Adding Support for a Concurrent Invocation Statement in JR

by

ESTEBAN PAULI

HONORS THESIS
Submitted in partial satisfaction of the requirements for the degree of
BACHELOR OF SCIENCE

in

Computer Science

in the

COLLEGE OF LETTERS AND SCIENCES
of the
UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

Professor Ronald A. Olsson

2004

Contents

List of Figures

Chapter 1

Introduction

The JR programming language has many constructs which make writing parallel programs and algorithms relatively easy. With little effort, and quite a bit of flexibility, a user can create multiple processes, protect against race conditions, do complicated message passing, do any of these across different virtual machines, and even do them over different physical machines without having to write too much code. JR is based on the SR language [2] and borrows heavily from its features.

However, there are some things JR cannot currently do. Consider a database system which stores duplicate copies of all the data on three computers. We want to make a number of requests, and are interested in which of the three machines is responding the fastest so that we can send our requests just to that server and not put an unnecessarily large load on the system. For the time being, assume we are interested in sending all the requests at once and waiting for them to be serviced. A good place to start might be the following:

```
entry = call server[0].get_entry(entry_key);
entry = call server[1].get_entry(entry_key);
entry = call server[2].get_entry(entry_key);
```

The effect this will have will be to send the request first to `server[0]`. When this request has been serviced, the request to `server[1]` will be made. When this second request has been serviced, we will make the request to `server[2]` and wait for it to complete. This is not really what we want since the above code is really doing one request at a time. We are doing three times as much work and not really gaining anything. With this in mind, we might try the following code:

```
send server[0].get_entry(entry_key);
send server[1].get_entry(entry_key);
send server[2].get_entry(entry_key);
```

This will achieve some of our goals, but not all of them. The above will send a request to `server[0]`, then to `server[1]`, and then to `server[2]`. It will not wait for one request to get back before it makes the next. In fact, after the requests are made, the program will continue, and any values that may be returned by other processes servicing these calls will be ignored. What we really want to do cannot be done without the concurrent invocation (`co`) statement, which is what we introduce below. With this new statement, which is based on SR's `co` statement, we will be able to do the following:

```
co entry = call server[0].get_entry(entry_key){break;}
[] entry = call server[1].get_entry(entry_key){break;}
[] entry = call server[2].get_entry(entry_key){break;}
```

With the use of the `co` statement, our example will work as follows. Each arm's invocation will be started concurrently. This behavior will be similarly to the case where we used the `send` statement. Once all the invocations have been started, we will block. As soon as one invocation is serviced, we will assign its return value to `entry` and stop waiting for the other invocations to be serviced. If we go back to the original problem of keeping track of the fastest server, we could write

```
co entry = call server[0].get_entry( entry_key){fastServer = server[0]; break;}
[] entry = call server[1].get_entry( entry_key){fastServer = server[1]; break;}
[] entry = call server[2].get_entry( entry_key){fastServer = server[2]; break;}
```

.

In addition to behavior described above, we now store whichever server serviced our invocation first in `fastServer`. In general, we will be able to start any number of invocations concurrently and execute arbitrary code as these invocations are serviced.

It should be noted that the behavior of the `co` statement can be simulated in JR without the new additions, but it would require the user to create a different interface for their classes in order to support the desired behavior (see Section 2.3). The addition of the concurrent invocation statement will make many tasks much easier to accomplish, will allow the programmer to write shorter, less complicated code, and will help capture the meaning of the program in the syntax.

The rest of this thesis is organized as follows. Chapter 2 shows more examples of the `co` statement. Chapter 3 shows the complete syntax of the `co` statement and explains its implementation. Chapter 4 highlights some implementations issues. Chapter 5 describes the work that has been completed and some possibilities for future work.

Chapter 2

Examples of the `co` Statement

2.1 Quicksort

This first example is adapted from [2] and shows how the `co` statement can help better express the algorithm being used. It also illustrates how the `co` statement can help us avoid changing the interface to our program.

When writing a non-parallel quicksort algorithm in Java, we might try to implement it as in Figure 2.1.

Realizing that the recursive calls can be done in parallel, in JR we could try the code given in Figure 2.2.

