CAL Runtime Internals

Contributors: Raymond Cypher
Last revised: August 3, 2007

Please Note: This document is not yet complete. It is provided in its current state in order to solicit ongoing feedback as it is written.

Contents

1 LECC Runtime

The LECC runtime is a non-interpreted Java runtime for CAL programs. It is expressed as a set of core classes, which define how a CAL program is evaluated as well as defining some critical built-in data types (ex. Integers, Booleans, etc.) Non-kernel code is generated by an LECC specific code generator from the expression form produced by the compiler (potentially with additional middle-tier optimizations applied).

The code generator directly produces Java bytecode which defines new classes derived from the core runtime classes. These new classes implement the CAL data types and CAL functions defined by the user.

2 Simplified Overview

At its most fundamental level the LECC machine is a graph reducer with the reduction logic defined in the core graph node classes. The logic of the defined CAL functions is implemented in a series of generated Java classes which extend the core runtime classes.

What follows is an overview of the base graph reducing machine.

2.1 Core Classes

RTValue
Everything in the LECC system is of type RTValue. RTValue is an abstract base class which represents two kinds of values: 1) RTData – a non-reducible instance of a value type, 2) RTFunction – a reducible function value..

RTData
RTData is an abstract class from which 'concrete' data value classes for primitive types are derived. Such classes are types and constructors for the data they represent. If a data type has only one constructor (for example a CAL_Integer), then a direct subclass with appropriate Java constructor can represent the value type. If a data type has a finite number of values (for example a CAL_Boolean), then an abstract class inherits from RTData which represents the CAL type, and within this are nested derived classes for each CAL constructor (e.g. CAL_Boolean.CAL_True and CAL_Boolean.CAL_False).

The Java values held in these classes are retrieved by access functions such as ‘getIntValue()’, ‘getByteValue()’, etc. The RTValue class actually implements versions of all these access functions. The base implementations in RTValue throw a runtime error and are meant to be overridden the the specific data classes. Providing the base implementations in RTValue allows for more efficient code generation as the generated code can simply invoke an access method on an instance of RTValue without having to down cast to the appropriate RTData subclass.

To cut down on memory allocations and overall memory usage some of these classes cache commonly used instances for re-use. For example the CAL_Byte class caches instances of itself for the values -128 through 127, covering all possible values. Other classes, such as CAL_Int, cannot cache all possible values but do cache a set of commonly used values.

```
public static final CAL_Int make(int i) {
    if (i >= -128 && i <= 127) {
        return cachedIntegers[i + 128];
    }
    return new CAL_Int(i);
}
```

RTFunction

RTFunction is an abstract class from which 'concrete' functional classes are derived. There are basically two forms of RTFunction: RTSupercombinator and RTResultFunction.

RTFunction.RTSupercombinator

This represents an unapplied function implementation. An RTSupercombinator is not reducible. This is the base class for the generated classes corresponding to CAL functions.

RTFunction.RTSupercombinator.RTCAF

This is the abstract base class for constant applicative forms. I.e. CAL functions with no arguments.

RTFunction.RTSupercombinator.RTCons

This is the abstract base class for generated classes corresponding to CAL data types. For a CAL data type an abstract type class representing the data type extends RTCons. Each data constructor is represented by a concrete class extending the type class. These concrete classes represent the functional form of the data constructor, with individual instances of the class representing realized instances of the data constructor.

RTFunction.RTResultFunction

This is the abstract base class representing functional forms which can be reduced. After reduction a reference to the result is maintained.

RTFunction.RTResultFunction.RTZeroArityFunction
This is the base class for functions with no arguments that are not considered constant applicative forms. At this point the only functions in this category are foreign functions with no arguments.

RTFunction.RTResultFunction.RTApplication
This class represents the application of an argument (RTValue) to a function (RTFunction). RTApplication instances are updated to the result of applying a supercombinator to an argument chain when the fully saturated chain is reduced.

RTExecutionContext
This class defines the context in which program evaluation is taking place. An instance of this type is passed through program evaluation.

2.2 Evaluation

A graph is evaluated by stepwise reduction until the result is in weak head normal form. A graph node which is already in weak head normal form will simply return itself when reduced.

For example the definition of the evaluate method in the RTValue class is:

```
public RTValue evaluate(RTExecutionContext ec) {
   // Base class implementation simply returns 'this'.
   return this;
}
```

In the RTResultFunction class evaluate is defined as:

```
public RTValue evaluate(RTExecutionContext ec) {
   RTValue newResult = result == null ? this : result;
   RTValue lastResult;

      // Attempt to reduce this result
      do {
         lastResult = newResult;
         newResult = newResult.reduce(ec);
      } while (lastResult != newResult);

      if (newResult != this)
         setResult (newResult);

   return newResult;
}
```

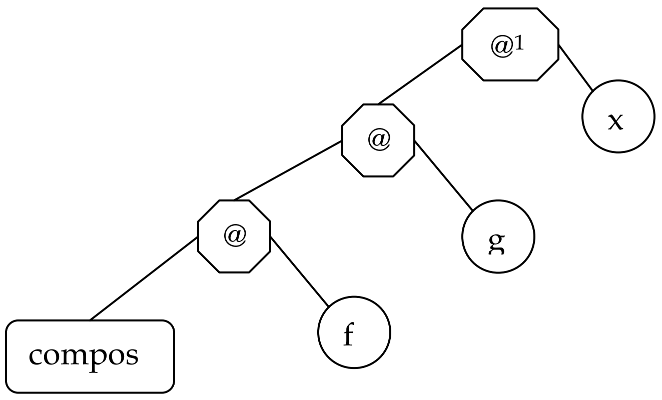
That is: the current node is reduced until reduction fails to produce a change in value. At this point the result reference is set and evaluation terminates.

Reduction of a graph value already in weak head normal form node does nothing. I.e. the node simply returns itself.

Reduction of a supercombinator application is accomplished by walking to the left hand side of the graph until an RTSupercombinator instance is found. The RTSupercombinator instance will be an instance of a generated class corresponding to a CAL function. The generated class implements an 'f' method which executes the logic of the CAL function. The 'f' method of the supercombinator instance is invoked with the root of the application chain passed as an argument. The generated code will extract the argument values from the application chain and apply the appropriate logic.

Take for example the CAL function compose:
```
compose f g x = f (g x);
```

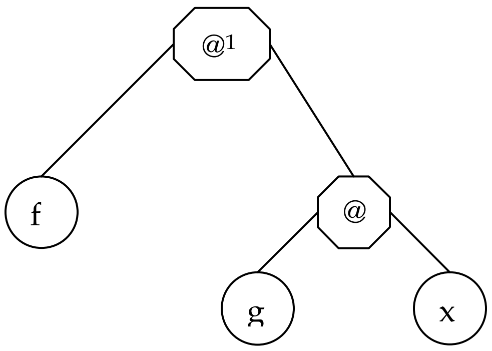
The graph for an application of compose would be:



RTApplication nodes are represented by @. The root of the application is the node marked as @1.

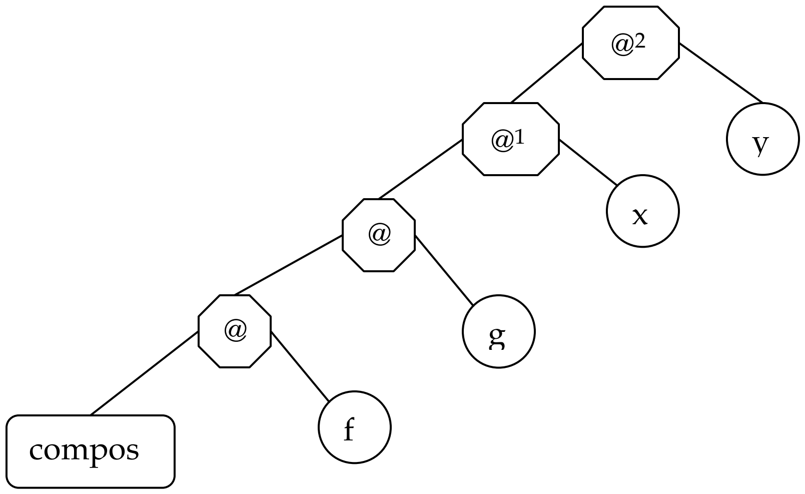
The 'f' method in the generated Java class will look something like:
```
public final RTValue f(RTValue $lastArg, RTExecutionContext $ec) {
   // Arguments
   RTValue x = $lastArg.getArgValue();
   RTValue g = ($lastArg =  $lastArg.prevArg()).getArgValue();
   RTValue f = $lastArg.prevArg().getArgValue();

   return f.apply(g.apply(x));
}
```

The ‘f’ method retrieves the arguments to the function by walking the application chain and extracting the argument sub-graphs. Finally the function logic is executed. In this case a new set of applications is built to form the graph *f (g x)*. The result graph will be:



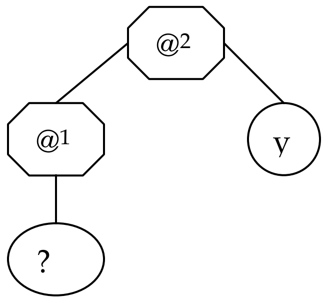
An under saturated supercombinator application (i.e. the case where there are not enough arguments for the CAL function to be evaluated) cannot be reduced. Trying to either evaluate or reduce such an application will simply return the application unchanged.

An over saturated supercombinator application is handled by finding the root of the sub-graph which represents a fully saturated application and first evaluating this sub-graph.
For example, an over saturated application of compose would look like:



In this case compose is applied to four argument when the arity of compose is only three. Reducing this graph involves first evaluating the sub-graph which has the @[1] node as its root. Once this sub-graph is in weak head normal form the original root (@[2]) can be reduced.

After evaluating the sub-graph with @[1] as the root we will have the following graph.



