Aldor meets Haskell

Chris Ryder (Supervised by Simon Thompson)

July/August 1999

Contents

1 Introduction

The aim of this project was to attempt to output a Haskell representation of the Aldor compiler's abstract syntax tree. The purpose of this is to enable the representation to be executed and to give an experimental platform in which to look at how to circumvent some of the limitations of the Aldor compilers type checker.

1.1 Overview of this report

Section 2 contains a description of how to setup and use the Haskell Aldor interpreter.

Section 3 describes how the Aldor compiler was modified to produce Haskell output.

Section 4 explains how the Aldor compiler represents the various aspects of the Aldor language.

Section 5 describes how the interpreter works, and also its capabilities and limitations.

2 Using the Haskell Aldor interpreter

To use the interpreter, you must first setup the `HUGSFLAGS` environment variable to include the path where the Haskell source files for the interpreter are stored e.g.

```
setenv HUGSFLAGS -P/usr/local/cs/pkg/hugs/share/lib:
                                  /home/cut/cr6/haskell/stable
```

Note that this environment variable must also include the directory containing the standard libraries for Hugs. This is because this path overrides Hugs builtin path.

During the initial stages of development, Hugs 1.4 was used. This proved to have a few problems (described later) so Hugs 98 was used instead. Hugs 98 allows command line configuration of some options which are only configurable by recompiling in Hugs 1.4 (described later). However, apart from this there appeared to be little difference. This is probably because the interpreter uses only basic Haskell code.

To use the interpreter, you need to generate a Haskell file from an Aldor source file. This is done like so

`axiomxl -Fhs` *file*`.as`

which results in a file called *file*`.hs` in the same directory (*file* can be any arbitrary name). To run the interpreter on this file, simply type `hugs` *file*`.hs`. This will automatically pull in all the necessary modules (assuming `HUGSFLAGS` is setup correctly) then present you with a prompt `Main>`. Type `runTree ast` to execute the tree and pretty print the result.

Note that the execution code only executes assignments. To evaluate any Aldor expression, `exp`, it is therefore sufficient to include the assignment

```
main := exp
```

(`main` can be any arbitrary choice of legal identifier).

There are two other functions that can be applied to trees. These are :

`showTree` This pretty prints a tree. The output of this function is much easier to read than the builtin `show` mechanism.

`executeTree` This takes a tree as an argument and returns the executed tree.

Hence, `runTree` is defined as `runTree = showTree . executeTree`.

3 A Haskell Representation of the AST

To output the abstract syntax tree as Haskell code, or more correctly as a Haskell data structure, two steps must be taken. The first step is to derive a Haskell type to represent abstract syntax trees. The second is to write the code to output particular abstract syntax trees.

3.1 Deriving a type

As a starting point, the `AbSyn` structure of the Aldor compiler was translated into Haskell. All `Syme` types were represented as `String`s and all `TForm` structures were ignored. This was because they are empty at the point at which Haskell code is produced. In addition to those nodes defined in the `AbSyn.h` file, a few extra nodes were added:

`Null` Represents an empty tree

`Error String` This is used to represent any nodes that haven't been output properly. This was used to allow incremental development of the Haskell output code. It is important to stress that these nodes do not represent errors, merely nodes which the code in the Aldor compiler does not yet know how to output correctly.

In addition to these extra nodes, `Apply` nodes were modified to remove some duplication. [1]

The abstract syntax tree was represented as an algebraic type, deriving from `Show`. Due to a known limitation in the Haskell interpreter, Hugs 1.4, [2] the tree could not derive from `Eq` which is needed to be able to type check the tree. As a workaround, the following definition was used.

```
instance Eq AST where
t1 == t2 = (show t1)==(show t2)
```

[1] An `Apply` node has the form `Apply AST [AST]` but it appears that the first `AST` is always the same as the first element of `[AST]`. Thus, the form was shortened to `Apply [AST]`.

[2] When attempting to compile the Haskell code in Hugs 1.4, if the `AST` type derives `Eq`, the error `ERROR "AldorAST.lhs": Compiled code too complex` occurs. There is an internal limit on the "complexity" of expressions. Hugs98 allows this limit to be adjusted, and hence allows this to compile. To be able to compile the Haskell code in Hugs98, you must increase this threshold using the `-c` option. A value of around 200 seems to work well.

It was also interesting to note that there were some node tags defined in `AbSyn.h` which appear not to have corresponding body definitions. They may use some generic body so, as a precaution, the Haskell output code represents them as `Error` nodes. So far, no `Error` nodes have been found in the output, which suggests these node tags may not be used.

