The Language Globs for Parallel Computation and Coordination

Fernando M Q Pereira                Jonathan K Lee

June 6, 2008

Abstract

This report describes globs, a programming language whose purpose is to facilitate the development of parallel applications. Globs provides powerful support to task parallelism: statements in the same block are executed in parallel. Additionally, Globs allows synchronous and asynchronous communication between processes. Asynchronous message passing is implemented via message boxes, in a way similar to what is done in Erlang. Synchronous communication is implemented via point-to-point channels, or via remote method calls.

1 Introduction

Multi-core computers are gradually becoming the standard option in terms of high performance computing. As a testimony of this fact, important hardware companies are adding multi-core machines in their main line of products. For instance, the new Sun’s Ultra Sparc T1 has eight cores [11], the Cell processor from the Sony/IBM/toshiba consortium has nine cores [10] and the NVIDIA Geforce 8800 GTX box has 16 cores [14]. The emergence of multi-threading architectures is naturally followed by the arise of new languages for parallel programming. As an example, in the two last years we saw the release of three of these languages: Fortress [1], Chapel [5] and X10 [6]. It is in the context of this new revolution that we are designing globs, a language for parallel computation and coordination of distributed processes.

Our main choices concerning the design of globs are motivated by three objectives:

- Globs must allow the easy development of parallel applications. Thus, the developer must not be burdened with extra syntax to fork threads
or to synchronize different flows of computation. Also, globs programs must be oblivious to the physical location of processors every time this information is not necessary for performance reasons.

- Globs programs must be easy to analyze and optimize automatically. We believe that not only the application developer, but also the compiler and execution environment are responsible for guaranteeing the high performance of parallel applications. Thus, globs is a statically typed language, and it is our intention to limit the number of run-time checks to the possible minimum.

- Globs must foster a programming model that forces the application developer to think in terms of parallel computations. Contrary to most of the well known languages, including languages specifically designed to facilitate concurrent programming, in globs parallel computations are the default, instead of sequential execution. Thus, statements in the same basic block are executed in parallel, and the arguments of tuples are evaluated concurrently.

We have implemented a prototype interpreter for globs programs, and a small collection of visualization tools that help in debugging applications. Our current implementation of globs provides guarded commands, parallel blocks and message boxes for asynchronous communication. The interpreter, as well as a small collection of benchmark programs is freely available at:

http://compilers.cs.ucla.edu/fernando/projects/globs

The examples discussed in this report were designed to illustrate the full power of the globs specification. The current implementation of globs does not provide the following features yet: arrays, dependent types, structural sub-typing, closures and synchronous communication.

The remainder of this document is organized as follows: Section 2 describes the general model of computation fostered by globs. General program language features, like closures and guarded commands are described in Section 3. The communication model is discussed in Section 4. Finally, section 8 concludes the paper.

2 The Globs Programming Model

We chose to provide globs with a execution model where parallel computations are the default, contrary to most programming languages, which foster a sequential model of computation. Thus, statements in the same
basic block, in globs, are executed in parallel. For instance, the class below has two fields of a generic type E. Its swap function exchanges the values of these fields:

```java
class SwapEx<E>(a: E, b: E){
    fun swap() -> () {
        a = b;
        b = a;
    }
}
```

The basic statement in globs is an assignment. During the execution of a block of commands, the left side of each assignment is evaluated in parallel. After evaluation, values are assigned, also in parallel. In this way, the swap function shown about is semantically correct, and it guarantees that neither the value of a, nor the value of b would be overwritten during the program execution.

For sequential computations, globs provides the let statement. Such statement is divided into two parts: a sequence of binding declarations, and a sequence of statements. Bindings are evaluated before the statements, but declarations and statements are evaluated in parallel among themselves. As an example, the program below declares, initializes and uses a stack:

```java
class Main {
    fun main() -> () {
        let {
            var s:Stack<String> = new Stack();
        } in {
            let {
                var _ = s.push("This is a datum.");
            } in {
                \out = "Data removed from the list: " + s.pop();
            }
        }
    }
}
```

Some statements are only useful for the side-effect. To execute such statements sequentially, we bind them to undefined names, as done in the second let block of the example above.