The reduced root @[1] is now an indirection to the result of the reduction. At this point reduction of the original root can be attempted again. Whether it can be reduced will depend on the result of reducing the sub-graph.

End of simplified overview.

3 Generated Java Code

3.1 CAL Functions

The LECC code generator generates a Java class for each CAL function. The generated class houses the logic defined in the body of the CAL function as well as implementing some helper methods which are used when reducing the graph.

Take the CAL function compose.
```
public compose f g x = f (g x);
```

The following is the generated Java class corresponding to compose.
It contains a static final field of type Compose, which is the singleton instance of this class.
The default constructor is made private to prevent any other instances of the class from being created.
The getArity() method returns the arity (i.e. number of required arguments) of the CAL function, and is used when reducing application graphs.
The methods getModuleName, getUnqualifiedName, and getQualifiedName all return Strings describing the CAL function and are used when debugging and generating error messages.

The f method is passed the root of an application graph and an execution context. The graph is decomposed to retrieve the argument values and the function logic is then executed.

```
public final class Compose extends RTSupercombinator {
   public static final Compose $instance = new Compose();

   private Compose() {
   }

   public final int getArity() {
      return 3;
   }

   public final java.lang.String getModuleName() {
            return "Cal.Core.Prelude";
   }

   public final java.lang.String getUnqualifiedName() {
      return "compose";
   }

   public final java.lang.String getQualifiedName() {
      return "Cal.Core.Prelude.compose";
   }

   /**
    * f
    * This method implements the function logic of the CAL function
    * Cal.Core.Prelude.compose
    */
   public final RTValue f(RTValue $lastArg, RTExecutionContext $ec){
      // Arguments
      RTValue x = $lastArg.getArgValue();
      RTValue g = ($lastArg = $lastArg.prevArg()).getArgValue();
      RTValue f = $lastArg.prevArg().getArgValue();
      return f.apply (g.apply (x));
   }
}
```

3.2 CAFs and Zero Arity Functions

CAFs (constant applicative forms) and zero arity functions require special handling. The ability of CAL to import foreign functions and use them in CAL code complicates the issue of CAFs and zero arity functions.

A zero arity foreign function is not necessarily a CAF. For example if the default constructor for a Java class is imported into CAL it will have an arity of zero.

However, treating this as a CAF would be incorrect as each invocation of the foreign function is expected to produce a different instance of the Java class.

Even when dealing with a zero arity non-foreign function there is a complication due to the ability of CAL to call external Java functions.
For example, the following code declares the Java method System.getProperty(String) as a foreign function called getSystemProperty. A CAL function called userID is defined which retrieves the system propery associated with the tag "userID".

```
foreign unsafe import jvm "static method System.getProperty"
getSystemPropery :: String -> String;
userID = getSystemPropery "userID";
```

Because the runtime can support multiple simultaneous users it cannot simply cache the first result of userID. This would result in all users having the same identity regardless of what their system property is set to. This is a situation where the execution context (an instance of RTExecutionContext) comes into play. The generated class for a CAF caches previously evaluated results with an association to the execution context. This means that a CAF will be evaluated once for each individual execution context.

The following fragment of Java code is from the generated class for a CAF. It shows the Map used to cache instances of the CAF. The Map is synchronized for thread safety and a weak hash map is used so that the lifetime of the cached value is tied to the lifetime of the associated execution context. The factory method 'make' will try to retrieve an existing value from the cache, based on the execution context. If there is no cached value then a new instance is created and added to the cache.

```
private static Map $instancesMap =
   Collections.synchronizedMap(new WeakHashMap());


public static final RTFunction make(RTExecutionContext $ec) {
   RTFunction newInstance =
      ((RTFunction)Test_Module.$instancesMap.get($ec));

   if (newInstance == null) {
      newInstance = (new TFullApp.General._0(Test_Module.$instance));
      Test_Module.$instancesMap.put($ec, newInstance);
   }
   return newInstance;
}
```

3.3 Type constructors and data constructors.

The generated Java for a data type consists of an outer class corresponding to the type constructor. Inner classes, which extend the type constructor class, are then generated for each data constructor. The data constructor classes contain a field for each field in the data constructor with appropriate access methods.
The following Java code fragment shows part of the generated Java code for a list data type:

```
data List a =
    Nil |
    Cons
        head :: a
        tail :: (List a)
    ;
```

The outer generated class is named TYPE_ List and extends the RTCons class. It corresponds to the CAL type constructor List. It should be noted that the type constructor class implements access functions for all the fields in the data constructors (in this case just head and tail). These implementations simply generate a runtime error and are intended to be overridden by the data constructor classes which contain these fields. The field access functions are implemented in the type constructor class to facilitate retrieving the field values without having to do a downcast from the type constructor class to the data constructor class. Avoiding the downcast, and corresponding runtime type check in the JVM, improves performance.

Each data constructor is assigned a unique ordinal value and the generated data constructor class implements a getOrdinalValue() method. This is used at runtime to efficiently resolve which data constructor a class instance represents.

The inner class corresponding to the Nil data constructor is very simple because there are no fields in the data constructor. There is a singleton instance of Nil which is used for all references to this data constructor.

The inner class for Cons, called CAL_Cons, is a bit more complicated. It contains two fields of type RTValue for the head and tail. There is also a static field which contains an instance of CAL_Cons. This instance is used when the data constructor is applied as a function. For example, it would be used for an application of the data constructor to a single field.

The default constructor is made private so that instances of CAL_Cons cannot be created without values for the two fields. The public constructor asserts that the provided field values are non-null.

Like the generated class for a function CAL_Cons contains an 'f' method. This method is used to evaluate an application of Cons to two arguments and simply creates a new instance of Cons.

The access functions for the head and tail fields are more complicated than might be expected. Rather than simply return the value of the field there is a check on the runtime type of the field and a possible reassignment. This is done to minimize the amount of memory used by the runtime and will be discussed further at a later point.

The class for Cons also contains a function called 'getFieldByIndex()'. This method is used when generating code for data constructor field selection expressions, which are discussed separately. Briefly, the getFieldByIndex method is declared in the RTCons class and is overridden by data constructor classes which have fields. The method takes three arguments: dcOrdinal is the ordinal of the expected data constructor, fieldIndex is the index of the desired field, and errorInfo gives information about the location of the call to getFieldByIndex. This call site information is used to generate a useful error message if the actual data constructor instance doesn't match the expected data constructor.

```
public abstract class TYPE_List extends RTCons {
    /** Constructor for TYPE_List */
        protected TYPE_List() {
        }

    /** Base implementation of 'head' accessor. */
        public RTValue get_head() {
            return badFieldAccessor("_head");
        }

    /** Base implementation of 'tail' accessor. */
        public RTValue get_tail() {
            return badFieldAccessor("_tail");
        }

        public static final class CAL_Nil extends TYPE_List {
            public static final CAL_Nil $instance = new CAL_Nil();

            private CAL_Nil() {
            }

            public int getOrdinalValue() {
                return 0;
            }

        }

        public static final class CAL_Cons extends TYPE_List {
            private RTValue _head;
            private RTValue _tail;

            public static final CAL_Cons $instance = new CAL_Cons();

            private CAL_Cons() {
            }

            public CAL_Cons(RTValue member0, RTValue member1) {
                assert ((member0 != null) && (member1 != null)) : (
                        "Invalid constructor argument for  Cons");
                _head = member0;
                _tail = member1;
            }

            public int getOrdinalValue() {
                return 1;
            }

            public final RTValue f (RTValue $lastArg, RTExecutionContext $ec) {
                // Arguments
                final RTValue $arg1 = $lastArg.getArgValue();
                final RTValue $arg0 =
                        ($lastArg = $lastArg.prevArg()).getArgValue();

                return new CAL_Cons($arg0, $arg1);
            }

            public final RTValue get_head() {
                RTValue _head$;

                if ((_head$ = _head) instanceof RTResultFunction) {
                        return _head = _head$.getValue();
                }
                return _head$;
            }

            public final RTValue get_tail() {
                RTValue _tail$;

                if ((_tail$ = _tail) instanceof RTResultFunction) {
                        return _tail = _tail$.getValue();
                }
                return _tail$;
            }

            public final RTValue getFieldByIndex(int dcOrdinal, int fieldIndex,
                       ErrorInfo errorInfo) {
                checkDCOrdinalForFieldSelection(dcOrdinal, errorInfo);
                switch (fieldIndex) {
                        case 0: {
                                return get_head();
                        }
                        case 1: {
                                return get_tail();
                        }
```

```
                        }
                        badFieldIndexInGetFieldByIndex(fieldIndex);
                        return null;
                }

        }
}
```

When the code generator encounters a data constructor application with all the necessary arguments it will generate code which simply creates a new instance of the data constructor class. Since the ‘f’ method of the data constructor class simply creates a new instance of the class without evaluating any arguments or doing any other work it is safe to generate such code regardless of whether the data constructor application occurred in a strict or lazy context. (For the exception to this see the section on strict data constructor fields)

A partial application of a data constructor will be handled by generating code to build an application graph. When the graph is reduced the ‘f’ method of the data constructor class will be invoked and will create the new data constructor instance.

There are two optimizations to the generated Java that come into play for data constructors with no fields. If a type constructor has multiple data constructors with no fields a single inner class is generated to represent the data constructors without fields. For example, the following data type has three data constructors, two of which have no fields.

```
data MyData =
    Data1 x :: Int | Data2 | Data3;
```

The generated Java code for this data type would include the following. A single inner class, called TagDC, is used to represent the Data2 and Data3 data constructors. The TagDC class has a single int field called tag, this field is used to determine which data constructor an instance of TagDC represents. The type constructor class has a static final field of type TagDC for each of the data constructors and a single instance of TagDC is used for all references to each data constructor.

```
public abstract class TYPE_My_Data extends RTCons {
        private static final TagDC Data2 = new TagDC(1);

        private static final TagDC Data3 = new TagDC(2);

        public static final class TagDC extends TYPE_My_Data {
                private final int tag;

                public TagDC(int tagVal) {
                        tag = tagVal;
                }
```

A further refinement comes into play when all the data constructors for a data type have no fields. In this case the LECC code generator treats all references to the data type as a reference to an int and replaces all data constructor references with the ordinal value for the data constructor. In this case there is no Java class generated for the data type.

