExp} \to \textbf{new}\ \mathit{Type}(\mathit{ConstExp}^{*,})$$

In this case the *Type* must be a nongeneric class type, the number of arguments must match the number of parameters for the class's constructor. For each parameter (whether **in** or **obj**), the argument's type must be a subtype of the parameter's type.

1.4.4 Choice Initialization Expressions

$$\mathit{InitExp} \to \Big(\textbf{if}\ \mathit{ConstExp}\ \textbf{then}\ \mathit{InitExp}\ \underline{(}\textbf{else if}\ \mathit{ConstExp}\ \textbf{then}\ \mathit{InitExp}\underline{)}^{*}\ \textbf{else}\ \mathit{InitExp}\ \underline{[}\textbf{if}\underline{]}\Big)$$

The type of the expression is the least supertype of all the *InitExp*s.

Restrictions The type of each *ConstExp* must be Bool.
If no such supertype exists, then there is an error.

1.5 Classes and interfaces

Each class declaration defines a family of types. Classes may be generic or nongeneric. A generic class has one or more generic parameters

$$\begin{aligned}\mathit{ClassDecl} \to {} & (\textbf{class}\ Name\ GParams^{?}\ (CParam^{*,})\ \underline{(}\textbf{implements}\ Type^{+,}\underline{)}^{?} \\ & \underline{(}ClassMember\underline{)}^{*}\ \underline{[}\textbf{class}\ \underline{[}Name\underline{]}\underline{]}\underline{)}\end{aligned}$$

- The $Name$ is the name of the class.
- The $GParams$ is only present for generic classes, which will be presented in a later section.
- The *Type*s are the interfaces that the class implements.

An interface defines a type. Interfaces may be generic or nongeneric. A generic interfaces has one or more generic parameters

$$IntfDecl \to \Big(\textbf{interface}\ Name\ GParams^{?}\ \underline{(}\textbf{extends}\ Type^{+,}\underline{)}^{?}\ \underline{(}IntfMember\underline{)}^{*}\ \underline{[}\textbf{interface}\ \underline{[}Name\underline{]}\underline{]}\Big)$$

- The $Name$ is the name of the class.

- The *GParams* will be presented in a later section.
- The *Type*s are the interfaces that the interface extends.

Constructor parameters represent objects to which this object is connected.

$$CParam \to \textbf{obj}\ Name : Type \mid \textbf{in}\ Name : Type$$

- Object parameters represent named connections to other objects. So for example if we have

  (class B(obj x : A) ...)
  obj a := (for i : 10 do new A())
  obj b := (for i : 10 do new B(a(i))

  Then object `b[0]` knows object `a[0]` by the name of `x`.

- In parameters are compile time constants and the corresponding argument must be such.

1.6 Class Members

Class members can be fields, methods, and threads. [Nested classes and interfaces are a possibility for the future.]

$$ClassMember \to Field \mid Method \mid Thread \mid ;$$

Fields are objects, arrays, or locations that are within objects. Field declarations therefore define the part/whole hierarchy.

$$\begin{aligned} Field &\to Access^{?}\ ObjectDecl \\ Access &\to \textbf{private} \mid \textbf{public} \end{aligned}$$

Method declarations declare the method, but not its implementation. The implementation of each must be embedded within a thread.

$$\begin{aligned} Method &\to Access\ \textbf{proc}\ Name(\underline{(}Direction\ Name :\ Type\underline{)}^{*,})\underline{]} \\ Direction &\to \textbf{in} \mid \textbf{out} \end{aligned}$$

The types of method parameters must be primitive.
Recommended order of declarations is

- public methods and fields, followed by
- private methods and fields, followed by
- threads.

There is no ‘declaration before use rule’. Name lookup works from inside out.

