Mystique:
An Imperative Reactive Language

David François Huynh

August 22, 2003

This document motivates and introduces the Mystique programming language to implementors and potential users of the language. This document is not a formal documentation of the language nor a tutorial to the language. You will notice that many aspects of Mystique's syntax are left to your imagination.

1 Motivation

Haystack is an information management platform that is built on top of a semistructured data model. At the bottom of the platform is a data store which hosts information in the RDF triple format. (In the Resource Description Framework (RDF), data is expressed as a set of RDF statements, each consisting of a triple of subject, predicate, and object.) The Haystack User Interface Framework is purposed to project the underlying information to the screen (or another output device) and to allow the user to interact with and modify the information beneath.

1.1 Adenine

The process of projecting information from the underlying RDF data store to the screen requires reading RDF data as well as massaging it into some forms suitable for displaying. Manipulation of RDF data was deemed to be best done in the Adenine programming language[1], as Adenine has native syntax for RDF resources and literals and for RDF queries, making it much less cumbersome to manipulate RDF data than, say, in Java. For instance, the following code written in Java as:

```
rdfStore.add(new Statement(
    new Resource("http://haystack.lcs.mit.edu/"),
    new Resource("http://www.w3.org/1999/02/22-rdf-syntax-ns#type"),
    new Resource("http://haystack.lcs.mit.edu/schemata/web#Webpage")
));
```

can be written more compactly in Adenine as:

```
rdfStore.add { <http://haystack.lcs.mit.edu/> rdf:type web:Webpage }
```

[1] Invented by Dennis Quan. See http://haystack.lcs.mit.edu/documentation/adenine.pdf.

(with the prefixes defined elsewhere first).

Like in most other imperative languages, each piece of Adenine code is executed until there is no code left to execute, and then a result is returned. That is, the execution process is inherently single threaded and linear. If Adenine were used to program UI elements, this execution characteristic would force UI programmers to handle the asynchronicity ever present in UI programming themselves. This work might be acceptable, as the same observation is already true for other languages like Java and C++, which have been used to program UIs but with the help of event handling architectures. However, Adenine, being primarily interpreted, is too heavy-weighed for encoding low-level event handlers. Even when Adenine code can be "precompiled" into Java byte code rather than interpreted, Adenine's lack of static typing prevents type related optimizations.

Nevertheless, it was still desirable to take advantage of Adenine's syntactic sugar for RDF data manipulation. A solution was quickly found by adapting the server HTML generation scheme: Adenine was used for extracting the RDF data to be presented and then for generating more RDF data that encodes the UI layouts for displaying the underlying data. That is, Adenine was used much like the various server scripting languages including ASP and PHP. The generated UI layout data is stored in the triple format in the RDF store rather than transfered to a browser as a stream of HTML. Regardless, the generated UI layout data is readily recognizable as a variant of HTML:

```
= ui ${
    rdf:type        slide:Paragraph ;
    slide:fontSize  "12" ;
    slide:color     "blue" ;
    slide:children (List
        ${  rdf:type     slide:Text ;
            slide:text   "Hello World!"
        }
        ${  rdf:type     slide:Image ;
            slide:source <http://haystack.lcs.mit.edu/icons/hello.gif>
        }
    )
}
```

While Adenine seemed appealing for encoding UI code, its selection for such purpose has been troublesome in retrospect:

Verbosity: A quick glance at the previous block of Adenine code tells us that the triple format (which makes it easy to manipulate RDF data in Adenine) is unsuitable for describing UI layouts in an HTML-like schema. If the same UI layout were written in HTML, it would be much shorter and more readable:

```
<p style="color: blue; font-size: 12">
    Hello World!
    <img src="http://haystack.lcs.mit.edu/icons/hello.gif">
</p>
```

Duplicated data: The generated UI layout data most certainly embeds data that has been extracted from the RDF store before. Such duplication of the data to be presented overburdens the RDF store unnecessarily. Furthermore, the layout structure of the generated UI layout data is also duplicated many times for presenting many things that make use of the same UI layout.

Stale data: When the data to be presented has been changed, the UI layout data still holds on to the previously extracted data. Stale data requires the UI to be explicitly refreshed by the user—this might be acceptable for web pages, but not for an interactive application like Haystack.