3.4 Records

Records are represented by a set of core classes which extend RTValue. Records represent a set of name/value pairs. A field name can be either textual or ordinal.

RTRecordValue
This is the base class for record instances. It represents a record literal value in the runtime. In CAL source, this corresponds to a record such as: `{field1 = "abc", field2 = 2.0}`.
Records involving record-polymorphic record extension, such as:
`{r | field1 = "abc", field2 = 2.0}`
are not represented directly using RTRecordValue objects. Rather, r is first evaluated to an RTRecordValue, and then the extension is made. (In a lazy context, this is done through reducing RTRecordExtension, in a strict context, this is done directly).

RTRecordValue.EmptyRecord
This class is has a single instance which represents the empty record value.

RTRecordValue.OrdinalRecord
Class used to represent ordinal records. I.e.:
- All the fields in the record are ordinal fields
- The record is not a TupleRecord or the EmptyRecord.

For example:
`{#1 = "abc", #3 = True}` is an OrdinalRecord
`{#2 = 100.0, #3 = True}` is an OrdinalRecord
`{#1 = "abc", #2 = 100.0, #3 = True}` is a TupleRecord and not an OrdinalRecord.

RTRecordValue.TextualRecord
Class used to represent textual records. I.e.:
- all the fields in the record are textual fields
- not the EmptyRecord (i.e. there is at least 1 textual field).

For example:
`{name = "abc", flag = True}` is a TextualRecord.

RTRecordValue.MixedRecord
Represents records having both ordinal and textual fields, where the ordinal part is not a tuple i.e.
-there is at least 1 ordinal field, and at least 1 textual field
-the ordinal fields are not consecutive i.e. #1, #2, ..., #n.
For example, `{#1 = 2.0, #3 = "FooBar", name = "abc", flag = True}` is a MixedRecord.

RTRecordValue.TupleRecord
Class used to represent tuple-records i.e.
-all the fields in the record are ordinal fields
-the ordinal fields are consecutive i.e. #1, #2, ..., #n.
-not the empty record
For example, `{#1 = "abc", #2 = 100.0, #3 = True}`

RTRecordValue.TupleMixedRecord
Represents records having both ordinal and textual fields, and such that the ordinal part is a tuple. I.e.:
-there is at least 1 ordinal field, and at least 1 textual field
-the ordinal fields are consecutive i.e. #1, #2, ..., #n.
For example:
`{#1 = 2.0, #2 = "FooBar", name = "abc", flag = True}` is a TupleMixedRecord.

CAL code which extends a record (ex. `{r | foo = 1.0, blah = “blah”}`) will generate different Java code based on whether the it occurs in a strict or lazy context. In a strict context 'r' would be evaluated to an RTRecordValue and then a new record would be created which would be the extended record.
For example, in a strict context the CAL expression above would generate the following java:

```
r.evaluate($ec).makeTextualRecordExtension(
      new String[]{“foo”, “blah”),
      new RTValue[]{RTData.CAL_Double.make(1.0),
                RTData.Cal_String.make(“blah”)})
```

As can be seen 'r' is evaluated then a class method on 'r' is used to create the new, extended, record passing in the new field names and values.

A special class is used to represent a record extension found in a lazy context. RTRecordExtension is the base class for this. It has derived classes for the different kind of extensions (i.e. textual, ordinal, tuple, mixed, and tuple/mixed) and holds the base record and the field names/values in an unevaluated state until the extension is evaluated.

3.5 Generated Java for CAL Language Constructs

3.5.1 Let variables

A let variable in CAL corresponds to a local variable in the generated Java code. For example:

```
    let
        foo = blah x y;
    in
        ….;
```

Would ideally generate a local variable like:

```
RTValue foo = Blah.$instance.apply(x).apply(y);
…
```

The graph corresponding to 'blah x y' is build and assigned to a local variable of type RTValue named 'foo'.

Unfortunately things are not quite so simple in practice. The Java bytecode format has a limit of 64kB on the size of the bytecode in a single method. Experience has shown that CAL functions can exceed this size when a large number of complex let variables are present. The solution is to lift the let variable definition into a separate method in the generated Java class. The let variable definition is analyzed to determine the free variables and these are then passed as arguments to the lifted function. The above code will generate an extra method 'foo$2$def' which can be invoked to build the lazy graph corresponding to the right-hand-side of the let binding.

```
private RTValue foo$2$def (RTValue x, RTValue y) {
   return Blah.$instance.apply(x).apply(y);
}

…
RTValue foo = foo$2$def (x, y);
…
```

A number of optimizing transformations are applied to let variables. The goal of these is to transform CAL code to a form from which the LECC runtime can generate more efficient Java code.
If a let variable is defined as a literal value (ex. x = 1.0;) the local variable is removed and all references to it are replaced by the literal value.

Similarly if a let variable is defined as another local variable or a data constructor or a function the local variable is removed and all references replaced with defining entity.
If a let variable is never referenced in the body of the let expression the variable is simply eliminated.
If a let variable is referenced only once the call to the lifted definition function is in-lined at the point of the let variable reference.
A let variable which is only reference in the binding of another let variable will have it's declaration moved into that binding.
Let variables are also floated inwards. This means that the let variable declaration will be moved as far inward in the function body as possible. In the case of the LECC runtime the let variables are duplicated to allow moving the let variables as for inward as possible.

For example start off with the following CAL code :

```
let
    a = 1.0;
    b = foo a;
    c = b + 1.0;
    d = c + 1.0;
    e = foo 2.0;
in
    if (b > 2.0) then
        c
    else if d > 2.0 then
        c
    else
        d + e;
```

Substituting the literal value 1.0 for references to 'a' and in-lining the definition of 'e' since it is only referenced once will produce the following. The local variable 'a' has been eliminated, corresponding to one less local variable in the generated Java code. The in-lining of the definition of 'e' now allows it to be compiled strictly offering another performance gain.

```
let
    b = foo 1.0;
    c = b + 1.0;
    d = c + 1.0;
in
    if (b > 2.0) then
        c
    else if d > 2.0 then
        c
    else
        d + (foo 2.0);
```

Floating the 'd' variable inwards means that we only incur the overhead of declaring a local variable, and building the associated graph, if the execution path containing d is taken. This would correspond to the following CAL code.

```
let
    b = foo 1.0;
    c = b + 1.0;
in
    if (b > 2.0) then
        c
    else
        let
            d = c + 1.0;
        in
            if d > 2.0 then
                c
            else
                d + (foo 2.0);
```

The 'c' variable can now be floated inwards. In this case we end up duplicating the variable, producing the following CAL code. This transformation itself doesn't produce more efficient code but allows further transformations to be applied.

```
let
    b = foo 1.0;
    in
    if (b > 2.0) then
        let
            c1 = b + 1.0;
        in
            c1
    else
        let
            c2 = b + 1.0;
            d = c2 + 1.0;
        in
            if d > 2.0 then
                c2
            else
                d + (foo 2.0);
```

The new variable 'c1' is only referenced once so its definition can be in-lined. This eliminates a local variable and allows for a more efficient compilation of the body (b + 1.0) since it is now in a strict context.

```
let
    b = foo 1.0;
    in
    if (b > 2.0) then
        b + 1.0
    else
        let
            c2 = b + 1.0;
            d = c + 1.0;
        in
            if d > 2.0 then
                c2
            else
                d + (foo 2.0);
```

3.5.2 If then else

The Java code generated for an if-then-else will depend on the context in which it is encountered.
When encountered in a lazy context a graph of the 'if' function will be built.
For example `foo (if a then b else c);` would generate java code:

```
   Foo.$instance.apply(If.$instance.apply(b).apply(c));
```

When encountered in a strict context the Java '?' operator is used. For example `case (if a then b else c) of -> …` would generate java code:

```
   switch (a.evaluate().getBooleanValue() ? b : c) ...
```

If an if-then-else is encountered in a context that will correspond to the top level of the corresponding generated Java method the Java if-then-else will be used.
For example:

```
foo x y z = if x then y else z;
```

The body of the generated Java function would be:

```
if (x.evaluate().getBooleanValue()) {
   return y;
} else {
   return z;
}
```

3.5.3 Case

The CAL compiler ensures that case expressions will only be encountered at the top level of a function body, lifting case expressions into separate functions if necessary. This means that the LECC code generator can generate a Java switch for CAL case expressions.
CAL supports case expressions on integers, characters, and data constructors. In Java a switch can only be performed on an int. CAL integers and characters have a direct correspondence to a Java int value. For data constructors the ordinal value associated with each data constructor is used.

In the generated Java code the expression being switched on is evaluated and assigned to a local variable. The switch is then based on the ordinal value of the result. The different switch alternates will then use the access functions to retrieve field values from the data constructor instance.

Take, for example, the following CAL code. A data type with two data constructors and a function which extracts the int field from the data constructors are declared.

```
data public MyData =
    public Data1
        x :: Int
        y :: String
    |
    public Data2
        z :: Int
;

foo  :: MyData -> Int;
public foo myData =
    case MyData of
    Data1 a b -> foo2 a a;
    Data2 a -> foo3 a a a;
   ;
```

The generated Java for the function body of 'foo' would be as follows. As can be seen the expression being switched on is evaluated and the result is bound to a local variable of type Type_MyData. The individual cases retrieve field values and assign them to local variables, which are used in the subsequent expression. Note that the alternate for ordinal zero (i.e. Data1) no local variable is declared for b. This is because the variable is never referenced.
Also the local variable $case does not have to be cast to an instance of Data1 or Data2 to retrieve the field values. As noted previously when discussing generated code for type constructors and data constructors the class corresponding to the type constructor has a default implementation access function for all fields in the data constructors.

```
Type_MyData $case;

switch(($case = (Type_MyData)myData.evaluate()).getOrdinalValue()) {
   case 0: {
      // Data1
      RTValue a = $case.get_x();
      return  Foo3.$instance.apply(a).apply(a).apply(a);
   }

   case 1: {
      // Data2
      RTValue a = $case.get_z();
      return  Foo3.$instance.apply(a).apply(a).apply(a);
   }

   default: {
     return badSwitchIndex(Map.Cal_Core_Prelude_map_3068_5);
   }
}
```

There are a variety of optimizations to the generated Java code for CAL case expressions.

