Language Design for CGRA project. Design 6 [Draft].

Theodore S Norvell
Electrical and Computer Engineering
Memorial University

October 8, 2010

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 statement with "with statement"
  - Added section on parallel access to locations

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 set of classes, interfaces, and objects.

$$\mathit{Program} \rightarrow \underline{(}\mathit{ClassDecl} \mid \mathit{IntDecl} \mid \mathit{ObjectDecl} \mid \mathit{ConstDecl} \mid ; \underline{)}^{*}$$

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, int
  - real16, real32, real64, real
  - bool
- 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 some domain to types. In order to be used, generic types must be instantiated.

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

$$\mathit{Type} \rightarrow \mathit{Name} \mid \mathit{Name}\ \mathit{GArgs} \mid \mathit{Type}[\mathit{Bounds}]$$

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

$$\mathit{Bounds} \rightarrow \mathit{ConstIntExp}$$

1.3 Objects

Objects are named instances of types.

$$\mathit{ObjectDecl} \rightarrow \mathbf{obj}\ \mathit{Name}\ \underline{[}: \mathit{Type}\underline{]} := \mathit{InitExp}$$

The Type may not be generic.

Initialization of an object can be an expression or an array initialization

$$\begin{aligned}\mathit{InitExp} \to\ & \mathit{Exp} \mid \mathit{ArrayInit} \mid \textbf{new}\ \mathit{Type}(\mathit{CArg}^{+,}) \\ & \mid \Big(\textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{InitExp}\ \underline{(}\textbf{else if}\ \mathit{Exp}\ \mathit{InitExp}\underline{)}^{*}\ \textbf{else}\ \mathit{InitExp}\ \underline{[}\textbf{if}\underline{]}\Big)\end{aligned}$$

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

$$\mathit{CArg} \to \mathit{Exp}$$

- If the object to be initialized is of a primitive type (such as **int32** or **real64**), the *initExp* should be a compile-time constant expression of a type assignable to the type of the object.
- If the object to be initialized is an array, then the *InitExp* should be an *ArrayInitExp*.
- If the object to be initialized is an object of non-primitive type, then the *InitExp* should be of the form **new** *Type*(*Args*) where the *Type* is a non-generic class type.
- Constructor arguments must either represent objects or compile time values, depending on whether the corresponding parameter is **obj** or **in**.
- In any case, the InitExp can be an if-else structure in which the expression is a compile-time constant assignable to **bool**.
- The *InitExp* must have a type that is a subtype of the *Type*.

1.4 Constants

A constant is simply a named constant expression

$$\mathit{ConstDecl} \to \textbf{const}\ \mathit{Name}\ \underline{[}: \mathit{Type}\underline{]} := \mathit{ConstExp}$$

The type, if present must be primitive. Constant expressions are always primitive.

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

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

- 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

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

- 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.

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

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

  (class B constructor(obj x : A) ...)
  obj a := (for i : 10 do new A())
  obj b := (for i : 10 do new B(a0))

  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.]

$$\mathit{ClassMember} \to \mathit{Field} \mid \mathit{Method} \mid \mathit{Thread} \mid \mathit{ConstDecl} \mid ;$$

Fields are objects that are within objects. Field declarations therefore define the part/whole hierarchy.

$$\mathit{Field} \to \mathit{Access}\ \textbf{obj}\ \mathit{Name}\underline{[} : \mathit{Type}\underline{]} := \mathit{InitExp}$$
$$\mathit{Access} \to \textbf{private} \mid \textbf{public}$$

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

$$\mathit{Method} \to \mathit{Access}\ \textbf{proc}\ \mathit{Name}(\underline{(}\mathit{Direction}\ \underline{[}\mathit{Name} : \underline{]}\ \mathit{Type}\underline{)}^{*,})\underline{]}$$
$$\mathit{Direction} \to \textbf{in} \mid \textbf{out}$$

The types of 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.]

$$\mathit{IntMember} \rightarrow \mathit{Field} \mid \mathit{Method} \mid \mathit{ConstDecl} \mid ;$$

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

$$\mathit{Block} \rightarrow \underline{(}\mathit{Statement} \mid ;\underline{)}^*$$

Statements as follow

- Assignment statements

$$\begin{aligned}
\mathit{Statement} &\rightarrow \mathit{ObjectIds} := \mathit{Expressions} \\
\mathit{ObjectIds} &\rightarrow \mathit{ObjectId}\ \underline{(}, \ \mathit{ObjectId})^* \\
\mathit{Expressions} &\rightarrow \mathit{Expression}\ \underline{(}, \ \mathit{Expression})^* \\
\mathit{ObjectId} &\rightarrow \mathit{Name} \mid \mathit{ObjectId}[\mathit{Expression}] \mid \mathit{ObjectId}.\mathit{Name}
\end{aligned}$$

The type of the ObjectId must admit assignment, which means it should be a primitive type, like **int32** or **real64**.

- Local variable declaration

$$\mathit{Statement} \rightarrow \mathbf{obj}\ \mathit{Name}[\,:\mathit{Type}] := \mathit{InitExp}\ \mathit{Block}$$

  Same restrictions as fields. The type may be omitted, in which case it is inferred from the initialization expression. The block part contains as many statements as possible. The scope of a local variable name is the block that follows it.

- Constant Declarations

$$\mathit{Statement} \rightarrow \mathit{ConstDecl}\ \mathit{Block}$$

  The block part contains as many statements as possible. The scope of a local constant name is the block that follows it.

- Method call statements

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

- Sequential control flow

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

- Parallelism

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

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

- Method implementation.

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

  - Restrictions
    - The 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 statement 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

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

  - 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 statements 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*)
      (**wh 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

$$(\textbf{co}\ a[i] := 0 \ ||\ a[j] := 0\ \textbf{co})$$

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:

$$(\textbf{co}\ (\textbf{with}\ l\ a[i] := 0\ )\ ||\ (\textbf{with}\ l\ a[i] := 1\ )\ )$$

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

$$(\textbf{co}\ s.p()\ a[i] := 0\ s.v()\ ||\ s.p()\ a[i] := 1\ s.v()\ \textbf{co})$$

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

4 Expressions

[[To Be Completed]]

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

$$\begin{aligned} GParams &\rightarrow \left\{GParam^{+,}\right\} \\ GParam &\rightarrow \textbf{type}\ Name\ \underline{[}\textbf{extends}\ Type\underline{]} \end{aligned}$$

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

6 Examples

(**class** FIFO {**type** T **extends** primitive}
constructor(**in** capacity : int)

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