3.2 Outputting Haskell

Continuing from previous work on the Aldor compiler [3], a hook into the command line to add the option `-Fhs` was implemented.

The code to output the Haskell representation is a recursive function that consists mainly of a large case statement. All node types have to have a case, since they need a different type constructor in the Haskell output. This means there is very little sharing of code (although it could be optimised a little more than it is). The first version of the code output the whole abstract syntax as a single line. This proved difficult for a human to read (indeed, the editor `vi` complained about the length of the line), although the Haskell interpreter, Hugs, had no problems with this format. Later versions of the code format the output in a slightly more human-readable form, breaking up the line and indenting to clarify the structure.

The compiler output first has two lines to import the definition of the `AST` structure (`AldorAST`) and definitions of the functions to act on the tree (`AbstractUtils`). Thus, the top of the Haskell output file looks something like this:

```
{-
Haskell representation of the AST from the Aldor compiler.
Produced from the file "test.as" on Tuesday Jul 27 1999 at 14:31
-}

import AldorAST -- For the types
import AbstractUtils -- For functions that act on the tree

ast :: AST
ast = ...
```

Haskell code is output on the basis of the abstract syntax tree present after macro expansion and scope binding but *before* type inference. A side effect of this is that the Aldor compiler will still type check a program, but only *after* the program has been output as Haskell. Thus, if a type error occurs, the Aldor compiler will tell you, but will have still produced Haskell code. This provides a way to compare the Aldor type checker with our own type checker, and also to work with programs rejected by the Aldor compilers type checker.

[3] A previous report, describing some of the internal structure of the Aldor compiler is available at `http://www.cs.ukc.ac.uk/people/staff/ep5/Aldor/chris_report.ps`

4 The Abstract Syntax Tree

4.1 The Haskell Representation

The Aldor compiler represents the Aldor program as an abstract syntax tree. Different combinations of nodes are used to describe different aspects of the language. This section describes some of the interesting nodes and how they are used.

4.2 How Aldor Uses Nodes

4.2.1 Assign nodes

`Assign` nodes are used by Aldor to represent assignments. Hence, they have two sub-trees to represent the left and right hand sides of the assignment. For example

```
a := 4
```

is represented as

```
Assign (Ident "a") (LitInt 4)
```

4.2.2 Apply nodes

`Apply` nodes are used by Aldor to represent n-ary applications. Hence, this node has a variable number of sub-trees. The first sub-tree is always the identifier of the function. The rest of the sub-trees are the arguments to the function. For example, the function application

```
... func(3, 4, 5);
```

is represented as

```
Apply [(Ident "func"), (LitInt 3), (LitInt 4), (LitInt 5)]
```

`Apply` nodes also have a second purpose. In line with the design philosophy of Aldor, as few primitives as possible are included in the system. This means that a function type is not represented as a primitive but rather as the application of the type constructor, `->`, to a tuple of type arguments [4]. For instance, the type of the function

```
func(i:Integer):Integer == i
```

is represented as

```
Apply [(Ident "->"),
       (Declare (Ident "i") (Ident "Integer")),
       (Ident "Integer")]
```

[4] This is discussed further in a paper by Simon Thompson and Erik Poll, available at `http://www.cs.ukc.ac.uk/people/staff/ep5/Aldor`.

This example shows the two representations of types. The simple representation of the return type `Integer` as an `Ident` and the more complex representation of the type of the function as a whole. This use of `Apply` nodes to represent types gets more complicated when functions take more than one argument. The type is then represented as an application of `->` to a `Comma` list of arguments and the return type. For example the type of the function

```
func2(i:Integer,j:Integer):Integer == i+j;
```

is represented as

```
Apply [(Ident "->"),
       (Comma [(Declare ...),(Declare ...)]),
       (Ident "Integer")]
```

Also, types that take arguments, such as lists, are also represented as `Apply` nodes. For instance the type

```
List Integer
```

is represented as

```
Apply [(Ident "List"),(Ident "Integer")]
```

4.2.3 Define nodes

`Define` nodes are used to represent the definitions in an Aldor program. Such a node has two sub-trees, the left and right hand sides of the definition. The left hand side is normally a `Declare` node, specifying the identifier and type. The right hand side is the body of the definition. For function definitions this is normally a `Lambda` node (see Section 4.2.5), whereas for simple declarations that take no arguments (e.g. `a:Integer == 3`) this is just the abstract syntax for the right hand side of the declaration. For example

```
a:Integer == 3
```

is represented like so

```
Define (Declare (Ident "a") (Ident "Integer")) (LitInt 3)
```

4.2.4 Declare nodes

A `Declare` node is used to represent declarations. A `Declare` node has two subtrees. The first is the identifier being declared. The second is the default type of the identifier. For instance, the declaration

```
a:Integer
```

would be represented as

```
Declare (Ident "a") (Ident "Integer")
```

4.2.5 Lambda

A `Lambda` node is a description of a function. It has three sub-trees. The first is a description of the parameters. The second is a representation of the return type. The third is the body of the function.