However, this does not really improve things. Since a `call` statement block until it has been serviced, we do not gain anything here. If we were to switch the `call`s with `send`s, we would have a different problem. Since the `send` statements do not block, our first call to quicksort would return after the first two recursions had been

```
private static void quicksort(int [] list, int begin, int end){
    /*check if done with this list*/
    if(begin == end)
        return;
    /*pick pivot*/
    ...
    /*recurse*/
    quicksort(list, begin, pivot);
    quicksort(list, pivot, end);
}
```

Figure 2.1: Quicksort written in Java

```
private static op void quicksort(int [] list, int begin, int end){
    /*check if done with this list*/
    if(begin == end)
        return;
    /*pick pivot*/
    ...
    /*recurse*/
    call quicksort(list, begin, pivot);
    call quicksort(list, pivot, end);
}
```

Figure 2.2: Quicksort in JR with `call` statements

```
private static op void quicksort(int [] list, int begin, int end, cap void () reply_to_me){
    /*check if done with this list*/
    if(begin == end){
        send reply_to_me();
        return;
    }
    /*pick pivot*/
    ...
    /*make new capabilities*/
    cap void () reply1 = new op void();
    cap void () reply2 = new op void();
    /*recurse*/
    send quicksort(list, begin, pivot, reply1);
    send quicksort(list, pivot, end, reply2);
    /*wait for replies*/
    for(int i = 0; i < 2; i++)
        inni void reply1(){}
        [] void reply2(){}
    /*reply to our caller*/
    send reply_to_me();
}
```

Figure 2.3: Quicksort in JR with `send` statements and changed interface

started, and we would be continue on, dealing with an array that is being sorted at the same time we might be trying to use it. Noting this, we might try to salvage our efforts by changing the interface as shown in Figure 2.3.

While the above would accomplish what we want, it is rather messy. First, it requires us to change the interface to our quicksort function. Furthermore, we have to write three times as much code for message passing. More importantly, the changes obscure the meaning of the program. While this third version is parallel and the previous two are not, the first two versions are shorter to write and their meaning is much more clear.

With the `co` statement, we can have the best of both worlds as illustrated in Figure 2.4.

```
private static op void quicksort(int [] list, int begin, int end){
    /*check if done with this list*/
    if(begin + 1 == end)
        return;
    /*pick pivot*/
    ...
    /*recurse*/
    co call quicksort(list, begin, pivot);
    [] call quicksort(list, pivot, end);
}
```

Figure 2.4: Quicksort in JR with the `co` statement

The only change between Figure 2.2 and Figure 2.4 is the fact that we change from plain `call` statements to a `co` statement with an arm for each `call`. This will (theoretically) execute the two arms in parallel and wait for them both to finish before continuing. Note how much cleaner this is than Figure 2.3 and how well it preserves the meaning of the algorithm.

2.2 Voting

This next example illustrates the use of the post-processing block. Here, we consider a group of friends trying to decide whether to go on a trip. Some of the friends are slower to decide, but they will all either stay home or go on the trip. Since they want as much time as possible to prepare, they want to know whether or not they are going as soon as a majority has reached the same decision.

In order to make this work, we can use the `co` statement. Each arm of the statement will ask a different friend whether or not they want to go. If they want to go, they service the call and reply with `true`. If they do not want to go, they reply with `false`. As each friend replies, we increment the number of people who want to go or stay based on their reply. If at any time a majority of the people have made the same decision, we stop waiting for the rest. Finally, we return `true` if the friends decided to go, and `false` otherwise.

The code for the vote-counting part of the algorithm is given in Figure 2.5.

Note that we use `break` statements outside of a loop inside the post-processing code. The `co` statement allows us to do this, as well as using `continue` statements outside of loops, to indicate when we are done with a post-processing block and

```
/*declare array of ops a class member*/
...
/*count votes*/
private static final int numFriends = 7;
private static op boolean check_if_go_on_trip(){
    int goCount = 0;
    int stayCount = 0;
    boolean vote;
    co vote = askFriend[0](){
        if(vote){
            goCount++;
            if(goCount > numFriends / 2)
                break;
        }
        else {
            stayCount++;
            if(stayCount > numFriends / 2)
                break;
        }
    }
    [] vote = askFriend[1](){
    ...
    [] vote = askFriend[6](){
        if(vote){
            goCount++;
            if(goCount > numFriends / 2)
                break;
        }
        else {
            stayCount++;
            if(stayCount > numFriends / 2)
                break;
        }
    }
    return goCount > stayCount;
}
```

Figure 2.5: Voting

whether or not we should wait for the rest of the invocations. Although the post-processing blocks are simple in this example, in general we can include any arbitrary code in this section, even another `co` statement.

