An Embedded Haskell Subset Implementation

Ian Lewis
ij_lewis@utas.edu.au
School of Computing, University of Tasmania, GPO Box 252-100, Hobart, Tasmania, Australia, 7001
Phone: +61 3 6226 2952, Fax: +61 3 6226 1824

Abstract

We provide an implementation of a Haskell [1] subset embedded within the Java programming language. The subset provides type inference, polymorphism, first-class functions, currying, and mixing of Haskell calls within Java expressions. These calls are evaluated lazily before returning to standard imperative evaluation.

The implementation is via the language Genesis [2]: a Java extension that allows for meta-programming and syntax creation. Genesis even allows for the subset to be used independently, so that source files containing purely Haskell subset code are translated into Java classes.

Keywords

Meta-programming, extensibility, and Haskell.

1 Introduction

There are many approaches to providing imperative forms within a pure functional language. Whilst much progress has been made in precisely this area, current solutions still provide significant initial barriers to use [3].

A somewhat unexplored technique is to provide an embedded functional language within a mainstream imperative language. Although, Haskell itself has been embedded in XML [4] and Elegant [5] supports many evaluation strategies. Of primary interest would be the ability to switch between imperative evaluation and lazy evaluation at the user's behest.

To this end, we provide an implementation of a subset of the popular functional programming language Haskell within a recent Java extension, Genesis.

Genesis is an extension to Java that supports compile-time meta-programming by allowing users to create their own arbitrary syntax. This is achieved through macros that operate on a mix of both concrete and abstract syntax, and produce abstract syntax. Genesis provides a minimal design whilst maintaining, and extending, the expressive power of other similar macro systems (such as MS^2 [6], JSE [7], and Maya [8]). The core Genesis language definition lacks many of the desirable features found in these systems, such as quasi-quote, hygiene, and static expression-type dispatch, but is expressive enough to define these as syntax extensions.

Other imperative languages that offer powerful meta-programming are unable to match Genesis' powerful syntax creation facilities. As we shall show, Genesis is expressive enough to allow both the embedding of our Haskell subset within Java and to allow the translation to Java of source files containing only Haskell code.

1.1 Overview

The remainder of this paper is presented as follows. In section 2, we provide a brief explanation of the language Haskell. In section 3, we specify the Haskell subset. In section 4, we present an implementation using Genesis. In section 5, we describe some extensions to the subset. In section 6, we provide an analysis of this work with a brief mention of further work.

2 Background

Haskell is a non-strict purely functional language that has obtained much popularity both from programmers and researchers. It has many similarities to ML [9] and Miranda [10]. Haskell has been used successfully to implement a variety of applications such as a Perl6 implementation, a LaTeX pre-processor, and even a few graphical games.

Its evaluation proceeds in a *lazy* fashion whereby calculations are deferred until they are determined to be guaranteed to be necessary. Apart from possible improved performance, this feature allows for the use of conceptually infinite datastructures. However, lazy evaluation complicates input/output and, as a result, much work has been done to provide sequential forms. The most successful approach found so far is via *monads* and Haskell specifies a special syntactic sugar, `do`-notation, to simplify their use. Nonetheless, monads are a somewhat awkward solution and the approach taken by this work may be in interesting alternative.

Haskell has been praised for its clean syntax but this can also act as a deterrent for those unfamiliar with it. This clean syntax combined with the Haskell standard library (the *prelude*) results in very concise programs when compared to other languages [11].