A case on a boolean value will generate an if-then-else in the Java code.

If no data constructor fields are used in any of the case alternates the local variable for the result of evaluating the case expression will be removed from the Java code.

A case in which all the alternates resolve to True or False resolves to generated code where a simple check of the ordinal value is performed. This pattern often occurs when writing a function that is checking the type of value. For example:

```
case x of
Data1 x y -> True;
Data2 x -> False;
Data3 -> True;
```

Would generate something like:

```
int type = x.evaluate().getOrdinalValue();
return (type == 0 | type == 2);
```

When the case expression is a data type which only has one data constructor the switch is eliminated from the generated Java code.

3.5.4 Data constructor field selection

CAL supports the ability to unpack individual data constructor fields without using a case expression. Using the syntax **a.DataConstructor.field** where 'a' is an expression that resolves to an instance of 'DataConstructor' the specified field is retrieved.
When encountered in a strict context the generated Java code evaluates the data constructor expression and uses the getFieldByIndex method to retrieve the desired field. For example **a.Cons.head** would generate something like:

```
((Type_List)a.evaluate()).getFieldByIndex(…)
```

When encountered in a lazy context a core runtime class, RTResultFunction.RTDataConsFieldSelection is used to represent the field selection. This class holds the data constructor expression, the data constructor ordinal, field index, and call site information. The generated Java code creates an instance of RTDataConsFieldSelection.

```
new RTDataConsFieldSelection (a, 0, 0, …);
```

When reduced the RTDataConsFieldSelection instance will evaluate the data constructor expression, and set its result reference to the retrieved field.

```
protected RTValue reduce(RTExecutionContext $ec) {
   if (dataConsExpr != null) {
      setResult (((RTCons)dataConsExpr.evaluate($ec)).
         getFieldByIndex (dcOrdinal, fieldOrdinal, errorInfo));
   }
```

```
    return result;
}
```

3.5.5 Foreign Functions

A function class extending RTSupercombinator will be generated to correspond to each foreign function. This class will be used for applications of the foreign function, the same as with non-foreign functions.

The f method in the generated class for a foreign function is slightly different than for a regular CAL function. The generated code forces the evaluation of the arguments to the foreign function, extracts the required Java value from each argument, calls the external Java method and wraps the result in a new graph node.

For example:

```
foreign unsafe import jvm "static method MyClass.myMethod"
    public myForeignFunction :: Int -> Int -> Int;
```

The body of the f method in the generated Java class for myForeignFunction would be:

```
return RTData.CAL_Int.make(
    MyClass.myMethod(
        arg1.evaluate().getIntValue(),
        arg2.evaluate().getIntValue()));
```

3.5.6 Primitive Functions

A primitive function is a function whose definition is built into the runtime. In CAL source a primitive function has a type declaration but no function definition.

```
        primitive public seq :: a -> b -> b;
```

In the LECC runtime these functions are implemented by core classes which extend RTSupercombinator. These classes implement all the methods necessary to participate in graph reduction in the same fashion as the generated function classes. When encountering a reference to one of these primitive functions the LECC code generator knows to use a reference to the appropriate built-in class, rather than a generated class.

Internally the LECC machine treats a specific set of primitive functions as primitive operations. These are operations that correspond to Java operators. Rather than having a built-in core runtime class that implements function logic the primitive operations are generated at runtime using the appropriate Java operators.

For example, the Prelude module declares the primitive function addInt. This function corresponds to the Java + operator on int values.

```
primitive private addInt :: Int -> Int -> Int;
```

A function class will be generated for this primitive function. It differs from the generated function class for other CAL functions in the body of the 'f' method. In this case the body of the f method in the generated Java class will use the Java '+' operator. As can be seen the arguments to the function are evaluated and the appropriate Java value is extracted.

```
return RTData.CAL_Int.make(
a.evaluate().getIntValue() + b.evaluate().getIntValue());
```

3.5.7 Tail recursion

The LECC code generator optimizes tail recursive CAL functions. Since Java doesn't have special support for tail recursion in the JVM the code generator converts the tail recursion to a loop when generating the Java code.
Take, for example, the following function which retrieves an element from a list. The line marked in bold is a tail recursive call.

```
subscript :: [a] -> Int -> a;
public subscript list index =
    case list of
    listHead : listTail ->
        if index == 0 then
            listHead
        else if index > 0 then
            subscript listTail (index - 1)
        else
            error "Negative index.";

    [] -> error "Index out of bounds.";
    ;
```

The following fragment of Java shows part of the 'f' method generated for subscript. The first line in bold shows that the method body is put inside a labeled unconditional loop. The other lines marked in bold show the location of the tail recursive call. Note that the arguments to the method are assigned new values and the execution returns to the top of the loop. In cases where there are interdependencies between the current values of the function arguments and the new values temporary variables are used as necessary to preserve correctness.

```
public final RTValue f2(RTValue list, RTValue index$,
                  RTExecutionContext $ec) {

   list = list.evaluate($ec);
   int index = index$.evaluate($ec).getOrdinalValue();
   TRLoop: while (true) {
      // Top level supercombinator logic
      TYPE_List $case1;

      switch (($case1 = ((TYPE_List)list))).getOrdinalValue()) {
         case 0: …;

         case 1: {
                        // Cal.Core.Prelude.Cons
            // Decompose data type to access members.
            RTValue listHead = $case1.get_head();
            RTValue listTail = $case1.get_tail();

            if (index == 0) {
               return listHead;
            } else {
               if (index > 0) {
                  list = listTail.evaluate($ec);
                  index = (index - 1);
                  continue TRLoop;
               } else …
```

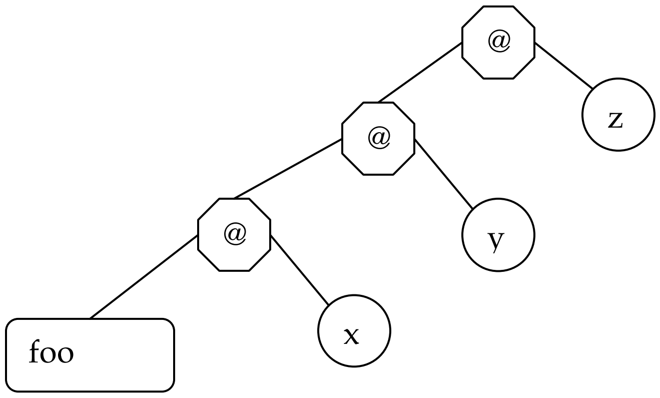
3.5.8 Record case

4 Optimizations to the Generated Java

4.1 N Argument Application Nodes

The majority of the graph building which takes place at runtime consists of building up applications. In the most basic form these applications are built up using the RTApplication class as described previously. However there is opportunity in this area for large gains in efficiency.

In many cases an application expressed in CAL code is the application of a known CAL function to the number of arguments required by that function. For example `foo x y z` where foo is a declared CAL function which takes 3 arguments. If encountered in a lazy context the generated Java will build a graph representing this application. Building the graph using RTApplication instances would result in the following.



As can be seen this involved allocating three new application nodes. It is significantly more efficient to allocate a single instance of a class which holds the function and the three arguments as fields. In addition, since it is known that the number of arguments supplied matches the number of arguments required by the function the reduction of an instance of such a class would be much simpler as no checks need to be done.

The LECC core runtime includes graph node classes for exactly this purpose. Originally a single class was used, which held a function and an array of arguments. However it was determined that separate allocation of the array used to hold the arguments was much more expensive than allocating an instance of a class which used fields to hold the arguments. There are a potentially infinite number of such classes, one for each number of arguments. Obviously the LECC runtime cannot have an infinite number of such application classes. However, analysis of existing CAL code showed that the vast majority of cases could be handled by having classes for up to fifteen arguments.

The class hierarchy for these n-argument application node classes is:
RTResultFunction.RTFullApp.General._1
RTResultFunction.RTFullApp.General._2
Etc.

Where the number in the class name indicates the number of arguments it holds.

The CAL expression `foo x y z`, as described above, would now translate into the following Java code.

```
        new  RTFullApp.General._3(foo, x, y, z)
```

This node can be held until reduced at some future point in time.

The base class for these full application nodes, RTFullApp, implements final versions of the methods getArity() and getArgCount(), simply returning zero for both values. This allows an instance of RTFullApp to appear as a zero argument function when involved in graph reduction.

These new application nodes have an effect on the Java code generated for CAL functions. Previously the function logic was implemented in the f method of the generated class. The f method was passed the graph root as an argument and decomposed the graph to retrieve the argument values for the CAL function. However, with the new graph nodes we don't have a graph root that could be passed to the f method. It is potentially possible to have the n-argument application nodes implement the prevArg() and getArgValue() methods, coupled with some internal state, in such a way as to allow decomposition as if it were a graph of general RTApplication nodes. In practice it is much more efficient to change the generated code for a CAL function to include a method that can be passed the CAL function arguments as individual method arguments. The original f method can now unpack the argument values and simply delegate to this new method.