1.7 Interface Members

Interface's members can be fields and methods. [Nested classes and interfaces are a possibility for the future.]

$$IntfMember \rightarrow \mathit{Field} \mid \mathit{Method} \mid ;$$

2 Threads

Threads are blocks executed in response to object creation.

$$\mathit{Thread} \rightarrow (\textbf{thread}\ \mathit{Block}\ \underline{[}\textbf{thread}\underline{]})$$

Each object contains within it zero or more threads. Coordination between the threads within the same object are the responsibility of the programmer. All concurrency within an object arises from the existence of multiple threads in its class. Thus you can write a monitor (essentially) by having only one thread in a class.

2.1 Statements and Blocks

A block is simply a sequence of statements and semicolons

$$\mathit{Block} \rightarrow \mathit{Command\ Block} \mid \mathit{LocalDeclaration} \mid ;\ \mathit{Block} \mid$$

Statements as follow

- Assignment statements

$$\begin{aligned} \mathit{Command} &\rightarrow (\mathit{ObjectIds}) := (\mathit{Expressions}) \\ \mathit{Command} &\rightarrow \mathit{ObjectId} := \mathit{Expression} \\ \mathit{ObjectIds} &\rightarrow \underline{(}\mathit{ObjectId}\underline{)}^{+,} \\ \mathit{Expressions} &\rightarrow \underline{(}\mathit{Expression}\underline{)}^{+,} \end{aligned}$$

  The *ObjectId* must represent a location. The corresponding expression must be convertable to that location's type via a widening conversion. If the same location occurs twice or more in an assigment, the expression must all have the same value. E.e. $a[i], a[j] := a[j], a[i]$ is a legitimate asssigment, since in the case that $i = j$, the two expressions will have the same value.

- Local variable declaration

$$\mathit{LocalDeclaration} \rightarrow (\textbf{obj} \mid \textbf{const})\ \mathit{Name}\underline{[} : \mathit{Type}\underline{]} := \mathit{Exp\ Block}$$

  Note that the initialization expression is a regular expression, not an init-Exp. The initialization is performed each time the declaration is encountered. The type may be omitted, in which case it is inferred from the initialization expression. In either case, the type must be primitive. The scope of a local variable name is the block that follows it. Local variables marked **const** may not be assigned to. Names must be unique within the thread.

- Method call statements

$$\begin{aligned} \mathit{Command} &\rightarrow \mathit{ObjectId.Name}(\mathit{Args}) \\ &\mid \mathit{Name}(\mathit{Args}) \\ \mathit{Args} &\rightarrow \underline{[}\mathit{Expressions}\underline{]} \end{aligned}$$

  Restrictions:

  - For the first syntax, the type of the *ObjectId* must be of class or interface type and the *Name* must name a public method of that type (including inherited methods).
  - For the second syntax, the *Name* must be the name of a (public or private) method of the class or interface type currently being defined.
  - The number of argument must equal the number of parameters declared for the method.
  - For each **in** parameter of the method, the corresponding argument must have either the same type as the parameter's type or a type that can be widened to the parameter's type.
  - For each **out** parameter of the method, the argument must refer to a location and the type of that location must be the same as the type of the argument, or type that can be widened to the argument type.

- Sequential control flow

$$\begin{aligned} \mathit{Command} \rightarrow & \Big(\textbf{if}\ \mathit{Expression}\ \textbf{then}\ \mathit{Block}\ \underline{(}\textbf{else if}\ \mathit{Expression}\ \textbf{then}\ \mathit{Block}\underline{)}^{*}\underline{(}\textbf{else}\ \mathit{Block}\underline{)}^{?}\ \underline{[}\textbf{if}\underline{]}\Big) \\ & \mid \Big(\textbf{while}\ \mathit{Expression}\ \textbf{do}\ \mathit{Block}\ \underline{[}\textbf{while}\underline{]}\Big) \\ & \mid \Big(\textbf{for}\ \mathit{Name} : \mathit{Bounds}\ \textbf{do}\ \mathit{Block}\ \underline{[}\textbf{for}\underline{]}\Big) \end{aligned}$$

- Parallelism (note that the 2 vertical bars here is a terminal)

$$\begin{aligned} \mathit{Command} \rightarrow & \Big(\textbf{co}\ \mathit{Block}\ \underline{(}||\ \mathit{Block}\underline{)}^{+}\ \underline{[}\textbf{co}\underline{]}\Big) \\ & \mid \Big(\textbf{co}\ \mathit{Name} : \mathit{Bounds}\ \textbf{do}\ \mathit{Block}\ \underline{[}\textbf{co}\underline{]}\Big) \end{aligned}$$

  In the second case, the *Bounds* must be compile-time constant.