PLC'06, June 26–29, 2006, Las Vegas, Nevada.

```
subset = (definition)+
definition ::= identifier = expr

expr ::=  if bExpr then expr else expr
          | \ identifier -> expr                // lambda function
          | let identifier = expr in expr
          | expr operator expr                  // arithmetic
operation
          | ( expr )
          | expr expr                           // function
application
          | identifier
          | literal                             // integer literals
          | []
          | expr : expr
          | head expr
          | tail expr

literal ::= digits+

operator ::= + | - | * | /                      // no precedences

bExpr ::= expr cOp expr | bExpr lOp bExpr
cOp ::= > | < | == | /=
lOp ::= && | ||
```

Figure 1: Haskell Subset Grammar

```
map = \f -> \xs -> if xs==[]
                   then []
                   else f (head xs) : map f (tail xs)
```

Figure 2: Declaration Example

3 Haskell Subset

The major goal of this work is to assess the viability of embedding within Java a lazily evaluated Haskell subset with clean syntax. As a result, a heavily restricted subset is all that is required for experimentation. The Haskell subset provides:

- integer literals and lambda functions (see section 3.1);
- lists: with the construction via cons, termination via the empty list (see section 3.1), and built-in functions for finding the head or tail of a list (see section 3.2);
- simple expressions: variable identifiers, let expressions, if-then-else expressions, infix expressions, and function application (see section 3.2); and
- infix operators for arithmetic and logical calculation.

It does not provide type signatures, currying, pattern-matching, list constructions, list comprehensions, tuples, where-clauses or any of a range of other sophisticated features such as type classes or the monadic `do`-notation. These features are possible to support, but are not required in this proof-of-concept implementation — much of Haskell can be expressed in more primitive Haskell constructs regardless [12].

No standard functions are part of the subset definition; it is part of the proof-by-implementation to define functions such as `map`, `take`, `foldr`, etc. in these specified primitives.

The evaluation of the subset is described in section 3.4, and its simplified type-system in section 3.5. Section 3.6 describes the embedding of this subset in Java.

The full grammar for the Haskell subset is shown in Figure 1, and further described in the following subsections.

3.1 Atoms

The subset provides four atomic elements (i.e. elements that cannot be further evaluated):

- integer literals;
- lambda functions of the form `\i->e`;
- the empty list (`[]`); and
- list construction via cons (**:**).

3.2 Expressions

The subset provides five general expressions:

- simple identifiers (e.g. `list`);
- arithmetic infix operators for addition, subtraction, multiplication, and division (e.g. `4+7*2`);
- let expressions; (e.g. `let x=4 in x*x`)
- if-then-else expressions (e.g. `if x>4 then x else 7`); and
- function application (e.g. `f (4*x) list`).

If-then-else expressions require a Boolean expression for their condition. This is provided via explicit grammar (see the final three lines of Figure 1) that provides basic comparison operators (greater-than, less-than, equals, not-equals) and Boolean operators for *and* and *or*. There is no general Boolean type (see section 3.5).

The subset provides `head` and `tail` for the deconstruction of list atoms that respectively return the first element of a list and its remainder. These functions are the only provided facilities for deconstruction of lists.

3.3 Declarations

Declarations bind a functional definition to an identifier. Multiple declarations can appear consecutively and do not require any special layout or separation. Effectively, a single equals-sign acts as a reference-point for determining the beginning of declarations. Figure 2 shows an example declaration.

3.4 Evaluation

The evaluation rules for the seven expression types are illustrated in Figure 3 where square brackets indicate evaluation and σ is the current environment (i.e. a list of variable bindings). These rules are generally straight-forward, but some cases benefit from a little explanation:

- both `head` and `tail` evaluations evaluate their argument to see if it results in a list atom: if so, evaluation proceeds with the appropriate component of this atom and if not, a run-time error occurs (this is the only run-time error produced by the subset);