Mixing of data and UI layout: The data to be presented is extracted and then embedded directly into the UI layout data. This paradigm makes it hard to support internationalization. Generally, internationalization requires user data to be substituted into different UI "templates", each designed for a different locale.

(Because Adenine's methods execute and "die away", the only way to keep state for live UI elements is through the RDF store. This is the reason why even if Adenine was made more performant, its use for UI still suffers from the above problems. It is also the reason newcomers to Haystack are confused with the UI framework.)

1.2 Data Sources

The second and third problems have been addressed in part through the introduction of "data sources". A data source (in Haystack) is a description of a computation that yields data, possibly to be fed into a UI element. For example, we can declare a data source that extracts the name of a person (whose URI is stored in `somePerson`), which is then fed into a text UI element as follows:

```
= ui ${
    rdf:type slide:Text ;
    slide:textDataSource ${
        rdf:type       data:LiteralPropertySource ;
        data:subject   somePerson ;
        data:predicate person:name
    }
}
```

This piece UI data is equivalent to:

```
= ui ${
    rdf:type   slide:Text ;
    slide:text (extract somePerson person:name ?x)
}
```

except that the former is dynamic in that the displayed text changes when the name of the given person changes, while the latter remains static. The former is a declarative "recipe" read by Java code which keeps the computed data up-to-date, while the latter is imperative Adenine code, executed once when the UI layout data is generated.

Data sources can be nested to allow encoding of complex computations. In the example above, we can use another data source to compute which person whose name is to be presented:

```
= ui ${
    rdf:type slide:Text ;
```

```
        slide:textDataSource ${
            rdf:type           data:LiteralPropertySource ;
            data:subjectDataSource ${
                rdf:type       data:ResourcePropertySource ;
                data:subject   someEmailMessage ;
                data:predicate mail:sender
            } ;
            data:predicate     person:name
        }
    }
```

This is equivalent to:

```
    = ui ${
        rdf:type   slide:Text ;
        slide:text
            (extract
                (extract someEmailMessage mail:sender ?x) person:name ?y)
    }
```

except for its dynamicity.

You can notice that the syntax of data sources also suffers from verbosity due to the triple syntax of Adenine. In fact, data sources suffer from several drawbacks:

Verbosity: Which *severely* reduces the readability of the code.

Doubling of predicates for UI elements: Each property that a UI element can take must now have a parallel property for specifying a data source (e.g. `slide:text` → `slide:textDataSource`). Each UI element must now handle both static and dynamic properties, and UI programmers must learn both cases.

No type checking for data sources: Each data source describes the computation of a particular type of data such as an RDF resource, an RDF literal, a `java.lang.Boolean`, a `java.util.List`, etc. However, since data sources do not have native syntax in Adenine (for all it matters, a data source is just another blob of RDF to the Adenine compiler), it is not possible to type-check them, especially when they are nested together. In the preceding example of nested data sources, the Adenine compiler cannot tell whether the outer data source expects an RDF resource from the inner one, and whether the inner one yields an RDF resource. The burden of type-checking rests on the programmer at design time, and on the Java code that handles each type of data source at runtime.

No reuse for results of data sources: It is also because data sources have no native syntax in Adenine that results from data sources cannot be referenced for reuse. For example, in the preceding example of nested data sources, if we were to display the age of the same person as well, we must specify another data source to recompute the person:

```
        = ui1 ${
            rdf:type slide:Text ;
            slide:textDataSource ${
                rdf:type           data:LiteralPropertySource ;
```

```
        data:subjectDataSource ${
            rdf:type       data:ResourcePropertySource ;
            data:subject   someEmailMessage ;
            data:predicate mail:sender
        } ;
        data:predicate     person:name
    }
}
= ui2 ${
    rdf:type slide:Text ;
    slide:textDataSource ${
        rdf:type           data:LiteralPropertySource ;
        data:subjectDataSource ${
            rdf:type       data:ResourcePropertySource ;
            data:subject   someEmailMessage ;
            data:predicate mail:sender
        } ;
        data:predicate     person:age
    }
}
```

We could theoretically reuse the RDF data that makes up the data source:

```
= innerDataSource ${
    rdf:type       data:ResourcePropertySource ;
    data:subject   someEmailMessage ;
    data:predicate mail:sender
}
= ui1 ${
    rdf:type slide:Text ;
    slide:textDataSource ${
        rdf:type               data:LiteralPropertySource ;
        data:subjectDataSource innerDataSource ;
        data:predicate         person:name
    }
}
= ui2 ${
    rdf:type slide:Text ;
    slide:textDataSource ${
        rdf:type               data:LiteralPropertySource ;
        data:subjectDataSource innerDataSource ;
        data:predicate         person:age
    }
}
```

but this approach still leads to two different Java objects being instantiated to implement the data source in its two instances of use. The two Java objects will do the work twice to retrieve the data source's result twice. The two contexts in which the data source is used prohibit blindly pooling the data source's implementation Java object.