- Method implementation (note that the vertical bar here is a terminal)

$$\begin{aligned} \mathit{Command} &\rightarrow \Big(\textbf{accept}\ \mathit{MethodImp}\ \underline{(}|\ \mathit{MethodImp}\underline{)}^{*}\ \underline{[}\textbf{accept}\underline{]}\Big) \\ \mathit{MethodImp} &\rightarrow \mathit{Name}(\ \underline{(}\mathit{Direction}\ \mathit{Name} : \mathit{Type}\underline{)}^{*,}\ )\ \underline{[}\mathit{Guard}\underline{]}\ \mathit{Block}_0\ \underline{[}\textbf{then}\ \mathit{Block}_1\underline{]} \\ \mathit{Guard} &\rightarrow \textbf{when}\ \mathit{Expression} \end{aligned}$$

  - Restrictions
    - The names, directions and types must match the declaration.
    - The guard expression must be boolean.
    - Each method may only be implemented once per class
  - Possible restrictions:
    - The guard may not refer to any parameters.
    - The guard may refer only to the in parameters.
  - Semantics: A thread that reaches an accept command must wait until there is a call to one of the methods it implements and the corresponding guard is true. Once there is at least one method the accept can execute, one is selected. Input parameters are passed in, Block_0 is executed and finally the output parameters are copied back to the calling thread. If there is a Block_1 it is executed next.

- Locking

$$\mathit{Command} \rightarrow \Big(\textbf{with}\ \mathit{Exp}\ \underline{[}\mathit{Guard}\underline{]}\ \textbf{do}\ \mathit{Block}\ \underline{[}\textbf{with}\underline{]}\Big)$$

  - Restrictions:
    - The *Exp* must refer to an object of type Lock.
    - The guard expression must be boolean
  - Semantics:
    - The block (including the guard) is executed as if atomically with respect to other with commands sharing the same lock
    - If there is no guard, it defaults to true.
    - If the guard is false, then the lock is unlocked and then everything starts again.
    - In summary the sematics is like this

      lock(*Exp*)
      (**while not** *Guard* **do** unlock(*Exp*) lock(*Exp*))
      Block
      unlock(*Exp*)

3 Parallel access to data locations

Each object of a primitive type, including array items, is a separate location. Access to a location is either a read access or a write access. Accesses are not considered to be atomic, but rather to take a span of time. As such two accesses that could overlap in time must not be to the same location — except that we will allow read accesses to overlap. If two accesses could overlap in time, we say that they "could happen at the same time". Suppose that a and b are two accesses from separate threads that could happen at the same time

- Parallel read accesses are allowed. If a and b are both reads, behaviour is well defined.
- Parallel write accesses are not allowed. If a and b are both writes, then behaviour is undefined.
- Parallel read and write accesses are not allowed. If a is a read and b is a write (or the other way around), then behaviour is undefined.

The compiler may or may not diagnose undefined behaviour. The reason is that aliasing makes it impossible to tell for sure whether the behaviour of a program is well defined or undefined. Consider

(**co** $a[i] := 0 \mid\mid a[j] := 0$ **co**)

In states where $i = j$, behavour is undefined. In states where $\not i \neq j$, but where i and j are in bounds, behaviour is well defined. Since the values of i and j are (in general) unknowable at compile time, the compiler is not in a position to diagnose undefined behaviour, although a good compiler may warn that it can not rule out undefined behaviour. Note that the fact that both statements are writing the same value makes no difference to whether the parallel accesses are well-defined.