$$\begin{array}{ll}
[i]_\sigma & \Rightarrow \sigma\ i \\
[e_1\ \mathbf{op}\ e_2]_\sigma & \Rightarrow [e_1]_\sigma\ \mathbf{op}\ [e_2]_\sigma \\
[\mathbf{if}\ b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]_\sigma & \Rightarrow \mathbf{if}\ [b]_\sigma\ \mathbf{then}\ [e_1]_\sigma\ \mathbf{else}\ [e_2]_\sigma \\
[\mathbf{let}\ i = v\ \mathbf{in}\ e]_\sigma & \Rightarrow [e]_{[i=[v]_\sigma]\sigma} \\
[e_1\ e_2]_\sigma & \Rightarrow \mathbf{let}\ (\lambda i \to e')_{\sigma'} = [e_1]_\sigma\ \mathbf{in}\ [e']_{[i=[e_2]_\sigma]\sigma'} \\
[\mathbf{head}\ e]_\sigma & \Rightarrow \mathbf{let}\ (x : xs)_{\sigma'} = [e]_\sigma\ \mathbf{in}\ [x]_{\sigma'} \\
[\mathbf{tail}\ e]_\sigma & \Rightarrow \mathbf{let}\ (x : xs)_{\sigma'} = [e]_\sigma\ \mathbf{in}\ [xs]_{\sigma'}
\end{array}$$

Figure 3: Evaluation Rules

$$\begin{array}{lll}
A \Leftrightarrow t & \Rightarrow t & (\therefore A = t) \\
t \Leftrightarrow A & \Rightarrow A \Leftrightarrow t & \\
[t] \Leftrightarrow [u] & \Rightarrow t \Leftrightarrow u & \\
(t \to u) \Leftrightarrow (v \to w) & \Rightarrow (t \Leftrightarrow v) \to (u \Leftrightarrow w) & \\
\mathbf{int} \Leftrightarrow \mathbf{int} & \Rightarrow \mathbf{int} &
\end{array}$$

Figure 4: Unification Rules

$$\begin{array}{lll}
\mathbf{type}(i) & \Rightarrow \mathbf{find}(i) & \\
\mathbf{type}(n) & \Rightarrow \mathbf{int} & \\
\mathbf{type}(e_1\ op\ e_2) & \Rightarrow \mathbf{int} \Leftrightarrow \mathbf{type}(e_1) \Leftrightarrow \mathbf{type}(e_2) & \\
\mathbf{type}(\mathbf{if}\ b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) & \Rightarrow \mathbf{type}(e_1) \Leftrightarrow \mathbf{type}(e_2) & \\
\mathbf{type}([]) & \Rightarrow [A] & \\
\mathbf{type}(e_1 : e_2) & \Rightarrow [\mathbf{type}(e_1)] \Leftrightarrow \mathbf{type}(e_2) & \\
\mathbf{type}(\mathbf{tail}\ e) & \Rightarrow [A] \Leftrightarrow \mathbf{type}(e) & \\
\mathbf{type}(\mathbf{head}\ e) & \Rightarrow A & [A] \Leftrightarrow \mathbf{type}(e) \\
\mathbf{type}(\lambda i \to e) & \Rightarrow A \to \mathbf{type}(e) & \mathbf{add}(i = A) \\
\mathbf{type}(i = e) & \Rightarrow \mathbf{type}(e) & \mathbf{add}(i = A) \\
\mathbf{type}(\mathbf{let}\ i = v\ \mathbf{in}\ e) & \Rightarrow \mathbf{type}(e) & \mathbf{add}(i = \mathbf{type}(v)) \\
\mathbf{type}(e_1\ e_2) \Rightarrow A & \Rightarrow A & \mathbf{type}(e_1) \Leftrightarrow (\mathbf{type}(e_2) \to A)
\end{array}$$

Figure 5: Type-Inference Rules

- a let-expression evaluation adds a new binding in the environment for its variable and its value is evaluated with the original environment; and
- function application must resolve its left-hand argument into a lambda function and then create a new binding in the environment from the lambda's variable and its second argument. This new binding must be composed with the environment created by evaluation of the left-hand argument and is used to evaluate the lambda function's expression.

By definition, the atomic elements cannot be further reduced by evaluation and return unchanged.

The evaluation proceeds in a lazy fashion only at those points in which a new binding is added to the environment (i.e. let-expressions and function application). When a new binding is created, the calculation of its value is delayed until it is known to be required.