Two contexts: Because UI layout data can now be generated once and reused many times (without the need to be regenerated to stay in sync with underlying data), the programmer needs to

be mindful of two different contexts, one in which the UI layout data is generated and one in which it is used to produce screen renderings. These two contexts often confuse newcomers to the UI framework.

These various drawbacks of data sources make their use unnatural and awkward to UI programmers, novice and expert alike. They are also limited in their expressive power. It is clear that a better solution is needed.

2 Approach

We have seen that data sources solve the problems of duplicated and stale data, but they themselves bring along another host of problems mainly because they have no native syntax and support in Adenine. In order to solve these new problems, we invent another programming language called *Mystique* that has native support for data sources. In fact, the functionality previously provided by data sources (i.e. keeping dynamically computed data up-to-date) is so central to Mystique that the concept of data sources independent from this new language (as something that the programmer must explicitly describe) disappears entirely. Put it another way, "data source" will not be a word in the vocabulary of a Mystique programmer.

A body of Mystique code is like an electronic circuit. It has a set of inputs and a set of outputs, and the outputs always reflect the inputs with respect to the internal logic of the code body. That is, if the inputs change, the outputs are recomputed (after some "propagation" delay) to whichever values the outputs should be as if the new inputs are the values with which the code body is fed originally.

2.1 Reactive vs. Constraint-Based

With the above description of Mystique, one might characterize it as a constraint-based language and might therefore assume that every body of Mystique code is written as a set of constraints—conditions that are maintained without explicit imperative instructions. To dispel such assumption, because each body of Mystique code is in fact written as a sequence of imperative statements, I would rather characterize Mystique as a *reactive* language. This reflects better the above analogy with electronic circuits. Electronic circuits are more often thought to be reactive to their inputs, rather than being sets of constraints.

Mystique's imperative syntax is chosen because I believe that most UI programmers are more familiar with imperative languages such as Java and C++ than with constraint-based languages.

To think of a body of Mystique code as a set of constraints is not only inaccurate with respect to the way Mystique code is written, but also to the way Mystique code is executed. A set of constraints specifies no order in which the constraints are applied while the statements in a Mystique code body have a fixed order of execution. The statements are essentially re-executed in that order when the inputs change.

In addition, "constraint" suggests two-way symmetric dependency while in Mystique, only the outputs depend on the inputs, not vice-versa.

2.2 Reactive vs. Rule-Based

One might also be tempted to characterize Mystique as rule-based. That is, a body of Mystique code consists of a set of rules that are applied over and over again until there is no change. Such a characterization is closer to the nature of Mystique than the constraint-based characterization, because rules specify only one-way dependency and exhibit an imperative nature.

However, the rule-based characterization is still inaccurate in two ways. First, obviously Mystique code is not written as a set of rules, as much as it is not written as a set of constraints. The second point is more involved and is elaborated below.

In typical rule-based systems, a body of data is transformed by rules again and again until no more transformation can be brought about by any of the rules. Changes can be originally triggered by external sources (e.g. the user edits a spreadsheet cell). More changes can then be caused by the application of rules. The two types of changes are not distinguished because both types can lead to application of rules and hence, further changes.

In Mystique, however, the two types of changes are distinguished. External changes lead to re-execution of Mystique code, while internal changes (caused by execution of code) do not evoke re-execution. Consider a body of Mystique code that contains the following statement:

```
x = x + 1
```

If Mystique were rule-based, that body of Mystique code would be stuck forever reapplying the above assignment "rule". However, the change to `x` caused by the assignment is an internal change with respect to that body of code, brought about during the execution of that code body itself. Such an internal change does not cause re-execution.