The programmer may prevent accesses from occuring at the same time using locks:

(**co** (**with** l $a[i] := 0$) || (**with** l $a[i] := 1$))

In this example, the parallel accesses are well defined because they can not take place at the same time.

Accesses may also be protected via other synchronization mechanisms. For example

(**co** $s.p()$ $a[i] := 0$ $s.v()$ || $s.p()$ $a[i] := 1$ $s.v()$ **co**)

Is well defined if the p and v methods of object s somehow prevent the accesses from occurring at the same time. Method calls serve as "synchonization points".

Here is one more example.

obj l : *Lock* := **new** *Lock*()

```
**obj** $s := 1$
(**co**
    (**with** $l$ **when** $s = 1$ $s := 0$ )
    $a[i] := 0$
    (**with** $l$ $s := 1$)
||
    (**with** $l$ **when** $s = 1$ $s := 0$ )
    $a[i] := 1$
    (**with** $l$ $s := 1$)
**co**)
```

In this case, the assignments to $a[i]$ can not happen at the same time and so the result is well defined (though nondeterministic). In particular the compiler can not move the assignment to $a[i]$ any earlier or later in the threads. This means that with statements, like method calls, serve as "synchronization points."

One might at first think that there is no need to protect the assignments $s := 1$, in the previous example, with **with** statements. This is not so; there is a read access in the other process that could happen at the same time. The following program's behaviour is undefined.

```
**obj** $l$ : *Lock*
**obj** $s := 1$
(**co**
    (**with** $l$ **when** $s = 1$ $s := 0$ )
    $a[i] := 0$
    $s := 1$ // Wrong. Access is unprotected.
||
    (**with** $l$ **when** $s = 1$ $s := 0$ )
    $a[i] := 1$
    $s := 1$ // Wrong! Access is unprotected.
**co**)
```

A compiler optimizing this program, might look at the sequence $a[i] := 1\, s := 1$ and decide to reorder the statements to $s := 1\, a[i] := 1$ and that would be legitimate in the sense that any change to an undefined program can not make it more wrong.

While sequential programming constructs impose an nominal order on execution, a compiler is welcome to reorder accesses that are to different locations (or that are both read accesses). For example the program $a := 1\, b := 1$ says that a should be written first, nominally. However as a and b are different locations, the command can be rewritten to $b := 1\, a := 1$. The compiler can assume that there are no parallel accesses to a or b that could happen at the same time, as such accesses would make the program undefined and there is nothing the compiler can do to an undefined program to make it more wrong. Thus the compiler is welcome to make any sequential optimizations to code that appears between synchronization points. The following are synchronization points:

- The start of a with command.
- The end of a with command
- The start of an accept command
- The return from an accept command
- Any method call (after argument evaluation)

4 Expressions

4.1 Syntax

Expressions

$$
\begin{aligned}
\mathit{Exp} &\rightarrow \mathit{Exp}0\ \underline{\underline{(}}\underline{(}\text{=>} \mid \text{<=} \mid \text{<=>}\underline{)}\ \mathit{Exp}0\underline{)}^{*}\ \underline{[}\textbf{as}\ \mathit{Id}\underline{]}\\
\mathit{Exp}0 &\rightarrow \mathit{Exp}1\ \underline{\underline{(}}\underline{(}\backslash/ \mid \text{or}\underline{)}\ \mathit{Exp}1\underline{)}^{*}\\
\mathit{Exp}1 &\rightarrow \mathit{Exp}2\ \underline{\underline{(}}\underline{(}/\backslash \mid \text{and}\underline{)}\ \mathit{Exp}2\underline{)}^{*}\\
\mathit{Exp}2 &\rightarrow \mathit{Exp}3 \mid \text{not}\ \mathit{Exp}2 \mid \tilde{}\ \mathit{Exp}2\\
\mathit{Exp}3 &\rightarrow \mathit{Exp}4\ \underline{\underline{(}}\underline{(}= \mid \tilde{}= \mid \text{not=} \mid < \mid \_< \mid >\_ \mid >\underline{)}\ \mathit{Exp}4\underline{)}^{*}\\
\mathit{Exp}4 &\rightarrow \mathit{Exp}5\ \underline{\underline{(}}\underline{(}+ \mid -\underline{)}\ \mathit{Exp}5\underline{)}{*}\\
\mathit{Exp}5 &\rightarrow \mathit{Primary}\ \underline{\underline{(}}\underline{(}* \mid / \mid \text{div} \mid \text{mod}\ \underline{)}\ \mathit{Primary}\underline{)}^{*}\\
\mathit{Exp}6 &\rightarrow -\mathit{Exp}6 \mid \mathit{Exp}7\\
\mathit{Exp}7 &\rightarrow \mathit{Primary}\\
\mathit{Primary} &\rightarrow (\mathit{Exp}) \mid \mathit{ObjectId} \mid \mathit{Literal}\\
\mathit{ObjectId} &\rightarrow \mathit{Name} \mid \mathit{ObjectId}[\mathit{Expression}] \mid \mathit{ObjectId}.\mathit{Name} \mid \textbf{this}
\end{aligned}
$$