3.5 Type System

The subset provides only four types:

- integers: *int*;
- functions: *A*→*B* where *A* and *B* can be any type;
- lists: [*A*] where *A* can be any type; and
- arbitrary types.

Whilst this is significantly restricted from full Haskell, it is more expressive than it appears at first glance. The composition of function and list types and the use of arbitrary types provides for quite a complicated set of types.

Perhaps the most seemingly restrictive of the missing features is the lack of a Boolean type. Use of the conditional `if` expression functions as normal, but use of Boolean expressions is restricted to this situation alone.

The subset provides no way to provide type signatures or to work with types directly — all declarations have their types automatically inferred.

Type Inference

The type inference algorithm utilizes a common state that maintains the most specific type for all variables in scope. Operations that affect this common state are:

- `add(identifier)`: introduce a new variable into the state with its type initialized to be the arbitrary type;
- `find(identifier)`: search the state for the specified identifier and return its type; and
- (`type` t) ⇔ (`type` u): unify the two supplied types (the rules for unification are shown in Figure 4).

The rules for type-inference for the seven expressions and four atoms are shown in Figure 5. The final five rules contain extra actions that may affect the common state, but do not reflect the resultant type. Lambda functions, let-expressions, and declarations introduce new identifiers via the `add` function. As they are straight-forward, the rules for inference of Boolean expressions are omitted.

Figure 6 contains a type-inference example for the map declaration from Figure 2.

3.6 Embedding

The Java embedding for the Haskell subset is shown in Figure 7. A declaration is enclosed in braces prefixed by the single keyword `fun` and may appear anywhere a class-body declaration can. A functional call is surrounded by parentheses preceded by `fun` and can appear at the expression level. Figure 8 contains an example of embedded Haskell subset declarations and their use within standard Java code.

```
map = ...                        add map = A
  \f -> ...                      add f = B
    \xs -> ...                   add xs = C
      if ...
        xs==[]
          xs                     => C
          []                     => [D]
          => [D]                 C ⇔ D => C = [D]
        []                       => [E]
        f (head xs) : map f (tail xs)
          f (head xs)
            f                    => B
            head xs
              xs                 => [D]
              => D               [F] ⇔ [D] => F = D
            => G                 B ⇔ (D->G) => B = (D->G)
          (map f) (tail xs)
            map f
              map                => A
              f                  => D->G
              => H               A ⇔ (D->G)->H
                                         => A = (D->G)->H
            tail xs
              xs                 => [D]
              => [D]             [I] ⇔ [D] => I = D
            => J                 H ⇔ [D]->J => H = [D]->J
          => [G]                 [G] ⇔ J => J = [G]
        => [E]                   [E] ⇔ [G] => G = E
      => [D]->[E]
    => (D->E)->[D]->[E]
  => (D->E)->[D]->[E]
```

Figure 6: Type-Inference Example

```
expression ::= ... | fun ( fun_expr )
class_body_declaration ::= ... | fun { subset }
```

Figure 7: Java Embedding Grammar

```
import genesis.Haskell.*;

class FunExample {
  fun {
    map = \f -> \xs -> if xs==[]
      then []
      else f (head xs) : map f (tail xs)
    take = \n -> \xs ->
      if (n==0) then xs else take (n-1) (tail xs)
    fib2 = \x -> \y -> x:fib2 y (x+y)
    fib = fib2 1 1
    square = \x -> x * x
  }

  public static void main(String[] args) {
    int x = fun(square 42));
    int y = 10;
    FunList result = fun(take y (map square fib));

    for (Iterator i = result.iterator(); i.hasNext(); ) {
      System.out.println(i.next());
    }
  }
}
```

