The Joccam Language

Joccam (Java-style Occam) is a brand new language which fuses two very important features from two existing programming languages: The process orientated design methodology (including channels, barriers, mobile processes etc.) from occam [] and the well known syntax of Java [].

The reason for the development of Joccam is mainly to try to push process oriented development. We believe that process orientation is the best way to program fine grained parallelism; especially when dealing with multi-core architectures, which are becoming increasingly popular.

The design of Joccam follows the following simple thesis: Attempt to provide the functionality of occam-pi but use a Java style (i.e., 100% match java if possible, else adopt the Java style for new constructs).

With the popularity of multi-core architectures this naturally remains a target for a process oriented language like Joccam, but in addition, we intend to provide multiple back-ends to the compiler; in particular, we are working on a C with MPI code generator, which lets your take advantage of process oriented design principles and language features as well as a distributed runtime environment. Other targets that are planned include java bytecode for portability reasons.

Please check back regularly as we will be updating this page as we progress. This work is supported by the UNLV Research Development Award 2008.

1 Joccam Examples

Although the development of Joccam is still in its early stages, the syntax is more or less determined. Here is an example of a piece of occam code with an ALT and an extended rendez-vous:

```
PROC foo(CHAN OF INT in1?,
          CHAN OF INT in2?,
          CHAN OF INT out!)
SEQ
  ALT
    INT y:
    in1 ? y
    out ! y
    INT y:
    in2 ? y
    out ! y
    INT y:
    in1 ?? y
    out ! y

which in Joccam would look like this:
```

```
The Joccam code is much closer to the well-known Java syntax, and in addition to having { } constructs, the keywords are no longer set in capitals, and indentation no longer matters.

We believe that this will prove to be a good motivator for learning process oriented design and programming.

2 The building blocks of Joccam

2.1 Data Types

In the spirit of keeping Joccam close to Java with respect to syntax, we have chose to use the Java atomic type system, rather than that of occam.

2.1.1 Atomic Types

In Java, the atomic types are: boolean, byte, short, char, int, float, long, double. Joccam uses the same names, but in addition, string is an atomic type (Joccam does not have the notion of objects, and since Java implements strings as an object, this seemed necessary).

In addition to the regular atomic types, Joccam has an additional number (currently 2) of atomic types inherited from occam. The first is a timer type, which in reality is the reading end of a timing process that is always willing to engage in communication. The second is a barrier type, which is a special construct that defines a full barrier. Processes enrolled (using the keyword enroll, which can be prefixed by a par or a for-loop, if needed) on a barrier can synchronize on it by using the sync keyword.

2.1.2 Type constructors

Java has but two type constructors, namely classes and arrays. Since Joccam is not object oriented we do not have the notion of classes or objects, but we do have arrays; arrays in Joccam look and behave similarly to their Java cousins. Only a few differences exist:

- The syntax for declaring a Joccam array requires the [..] to be placed on the type and not on the name of the variable/parameter.
- Empty dimensions are allowed only on parameters or on variables if assigned an array value whose dimensions can be determined at compile time.
- Java only allows for array constants at variable declaration time. Joccam treats array
constants like any other constant.  

- A number of useful array operations like size must be made available; we haven't determined if a size of an array should be accessed as

  \[ s = \text{array.size} \]

  or as

  \[ s = \text{size(array)} \]

  At the moment the first one is favoured.

- occam implements array slicing, we have not yet determined if we are going to support this in Joccam.

Records are not found in Java (Records can be implemented as classes without any methods), but since we do not have classes in Joccam, it is necessary to have at least records available. An example of a record ld be:

```occam
type record myRecord {
  int a;
  double d;
}
```

Like arrays, records constant values. An example of a constant value of the above record could be:

```occam
myRecord{4, 5.7}
```

An additional two type constructors exist, namely protocol and process types. A protocol determines which types a channel can carry; We have not determined the exact syntax for protocols as of yet. A process type, or a proc type looks like this:

```occam
type proc is (.....);
```

where ..... is a list of formal parameters of a process definition. A process definition is needed when implementing mobile processes, not only for specifying that a mobile process' declarations 'implements' this process type, but also for specifying the type that travels across the channels that will carry the mobile when it is in its suspended state (i.e., not executing).

**Room for discussion:** arrays are not declared using the type word array, whereas records are.

### 2.2 Declarations

#### 2.2.1 Constants

In Java the concept 'constant' does not exist explicitly; instead, a final field cannot be assigned to, so in effect, it behaves like a constant; However, it is not possible to declare local constants within a method.

In occam a constant is prefixed the keyword VAL; VAL constants can be local as well as parameters.
In Joccam we have decided to call a spade a spade and simply use the word `const`. Constants can be declared at the top level, along with the type declarations; as well as be formal parameters, and local. If a constant is defined locally or globally it must be initialized at declaration time:

```plaintext
const int WORKING_TIME = 100000;
```

whereas when used as a formal we could have:

```plaintext
proc void foo(const int i) { ... }
```

### 2.2.2 Local and Parameters

A local or a parameter in Joccam is declared in the exact same was as in Java:

```plaintext
[<modifiers>] <type> [<array_dims>] name [ = initializer]
```

where currently the only two modifiers we have are `const`, used for declaring a constant (which then requires the initializer to be present), and `mobile` used to declare the variable as holding a mobile type.

### 2.2.3 Channel Declarations

Channels are a new concept not found in Java, but plays an integral role in occam, and for that matter, in any process oriented design. A channel carries values of a certain type in one direction, from or or more senders to one or more receivers (channels can be one-to-one, one-to-many, many-to-one, or many-to-many). If a channel is has many senders or many receivers (or both) was say that it is `shared`. When declaring channels we must declare which end of it is shared, and also specify the type that it carries. For example:

```plaintext
shared write chan<Agent_message> report
```

declares a channel called `report`, which carries values of the `Agent_message` type, and is shared on the writing end (i.e., it can have multiple writers. When a channel is declared shared, it must be `claimed` by a process before it can be written to/read from. (More about claim later)

In addition, we have channel-end types and expressions as well. A channel-end type is simply a channel type followed by a `.read` or a `.write`, indicating which end of a channel type we are interested in. Likewise, if a variable contains a whole channel, and not just a channel-end, `.read` and `.write` can be specified. If a `.read()` operation is applied to a whole channel, the read end is automatically picked. The same holds for the `.write()` operation.

### 2.3 Statements

In the tradition of keeping the syntax as close to Java as possible, the majority of statements in Joccam have the same syntactical structure as statements in Java.

- Assignment
Java allows assignment like \( a = b = c = 6; \), and occam allows assignments like \( a, b, c = 7, 5, d \). Either type can cause aliasing problems, but at the moment Joccam only supports a single assignment per =, though we will probably end up supporting both types of assignment, but not a mixture (That would probably not be syntactically viable anyhow!).

- **Choice**

  Joccam will support Java-style if statements, both with and without an else clause. Whether a ternary expression will be supported has not yet been determined. We might introduce an elseif keyword, but that is only a thought at the moment. Java-style switch/case statements will be implemented in Joccam, and in addition we might implement the case statement of occam that allows to switch on a channel read. The extension to Java's switch statement then would require that the case labels can be not only integral constants, but also union constants.

- **Repetition**

  Java currently has standard for, do, and while loops, in addition to iterators. Joccam will implement do and while loops according to the standard Java syntax; for loops are a little more difficult as the occam equivalent is somewhat different in that a for loop can be prefixed with the par construct, allowing for the body of the loop to be done in parallel. This would ultimately require some analysis of the body of the loop, or a pre-execution of the loop to determine the index set. This is only possible if the body of the loop cannot change the variables that are used in the loop condition. Since we do not allow global variables, and use a call-by-value call semantics, it should be reasonably simple analyze the assignments of the body to assure that if the loop is to be executed in parallel no loop variables may be re-assigned. An example of how to do this has been provided by Neil Brown and works as follows:

  ```
  par for (i=,j=0;i<100;i+=2,j=j+2*i)
  statements
  ```

  would require the index set

  \[(i,j) = \{(0,0), (2,4), (4,12) \ldots\}\]

  to be pre-computed, and then the statements executed in parallel over this index set. As far as iterators, we have not made any decisions at all.

- **Blocks and par**

  occam provides both a PAR and a SEQ construct. Examples are:

  ```
  SEQ
  a := 5
  b := 7
  ```

  and

  ```
  PAR
  a := 5
  b := 7
  ```
where the assignments in the **SEQ** block are done in sequence (i.e., sequentially), and the assignments in the **PAR** block are done in parallel. In Joccam there is no need for the **SEQ** construct, as by default, everything is done sequentially, so a simple Java block using `{ and } is sufficient. In Joccam a block prefixed by the **par** keyword will be a parallel block.

- **Communication**

  The notion of channel communication does not exist in Java; this is 100% inherited from occam. Thus this requires a brand new Java-like syntax. In occam a channel can be read or written in the following way:

  ```
  in ? x
  out ! x
  ```

  The first statement reads a value from the **in** channel into **x** and the second writes the same value to the **out** channel.

  The suggested syntax for joccam is as follows:

  ```java
  x = in.read();
  out.write(x);
  ```

  Though Joccam does not have objects, we chose this 'method invocation' style to stay close to something Java programmers know.

  Extended rendez-vous in occam (a construct where the writer/sender is not released from the synchronous call until the code in the extended rendez-vous block has been executed) look like this (If more than one process must be executed in the rendez-vous block a **PAR** or a **SEQ** block is needed):

  ```
  in ?? x
  SEQ
  out ! x
  out ! x
  ```

  this, in Joccam, would look like this:

  ```java
  x = in.read({out.write(x); out.write(x);});
  ```

  Alternatively the rendez-vous block could be prefixed a **par**.

- **Barriers**

  A barrier in occam is an atomic typed, and also so in Joccam. Barriers are declared, enrolled on, synchronizes on and temporarily resigned from in he following way:

  ```java
  barrier b;
  ...
  enroll(b) {
  }
  ...
  sync(b);
  ...
  ```
resign(b) {

};

Alternatively a par can be placed infront of the enroll statement. Barriers can also be passed as parameters to function.

- skip and stop

Both skip (which is a no-operations process), and stop (which is a process that refuses to engage in any action, thus deadlocking the process) exist only in occam. Typically skip can be replaced by a ; in Java, but both are statements in Joccam.

- Mobile invocation, suspension, and resumption

The concept of mobiles does not exist in Java, but is the latest extension to occam-pi. There are two classes of mobiles: mobile channels and mobile processes. For now we shall restrict Joccam to have mobile processes, and we will later return to the mobile channels (and mobile channel bundles, as well as issues with arrays of mobile).

2.4 Expressions

Expressions in Joccam (except the addition of channel reads and timer reads) are similar to Java expression.

3 Modularity

For Joccam to be a proper language some sort of modularity and import/include mechanism must be provided. The Java package model is slightly complicated to implement, and relies somewhat on the object oriented aspects of the language. Occam uses an include statement on a per file basis; something is much easier to implement. For joccam we shall stick to the occam way and simply include files as needed.

4 I/O

To come later

5 Java Call Backs

To come later

6 Multiple Back-ends

Currently we have a occam and a c with MPI back-end in the works.