Returning to our example of the compose function the generated f methods would look like the following.

```
/**
 * f
 * This method implements the function logic of the CAL function
 * Cal.Core.Prelude.compose
 */
public final RTValue f(RTValue $lastArg, RTExecutionContext $ec) {
   // Arguments
   RTValue x = $lastArg.getArgValue();
   RTValue g = ($lastArg = $lastArg.prevArg()).getArgValue();
   RTValue f = $lastArg.prevArg().getArgValue();
   return f3L(f, g, x, $ec);
}

/**
  * f3L
  * This method implements the function logic of the CAL function
  * Cal.Core.Prelude.compose
  */
public final RTValue f3L(RTValue f, RTValue g, RTValue x,
                    RTExecutionContext $ec) {
   Return f.apply (g.apply (x));
}
```

As can be seen the f method now unpacks arguments and delegates to the f3L method. The 3 in the new method name denotes the number of CAL arguments it takes, the actual number of arguments is four since the execution context is also passed. The L indicates that this is the method called when resolving a lazy application of the CAL function.

4.2 Function application in a strict context

When dealing with an application in a strict context the code generator can make avoid building graph if the left hand side of the application is a known CAL function.

Take, for example the following application.

```
foo x y z
```

When encountered in a strict context the generated Java code can take two forms. If foo is a local variable or an argument to the function containing the application the generated Java would be:

```
foo.apply(x).apply(y).apply(z).evaluate()
```

If foo is a declared CAL function the code will be slightly different as the singleton instance of the function class for foo will be used in the graph.

```
Foo.$instance.apply(x).apply(y).apply(z).evaluate()
```

The previously described optimization will generate more efficient code by using an n-argument application node.

```
new RTFullApp.General._3(foo, x, y, z).evaluate()
```

However this still involves the allocation of an application node that will immediately be reduced an become an indirection to the result.

More efficient Java code would be:

```
        Foo.$instance.f3L(x, y, z).evaluate()
```

The allocation of the application node is replaced by a direct call to the n-argument 'f' method of the function class. Calling the evaluate method on the result ensures that the result will continue to be reduced if it is not yet in weak head normal form.

4.3 Foreign functions in a strict context

When an application of all the necessary arguments to a foreign function is encountered in a strict context an optimization specific to foreign functions can be applied.

For example:

```
foreign unsafe import jvm "static method MyClass.myMethod"
    public myForeignFunction :: Int -> Int -> Int;
```

The body of the f method in the generated Java class for myForeignFunction would be:

```
return RTData.CAL_Int.make(
   MyClass.myMethod(
    arg1.evaluate().getIntValue(),
    arg2.evaluate().getIntValue()));
```

An application of myForeignFunction in a strict context would produce the following:

```
MyForeignFunction.$instance.f2L(arg1, arg2).evaluate()
```

However, we know that the 'f2L' method of the MyForeignFunction class is going to simply call MyClass.myMethod. This allows us to in-line to the call to the external method, generating the Java expression. Also we know that the result of a foreign function will always be an irreducible value, so there is no need to call the evaluate method on the result. This means that we can simply generate:

```
RTData.CAL_Int.make(
MyClass.myMethod(
arg1.evaluate().getIntValue(),
arg2.evaluate().getIntValue()))
```

4.4 Primitive operations in a strict context

As with foreign functions an application of a primitive function which corresponds to a primitive operation can be optimized when encountered in a strict context. Thus an application of addInt a b in a strict context can generate:

```
RTData.CAL_Int.make (a.evaluate().getIntValue() + b.evaluate().getIntValue()))
```

Rather than:

```
AddInt.$instance.f2L(a, b).evaluate()
```

4.5 General applications in a strict context

As previously discussed a function application in a strict context can be optimized to eliminate building the application graph and simplify the graph reduction. It would be nice to extend this optimization to situations when the left hand side of the application is not a known function. For example:

```
foo  f x y =
    case (f x y) of …
```

In this function the application 'f x y' occurs in a strict context. However, we don't know what 'f' is and thus must resort to building the application graph and then reducing. It is possible that at runtime the argument 'f' will be an instance of a CAL function and the optimization could apply. There is a way to take advantage of this optimization at runtime when f is an instance of a CAL function which takes two arguments. The solutions is to implement base versions of the multi-argument 'f' methods in the RTValue class. These base implementations simply build an application graph of 'this' applied to the n arguments. This means that for applications in a strict context we can optimistically generate code that assumes the left hand side is an instance of a function class.

```
f.f2L(x, y).evaluate()
```

If 'f' is an instance of a CAL function class taking two arguments it will have overridden the f2L method and will execute the function logic. If 'f' is not an instance of a CAL function class the base implementation of the f2L method will simply build the appropriate application graph which will then be reduced.

4.6 Optimization of partial applications

The previous optimization for general applications in a strict context optimistically treats the left hand side of an application as an instance of a CAL function class, relying an polymorphism in the Java runtime classes to produce the correct behavior for the actual type of the left hand side. This means that if the left hand side is indeed an instance of a function class that reduction will proceed in an efficient fashion.

Another case that can be optimized is when the left hand side of the application at runtime is a partial application of a function. i.e. an application of a function instance to fewer arguments than it requires. To this end a special set of application node classes exist to represent partial applications. These nodes have fields to hold the original function and arguments.

For example the class RTSupercombinator.RTPartialApp._3._2 is used to represent an application of a three argument function to two arguments. Partial application nodes for all partial applications of functions of up to fifteen arguments are part of the core runtime classes.

The partial application node classes mimic the behavior of a function instance of n arguments where n is the number required by the original CAL function. So the node class for applications of three argument functions to two functions implements the following methods, allowing it to appear to be an instance of a one argument function. As can be seen the getArity() and getArgCount() methods indicate that the node is a one argument function. To this end it implements the 'f' and 'f1L' methods which simply combine the single argument with the two arguments being held in the node class and call the held functions 'f3L' method.

```
public final int getArity () {
   return 1;
}
```

```
public final int getArgCount () {
   return 0;
}
public final RTValue f(RTValue lastArg, RTExecutionContext ec) {
   RTValue arg3 = lastArg.getArgValue();
   return function.f3L(arg1, arg2, arg3, ec);
}

public RTValue f1L(RTValue arg3, RTExecutionContext ec) {
   return function.f3L(arg1, arg2, arg3, ec);
}
```

This means that the LECC code generator can handle a partial application of a known function by creating one of these multi-argument partial application nodes. Then when additional arguments are applied the previous optimization, which optimistically directly calls the appropriate 'fNL' method will come into play.

4.7 Eagerly Evaluated Functions

The CAL compiler maintains a set of known functions which can be eagerly optimized. This means that an application of such a function in a lazy context will be treated as if it was in a strict context when it can be determined at compile time that all the arguments to the function are in weak head normal form or can in turn be eagerly evaluated.

This optimization is for functions where, when the arguments are already evaluated, it is cheaper to simply evaluate the function than it is to build a graph of the application. Functions optimized in this fashion must be safe. That is, there are no side effects, no possibility of a thrown exception, etc. This is because the optimization can cause a change in reduction order, which is only safe if the function has no side effects and can never fail (i.e. throw an exception).

Examples of functions optimized in this manner are given in the appendices.

5 Strictness

5.1 Strict Function arguments

CAL provides the ability to flag certain parameters as strict by prepending them with an exclamation point (or "pling"):

```
myHead :: [a] -> a;
public myHead !list =
    case list of
    firstElement : _  -> firstElement;
    []     -> error "empty list.";
    ;
```

In the definition of `myHead` above, for example, the `list` parameter has been flagged as strict. This indicates that the value passed into head as `list` will be evaluated until it is in "Weak Head Normal Form" before the function itself is evaluated.

Plinged arguments are evaluated in left-to-right order before the body of the function is returned.

Obviously marking some arguments as strict will have an effect on the generated Java code for the function. The simplest way to implement this functionality would be to simply add statements to the beginning of the 'f' methods in the generated classes which force the evaluation of the strict arguments. However, from the standpoint of other optimizations this is not ideal. In actuality the handling of strict arguments is achieved by introducing a new method into the generated Java class. This new method implements the logic of the CAL function with the assumption that all strict arguments are already in weak head normal form. Both the 'f' and 'fNL' methods can force the evaluation of strict arguments and then delegate to this new method.

So now the generated Java class for a CAL function has three entry points for executing the logic of the CAL function. The entry point used when reducing a general application graph is the 'f' method, which takes the root of an application graph and extracts the function argument values. The entry point used when all function arguments are available, but strict arguments are not known to be in weak head normal form, is the 'fNL' method where the N will be the number of CAL function arguments. The entry point used when all function arguments are available and strict arguments are in weak head normal form is the 'fNS' method where N will be the number of CAL function arguments.

For example the CAL function 'compose' where the first argument is strict:

```
compose :: (b -> c) -> (a -> b) -> (a -> c);
public compose !f g x = f (g x);
```

The three entry points in the generated class would be:

```
public final RTValue f (RTValue $lastArg, RTExecutionContext $ec) {
   // Arguments
   RTValue x = $lastArg.getArgValue();
```

```
    RTValue g = ($lastArg =  $lastArg.prevArg()).getArgValue();
    RTValue f = $lastArg.prevArg().getArgValue();

    return f3S (f.evaluate(), g x, $ec);
}

public final RTValue f3L (RTValue f, RTValue g, RTValue x, RTExecutionContext $ec) {
    return f3S (f.evaluate(), g, x, $ec);
}

public final RTValue f3S (RTValue f, RTValue g, RTValue x, RTExecutionContext $ec) {
    return f.apply(g.apply(x));
}
```

The new entry point (fNS) allows the code generator to keep generating efficient code for function applications in a strict context. The code generator can generate strict code at the call site for any function arguments marked as strict and directly call the 'fNS' method of the generated class.

5.2 N Argument Strict Application Nodes

With the ability to mark function arguments as strict an opportunity arises for an optimization similar to the N argument application nodes described previously. If the code generator encounters an application of a known function to the necessary number of arguments and can determine that any strict arguments are already in weak head normal form an alternate version of the N argument application node is used.