To better understand the concept of internal changes, one could think that the reason a local variable is reused within a body of Mystique code is convenience. Consider the following body of code:

```
Integer x = ...
...
if ...some condition...
    x = x + 1

...code that depends on x...
```

We can certainly rewrite that body of code without the second assignment statement as follows:

```
Integer x2 = ...
...
if ...some condition...
    Integer x = x2 + 1

    ...code that depends on x...
else
    Integer x = x2

    ...code that depends on x...
```

but the former version of the code is more compact and no less comprehensible.

3 Functions

There have been a lot of references to "body of Mystique code". In this section, we formalize this concept by introducing the fundamental unit of Mystique code: the Mystique function. A Mystique function takes zero or more parameters and returns zero or more results. Parameters can be *positional* (matched with call arguments by positions) or *named* (matched with call arguments by names). Results can only be positional. The syntax for Mystique functions is as follows:

```
<function> ::=
    "function" <annotated-URI> "(" [ <parameter-list> ] ")"
    [ "returns"  "(" [ <result-list> ] ")" ]
    [ "accesses" "(" [ <access-list> ] ")" ]
        <body-of-code>

<parameter-list> ::=
      <positional-parameter>
    | <positional-parameter> "," <parameter-list>
    | <named-parameter-list>

<named-parameter-list> ::=
      <named-parameter>
    | <named-parameter> "," <named-parameter-list>

<result-list> ::=
      <result>
    | <result> "," <result-list>

<access-list> ::=
      <access>
    | <access> "," <access-list>

<positional-parameter> ::=
    <type> <identifier> [ "optional" ]

<named-parameter> ::=
    <type> <identifier> "as" <annotated-URI> [ "optional" ]

<result> ::=
    <type> <identifier>

<access> ::=
    <type> <identifier>
```

We won't define `<annotated-URI>` for now.

3.1 Positional Parameters and Results

Here is an example of a function with two positional parameters and two results:

```
function myUtilities:minMax (
    Integer i, Integer j
) returns (
    Integer min, Integer max
)
    if i < j
        min = i
        max = j
    else
        min = j
        max = i
```

Note that unlike in most other languages, each result of a Mystique function is named by a variable. This function calculates both the minimum and the maximum of a pair of integers. One can use this function in three ways, as shown here:

```
x, y = myUtilities:minMax(10, -2) // x is set to -2, y to 10
x,   = myUtilities:minMax(4, 6) // x is set to 4
 , y = myUtilities:minMax(2, 9) // y is set to 9
```

Optional positional parameters can be marked with the keyword `optional` and can occur in any order with respect to non-optional positional parameters. E.g.

```
function strings:printf(
    String       format,
    Vector       substitutions,
    Locale       locale optional,
    OutputStream os
)
    if locale == null
        locale = ...

    ...
```

One can call this function without providing the parameter `locale` as follows:

```
string:printf(aFormat, someSubstitutions, , anOutputStream)
```

Notice the two consecutive commas. If the optional parameter to be neglected is at the end of the parameter list, then no extra comma is needed. For example, the following function:

```
function strings:integerToString(
    Integer i,
    Integer base optional
```

```
) returns (String s)
    if base == null
        base = 10

    s = ...
```

can be called as follows:

```
strings:integerToString(102)
```

if the intended base is 10.

3.2 Named Parameters

A named parameter is marked with the keyword `as` followed by an annotated URI. E.g.

```
function files:copy(
    File        f1              as files:fromFile,
    File        f2              as files:toFile,
    Boolean     overrideIfExist,
    Certificate authentication,
    File        f3              as files:logFile optional
)
    ...
```

To use this function to copy a file, one must provide four parameters, two positional and two named. The parameter `overrideIfExist` must be provided before the parameter `authentication`. The parameters `f1` and `f2` can be provided in any order among themselves and among the two positional parameters. The parameter `f3` can be skipped altogether. Here is an example of how to call this function:

```
files:copy(
    destination as files:toFile,
    true,
    source as files:fromFile,
    aCertificate
)
```

3.3 Currying

Mystique provides native support for currying (as compared to Adenine): when a function is called with some non-optional parameters (named or positional) not provided, the result is a new function that takes the missing parameters (both optional or non optional). For example, the following call:

```
file:copy(
    destination as files:toFile,
    true
)
```

returns a new function that acts just like this function:

```
function file:copy2(
    File        f1              as files:fromFile,
    Certificate authentication,
    File        f3              as files:logFile optional
)
    file:copy(
        destination as files:toFile,
        true,
        authentication,
        f1 as files:fromFile,
        f3 as files:logFile
    )
```

Note that optional named parameters can always be omitted while currying functions, but optional positional parameters have to be provided or defaulted if they appear in front of non-optional positional parameters that are provided. For example, if the positional parameter `overrideIfExist` were optional, one cannot produce a curried function that still takes that parameter but already has the parameter `authentication` provided, because the following call is illegal:

```
file:copy(aCertificate)
```

4 Variables

There are four types of variable in Mystique:

- Parameter variables
- Result variables
- Local variables
- Dynamic variables

A parameter variable is passed by reference and has the scope of the function and the lifetime of the function's invocation. The parameter variable can be reassigned within the code body of the function.