Figure 8: Embedded Haskell Usage

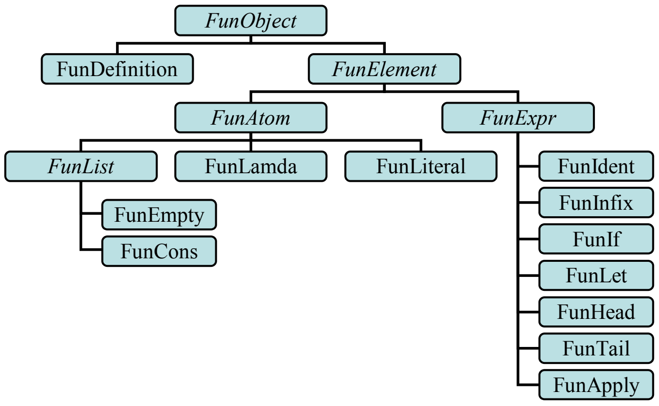

Figure 9: Haskell Subset Class Hierarchy

```
abstract class FunObject extends AbstractSyntax
                          implements Cloneable {
  protected FunDefinition environment;

  public abstract Creation createSelf();
  public abstract FunObject eval();
  protected abstract FunType funType(TreeMap variables)
    throws TypeMismatch;
}
```

Figure 10: Base Class Definition

4 Implementation

The implementation is provided via a number of Java classes shown in a hierarchy in Figure 9 (with abstract classes shown in italics). Each of these classes is used for a variety of purposes:

- to drive the parse at compile-time and provide syntax checking;
- for compile-time type-checking;
- at run-time to represent the structure of the functional program; and
- for run-time lazy evaluation.

4.1 Base Class

The base class is `FunObject` (see Figure 10) which provides three abstract methods for translation, functional type-checking, and evaluation: `createSelf`, `funType`, and `eval`. These methods have the following characteristics:

- Calling the `createSelf` method on a functional object will produce a Java creation expression (i.e. a sequence of `new` expressions) that when executed will reproduce the datastructure fully. This is used at compile-time to create run-time code that will store the functional objects.
- The `funType` method implements the type-inference algorithm specified in section 3.5. It uses a `TreeMap` datastructure to maintain the shared state.
- The `eval` method implements lazy-evaluation as described in section 3.4.

The `FunObject` class contains one instance variable that is the functional object's current environment — this is used at run-time during the evaluation process. The class also contains an overloaded `eval` method that updates the environment before evaluation occurs, and an overloaded `funType` method that initialises a `TreeMap` before typing.

4.2 Definitions

The class `FunDefinition` associates an identifier with its value (a `FunObject`). Apart from being used for actual definitions, this class is used to encapsulate the environment associated with each functional object. The environment of a definition is used for chaining. These chains may (and often do) contain circular references.

The value contained within a definition typically contains its own unique environment that is used for its lazy evaluation. Once it has been determined that the identifier within a definition is required for evaluation to proceed, the expression contained within its value must be evaluated (and the value is duly updated to reflect the result of this evaluation).

4.3 Elements

The `FunElement` class merely acts as a superclass for atoms and expressions in order to clearly differentiate them from definitions; it does not provide any further functionality than the `FunObject` class.

4.4 Atoms

The class `FunAtom` defines the abstract method `eval` as returning the current object. As all atomic objects share this behaviour the `eval` method is also specified as `final`.

4.5 Lists

The `FunList` class acts as superclass to the list construction classes. It also provides an `iterator` method and a custom iterator class for traversal of any function list that is returned after evaluation. Such lists are not fully evaluated upon return and require an iterator to evaluate each element on demand. Thus, lists are lazily evaluated even within Java code.

4.6 Expressions

The `FunExpr` class merely acts as a superclass for the seven expression classes for: identifiers; if-expressions; let-expressions; infix expressions; head; tail; and function application.

4.7 Syntax

The macros defining the Haskell subset syntax (from Figure 1) are shown in Figure 11 (with macros for Boolean expressions, operators, and for lists of definitions omitted). All simply create objects from our class hierarchy to guide the parse. The function application syntax macro examines its left-hand argument in order to catch applications of `head` or `tail`. The final macro allows parentheses to be used for explicit precedence.

4.8 Translation

Translation of Haskell subset code requires two macros, one for the group of declarations, and the other for functional calls. These macros are illustrated in Figure 12.