The set of multi-argument application node classes built into the runtime is expanded to differentiate between the lazy and strict application nodes for each number of arguments. The arguments held by the strict version of the application node are assumed to be already evaluated, if the function being applied marks them as strict. The lazy version of the application node doesn't assume this.

The class hierarchy for the multi-argument application nodes is:

```
RTResultFunction.RTFullApp.General._1._L
RTResultFunction.RTFullApp.General._1._S
…
```

The _L and _S class differ only in there implementation of the reduce method. In the _L node the reduce method will call the 'fNL' method of the function class, while the _S class will call the 'fNS' method.

5.3 Strict Data Constructor Fields

As with the arguments to a function the fields of a data constructor can be marked as strict by using an exclamation point (or "pling"). This means that at the point the data type instance is created the strict fields will be evaluated, in the order of declaration.

```
data MyDataType =
      MyDC1 x :: !Int  y :: !Boolean
      |
      MyDC2 s1 :: String  s2 :: !String
      ;
```

Code generation will generate Java code which evaluates the data constructor fields before creation the data constructor instance. For example, `MyDC1 a b`, encountered in a strict context would generate:

```
new TYPE_MyData.CAL_MyDC1 (a.evaluate(), b.evaluate());
```

It should be noted that the evaluation of the strict arguments does not occur until the data constructor instance is created. Thus a partial application of a data constructor would not result in any field evaluation. For example, `MyDC1 a`, would result in:

```
TYPE_MyData.CAL_MyDC1.$instance.apply(a)
```

Where a is not evaluated to weak head normal form.

The 'f' method of the data constructor class is adjusted to force the evaluation of any strict fields.

```
public final RTValue f (RTValue $lastArg, RTExecutionContext $ec) {
        // Arguments
        final RTValue y = $lastArg.getArgValue();
        final RTValue x = ($lastArg = $lastArg.prevArg()).getArgValue();

        return new CAL_MyDC1 (x.evaluate($ec), y.evaluate($ec));
}
```

The usage of strict fields also changes the code generated at the site of a data constructor application. Previously the application of a data constructor to the requisite number of arguments resulted in code which simply created a new instance of the data constructor class, regardless of whether the application occurred in a strict or lazy context. This still holds true for data constructors with no strict fields. If a data constructor with strict fields is applied in a strict context the generated code has to do the evaluation of the strict field arguments. For example:

```
new TYPE_MyData.CAL_MyDC1(x.evaluate($ec), y.evaluate($ec))
```

In a lazy context we can't evaluate the strict field arguments, they shouldn't be evaluated until the application is reduced to a data constructor instance. In this case the generated code builds an application node:

```
new RTFullApp.General._2._L (x, y)
```

5.4 Eager

The primitive function `Prelude.eager` can also be used to control strictness in CAL.

```
Primitive public eager :: a -> a;
```

'eager' is used to force a strict evaluation of its argument to weak-head normal form.
This forced evaluation occurs even when the application of eager is in a lazy context. (In a strict context the expression will be evaluated to weak-head normal form anyways and so calling 'eager' is redundant and will generate identical underlying code).

eager should be used with care since forcing the evaluation changes the normal reduction ordering and can affect laziness. eager should only be used in cases where the argument expression is known to have no side effects (including throwing an exception).

With proper care eager can provide a significant performance boost since forcing the strict evaluation
of the argument means we avoid building a lazy graph for it.

5.5 Let Variables and Strictness

As has been previously mentioned in the section on code generation for let variables the right-hand-side of a let binding is lifted into a separate function to avoid issues with size limitations in the Java bytecode format.

Thus the function:

```
f1 !a b =
    let
        x = foo a a b;
    in
        if (a > b) then x + a else x + b;
```

Would, in essence, be transformed to two functions:

```
xdef !a b = foo a a b;
```

```
f1 !a b =
    let
        x = xdef a b;
    in
        if (a > b) then x + a else x + b;
```

When lifting the right-hand-side of the original let binding into a separate function we preserve strictness of any variables now being passed to the lifted function as arguments. This is based on the strictness of any arguments to the original containing function, strictness of other let variables, strictness of data constructor fields, etc.

We can treat a let variable as strict if the right-hand-side of the let binding is an application of the primitive function 'eager' or if the expression is one for which laziness can be ignored. We can ignore laziness in the case of literal values, expressions which are known to already be in weak head normal form, and applications of functions tagged for eager evaluation where the arguments can have laziness ignored.

The fact that let variable definitions are lifted causes some difficult in generating efficient code, since the right-hand-side of the binding will always be a simple application of the let variable definition function. What is actually desired is to compile the original right-hand-side in a strict fashion. To simplify matters the code generator will generate two versions of the lifted let variable definition function. In one the body will be compiled lazily and in the other strictly. If laziness can be ignored for a let variable definition the strict version of the definition function will always be called. Otherwise use of the strict or lazy version will depend on the context of the application of the let variable definition function. Recall that these applications can be in-lined in cases where a variable is referenced only once an can therefore be in either a strict or lazy context.

The previous CAL code would then generate the following Java methods for the let variable x:

```
private RTValue x$2$def_strict (RTValue a, RTValue b, RTExecutionContext $ec) {
   return Foo.$instance.f2S (a, b, $ec).evaluate($ec);
}
private RTValue x$2$def_lazy (RTValue a, RTValue b, RTExecutionContext $ec) {
   return RTFullApp.General._2._L(Foo.$instance, a, b);
}
```

6 Unboxed values

In the LECC values for foreign types in weak head normal form are represented by built-in classes derived from RTData. Instances of these classes hold a value of the foreign type an are used to allow the foreign type to participate in graph building and reduction. While this boxing of values is necessary for usage in graphs it can be quite expensive in terms of performance.

Take, for example, the following CAL functions.

```
add1 :: Int -> Int;
add1 !x = x + 1;

add2 :: Int -> Int;
add2 !x = x + 2;

foo = (add1 1) + (add2 2);
```

Which would generate the following Java methods:
add1:

```
RTValue f2S (RTValue x, RTExecutionContext $ec) {
   return RTData.CAL_Int.make(x. getIntValue() + 1);
}
```

add2:

```
RTValue f1S (RTValue x, RTExecutionContext $ec) {
   return RTData.CAL_Int.make(x. getIntValue() + 2);
}
```

Foo:

```
RTValue f (RTValue a, RTValue b, RTExecutionContext $ec) {
   return RTData.CAL_Int.make(
      Add1.$instance.f1S(RTData.CAL_Int.make(1), $ec).evaluate($ec).getIntValue()
      +
      Add2.$instance.f1S(RTData.CAL_Int.make(2), $ec).evaluate($ec).getIntValue()
      );
}
```

Looking at the generated Java methods for each of the functions it can be seen that the generated logic for foo will make two instances of CAL_Int to pass the values 1 and 2 to add1 and add2 respectively. These CAL_Int instances are then immediately discarded as add1 and add2 immediately extract the boxed value.

6.1 Function arguments

Strict function arguments of foreign type are given special handling in the LECC code generator. In this case the generated 'fNS' method declares these arguments as being of unboxed type, rather than RTValue. The 'f' and 'fNL' methods will now evaluate and unbox these arguments and pass the unboxed value to the 'fNS' method. This allows the body of the 'fNS' method to be generated assuming that any strict functions that have an unboxed form will be in unboxed form.

So the function add1 described above would generate the following three methods.

```
public final RTValue f(RTValue $lastArg, RTExecutionContext $ec) {
   // Arguments
   RTValue x$L = $lastArg.getArgValue();
   return f1S(x$L.evaluate($ec).getOrdinalValue(), $ec);
}

public final RTValue f1L(RTValue x$L, RTExecutionContext $ec) {
   return f1S(x$L.evaluate($ec).getOrdinalValue(), $ec);
}

public final RTValue f1S(int x, RTExecutionContext $ec) {
   // Top level supercombinator logic
   return RTData.CAL_Int.make(x + 1);
}
```

With this change to the 'fNS' entry point for the CAL function logic a corresponding change is necessary for code generated at the call site. A function application in a strict context now needs to pass an unboxed value to the 'fNS' method. If the argument value is not already in unboxed form it will need to be evaluated and unboxed. Of course, if the argument value is already in unboxed form it can be passed directly. So an application of `'add1 a'` in a strict context would generate:

```
Add1.$instance.f1S(a.evaluate($ec).getOrdinalValue(), $ec).evaluate($ec)
```

While an application of `'add1 5'` would generate:

```
'Add1.$instance.f1S(5, $ec).evaluate($ec)
```

6.2 Multi-Argument Application Nodes with Unboxed Values

In cases of function application in a lazy node we need to build a graph representing the application. Ideally if the strict/unboxable arguments are already in weak head normal form we could hold then in their unboxed state, rather than box them only to unbox them when the graph is reduced. This is accomplished by generating special purpose multi-argument application node classes. For CAL functions with strict unboxable arguments the generated Java class has a static inner class called RTAppS which is a function specific application node which holds any strict and unboxable arguments in their unboxed form.

6.3 Function Return Values

Passing function arguments as unboxed values naturally leads to a desire to have an unboxed value returned from a function as well. To this end the LECC code generator generates a fourth entry point for the CAL function logic in the generated Java class. The new method is named 'fUnboxedNS' where N is the number of arguments to the CAL function. This methods return type will be the unboxed type of the CAL function. For add1 the method would be:

```
public final int fUnboxed1S(int x, RTExecutionContext $ec) {
   // Top level supercombinator logic
   return x + 1;
}
```

As can be seen, in this case an unboxed value can be returned by simply omitting a boxing step. In cases where the return value would normally be a suspension the 'fUnboxed' method will force evaluation and then unbox and return the result. This allows the code generator to always generate a call to the 'fUnboxedNS' method when an unboxed value is required.