A result variable also has the scope of the function and the lifetime of the function's invocation. It is automatically assigned `null` initially, and in order to return a result, the function's code must reassign some other value to the variable.

A local variable has the scope and lifetime of whichever control construct (e.g. `if`, `while`) that contains it.

A dynamic variable has the lifetime of whichever construct `let` that declares it. For example, the variable `locale` in the following code body:

```
...code segment A...

let Locale locale = ...
    ...code segment B...

...code segment C...
```

has the lifetime of the construct `let` shown. That variable has the scope of the code segment B, encompassing the code bodies of all functions that B ultimately invokes. In order to access a dynamic variable in its code body, a function must explicitly declare such an access through the use of the keyword `accesses` in its signature. E.g.

```
function :getCurrentTimeAsString(
) returns (
    String s
) accesses (
    Locale locale
)
    ...
```

5 Types

Mystique is *semi*-statically typed. The type of a Mystique expression is not known at compile time only if the expression contains a function application. There is no guarantee that the function's signature is known at the time its application is compiled. This is because Mystique functions are compiled independently from one another and they can be recompiled and rebound at runtime.

Rather than making Mystique dynamically typed, we want to keep it semi-statically typed so that we can employ type-related optimizations where applicable. For this reason, we require variables and functions to be declared with types.

Each Mystique type corresponds to a Java type. For example, in preceding code samples, the Mystique types `String` and `Boolean` correspond to the Java types `java.lang.String` and `java.lang.Boolean` respectively. These two types are imported in the default compilation environment (much like they would be in Java compilation). To import Java types, use the directive `@importJava`:

```
@importJava edu.mit.lcs.haystack.rdf.Utilities
@importJava edu.mit.lcs.haystack.ozone.*
```

These work just like their Java `import` counterparts.

There is no support for creating a new type in Mystique. Because Mystique can manipulate objects but not create classes, it is considered object-based rather than object-oriented.

6 Basic Constructs

Mystique has the usual basic constructs that you would expect from any serious language, and some more. The following sections list and detail their syntaxes.

6.1 if-elseIf-else

The syntax of the construct `if-elseIf-else` is straightforward:

```
<if-elseIf-else> ::=
    "if" <expression>
        [ <body-of-code> ]
    ( "elseIf" <expression>
        [ <body-of-code> ] )*
    [ "else"
        [ <body-of-code> ] ]
```

Example:

```
if a > 10
    :foo(a)
elseIf a > 5
    :foo(-2)
```

6.2 while

Syntax:

```
<while> ::=
    "while" <expression>
        [ <body-of-code> ]
```

Example:

```
while a > 0
    ...
    a = a - 1
```

6.3 for-in

Syntax:

```
<for-in> ::=
    "for" <type> <identifier> "in" <expression>
        [ <body-of-code> ]
```

Example:

```
for Integer x in someList
    ...
    a = a - 1
```

6.4 break and continue

`break` and `continue` can be used within any construct `while` or `for-in`. Their effects are obvious.

6.5 stop

Syntax:

```
<stop> ::= "stop"
```

Use `stop` to stop a function from continuing its execution. This is similar to the construct `return` in most languages, but we have adopted `stop` for two reasons. First, `return` is usually used to pass a value back to the caller of a function; since in Mystique, one can have several results and can already assign a value to each result variable, there is no need for that particular function of `return`. The second reason is that while `return` suggests the return of execution to the caller, this might not always happen in Mystique—we will explain this point later.