The parameters are represented as a `Comma` list of `Declare` nodes. This is even true of "no-arg" functions, where the parameters are represented by an empty `Comma` list.

The return type is the abstract representation of the functions return type, usually just an `Ident` node.

The body of the function is represented using a `Label` node. A label node has two sub-trees. The first is the identifier of the function, the second is the actual body of the function. For instance the function

```
func (i:Integer):Integer == i
```

is represented like so

```
Lambda (Comma [(Declare (Ident "i") (Ident "Integer"))])
       (Ident "Integer")
       (Label (Ident "func") (Ident "i"))
```

5 The Haskell Aldor interpreter

5.1 Limitations and capabilities

The Haskell Aldor interpreter has the ability to :

- Type check assignments.
- Type check arguments to functions, including functions as arguments and type arguments.
- Execute simple arithmetic (on `Integer` and `Float`).
- Execute recursive functions (only if the terminating condition can be evaluated).
- Execute functions that have functions and/or types as arguments.

The interpreter has a number of limitations (described below). These limitations are not caused by any fundamental problem. Rather, they are a result of the limited time available on this project (8 weeks). Because of the short time available, it was necessary to restrict the functionality to a small subset of the language. This also lead to the decision to start with a very small implementation and add functionality as time permitted.

- The definitions are not checked to ensure that their declared type is the same as their actual type.
- The code to execute Aldor abstract syntax trees is capable of very simple operations on lists. However, the type checker does not have support for lists, so fails when it encounters a list.

- There is very little of the `axllib` implemented. Only simple arithmetic on `Integer` and `Float` and limited support for `Boolean` types.

- Overloading of identifiers is not permitted.

The Haskell code for the interpreter is split into seven files. A brief description of each file is shown below.

`AldorAST.lhs` This file contains the definition of the Haskell type `AST` which is the Haskell representation of Aldor abstract syntax trees.

`AbstractUtils.hs` This file is automatically imported in the output of the Aldor compiler. Hence, any exports from this module can be used on the `AST` structure. All modules that are used on trees are imported by this module. By default, it imports all the other files in this list except `AldorAST.lhs`. It is possible to import your own modules in this module, allowing your own routines to be used on the abstract syntax trees.

`RecurseAST.hs` This file contains utility functions to do common tasks like passing a function over the `AST` structure. These are mainly long lists of case distinctions, and thus save a lot of work in the more interesting functions.

`PrintAST.hs` This file contains code to pretty print the `AST` structure. This is a much easier form to read than the builtin `show` function.

`Definitions.hs` This file contains the code to build a table of all the definitions in a program (e.g. all the function definitions). This is used for both type checking and execution.

`TypeCheckAST.hs` This file contains the code to do the type checking of the abstract syntax tree.

`ExecuteAST.hs` This file contains the code to do the execution of the abstract syntax tree.

As far as understanding the operation of the interpreter, the interesting files are `Definition.hs`, `TypeCheckAST.hs` and `ExecuteAST.hs`. Each of these files will now be described in more detail.

5.2 RecurseAST.hs

Although this module is not interesting as far as understanding the operation of the interpreter, it is worth mentioning in passing. This module contains four functions. All are used to thread monadic functions through the `AST` structure. The function that is passed through has must have the type `(a,AST) -> (a,AST)`. This is a function that may change the structure `a` as a side effect, and hence, is a mondic function. The functions in `RecurseAST.hs` make sure that no changes to the structure `a` are lost (by ensuring the result of one function call is passed as an argument to the next function call).

The four functions are :

`applyToList` This function applies a monadic function to a list of `AST`'s.

applyToSubTree This will apply a function to all the sub-trees of a node. This is a large function due to the large number of node kinds in the AST structure.

applyToTreeBU This applies a function to all the nodes in a tree in a bottom up pass.

applyToTreeTD This applies a function to all the nodes in a tree in a top down pass.

5.3 Definition.hs

The purpose of the code in this file is to produce a table of all the definitions in the Aldor program. Because of the limited time available for the project, it was decided not to allow the use of overloading. There is no fundamental problem with allowing overloading, but it significantly complicates both type checking and execution.