Chapter 3

Design and Implementation of the co Statement

3.1 Definition of the co Statement

3.1.1 Syntactic Definition

The `co` statement is defined by the grammar given below.

```
CoStatement ::= co CoArm {'[]' CoArm}*
CoArm ::= [id =] CallStatement (SimpleBlock | ';')
CoArm ::= SendStatement (SimpleBlock | ';')
```

Here, a `SimpleBlock` is the same as in the rest of JR (see [3]), with the exception that we allow `break` and `continue` statements anywhere inside the block, even outside a loop or `switch` statement. The behavior of these statements is explained in Section 3.1.2. We also refer to this block as the post-processing code associated with the arm. The `CallStatement` and `SendStatement` are the same as defined in the rest of the JR specification. These are also referred to as invocation statements and the invocation of the corresponding arm.

3.1.2 Semantic Definition

The `co` statement behaves as follows.

1. The `call` or `send` statements of all arms are invoked. No checks are performed to see if any invocations have been serviced until they have all been invoked.

2. Once an arm's invocation has been serviced, we continue with the next two sub-steps.

   (a) If there was an assignment associated with the arm's invocation, do the assignment.

   (b) If there was post-processing code associated with the arm, execute it.

   - If a `break` statement is encountered in the post-processing code outside a loop or `switch` statement, stop waiting for the rest of the invocations and continue with the code following the `co` statement.
   - If a `continue` statement is encountered in the block outside of a loop, stop executing this block and continue to wait for the rest of the invocations.

3. Once all the arms' invocations have been serviced, execute whatever code follows the `co` statement.

The important thing to note here is that all the invocations are started before we check if they have been serviced. For example, if only one invocation has been started and a different thread services it before we have had a chance to invoke the rest, we ignore the fact that an arm has completed until all the other arms have been invoked. Only then do we execute any assignment or post-processing block associated with the invocation that completed. Also note that the thread which checks if invocations have completed and executes the appropriate post-processing code is the same thread that is executing the `co` statement; no new threads are created to do this.

3.2 Implementation

3.2.1 The Basic Idea

The main idea behind the implementation of the `co` statement is to do the semantic translation exemplified by Figure 2.3. As a further example, consider the code below.

```
class Main{
    double op f(double arg1);
    String op g(double arg1, boolean arg2);
    MyType op h();
    void op foo(AnotherType arg1);
    ...
    void go(){
        double x;
        String str;
        MyType mt;
        AnotherType at;
        ...
        System.out.println("Start co");
        co x = f(2.3){System.out.println("f");}
        [] str = g(0.1, true){System.out.println("g");}
        [] mt = h(){System.out.println("h");}
        [] foo(at){System.out.println("foo");}
        System.out.println("Done with co");
        ...
    }
    ...
}
```

What we want to do, is translate this to the following (only the relevant parts are shown). Note the similarity to Figure 2.3.

```
        System.out.println("Start co");
        cap void (double) f_tmp = cocall f(2.3);
        cap void (String) g_tmp = cocall g(0.1, true);
        cap void (MyType) h_tmp = cocall h();
        cap void () foo_tmp = cocall foo(at);
        for(int JR_CO_COUNTER = 0; JR_CO_COUNTER < 4; JR_CO_COUNTER++){
            inni void f_tmp(double retVal){
                x = retVal;
                {System.out.println("f");}
            }
            [] void g_tmp(String retVal){
                str = retVal;
                {System.out.println("g");}
            }
            [] void h_tmp(MyType retVal){
                mt = retVal;
                {System.out.println("h");}
            }
            [] void foo_tmp(void){
                {System.out.println("foo");}
            }
        }
        System.out.println("Done with co");
```