6.6 let

Syntax:

```
<let> ::=
    "let" <type> <identifier> "=" <expression>
        [ <body-of-code> ]
```

Example:

```
let Locale locale = new(Locale("fr_CA"))
    ...
```

7 Reactivity

We have discussed enough of the basic syntax of Mystique that we can now turn our attention to the fundamental behavior aspect of the language: its reactive nature to changes. If Mystique functions were entirely functional—if they bring about no side-effects, then the outputs of a function can be updated to reflect its inputs simply through full re-execution of that function (and any function that it subsequently calls). Optimizations can be introduced to re-execute only the affected code.

However, Mystique functions do bring about side-effects. While Mystique does not have any native first-class data structure, Mystique can be used to manipulate Java data structures, upon which side-effects can be brought about.

It is imperative that Mystique programmers be able to control the execution of code that causes side-effects. Full re-execution of all code upon any change is certainly not acceptable, nor is it desirable to employ partial re-execution optimizations that Mystique programmers do not understand or have no control over. We must provide language support (in syntax and semantics) for specifying which pieces of code should be re-executed upon which kinds of change.

7.1 Invocation Records and Invocation Trees

Let us consider the execution of a Mystique function. This function might call other functions (or itself), which might subsequently call yet other functions (or themselves again). Every time a function is called, an *invocation record* is made to store the context in which the function is being called, including information such as actual parameter values. An invocation record is almost like a stack-frame in compiler terminologies.

When a function is called several times, there are several invocation records for that function, each record storing a different context and corresponding to a different invocation. Every loop is "unrolled" so that the function calls in its code body have several invocation records corresponding to the several iterations of that loop. Considering all of the invocation records together starting from one function—a *root function*, we have a tree of invocation records—an *invocation tree*. The invocation tree is just a generalization of the call-stack: if stack-frames in a call-stack are not deallocated, then the call-stack would become an invocation tree.

Note that an invocation tree includes only the invocations that are actually made under a particular set of conditions in which the root function is called. That is, we would have a different invocation tree if the actual parameters to the root function were different, or if certain environment settings (such as the values of dynamic variables) were different.

The branches from each node in an invocation tree are ordered by the order of invocation. That is, if function A invokes function B, function C, and then function D, then there is one invocation record of A which has three child nodes, one invocation record of B, one of C, and one of D in that order. For convenience, we will order the invocation records from left to right, with the records to the left being the ones invoked first.

Note that *all* non-leaf nodes of invocation trees are invocation records of Mystique functions, while leaf nodes of invocation trees are *mostly* calls to Java code. It is unlikely that a Mystique function calls no other Mystique function or Java-implemented function. Hence, it is unlikely that the invocation record of a Mystique function is a leaf node.

7.2 Propagation of Changes

Changes always originate from calls (from Mystique code) to Java code. For example, the results of a call to an RDF store to extract RDF data can change as the data in the store changes. The result of a call to a Java "current time" function changes periodically. These changes always affect leaf nodes of invocation trees because they originate from Java code. They are what subsection 2.2 refers to as external changes.

External changes bring about internal changes as code is being re-executed. Consider the following Mystique function:

```
function util:printTime()
    Clock c = new(Clock("second"))
    String s = c.getTimeAsString("hh:mm:ss")
    System.out.println("The current time is ".append(s))
```

The `Clock` object stored in the variable `c` is instructed to update every second by the single argument to its constructor. Consequently, the result from the call `getTimeAsString("hh:mm:ss")` changes every second. Such is an external change since the method `getTimeAsString()` is implemented in Java and the mechanism by which it can generate changes has been abstracted away from all of our Mystique code. However, that external change in turn changes the content of the variable `s`—this is an internal change. This change in turns causes the result of the method `append()` to change, and so on.

7.3 Re-execution

When the execution of a function is viewed as an invocation tree, it is easy to see what needs to be re-executed upon an external change. As mentioned, an external change always occurs at a leaf node. From this leaf node, we traverse up the invocation tree to the parent node—the caller of the function whose results have just changed. We go through the code of the caller starting right after the call and *update* any expression or statement that is affected directly or indirectly by the change. If such updates cause the results of the caller to change, then we traverse up the invocation tree again and repeat the updating process with the caller's caller, and so forth.

A function call is affected (and hence will be updated) if any of its actual parameters has changed. Updating a function call involves updating those expressions and statements in the function's code body that are affected by the change(s) to the actual parameters.