Functions in globs are atomic by default. We chose to design globs in such a way that forces the software developer to use small functions. In
this way, we hope to increase the number of commands that can be executed in parallel in the same program. Therefore, globs has not conditionals statements. Iterations via loop blocks were restricted to only dimensions that are known at compilation time. The main construct for controlling the execution flow in a program is the guarded function. A guarded function augments the function declaration with a boolean expression. The function body is executed only if the guard is true. A function signature is defined by the function name plus the list of parameters of the function. The same function signature may denote different implementations. During a function call, the guards are evaluated, and the function with the first true guard is executed. The order of evaluation is given by the order in which functions are declared in the program text. For instance, the function below returns the absolute value of an integer number:

```plaintext
fun abs (in: Int) -> (out: Int) when in >= 0 {
  out = in;
}

fun abs (in: Int) -> (out: Int) {
  out = -in;
}
```

Globs is a statically typed, object oriented language. The main unit of compilation is a class. Classes have state, which is given by their internal field variables. Globs has no explicit construct for coercion, and all operations and values are type-checked during compilation. The choice for a statically type design is motivated by performance. Static type-checking allows the globs compiler to remove dynamic checks from programs. Arrays in globs must have size known statically in order to allow the globs compiler to avoid inserting bound checking in operations that access arrays. In order to handle arrays whose size is not known by the application developer during code writing, globs provides dependent types. The program below sums the elements of two vectors:

```plaintext
fun sumVector<N:Int>(v1[N]:Int, v2[N]:Int) -> (vr[N]:Int) {
  foreach(i) when 0 < i < N {
    vr[i] = v1[i] + v2[i];
  }
}
```
3 Programming Language Features

Globs defines three main kinds of data structures:

**class:** classes are the main compilation unit of globs. A program is a collection of one or more classes. A class contains fields and methods, each of which can be either public or private. A class declaration may contain type variables, which are used to declare generic and dependent types.

**array:** arrays are collections of elements of the same type that can be accessed via indexing. Arrays in globs must have size and type declared at compilation time. The size of an array may be bound to a type variable.

**record:** records are tuples whose fields have names. They are used for passing parameters in and out of functions.

The parallel execution of basic blocks leads naturally to a value-result parameter passing style. This occurs because it may be necessary to return multiple values that result from the execution of many statements in the same function. Every function declaration in globs contains two records. The first record defines the names of the input parameters, and the second record defines the names of the output parameters. For example, the function below returns the first and the last element of a list:

```scala
fun getFirstLast<E>(l:List<E>) -> (first:E, last:E)
  when l.size() > 1 {
    first = l.getFirst();
    last = l.getLast();
  }
```

One of the main uses of output records is to allow the globs developer to bind variables to the return values of functions, as we show in the example below:

```scala
let {
  var _:(f1:Int, l1:Int) = XXX.getFirstLast(l);
  // - or -
  var r:(f:Int, l:Int) = XXX.getFirstLast(l);
} in {
  \out = "First element = " + f1 + ", last element = " + l1;
  \out = "Alternatively: first = " + r.f + ", last = " + r.l;
}
```
In a way similar to OCaml [4], globs uses structural sub-typing. For sub-typing purposes, two functions are considered the same if they have the same signature. A class $C_<$ is a subclass of class $C$ if $C_<$ contains all the public function and field signatures present in $C$.

The most important mechanism of code reuse in globs is inheritance. If a class $C_1$ extends a class $C_0$, then $C_1$ inherits the implementation of all the public methods and fields present in $C_0$. Globs does not provide multiple inheritance, like C++ [15] does. Instead, globs uses traits [7], as a way to declare new type signatures, and as a way to simulate multiple-inheritance. Traits declare method signatures and they may provide implementation for some methods; however, traits cannot contain state in the form of class fields.