This will behave as we want the `co` statement to behave. First all the invocations will be started. However, instead of using the `call` or `send` methods normally used internally by JR, we will use the new `cocall` and `cosend` variations. These will return capabilities which will let us know when each invocation has been serviced. The implementation of this is explained in Section 3.2.4. Once all the invocations have been started, we go into a loop that repeats once per arm. During each repetition, it services one of the return capabilities. If there was an assignment associated with the original invocation, we do the assignment as the first step in the body of each `inni` arm. When there is post-processing code for a particular `co` arm, we insert that block inside the body of the corresponding `inni` arm. Since we are now inside a loop, `break` statements inside the post-processing blocks will exit the loop, effectively terminating our wait for the rest of the invocations to be serviced. Similarly, a `continue` statement will end the current iteration of the loop and go to the next. Since the return capabilities will be unique and will only be invoked once each, we are guaranteed that the post-processing code for each arm is invoked only once. This is the behavior we wanted for the `co` statement.

3.2.2 Generating Auxiliary Classes

During the translation of a JR program, in addition to translating individual JR source files to Java source files, the translator produces many auxiliary classes. These define and implement many of the objects and methods required by the translations of the user's code. Of interest in this section is what happens when the user declares an operation (capabilities and process operations are also important for a `co` statement, but the concepts explained here extend easily to these other cases). If in a file named `main.jr` a user has the declaration

```
private static op int gcd(int a, int b);,
```

the translator will put the line

```
private static Op_intXintToint.JRProxyOp op_gcd_intXintToint;
```

in the file `main.java`. Furthermore, it will produce the file `Op_intXintToint.java` and other auxiliary files which implement all the behavior necessary for anything the user might want to do with such an operation. However, the `co` statement requires something which is not defined here: we need all the auxiliary code necessary for dealing with objects of type `cap int (void)` (remember the translation we did in Section 3.2.1).

In order to implement the `co` statement, the translator needs to generate all these auxiliary files. Any time an operation, a capability, or a process operation with a return type of `t` is encountered, in addition to generating the original classes and functions, the translator now also generates the classes necessary for dealing with a capability of type `cap void (t)`. It also adds to the original and new classes a set of `cocall` and `cosend` functions. These are the key to implementing the `co` statement and are explained in Section 3.2.4.

3.2.3 Synthesizing the Parse Tree

One of the most important steps of the `co` statement is what happens during compilation. Unlike with most other statements where there is a one-to-one correlation between the input file and the parse tree, the `co` statement inserts many synthesized nodes into the tree. The basic algorithm is as follows:

```
returnCaps <- empty list
invocations <- empty list
inniArms <- empty list
armCount <- 0
for each arm a in costmt
    a.assignment <- parseAssignment()
    a.invocation <- parseInvocation()
    a.ppc <- parsePostProcessingCode()
    retType <- a.invocation.returnType
    curRetCap <- new cap(retTypeTovoid)
    insertParseTreeNode(new varDeclaration(curRetCap))
    curCoArmInv <- a.invocation((cocall | cosend))
    invocations <- append(invocations, curCoArmInv)
    insertParseTreeNode(new Assignment(curRetCap, curCoArmInv))
    returnCaps <- append(returnCaps, curRetCap)
    inniBlock <- new Block()
    inniBlock.addStatement(a.assignment)
    inniBlock.addStatement(a.ppc)
    curInniArm <- new inniArm(curRetCap, inniBlock)
    armCount <- armCount + 1
inniStmt <- new inniStatement(inniArms)
return new coStmt(invocations, innitStmt, armCount)
```

We start by creating empty lists for the return capabilities, the original invocations the user wanted, and the invocations needed by the `inni` statement. We also set the arm count to 0. Then for each arm of the `co` statement we proceed as follows. First we parse the individual parts of each arm. Next we construct the capability we will need which will take a single argument of the type the current invocation should return and will return a void value. We insert this declaration into the parse tree. We then create an invocation statement corresponding to the `cocall` or `cosend` equivalent (see Section 3.2.4) of the original invocation. We add an assignment expression to the parse tree which calls this invocation and assigns the return value to our capability. Next, a new block is created. If the current arm had an assignment or post-processing code, we insert these into our new block. We continue by creating a new arm for an `inni` statement consisting of the return capability and our new block. We add this to our list of inni arms, and finally, we increment the arm count. Once we have gone through all the arms we create a new `inni` statement with all the arms we have created. Finally, we return a new `co` statement which has the list of invocations, the inni statement, and the arm count. This new `co` statement will be inserted into the parse tree by the calling function. When the semantic analysis and code generation are done, the declarations of the capabilities and the invocations will be processed as they normally would. The `inni` statement is processed as part of the processing for the `co` statement.