To translate declarations, we perform the required type-inference checking and, if successful, replace the embedded Haskell with a static-initialiser block containing a call to a run-time class (defined in Figure 13) with a re-creation of the compile-time datastructure representing the declarations. The run-time class remembers this datastructure as the common environment and ensures that it is correctly self-referential.

For functional calls the type of the functional expression is evaluated, and translated into a call to a specialised evaluation function (contained in the run-time class) for either integers or lists as appropriate. Functional calls resulting in a lambda function are considered to be erroneous.

The `createSelf` method is responsible for detecting unbound variables within functional calls (assumed to be Java identifiers) and their subsequent translation to a call to the run-time function `toFun`. The run-time class provides overloaded `toFun` methods for conversion from integers, collections, and arrays.

Figure 14 demonstrates the resultant translation of the embedded Haskell example from Figure 8.

5 Extensions

The Haskell implementation contains a few extra forms that are not specified in the subset. These demonstrate the ease of extending and the power available within the Genesis implementation. Genesis allows use of quasi-quotation for user-defined abstract syntax and this is used extensively throughout this section.

5.1 Function Declarations

Haskell provides a syntactic sugar for function declarations that does not require the use of lambda functions. For example, instead of `f=\x->\y->x+y` we could write `f x y=x+y`. Using quasi-quotation, we can provide this extension to our Haskell subset by using a mixture of Haskell and Genesis forms as shown in Figure 15. It simply constructs lambda functions from the identifiers in the argument list (in reverse). Trivial Genesis code for creating the list of functional identifiers is omitted.

```
macro FunDefinition (FunIdent i, =, FunExpr e) {
  return new FunDefinition(i, e);
}
macro FunLambda (\, FunIdent i, ->, FunExpr e) {
  return new FunLambda(i,e);
}
macro FunLiteral (LiteralInt x) {
  return new FunLiteral(x);
}
macro FunCons (FunExpr e, :, FunExpr f) {
  return new FunCons(e, f);
}
macro FunEmpty ([]) {
  return new FunEmpty();
}
macro FunIdent (Identifier i) {
  return new FunIdent(i.toString());
}
macro FunInfix (FunExpr e, FunOperator op, FunExpr f) {
  return new FunInfix(e, op, f);
}
macro FunIf (if, FunBExpr b, then, FunExpr e, else, FunExpr f)
{
  return new FunIf(b, e, f);
}
macro FunLet (let, FunIdent i, =, FunExpr e, in, FunExpr f) {
  return new FunLet(i, e, f);
}
macro FunExpr (FunExpr e, FunExpr f) {
  if (e instanceof FunIdent) {
    FunIdent i = (FunIdent) e;

    if (i.equals("head")) return new FunHead(f);
    if (i.equals("tail")) return new FunTail(f);
  }
  return new FunApply(e, f);
}
macro FunExpr ("(", FunExpr e, ")") {
  return e;
}
macro FunDefinition (FunDefinition d, FunDefinition e) {
  d.environment = e;
  return d;
}
```

Figure 11: Haskell Subset Macro Definitions

```
macro ClassDeclaration (fun, "{", FunDefinition d, "}") {
  // loop thru all declarations & check types...

  return {{
    { FunRuntime.setEnv(`(d.createSelf())); }
  }};
}

macro MethodCall (fun, "(", FunExpr e, ")") throws
TypeMismatch
{
  FunType type = e.funType();
  if (type instanceof FunTypeInt) {
    return {{ FunRuntime.evalInt(`(e.createSelf())) }};
  } else if (type instanceof FunTypeList) {
    return {{ FunRuntime.evalList(`(e.createSelf())) }};
  }

  throw new TypeMismatch("function call of type: " + type);
}
```