Combining unboxed function arguments with unboxed return values and strict function arguments can result in highly efficient generated Java code. For example the function 'foo' defined at the beginning of the section on unboxed values can generate the following an 'f' method.

```
RTValue f (RTValue a, RTValue b, RTExecutionContext $ec) {
   return RTData.CAL_Int.make(
      Add1.$instance.fUnboxed1S(1, $ec)
      +
      Add2.$instance.fUnboxed1S(2, $ec)
      );
}
```

6.4 Data Constructor Fields

As with function arguments there are a variety of optimizations around strict data constructor fields having a type which can be unboxed.

Since strict fields will always be evaluated to weak head normal form before a data constructor instance is created the generated Java class can hold these fields in their unboxed form. This leads to changes in the generated access functions for the field.

```
data MyData =
    Data1 x :: !Int;
```

The generated class for Data1 will have a field of type int to hold the unboxed value of x. Two different access functions for x will be generated. One will return an unboxed value and the other will return a boxed value.

```
public RTValue get_x() {
   return RTData.CAL_Int.make(x);
}
public int get_x_As_Int() {
   return x;
}
```

Similarly the 'getFieldByIndex()' method will box x before returning and a 'getFieldByIndex_As_Int()' method will also be generated. Because the code which uses the getFieldByIndex method calls the method on instances of RTCons the RTCons base class now needs to implement a base version of getFieldByIndex_As_... for each unboxed type (i.e. int, long, boolean, String, Object, etc.). As with the base version of getFieldByIndex these functions simply generate an error if called.

Taking the ability to retrieve a data constructor field in an unboxed form a step further it makes sense to generate unboxed access methods even if the field is not strict. In the case of non-strict fields the access function forces evaluation and returns the unboxed value. This allows unboxed field values to be retrieved through a single mechanism, rather than generating different code based on data constructor field strictness.

The ability to consistently retrieve unboxed field values when unpacking data constructors in a case expression or a data constructor field selection expression allows more efficient Java code to be generated. Take the following function:

```
foo :: MyData -> Int;
foo a =
     case a of
     Data1 x -> x + x + 1;
     ;
```

Because the field v is only used in an unboxed fashion and is guaranteed to be evaluated and unboxed we can simply use the unboxed access method and declare a local variable of type int, rather than RTValue.

```
RTCons $case = (RTCons)a.evaluate();
switch ($case.getOrdinalValue()) {
   case 0: {
      private int x = ((Data1)$case).get_x_As_Int();
      return RTData.CAL_Int.make(x + x + 1);
   }
}
```

6.5 Unboxed Let Variables

As with function arguments and data constructor fields it can be desirable to work with let variables in their unboxed form. Consider the following example:

```
foo :: Int -> Int -> Int -> Int;
foo !a !b !c =
    let
        x = a + b + c;
    in
        x + x;
```

Analysis of the function shows that the let variable x will always be evaluated and will always be used in an unboxed fashion. As described previously the generated Java code lifts the let variable definition into two methods, one for strict evaluation and one for lazy. A third version of the lifted method is added for producing unboxed values. The example above would produce the following three methods for the definition of x:

```
private RTValue xDef_Lazy (int a, int b, int c) {
   return RTData.CAL_Int.make(a + b + c);
}
private RTValue xDef_Strict (int a, int b, int c) {
   return RTData.CAL_Int.make(a + b + c);
}
private RTVaue xDef_Unboxed (int a, int b, int c) {
   return a + b + c;
}
```

Note that the arguments to the x definition functions are passed as unboxed int values. This is because the in the original function foo the arguments a, b, and c were marked as strict. Also note that the xDef_Lazy and xDef_Strict methods are identical because laziness can be ignored for the + operator if the arguments are already in weak head normal form.

Prior to the introduction of the unboxed version of the x definition method the generated body of foo would have looked like:

```
private RTValue x = xDef_Strict (a, b, c);
return RTData.CAL_Int.make (x..getIntValue() + x.getIntValue());
```

With the unboxed version of the definition method the body would be:

```
private int  x = xDef_Unboxed (a, b, c);
return RTData.CAL_Int.make (x + x);
```

This version saves boxing the x value once and unboxing it twice. As with the unboxed entry points for functions there will always be an unboxed version of the let variable definition method if the variable type can be unboxed.

7 Memory Usage

The garbage collection of the Java virtual machine handles recovering unused graph nodes, eliminating the need for any manual garbage collection in the LECC runtime. However, the memory management in Java also provides some challenges.

7.1 Tail Recursive Functions

As previously discussed tail recursive functions are optimized by conversion into a loop in the generated Java code. However this does produce a potential memory leak which must be dealt with.

Normally the root node of the graph being reduced would have its result pointer updated after each stepwise reduction. At this point all other fields in the node are set to null as the values are no longer needed. Doing this allows the Java garbage collector to recover any graph nodes which are no longer in use. However, the body of the 'f' method in a tail recursive function consists of a loop. Thus no overwrite of the root node occurs until the end of the final tail call. Consequently the root node still holds pointers to the original arguments, preventing them from being garbage collected.

The solution is to null out the fields in the root node before entering the loop in the function body. Of course to do this the 'f' method in a generated function class must have access to the root node. Rather than add a new parameter to the 'f' method type signature the root node is passed via the execution context. In the 'reduce' method of application node classes a reference to the node is set into the execution context immediately before the 'f' method of the function class is called. The 'f' method can then retrieve this root node and use the 'clearMembers' method to force the node to null out its fields.

7.2 Dynamic removal of indirection chains

In Java memory cannot be arbitrarily overwritten with new values. Thus an application node cannot be changed into a value node by overwriting. This necessitates having a reduced graph node redirect to its result.

This results in the fields being held by record and data type instances often being a graph node which simply redirects to the result value. These nodes use up memory without providing any actual functionality or value. To deal with this situation the generated access functions for retrieving field values remove these indirections.

For example:

```
public RTValue get_value() {
        RTValue _value$;

        if ((_value$ = _value) instanceof RTResultFunction) {
                return _value = _value$.getValue();
        }
        return _value$;
}
```

This is the access function for the 'value' field in a CAL data type instance. If 'value' is an instance of RTResultFunction the field is updated to the result of calling 'getValue'. If the RTResultFunction instance has already been reduced getValue will return the result. If it hasn't been reduced getValue will simply return the node itself.

7.3 Consolidation of shared fields in type/data constructors.

Often the data constructors for a data type will have common fields. For example:

```
data MyData =
    MyData1  name :: String  intValue :: Int
    |
    MyData2  name :: String  boolValue :: Boolean
    ;
```

Both data constructors have a field called 'name' of type String.
The generated Java for a data type consists of an outer class corresponding to the type constructor. Inner classes, which extend the type constructor class, are then generated for each data constructor. Generally the data constructor classes contain a field for each field in the data constructor. In the case of all the data constructors sharing a common field the field and access methods will be generated as members of the containing data type class. This avoids code duplication between the data constructor classes and reduces the overall size of the generated classes.

8 Strongly connected functions and function groups.

Strongly connected (i.e. mutually referential) functions produce different Java code than regular functions. Instead of generating a separate class for each function a single class that represents the entire function group is generated. This is done to avoid problems with initialization of static members in mutually dependent Java classes.

```
function1 :: Double -> Double -> Double;
function1 x y = if x > 0.0 then x else function2 y;

function2 :: Double -> Double;
function2 x = if x > 0.0 then x else function1 x 0.0;
```

The generated class will be named after the first encountered function in the function group.

The generated class will contain a 'tag' field of type int. The tag value is used to determine which CAL function an instance of the generated class represents. It will also contain an 'arity' field of type int, which indicates the arity of CAL function the instance represents.

There will be a static final field for each represented function which holds an instance of the class. These are named $instance_functionName, where functionName is the name of the CAL function.

```
	public static final Function1 $instance_function1 = new Function1(0, 2);
	public static final Function1 $instance_Function2 = new Function1(1, 1);
```

As can be seen each instance field is initialized with an instance of the function class, created with tag and arity values.

The generated 'f' methods for each CAL function are named by prefixing with the associated function names. Regular 'f' methods are generated which switch on the tag field value to dispatch to the appropriate function specific method. For the arity specific methods (i.e. f1S, f1L, f2S, etc.) the switch only contains cases for the tag values associated with functions with that arity. The strict versions (ex. f2S) have a fall through return which throws an error indicated an inconsistent internal state as the generated code should never call the f2S method on an instance representing a 1 arity function. The lazy versions (ex. f2L) behave slightly differently. For these the fall through return is to call the super class version of the same method. This is necessary because of the optimization described in the section 'General applications in a strict context'. This optimization can generate code where a lazy method may be directly called on an instance of the generated class which represents a function of a different arity.

```
	public final RTValue f(RTValue $lastArg, RTExecutionContext $ec) throws CALExecutorException {
		switch (scTag) {
			case 0: {
				return function1_f($lastArg, $ec);
			}
			case 1: {
				return function2_f($lastArg, $ec);
			}
		}
		return RTValue.badValue("Bad scTag in \'f\'.");
	}

	public final RTValue f1S(RTValue $arg0, RTValue $arg1, RTExecutionContext $ec) throws
CALExecutorException {
		switch (scTag) {
			case 1: {
				return function2_f2S($arg0, $arg1, $ec);
			}
		}
```

```
                return RTValue.badValue("Bad scTag in \'f\'.");
        }

        public final RTValue f1L(RTValue $arg0, RTValue $arg1, RTExecutionContext $ec) throws
CALExecutorException {
                switch (scTag) {
                        case 1: {
                                return function2_f2L($arg0, $arg1, $ec);
                        }
                }
                // This is an oversaturated lazy application.
                // Usually this occurs when dealing with a lazy application of a function type argument.
                // Defer to the base implementation in the super class.
                return super.f2L($arg0, $arg1, $ec);
        }
        public final RTValue f2S(RTValue $arg0, RTValue $arg1, RTExecutionContext $ec) throws
CALExecutorException {
                switch (scTag) {
                        case 0: {
                                return function1_f2S($arg0, $arg1, $ec);
                        }
                }
                return RTValue.badValue("Bad scTag in \'f\'.");
        }

        public final RTValue f2L(RTValue $arg0, RTValue $arg1, RTExecutionContext $ec) throws
CALExecutorException {
                switch (scTag) {
                        case 0: {
                                return function1_f2L($arg0, $arg1, $ec);
                        }
                }
                // This is an oversaturated lazy application.
                // Usually this occurs when dealing with a lazy application of a function type argument.
                // Defer to the base implementation in the super class.
                return super.f2L($arg0, $arg1, $ec);
        }
```

Generated application node classes for functions which are grouped into a single class are named differently to indicate which function they are associated with. For example, instead of RTAppS the generated node class would be called RTAppS_Function1.