In the case a control construct such as `if-elseIf-else` needs to be updated, the invocation tree might be altered as the code path is re-routed. For example, consider the following short-circuit logical-and function with a call-back:

```
function logic:and(Boolean x, Boolean y, Function callback)
    if x
        if y
            callback(true)
            stop
    callback(false)
```

When this function is updated because `y` changes from `true` to `false` (while `x` remains `true`), the inner `if` construct no longer executes its consequent.

7.4 Optimizations for Updates

We can employ the following optimizations:

- Updating a short function with no side effect can be optimized by re-executing the entire function again.
- Changes can be encoded as minute deltas and processed as such (e.g. when a list has a new addition, process just the addition, not the entire list again).

8 Advanced Constructs

We need to introduce more constructs that give programmers more control over when code bodies are executed and updated.

8.1 initially-normally-finally

The construct `initially-normally-finally` allows programmers to specify code blocks that are executed once rather than updated upon changes. Its syntax is:

```
<initially-normally-finally> ::=
    "initially"
        [ <body-of-code> ]
    "normally"
        [ <body-of-code> ]
    [ "finally"
        [ <body-of-code> ] ]
```

If an `initially-normally-finally` construct were considered a function, then right after its invocation record is constructed, the code body after `initially` is executed once and never updated. Just before the invocation record is destroyed, the code body after `finally` is executed. The code body after `normally` is executed and updated as usual.

8.2 observe-ignore

The construct `observe-ignore` allows programmers to specify which code blocks depend on which variables. Its syntax is:

```
<observe-ignore> ::=
    "observe" <list-of-identifiers> [ "ignore" <list-of-identifiers> ]
        [ <body-of-code> ]
  |
    "ignore" <list-of-identifiers> [ "observe" <list-of-identifiers> ]
        [ <body-of-code> ]

<list-of-identifiers> ::=
    <identifier>
  | <identifier> "," <list-of-identifiers>
```

All expressions and statements within the body of code are marked to be affected by changes to the variables named by the identifiers following `observe`, and to be unaffected by changes to the variables named by the identifiers following `ignore`, regardless of whether those expressions and statements refer to those variables. For example, the following function:

```
function :onClickHandler(Event event)
    if event != null
        observe event
            System.out.println("Click")
```

prints out "Click" whenever `event` changes somehow. Because the last statement does not refer to `event`, if we don't use `observe event`, this statement would not be re-executed when `event` changes.

8.3 try-catch-finally and throw

Mystique code can throw and catch exceptions just like Java code can, except that a thrown exception does not cause the current code path to unwind until an appropriate exception handler is found. Rather, all code that has been executed thus far (until when the exception is thrown) is "held" and subsequently updated if any change affects them.

9 Syntax Containers

This section diverges from the reactive nature of Mystique and touches slightly on an entirely different aspect of the language that also makes programming UI in Haystack easier: native support for XHTML and SVG syntaxes. Mystique's syntax incorporates other syntaxes in *syntax containers*. It is best to illustrate syntax containers through code samples, such as the following two:

```
someXHTML = xhtml:{
    <p>Hi, my name is <b>David</b>.</p>
}

someSVG = svg:{
    <svg width="140" height="170">
        <circle id="theCircle" cx="70" cy="95" r="50"
            style="stroke: black; file: none" />
        <path id="thePath" d="M 75 90 L 65 90 A 5 10 0 0 0 75 90"
            style="stroke: black; file: #ffcccc" />
    </svg>
}
```

Here, `xhtml:{ ... }` and `svg:{ ... }` are the two syntax containers.

We will make slight changes to the syntaxes of XHTML and SVG to support substitutions. This support is both for internationalization and for dynamic computations of XHTML and SVG attributes. For examples:

```
someXHTMLinEnglish = xhtml:{
    <p>Hi, my name is <b>{%1}</b>.</p>
}
someXHTMLinFrench = xhtml:{
    <p>Bonjour, je m'appelle <b>{%1}</b>.</p>
}

someSVG = svg:{
    <svg width="140" height="170">
        <circle id="theCircle" cx="70" cy="95" r="50"
            style="stroke: black; file: none" />
        <path id="thePath" d="M {%1} {%2} L 65 90 A 5 10 0 0 0 75 90"
            style="stroke: black; file: #ffcccc" />
    </svg>
}
```

We can now use either `someXHTMLinEnglish` or `someXHTMLinFrench` depending on the locale and simply substitute in the name where `{%1}` is. As for the SVG data, we can substitute in the origin of the path where `{%1}` and `{%2}` are.