3.2.4 Implementation of `cocall` and `cosend` methods

Thus far we have assumed that we have `cocall` and `cosend` functions and they work as we need. This section explains what they do and how they work. The full details of how the regular `call` and `send` methods work can be found in [3] or [1].

Recall from Section 3.2.2 that every time the JR compiler encounters a capability, operation, or op method, it generates a Java class that contains all the functionality needed by objects of that particular type. Among other things, this class contain `call` and `send` methods that are used whenever a `call` or `send` statement is encountered in the JR code. In order to implement the `co` statement, we added a set of `cocall` and `cosend` functions to be used instead of their standard counterparts when a `call` or `send` statement is used as an invocation in a `co` statement.

First we will consider the `call` and `coccall` methods that appear as part of the generated classes.

When the JR compiler runs across the line

```
op int f(int);
```

it produces a Java class named `InOp_intToint_impl`. The parts of the `call` method of this class which are relevant to us look as follow:

```
public int call(long JRtimestamp, int arg0) throws /*... exceptions*/
    {
      ...
      Op_intTovoid.JRProxyOp retOp =
          new Op_intTovoid.JRProxyOp(new InOp_intTovoid_impl(false));
```

```
  this.send(JRtimestamp, arg0, retOp, null, null);
  ...
  Recv_intTovoid ret;
  ...
  ret = retOp.recv();
  ...
  return ret.arg0;
}
```

The important thing to note here is the creation of an operation named `retOp` which is of type `Op_intTovoid.JRProxyOp`. The `int` part of the type come from the fact that our operation returned a value of type `int`. The `void` part comes from the fact that we do not expect any further return values. When the `send` method is supplied a reference to this object, it will send a message through it which alerts the `call` method when the invocation has been serviced. Once the call to `send` is made, we invoke `retOp`'s `recv` method, which blocks until the `send` method sends its message. The return value the user is expecting is passed as the argument of this operation, and this is the value that is returned.

In order to modify this to work as it should within the context of a `co` statement, we make the `cocall` method behave as follows:

```
public Cap_intTovoid cocall(long JRtimestamp, int arg0) throws /*...*/
{
  ...
  Op_intTovoid.JRProxyOp op = new Op_intTovoid.JRProxyOp(new InOp_intTovoid_imp());
  Cap_intTovoid retOp =
      new Cap_intTovoid (op);
  this.send(JRtimestamp, arg0, op, null, null);
  return retOp;
  ...
}
```

First note that the return type has changed. Rather than returning an `int`, we now return a capability that takes an `int` as an argument and has return type `void`. This is because the `cocall` itself will no longer block until the invocation has been serviced. Rather, it will return a capability which will be used by an `inni` statement (see Section 3.2.1). The `retOp` object created in the original `call` statement is created just as before but with a new name of `op`. Just as before, we pass this to the `send` method. However, rather than invoking its `recv()` method, we create a capability that refers to this object and return it to the caller. As mentioned above, an `inni` statement will wait for this to be serviced, allowing us to use `call` statements within a `co` statement and seeing the behavior we expect.

The `send` method in our class needs to undergo slightly more complicated changes since it does more of the work. For our class above, the `send` statement looks like this:

```
public void send(long JRtimestamp, int arg0, Op_intTovoid.JRProxyOp retOp,
    edu.ucdavis.jr.RemoteHandler handler, Exception thrown) throws ... /*exceptions*/
{
    Recv_intToint newmsg =
        new Recv_intToint(arg0, retOp, handler, thrown, jrvm.setTimestamp(JRtimestamp));
    synchronized (this)
    {
      synchronized (this.wait_list)
      {
        this.wait_list.add(newmsg);
      }
    }
    ... /*code that waits for the 'wait_list' to be empty*/
}
```

What happens here is slightly more complicated than before. First note that the return type has been changed to `void` since making an assignment from a `send` statement is not legal in JR. Any information that is to be shared must go through the arguments that were passed to `send`. The first thing the `send` method does is create a new message object with a reference to `retOp`. Note that this is what `call` and `cocall` passed to `send`. Also note that when the JR compiler encounters a `send` statement, it passes a `null` reference for `retOp`. The message is then added to a list of messages which the `send` method must wait for before it returns (if the reference is `null`, it is 'serviced' immediately). Once all of this has been completed, the `send` method is completed.