Figure 12: Haskell Embedding Macro Definitions

```
class FunRuntime {
  private static FunDefinition environment;

  public static void setEnv(FunDefinition e) {
    environment = e;
    // force the environment to be self-referential (for
recursion)
  }

  public static FunList evalList(FunObject obj) {
    return (FunList) obj.eval(environment);
  }

  public static int evalInt(FunObject obj) {
    return ((FunLiteral) obj.eval(environment)).value;
  }

  public static FunAtom toFun(int x) { ... }
  public static FunAtom toFun(Collection list) { ... }
  public static FunAtom toFun(Iterator i) { ... }
  public static FunAtom toFun(int[] array) { ... }
}
```

Figure 13: Run-time Class Definition

```
import genesis.Haskell.*;

class FunExample {
  { FunRuntime.setEnv( /* output from createSelf method */
); }

  public static void main(String[] args) {
    int x = FunRuntime.evalInt(new FunApply(
      new FunIdent("square"), new FunLiteral(42)
    ));
    int y = 10;
    FunList result = FunRuntime.evalList(new FunApply(
      new FunApply(new
FunIdent("take"),FunRuntime.toFun(y)),
      new FunApply(
        new FunApply(
          new FunIdent("map"), new FunIdent("square")
        ),
        new FunIdent("fib")
      )
    ));

    for (Iterator i = result.iterator(); i.hasNext(); ) {
      System.out.println(i.next());
    }
  }
}
```

Figure 14: Embedded Haskell Translation

5.2 Operator Currying

Haskell allows binary operator application to omit either parameter to provide a partial application. For example the expression `(1+)` returns a function that adds one to its argument. Figure 16 shows the simplicity of adding this to Haskell subset.

5.3 List Constructions

List constructions allow lists of a fixed length to be created without the use of cons and the empty list. For example, instead of `1:x+y:[]` we could write `[1, x+y]`. Figure 17 demonstrates the definition of list

```
macro FunDefinition
(FunIdentifier ident, FunIdents args, =, FunElement expr) {
  FunExpr lambdas = expr;

  forall (FunArgument arg) in args.reverse() {
    lambdas = {{ \`arg -> `lambdas) }};
  }

  return {{ `ident = `lambdas }};
}
```

Figure 15: Function Declaration Macro Definition

```
macro FunLambda ("(", FunOperator op, FunExpr e, ")") {
   return {{ \x -> x `op `e }};
}

macro FunLambda ("(", FunExpr e, FunOperator op, ")") {
   return {{ \x -> `e `op x }};
}
```

Figure 16: Operator Currying Macro Definitions

```
macro FunList ([, FunExprs es, ]) {
  FunList ret = new FunEmpty();

  forall (FunExpr e) in es.reverse() {
    ret = new FunCons(e, ret);
  }

  return ret;
}
```

Figure 17: List Construction Macro Definition

```
macro FunApply ([,FunExpr e, |, FunIdent i, <-, FunExpr f,]) {
   return {{ map (\`i -> `e) `f }}
}
```

Figure 18: Simple List Comprehension Macro Defn

```
module HaskellTest where

range a b = if (a <= b) then (a : range (a+1) b) else []

main x = [ a * a | a <- range 1 x ]
```

Figure 19: Standalone Haskell Module

```
public class HaskellTest {
  { FunRuntime.setEnv( /* output from createSelf */ ); }

  public static void main(String[] args) {
    if (args.length != 1) return;

    System.out.println(FunRuntime.evalList(new FunApply(
      new FunExpr("main"), FunRuntime.toFun(args[0])
    ));
  }
}
```

Figure 20: Standalone Haskell Translation

constructions with the omission of the trivial Genesis code for creating a comma separated list of expressions.

5.4 Simple List Comprehensions

Simple single-source list comprehensions can be provided by translation into use of the `map` function. This translation is so simple it can be provided in a single line as shown in Figure 18.

5.5 Standalone Usage

Genesis is capable of translating Haskell subset files that contain no Java constructs whatsoever. This standalone usage requires the specification of a module name within the source file — this is used to name the resultant Java translation class. An example of this syntax is shown in Figure 19. Note that this example uses some of the other extended forms defined in the preceding subsections.