The default behavior for the code generator is to group strongly connected functions into a single generated class. However, this is the minimal level of grouping which occurs. Additional, more arbitrary, conditions for grouping can be applied, all the way up to generating entire modules into a single class. Potentially this ability could be used to group functions so as to minimize class loading time.

9 Debugging

The CAL runtime has built-in facilities for debugging. Client applications such as ICE (Interactive CAL Environment) can use these facilities to provide debugging capabilities.

9.1 Enabling Debugging

In order to use the debugging facilities the CAL modules must be compiled with debugging turned on. Setting the system property org.openquark.cal.machine.debug_capable will cause the generated Java classes to contain extra debugging functionality. Basically this consists of inserting a block of code into the beginning of the 'f' methods that implement the function logic. This code block uses the execution context to determine if any debug processing is needed for the current function. Debug processing includes things such as tracing out the state at the function entry or halting on a breakpoint.

The following code fragment shows the code block at the beginning of the 'f2S' method. As can be seen the function name is passed to the execution context method 'isDebugProcessingNeeded'. If the execution context returns true then the execution context method 'debugProcessing' is called. 'debugProcessing' is passed the name of the CAL function as well as the values of the arguments being passed to this invocation of the function.

```
public final RTValue f2S(RTValue dataSet, RTValue elemName, RTExecutionContext $ec) {
   if ($ec.isDebugProcessingNeeded("BusinessObjects.Boson.DataSet.buildXmlForDataSetWithName")) {
      $ec.debugProcessing(
         "BusinessObjects.Boson.DataSet.buildXmlForDataSetWithName",
         new RTValue[] {dataSet, elemName});
   }
```

Once the generated Java includes the additional debugging code described above the debugging facilities can be accessed through the execution context. The execution context (class RTExecutionContext) has methods for handling tracing of the program state as well as for setting breakpoints, halting, and stepping.

9.2 Function Tracing

Function tracing causes the runtime to send a message to standard error whenever a function is evaluated. The message consists of the function name and other optional values. Function tracing can be turned on for all functions or for specific functions. If tracing is turned on for all functions a set of regular expressions can be supplied to filter the output. Additionally options can be set to include the values of the function arguments and the name of the current thread in the trace message.

A client application accesses the tracing functionality via the execution context. The execution context has methods that allow setting and examining the various options around tracing. The following functions in the execution context (class ExecutionContextImpl) are used to manipulate function tracing:

```
/**
 * @return a Set of String containing the qualified names of functions for which tracing has been specifically enabled.
 */
public final Set getTracedFunctions ();

/**
 * Add a new function to the set of functions for which tracing is specifically enabled.
 * Note: this will supersede the general tracing setting.
 * @param tracedFunction – the qualified name of the function for which tracing should be enabled.
 */
public final void addTracedFunction (String tracedFunction);

/**
 * Remove a function from the set of functions for which tracing is specifically enabled.
 * @param tracedFunction – the qualified name of the function to remove from the set of traced functions.
 */
public final void removeTracedFunction (String tracedFunction);

/**
 * Remove all functions from the set of functions for which tracing is specifically enabled.
 */
public void clearTracedFunctions ();

/**
 * @return true if function tracing information should be printed to the console.
 * If the current machine configuration is not capable of tracing this flag has no effect.
 */
public final boolean isTracingEnabled ();

/**
 * Set the value of the flag which controls tracing information for all functions to the console.
 * If the current machine configuration is not capable of tracing this flag has no effect.
 * @param tracingEnabled
 */
public final void setTracingEnabled (boolean tracingEnabled);

/**
 * @return true if function tracing should also show the name of the thread that is evaluating the given function.
 */
Public final Boolean traceShowsThreadName ();

/**
 * Set flag indicating whether function tracing should also show the name of the thread that is evaluating the given function.
 */
public final void setTraceShowsThreadName (boolean traceShowsThreadName);

/**
 * @return true if function tracing should also show the arguments of the function being traced. This has the effect of
 * calling DebugSupport.showInternal on the arguments and does not modify the program evaluation state.
 */
Public final Boolean traceShowsFunctionArgs ();

/**
* Set flag indicating whether function tracing should also show the arguments of the function being traced. This has the effect of
```

```
 * calling DebugSupport.showInternal on the arguments and does not modify the program evaluation state.
 */
public final void setTraceShowsFunctionArgs (boolean traceShowsFunctionArgs);

/**
 * Set the list of regular expressions to use for filtering the trace output.
 * @param patterns – The list of regular expressions to use for the filter.
 */
public void setTraceFilters (List patterns);

/**
 * @return an immutable version of the list of regular expressions used to filter trace output.
 */
public List getTraceFilters ();
```

9.3 Textual representation of CAL values

As described in the section on function tracing the values of function arguments can optionally be displayed. This requires a textual representation for a CAL value. The representation used is the same as that displayed by the CAL function Debug.showInternalGraph.

Debug.showInternalGraph displays the internal representation of a value, including information about shared nodes and indirection nodes. This representation is subject to change, and should not be relied upon in production code. It is intended to assist in debugging, and for the purposes of understanding the lazy evaluation behavior of CAL.

It is important to note that link showInternalGraph does no evaluation of the value argument, and so does not change the order of reduction in any way. This is unlike Debug.show, or most other means of displaying a value of a CAL computation. This can be a great advantage, however, the drawback is that showInternalGraph is displaying an internal form and thus can be less readable, and less customizable than using other means.

showInternalGraph attempts to show more of the graph structure of the value. If a node is shared, its first appearance will be marked e.g. <@nodeNumber = nodeText>, and subsequent appearances will just displayed as <@nodeNumber>. Indirections are shown using an asterix (*).

9.4 Breakpoints

When running in debug mode breakpoints can be set, by a client tool, on CAL functions.

The ExecutionContextImpl class caches the set of breakpoints, when debugProcessing() is called from a function for which a breakpoint has been set the execution context will build and cache an instance of ExecutionContextImpl.SuspensionState. This object contains information about the state of the program at the point of suspension. The contained information includes the name of the function, the names and types of the function arguments, the current values of the arguments, and the current stack trace. The argument values are held as instances of CALValue, and can be displayed by calling toString() on the argument value which will produce a textual representation of the CAL value as described in section 9.3.

Once a SuspensionState object has been cached the runtime thread will notify any waiting threads and then go into a wait state itself. A client can use this functionality by executing the CAL code on a separate thread from the main client thread. When execution is started the main thread can wait on the runtime thread. The main thread will then be notified when execution is complete or a breakpoint is reached. When the main thread is notified it can check, via the execution context, whether the execution is in a suspended state and can access and display the cached suspension state. Execution of the CAL code can be resumed by signaling the waiting runtime thread by calling ExecutionContextImpl.resum().

Methods in ExecutionContextImpl allow for adding, removing, and retrieving breakpoints, as well as querying whether execution is suspended.

9.5 Stepping

The ExecutionContextImpl class as a state flag for stepping, which can be accessed through the isStepping() and setStepping() methods. When the stepping flag is set to true the runtime will behave as if a breakpoint is set on every CAL function. In other words, execution will suspend and wait on the client when entering every function. See the section on breakpoints for a further description of the suspension behaviour.

10 Appendices

10.1 Functions Marked for Eager Evaluation

As mentioned in section 4.7 there are specific CAL functions that are given special treatment in the runtime. Essentially the runtime will ignore laziness and immediately evaluate these functions if, an only if, it can be determined at compile time that all the arguments to the function are in weak head normal form or can in turn be eagerly evaluated.

An example of a function that can be eagerly evaluated would be Prelude.addInt. If both arguments to this function are already evaluated it is cheaper to simply add them and create a new data instance containing the result than to create a suspension to be evaluated later. Since addInt cannot fail, i.e. it will always return a value for any two int arguments, there are no possible side effects to eager evaluation and it is safe. Most of the functions which correspond to basic operations on built-in types (ex. Int, double, long, byte, etc) are similarly optimized.

Some operations on the built in types cannot be optimized in this fashion. For example Prelude.divideLong. This function does not qualify for eager evaluation because it can throw an exception if division by zero takes place. This side effect could change flow of program evaluation, especially if the suspension is not always evaluated. The runtime does try to optimize such functions when it can bet determined at compile time that the second argument is guaranteed to be non-zero, but this is a narrowing of the general requirement for eager evaluation.

An example of an operation on a basic type which cannot be optimized is Prelude.maxInteger. This is a foreign function which acts on instances of the BigInteger object. If one of the instances is null an exception is thrown. A null Integer cannot be directly created in CAL source, but it can be marshaled from the result of a foreign function call. The throwing of an exception is a side effect which would change the control flow of the program so functions like this cannot be optimized.

Foreign functions which correspond to a static final Java field can also be optimized. For example; Prelude.notANumber is declared as the foreign field java.lang.Double.NaN. Again there are no side effects to eagerly evaluating this function so it can also be optimiized.

11 Revision History

August 3, 2007: Updated formatting to use outline numbering. Added section for debugging. Removed the use of color in diagrams.