In order to get the behavior we want inside a `co` statement, we create the `cosend` method as follows:

```
public Cap_intTovoid cosend(long JRtimestamp, int arg0,
 Op_intTovoid.JRProxyOp retOp, edu.ucdavis.jr.RemoteHandler handler,
 Exception thrown) throws ...
{
    Op_intTovoid.JRProxyOp op = new Op_intTovoid.JRProxyOp(new InOp_intTovoid_impl());
    op.send(JRtimestamp, 0);

    Recv_intToint newmsg =
        new Recv_intToint(arg0, retOp, handler, thrown, jrvm.setTimestamp(JRtimestamp));
    synchronized (this)
    {
      synchronized (this.wait_list)
      {
        this.wait_list.add(newmsg);
      }
    }
    ...
    return new Cap_intTovoid(op);
```

```
}
```

Like the `cocall` method, the `cosend` method changes the return type of its regular counterpart to a capability. The main change in the body is the creation of a new operation. As soon as it is created, we do a `send()` on it to signify that we are no longer waiting for it. At the end of the function, we create a new capability from this operation and return it to the caller. As with the `cocall`, this is the capability the `inni` statement will wait for. Note that the rest of the function remains unchanged. We still create a new message object and add it to our list. Since `retOp` should always be `null`, we will never wait for it to be serviced. We only leave this in is so that in the future if the `cosend` method is needed for a different function, it will already be more like the `send` method and will require less modification.

With the changes described above, the `cocall` and `cosend` methods for operations work as we intended. The `cocall` and `cosend` methods for capabilities call the respective methods for the operation to which they refer, so the changes there are even smaller. Process operations require the creation of a new thread for the `cocall` methods just as in the `send` method. The difference between this and the case of operations is minor, so it is not outlined here.

Chapter 4

Implementation Issues

4.1 Support for `break` and `continue` Statements in Post-Processing Code

The JR compiler now needs to allow `break` and `continue` statements anywhere inside the post-processing code of the `co` statement. Recall from Section 3.1.2 that the desired behavior upon encountering a `continue` statement inside a post-processing block is to stop the execution of that block and then continue waiting for the rest of the invocations to be serviced. Similarly, upon encountering a `break` statement, execution of that block should terminate and we should stop waiting for the rest of the invocations. Based on how the `co` statement is implemented, one might think that since the `break` or `continue` statements will ultimately be inside a loop in the resulting Java code, there would not be a problem. While it is indeed true that the resulting Java code would compile without problems, when the JR compiler runs across one of these statements, it would complain if the statement is not yet in a legal context.

To allow `break` and `continue` statements anywhere would stop the JR compiler from complaining when encountering such a situation, but it would allow users to put the statements in places where they would have no meaning. Another solution might be to allow them to be placed in the blocks of the arms of `inni` statements since the semantic translation maps post-processing blocks to `inni` arm blocks, but this would lead to the same problem as before.

The solution we implement is based on the second solution mentioned above. What we do is create a new type of parse tree node. It extends the parse tree node associated with the `inni` arm and implements the loop interface required to allow `break` and `continue` statements. We then add this context to the list of contexts where `break` and `continue` statements are allowed and everything works as intended.

4.2 Changing Capability Types

Perhaps one of the most complicated parts of the implementation of the `co` statement is ensuring the synthesized capabilities have the correct type. As a reminder, when the JR compiler encounters a `co` statement, for each arm it must insert a capability into the parse tree that takes as an argument the return type of that arm's invocation and has a void return type. As an example, when the code below is encountered, capabilities of type `intTovoid` and `charTovoid` must be synthesized.

```
cap int (boolean) f;
cap char (String) g;
...
co f(true);
[] g("Hello");
```

In the example above, we already know the return types of `f` and `g`. It would not be too difficult to synthesize the required capabilities in this case. However, we cannot always know the return types of these capabilities on the first pass (for example, `f` and `g` could be class variables and be declared after their use in a `co` statement).