Operator precedence is as above. Binary operators

=> <= <=> \/ or /\ and + - * / div mod

are left associative. The following are synonymous (see also section7).

=>		$\Rightarrow$	
<=		$\Leftarrow$	
<=>		$\Leftrightarrow$	
\/	or	$\vee$	
/\	and	$\wedge$	
~=	not=	$\neq$	$\neg =$
~	not	$\neg$	
_<		$\leq$	$\leqslant$
>_		$\geq$	$\geqslant$

Binary operators

```
=   ~=   <   _<   >   >_
```

are "chaining", meaning that any sequence of these operators is 'anded' together. For example expression

$$\text{a \_< b < c = d \~{}= e}$$

is logically equivalent to

$$\text{a \_< b and b < c and c = d and d \~{}= e}$$

except that for the former, all operands are first converted to a common type.

Expressions are not necessarily evaluated from left to right or in any particular order. Associativity of associative operators may or may not be respected. Logical operators are not (necessarily) short-circuiting.

A constant expression is the same as an expression except it must be evaluable at elaboration time. Thus a constant expression can depend on **in** parameters of the constructor.

$$ConstExp \rightarrow Exp$$

4.2 Object ids

An object id that is a Name should refers to a local variable, a field, array, or object. An object id of the form *ObjectId*[*Expression*] refers to an item of an array. An object id of the form *ObjectId.Name* refers to a field of an object. The object id **this** refers to the current object.

4.3 Literals

Decimal integer literals are written as a series of decimal digits with any number of internal underlines. For example "0", "123", "123_456".

Integer literals in bases 2, 8, and 16 are also allowed. These are written with the base first, followed by a # symbol, followed by the digits that make up the literal. Again underscores can appear internally to provide spacing. Here is the lexical syntax of integer literals

$$
\begin{aligned}
DecimalLiteral &\rightarrow D \mid D(D \mid \_)^{*} D \\
HexLiteral &\rightarrow \texttt{16\#}(H \mid H(H \mid \_)^{*} H \\
OctLiteral &\rightarrow \texttt{8\#}(O \mid O(o \mid \_)^{*} O \\
BinLiteral &\rightarrow \texttt{2\#}(B \mid B(B \mid \_)^{*} B \\
D &\rightarrow 0 \mid 1 \mid \cdots \mid 9 \\
H &\rightarrow 0 \mid 1 \mid \cdots \mid 9 \mid a \mid \cdots \mid f \mid A \mid \cdots \mid F \\
O &\rightarrow 0 \mid 1 \mid \cdots \mid 7 \\
B &\rightarrow 0 \mid 1
\end{aligned}
$$