Similar to Scala [13], globs provides closures. The main appeal of closures for parallel computations is the possibility of writing many parallel algorithms as combinations of high order functions such as map and reduce [12]. Methods are first order values. In order to pass a method as a parameter of a method call, we convert the method text to a class with an eval method. For instance, below we show how a method swap can be converted into a class.

```javascript
fun swap<T>(x_in: T, y_in: T) -> (x_out: T, y_out: T) {
  x_out = y_in
  y_out = x_in
}

class swap<T> {
  fun eval(x_in: T, y_in: T) -> (x_out: T, y_out: T) {
    x_out = y_in
    y_out = x_in
  }
}
```

4 Inter-Process Communication

In globs, the actual location of objects is transparent to the application developer. Objects are distributed among processors, and they communicate via three different mechanisms: (i) remote method invocation; (ii) point-to-point channels and (iii) asynchronous message boxes. Each of these mechanisms, which provides different advantages, is discussed in this section.
Remote method invocation is the main mechanism for distributed communication in globs. Remote objects in globs are instances of any class that implements the trait `Remote`. The actual location of a remote object is transparent to the globs developer. Similar to the Java Programming Language [9], objects must be serialized before passed as parameters of remote methods. Objects that can be passed as arguments of remote method calls must implement the trait `Serializable`. The serialization of object state is the responsibility of the application developer. The trait `Serializable` defines two methods with empty body: `readObject()` and `writeObject`, which must be explicitly implemented by the application developer. Stubs for remote objects are passed among processors via communication channels.

Communication channels are a more efficient way for processes to share information than remote method invocation. Globs provides typed communication channels as defined in Microsoft’s experimental language Sing# [8]. These communication points are designed after the six principles described by Fähndrich et al. in [8]: channels are bidirectional; memory regions can be transfered by reference; channels are ruled by static contracts; channels end-points can be send through channels; sending and receiving on channels require no dynamic allocation of memory and send operations do not block the sender. Contrary to singularity, globs does not define new syntax for the specification of channels. Channels must implement the trait `Channel<E>`, which defines a channel that can send only elements of type `E`. This trait contains the implementation of two methods: `send<E>(e:E)` and `recv<E>:E`. Sing# allows different types of messages to be sent over channels. This is not possible in globs: for maximum efficiency, channels can transport only messages of one single type. More complicated synchronous protocols must be implemented via remote message invocation. In order to be sent through a channel, an object must implement the trait `Serializable`. Remote objects are sent by reference, that is, their stub is sent through the channel, whereas other objects are passed by value. The code below is an example of how remote objects are communicated between processes via channels:

```javascript
let { // The sender side:
    var c:Channel<MessageTest> = new ChannelImpl<MessageTest>();
    var out:MessageTest = new ClassTest("Hello!");
    var in:MessageTest = null;
} in {
    () = c.send(out, "ch_id"); // non-blocking
    in = c.recv("ch_id"); // blocking
}
```
let {
    var c:Channel<ClassTest> = new ChannelImpl<ClassTest>();
    var in:ClassTest = null;
    var out:ClassTest = new ClassTest("Ack.");
} in {
    in = c.recv("ch_id");
    () = c.send(out, "ch_id");
}

Finally, for asynchronous communication, globs provide message boxes, similar to the language Erlang [2]. Message boxes are the asynchronous mirror of channels. Class that represent message boxes must implement the trait MessageBox<E>, with methods send<E>(e:E) and recv<E>:E. However, while channels must ensure that calls to the methods send and recv are always alternated, this constraint does not exist on message boxes. The recv method blocks if the message box is empty. For non-blocking access, the receiver can verify the contents of the message box via the method hasMsg():Bool. As an example, the code below implements the dining philosophers problem using an algorithm similar to that described by Carriero and Gelernter [3].

class Main {
    fun main () -> () {
        let {
            var MessageBox<Int> tickets = new MessageBoxImpl<Int>();
            var MessageBox<Int> chops = new MessageBoxImpl<Int>();
            in {
                // Put four tickets into the ticket master, to avoid deadlock:
                var x1:() = tickets.send(1);
                var x2:() = tickets.send(1);
                var x3:() = tickets.send(1);
                var x4:() = tickets.send(1);
                // We have five chopsticks:
                var y1:() = chops.send(1);
                var y2:() = chops.send(1);
                var y3:() = chops.send(1);
                var y4:() = chops.send(1);
                var y5:() = chops.send(1);
                // Initialize the philosophers:
                var p1:Phil = new Phil(1, tickets, chop);
            }
        }
    }
}
var p2:Phil = new Phil(2, tickets, chop);
var p3:Phil = new Phil(3, tickets, chop);
var p4:Phil = new Phil(4, tickets, chop);
var p5:Phil = new Phil(5, tickets, chop);
} in {
    () = p1.eat();
    () = p2.eat();
    () = p3.eat();
    () = p4.eat();
    () = p5.eat();
}