Rather than distinguishing between the two above cases, we assume we never know the return type in advance. To compensate, we assume it is of type `int` (any other type would work equally well). We synthesize all the variables and statements we need under this assumption. Then we pass a reference to this type to the corresponding arm. When we check the arm for semantic correctness, we can calculate the correct type. We then use this information to change the reference of our synthesized type to the appropriate type.

Though relatively straightforward to describe, this solution is somewhat complicated to implement. We must first ensure that synthesized capability and its corresponding arm share a reference to the same Type object. Furthermore, we must ensure that everything is checked for semantic correctness in the right order: first the arms of the `co` statement, then the synthesized assignments, and finally the synthesized `inni` statement. If things are not done in this order, the JR compiler may complain about incorrect types, usage of undeclared variables, and usage of uninitialized variables. Doing things in the above order results in correct generated code and the desired behavior.

4.3 Naming Conflicts

The implementation of the `co` statement requires many synthesized variables. One of the potential pitfalls with this is naming conflicts. More specifically, every arm of every `co` statement requires a synthesized capability. In order to avoid naming conflicts, we keep a global count of the number of synthetic capabilities created for the current file. We then simply append the number to the string "JR_co_retop." In order to avoid naming conflicts accross different files, we also append the name of the class and file where the statement appears. Although it is not a difficult or innovative solution, its description is included here for completeness.

4.4 Generating the Right Classes

As mentioned in Section 3.2.4, all JR types which can be used in a `call` or `send` statement now have internal `cocall` and `cosend` functions. These return capabilities can potentially be of types which may not otherwise appear in the user's code. If this is the case, then the corresponding classes will not be generated, and the generated Java code will not compile. To compensate, we err on the side of caution. Any time an operation, a capability, or an op method is encountered by the JR compiler, it now generates not only the class associated with the parsed type, but also a class for a capability that takes the parsed return type as an argument and has a `void` return type. Since this new class requires a capability of type `voidTovoid`, we always generate this class. Furthermore, since the return type of the synthesized capabilities is always `void`, we do not have to worry about any other cases.

We implement the above by inserting synthesized nodes into the parse tree. For example, if a class has an operation as a field, we add the synthesized capability as a field. If a class method declares a local capability, we give it a new capability of our desired type. When the tree is traversed for code generation, all the correct classes are generated. Although it works, this is not the best solution. It requires inserting many synthetic nodes into the parse tree. A more efficient and natural solution would be to simply generate the required classes without inserting anything extra into the parse tree. We hope to change the implementation to this more efficient method in the near future.

Chapter 5

Conclusions and Future Work

The JR language now has full support for a concurrent invocation statement. We were able to implement all of the features and behavior described in Section 3.1. The programs described in this document and other test cases that have been developed work as intended.

There are several areas for further work where we hope to continue. An important aspect that needs to be considered is exception handling. Since we may finish with a `co` statement before every arm's invocation has been serviced, there is no way to catch an exception the rest of the invocations may throw. We need to design and implement an exception-handling scheme similar to that available to `send` statements. Another area for future work is that of allowing quantifiers with the `co` statement. This would be used to make code like that in Figure 2.5 cleaner. For example, the `co` statement in that code could look like this:

```
co ((int i = 0; i < numFriends; i++)) vote = askFriend[i](){
    if(vote){
        goCount++;
        if(goCount > numFriends / 2)
            break;
    }
    else{
        stayCount++;
        if(stayCount > numFriends / 2)
            break;
    }
}
```

This code would behave just as before, but we could now allow for a dynamic number of arms. A third feature that could be added would be to allow creation of virtual machines in place of the invocation statements. Along those same lines, we may want to allow arbitrary code and create a `cobegin` statement. This would be much more complicated as it would require the creation and management of new threads. As for some shorter-term goals, we need to go back and clean up the JR implementation (e.g., as suggested in Section 4.4). We also need to do some performance evaluation to see the effects of the new features on the run-time performance of JR programs.

Bibliography

[1] The JR Concurrent Programming Language, 2004. `http://www.cs.ucdavis.edu/~olsson/research/jr`.

[2] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1993.

[3] R. A. Olsson and A. W. Keen. *The JR Programming Language: Concurrent Programming in an Extended Java*. Kluwer Academic Publishers, Boston, to appear in 2004.