Floating point (or real) literals are defined by the following lexical rules

$$\begin{aligned}
\textit{RealLiteral} \rightarrow\ & \texttt{.}\ \textit{DecimalLiteral}\ \underline{[}\textit{Exp}\underline{]} \\
\mid\ & \textit{DecimalLiteral}\ \texttt{.}\ \underline{[}\textit{Exp}\underline{]} \\
\mid\ & \textit{DecimalLiteral}\ \texttt{.}\ \textit{DecimalLiteral}\ \underline{[}\textit{Exp}\underline{]} \\
\mid\ & \textit{DecimalLiteral}\ \textit{Exp} \\
\textit{Exp} \rightarrow\ & \underline{(}\texttt{e} \mid \texttt{E}\underline{)}\ \underline{[}+ \mid -\underline{]}\ \textit{DecimalLiteral}
\end{aligned}$$

4.3.1 Restrictions

Decimal, binary, octal, and hex, literals must be representable in 63 bits, i.e. they must be less than $2^{63} = 9,223,372,036,854,775,808$. Real literals must be representable in 64 bits on both the compiling and the target machines.

4.4 Types

The arithmetic primitive types include

- Int8, Int16, Int32, Int64
- Real16, Real32, Real64

and have the following one step widening relationships

- Int8 → Int16 → Int32 → Int64
- Real16 → Real32 → Real64
- Int8 → Real16
- Int16 → Real32
- Int32 → Real64

4.4.1 Object IDs and this

The type of an ObjectID is the discussed in [[section on object IDs above]]

The type of **this** is the same as the class that it occurs in. It is an error for **this** to appear outside of a class declaration.

4.4.2 Literals

The type of any real literal is of type Real64. However, when an initialization of an object consists soley of a real literal, and the object is explicitly typed, the literal is typed as the type of the object, provided that type is Real16 or Real32. Effectively there is an implicit 'as'. For example

obj a : Real32 := 1.0

is equivalent to

obj a : Real32 := 1.0 **as** Real32

The type of any integer literal (i.e. decimal, binary, octal, or hex literal) is Int32 unless its value is over $2^{31} = 2,147,483,648$, in which case it is of type Int64. However, when an initialization of an object consists soley of an integer literal, and the object is explicitly typed, the literal is typed as the type of the object, provided that type is Int8, Int16, or Real16 or Real32. Effectively there is an implicit 'as'. For example

obj a : Real32 := 1 **obj** b : Int8 := 0

is equivalent to

obj a : Real32 := 1 **as** Real32 **obj** b : Int8 := 0 **as** Int8

4.4.3 Binary and chaining operators

When applying binary operations, operands are generally widened to the narrowest common widening (ncw). For example if a Real32 is compared to an Int32, the narrowest common widening is Real64, so both operands are converted to Real64 and are compared at that type. Note that the ncr is not always defined. In particular ncw(Int64, t) is not defined for any $t \in \{\text{Real16}, \text{Real32}, \text{Real64}\}$.

An integral type is any of the types Int8, Int16, Int32, Int64.

Binary operators behave according to this table in which B stands for Boolean, A for arithmetic, I for integral, ncw for narrowest common widening, and ncrw stands for narrowest common real widening.

Operator	Left	Right	Operation Type	Result Type
<= => <=> /\ \/	B	B	B	B
= ~= > < >_ _<	A	A	ncw	B
+ - *	A	A	ncw	ncw
/	A	A	ncrw	ncrw
div mod	I	I	ncw	ncw

For the comparison operations, when multiple operations are chained together, as in $a \leq b < c$, all operands are widened to the narrowest common type before any comparisons are made. If that narrowest common widening does not exist, then it is an error.

For division with /, if both operands are integral, then the result is the narrowest real type that is wider than both operands.

Restrictions:

- A binary operation is not allowed if the types of the operands are not as shown in the above table.

- A binary operation is not allowed if the ncw or ncrw is not defined for the operand types.

4.4.4 Unary operators

Unary operators do not force conversions. The result type is the same as the operand type.

Operator	Left	Result
~	B	B
−	A	A

Restriction:

- A unary operation is not allowed if the type of the operand is not as shown in the above table.

4.4.5 'As' expressions

The 'as' construct forces a value to be reinterpreted as another type. The identifier must be one of the primitive types. Any expression of an arithmetic type can be reinterpreted as a value of any other arithemetic type.