Each standalone Haskell file must declare the function `main`. The definition of this function is permitted to contain any number of integer or integer list arguments that can be specified from the command line.

The translation of standalone Haskell follows much the same process as described in section 4.8. However, the surrounding class is created by the translator. Also created is a Java `main` method that accepts arguments from the console, performs a functional calculation, and outputs results to the console as demonstrated in Figure 20.

The Genesis compiler can be executed with only the Haskell subset syntax and without loading the default Java syntax using a command-line switch. In this form, the compiler acts as only a Haskell subset compiler.

6 Analysis and Conclusion

This work successfully embeds a Haskell subset within Java with clean syntax and lazy evaluation. The advantages of type-inference, polymorphic types, first-class functions, currying, and lazy evaluation are all supported.

Java code can contain calls to Haskell subset functions enabling the programmer to switch between imperative and lazy evaluation at will.

The Haskell subset is also a clear demonstration of the power and flexibility of macro definitions in Genesis. Not only is the Haskell syntax matched exactly (ignoring true Haskell layout rules), the implementation itself is relatively straightforward due to Genesis allowing multiple uses of its user-definable abstract syntax classes.

One might consider implementing a Haskell subset in a similar language to Genesis. Neither MS^2 nor Maya have the ability to express the required syntax in any form. MS^2 requires each macro to begin with a name and Maya is not capable of supporting the Haskell syntax due to its outside-in evaluation strategy. Even if it were, it is unlikely that Maya's LALR parser could handle the conflicts between functional expressions and standard Java expressions. While it might be possible to represent the syntax in JSE, this would be by pushing a sequence of unstructured tokens to a hand-coded parser.

Using Genesis' quasi-quotation facility many simple extensions to the subset are possible such as arithmetic operator currying, function declarations, and syntactic sugar for list constructions.

6.1 Planned Future Work

Planned and possible future work includes:

- the implementation of pattern-matching;
- extensions to the type-system, culminating in the support of type classes;
- addition of user-defined operators;
- allowing Haskell code to contain calls to Java methods; and
- ultimately, the implementation of the entire Haskell language specification.

References

[1] Simon Peyton Jones (editor): *Haskell 98 language and libraries: the Revised Report*, Cambridge University Press, January 1999; revised December 2002.

[2] Ian Lewis: *Genesis: An Extensible Java*, PhD Thesis, University of Tasmania, December 2005.

[3] Simon Peyton Jones: Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell, In, *Engineering theories of software construction*, IOS Press, Manfred Broy, Ralf Steinbruggen, pp. 47–96 , July 2002.

[4] Erik Meijer and Danny von Velzen: Haskell Server Pages: Functional Programming and the Battle for the Middle Tier. In, *Electronic Notes in Theoretical Computer Science*, Volume 41, Number 1, Elsevier Science, 2001.

[5] Lex Augusteijn: *Defintion of the Programming Language Elegant, Release 7.2*. Philips Research Laboratories, Eindhoven, the Netherlands, 1999.

[6] Daniel Weise and Roger Crew: Programmable Syntax Macros. In, *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI '93)*, pp. 156–165, Albuquerque, New Mexico, June 1993.

[7] Jonathan Bachrach and Keith Playford: The Java Syntactic Extender (JSE). In, *Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, a\nd Applications*, pp. 31–42, Tampa Bay, Florida, 2001.

[8] Jason Baker: *Macros that Play: Migrating from Java to Maya*, Master's Thesis, University of Utah, December 2001.

[9] Robin Milner, Mads Tofte, and Robert Harper: *The Definition of Standard ML*, MIT Press, 1991.

[10] David A. Turner: Miranda: a non-strict functional language with polymorphic types. In, *Functional Programming Languages and Computer Architecture*, Springer-Verlag, 1985

[11] Paul Hudak and Mark P. Jones: *Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity*, July 1994.

[12] Simon L. Peyton Jones: *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

.

.