In order to avoid deadlocks, our implementation uses a message box that contains, at any time, only four messages representing tickets for the meal room. In this way, only four processes can compete for the five shared chopsticks, and at least one of them will get two chopsticks.

class Phil {
    var id: Int = -1;
    var MessageBox<Int> tickets = null;
    var MessageBox<Int> chops = null;

    fun Phil(new_id: Int, MessageBox<Int> t, MessageBox<Int> c) -> () {
        id = new_id;
        tickets = t;
        chops = c;
    }

    fun eat() -> () {
        let {
            var ticket:Int = tickets.recv();
        } in {
            let {
                var chopstick1:Int = chops.recv();
                var chopstick2:Int = chops.recv();
            } in {
                let {
                    
                
```
var x:() = chops.send(chopstick1);
var y:() = chops.send(chopstick2);
} in {
    \out = id + " is done eating."
    () = tickets.send(ticket);
}
)

5 Implementation and Evaluation

We implemented an interpreter using javacc to evaluate two of the core features of our language – parallel assignment and message boxes. The interpreter is currently single threaded, although we did some experiments with multi-thread implementations. We feel that it will be much simpler to implement the multithreaded aspect when we compile directly to java. Currently, the parallel assignments are evaluated in the order in which they appear in the program text. Intermediate values are written to temporary variables, and the actual variables are assigned at the end of each parallel block. Although globs is a statically typed language, our interpreter currently perform type-checks during program execution.

Message boxes were implemented using the filesystem and counters to keep track of data in the file. In our current implementation of the globs interpreter, each message box is a file in the operating system filesystem. To facilitate debugging, each message is stored as a line in the text file. We keep track of how many messages have been read via a counter, also stored in a file, and we ensure that each thread only reads message past the number of lines indicated in the counter file.

6 Future Work

As future work, we plan to refine the grammar of our prototype language, as the current version does not enable all the features in the language specification, such as parametric polymorphism, dependent types and tuples. We would also like that to code a prototype compiler that translates globs directly to Java, as we believe it will be easier to implement the threaded
aspect of the language. This would also allow us to add remote method
invocations and more easily support generics and even arrays.

We also intend to implement compiler optimizations to this language.
With parallel assignments and guarded methods, we believe there will be
several opportunities to perform interesting static analyses to improve the
code produced during compilation.

7 Experiences

The decision of executing globs via an interpreter, instead of compiling it
directly to Java came with mixed results. While it made it easy to pinpoint
bugs and bad decisions in our example programs, it turned out to take much
longer to implement than expected. Furthermore, interpreting the abstract
syntax tree of globs programs made it difficult to add actual parallelism to
the language.

Due to not stabilizing our grammar very early, our implementation
started to quickly become a collection of kludges as we tried to finish the
interpreter early so that we could focus on implementing other features
described in the language specification. Ultimately we had to thin our
grammar to remove many of the features that we were initially planning
to implement.

One important experience acquired during the development of the globs
execution machinery concerns the importance of the solid development of
the language grammar before moving to the implementation of the language
interpreter. Having to change the grammar continuously during the imple-
mentation of the interpreter compromised our ability to quickly test globs
applications.

8 Conclusion

This report has described the language globs, which aims at the development
of parallel applications. The implementation of globs is still on very early
stage; however, we have been already able to implement programs that use
parallel statements to solve problems such as leader election in a ring and
the dining philosophers. It is our intention to use globs programs in the
development of applications for plasma simulation. In order to meet this
purpose, our next priority is implementing arrays and adding dependent
types to our current implementation of globs.
References