Restrictions:

- The 'as' construct is not allowed if the operand is not of arithmetic type.
- The 'as'construct is not allowed if the identifier on the right is does not resolve to one of the arithmetic types.

4.5 Semantics

Each integral type is associated with a particular range of values. These are

- Int8 $\{-128, .., 127\}$
- Int16 $\{-32\_768, .., 32\_767\}$
- Int32 $\{-2\_147\_483\_648, .., 2\_147\_483\_647\}$
- Int64 $\{-9\_223\_372\_036\_854\_775\_808, .., 9\_223\_372\_036\_854\_775\_807\}$

The semantics of operations on intergral types is simply that the result is the mathematical answer if representable in the ncw type and is not defined otherwise. For example

$$(1 \text{ as Int8}) + (127 \text{ as Int8})$$

is not defined since the ncw type is Int8 and 128 lies outside its range.

Real types similarly have ranges (sets of representable values) associated with them, but these ranges are implementation dependent. We do insist that widening conversions from integral types can be made without loss of information. For example every integer in Int16 must be exactly representable in type Real32, because there is a widening conversion from Int16 to Real32. These ranges have a minimum and maximum value. We say that a type has a value that approximates the real value x if x is between the minimum and maximum values for the type. Arithmetic operations on reals are defined as long as the ncw type has a value that approximates the mathematical value.

The 'as' construct is evaluated as follows.

- Conversions from integral types to integral types is defined only if the value of the operand can be represented in the new type.
- Conversions from real to real types are defined only if the value of the operand can be approximated by a value in the new type.
- Conversions from real types to integral types involve rounding the value of the operand down to the closest integer that is less than or equal to the value of the operand. The conversion is defined only if that integer is representable in the new type.

5 Genericity

Classes and interfaces can be parameterized by "generic parameters". The effect is a little like that of Java's generic classes or C++'s template classes. Classes and interfaces may be parameterized, in general, by other classes and interfaces, values of primitive types, for example integers, and objects.

Programs using generics can be expanded to programs that do not use generics at all. For example a program

```
(class K ... class)
(class G{ type T } ...T... class)
obj g : G{K} := ...
```

Expands to

```
(class K ... class)
obj k : K
(class G0 ...T... class)
obj g : G0 := ...
```

Generic parameters may be one of the following

- Nongeneric Types
- Nongeneric Classes

$$GParams \rightarrow \{\ GParam^{+,}\ \}$$

$$GParam \rightarrow \textbf{type}\ Name\ \underline{[}\textbf{extends}\ Type\underline{]}$$

$$GArgs \rightarrow \{\ Type^{+,}\ \}$$

6 Examples

```
 (class FIFO {type T extends primitive} (in capacity : int16)

  public proc deposit(in value : T)
  public proc fetch(out value : T)

  private obj a : T(capacity)
  private obj front := 0
  private obj size := 0

  (thread
      (wh true
          (accept
              deposit( in value : T ) when size < capacity
                  a[ (front + size] % capacity ) := value
                  size := size + 1
          |
              fetch( out value : T ) when size > 0
                  value := a[front]
                  front := (front + 1) % capacity
                  size := size - 1
          accept)
      wh)
  thread)
class)
```

7 Lexical issues

Comments are either single-line, C++ style

```
//This is a comment
```

or nesting multiline Pascal style

```
(* this (* is also
```

a *) comment *)

The former can be ended by the end of the file. The latter can not.

Some operators have unicode or keyword equivalents.

Ascii	**Keyword**	**Unicode glyph**	**Unicode code point (hexadecimal)**
`/\`	`and`	∧	2227
`\/`	`or`	∨	2228
`~`	`not`	¬	00ac
`~=`	`not=`	≠	2260
`_<`		≤ or ⩽	2264 or 2a7d
`>_`		≥ or ⩾	2265 or 2a7e
`=>`		⇒	21d2
`<=`		⇐	21d0
`<=>`		⇔	21d4
`'`		′	0027, 02B9, 02BC, 2019, 2032, or FF07