To make the code easier to read, a number of types are defined first. These are

```
type Type = AST
type Param = String
data Definition = FuncDef String [Param] Type AST |
                  IdDef String Type AST |
                  NotDefined
type DefTable = [Definition]
```

The FuncDef constructor of Definition is used to describe the definitions of functions. The String argument is the identifier of the function, the [Param] argument is the list of parameters of the function, the Type argument is the type of the function, and the AST is the body of the function. The IdDef constructor is used to describe the definition of simple identifiers that take no arguments. The table, DefTable, is implemented as a list to make manipulating the table easier, although it may not be the most efficient method of storage.

The main entry point into the code in this module is the function addDef, which takes a table of definitions and a piece of abstract syntax and returns the table with the definition added to to the head of the table. This function should be passed only Define nodes. For all other node types the table is returned unchanged. When a Define node is passed as an argument, addDef then uses the function makeDef to build a representation of the definition and appends it to the head of the list of definitions.

makeDef decides if the definition is the definition of a function or of a simple identifier (e.g. a:Integer==3). If the right hand side of the Define node is a Lambda node it is treated as a function definition, otherwise it is treated as a definition of a simple identifier. The function then builds a representation of the appropriate type from parts of the abstract syntax (see Section 4.2.3).

5.4 ExecuteAST.hs

The main entry point for executing the program's AST structure is the function executeTree. This function is a wrapper around the function execTree.

`execTree` takes a tuple of the type `(DefTable,AST)` and returns another tuple of the same type. This argument format is used to allow the use of the functions in `RecurseAST.hs` (see Section 5.2) to thread the `execTree` function through an `AST` structure. The `execTree` function ignores all nodes except the following:

`Assign` When an `Assign` node is found, the function `typeCheck` (see Section 5.5) is called. If this succeeds the function `evalAssign` (described below) is used to evaluate the node and the result is returned. This returned result will replace the original `Assign` node in the `AST` structure. (Remember, that a modified copy of the original `AST` structure is returned by the `executeTree` function).

`Apply` These nodes are treated just like `Assign` nodes, except that if the type check succeeds then the function `evalApply` (described below) is used to evaluate the node, and the result is returned.

`Ident` For these nodes, the function `evalIdent` is used to evaluate the node.

`Define` For these nodes are passed straight to the function `addDef` (described in Section 5.3).

`Not` These nodes are evaluated using the function `evalNot`.

`Test` The function `evalTest` is used to evaluate these nodes.

`If` These nodes are evaluated using the `evalCond` function.

There are several specialised functions that are used to evaluate the different nodes. These are described below:

`evalAssign` The parameters to this function are a table of definitions and the `Assign` node. If the right hand side of the node is an `Ident` node, the function `evalIdent` is used to evaluate it. If the right hand side is an `Apply` node, the function `evalApply` is used. For all other cases, the `Assign` node is returned unchanged.

`evalApply` This function decides what type of operation is being applied, then uses the appropriate function to evaluate the given `Apply` node. The function knows about four kinds of functions.

- User functions are those functions that have been defined in the program. These are evaluated using the function `evalUserFunc` (see Section 5.4.1).
- Library functions are those operations which are defined outside the program file being evaluated. Examples are operations such as `first` which are defined in `axllib`. For these types of operation, the function `evalLibFunc` is used.
- Binary operations (such as `+` and `-`) are evaluated using the function `evalBinOp`.
- For unary operations, `evalUnaryOp` is used.

Before an `Apply` node is evaluated, the arguments to the application are evaluated. This is done so that simple declarations and identifiers (such as `Ident "1"`) are resolved before the application itself is evaluated. The arguments are evaluated by using the `applyToList` function to apply `execTree` to each element of the list of arguments.

`evalIdent` Simple, builtin identifiers, such as `0` and `1` are converted into literals. For any other identifiers, the identifier is looked up in the table of definitions and its value is returned as the result of the evaluation. If the identifier is not in the definition table, the `Ident` node is returned unchanged.

`evalNot` This assumes the argument to be inverted is already evaluated. The function then uses pattern matching to invert the node.

`evalTest` This, like the `evalNot` functions, is essentially implemented by a few pattern matches.

`evalCond` This first evaluates the condition part of the `If` node using `execTree`. If the condition evaluates to `Ident true`, the "then" part of the `If` node is returned. If the condition evaluates to `Ident false`, the "else" part is returned. If the condition cannot be evaluated, the `If` node is returned unchanged. This can be a problem in a recursive function, since if the `If` node is returned unchanged, the interpreter may try to repeatedly evaluate the node.

5.4.1 Evaluating user functions

The function `evalUserFunc` is used to evaluate user defined functions. To do this, it first looks up the definition of the user function. It then extracts the definitions of the parameters from the definition of the user function. These are then converted into a list of identifier/value pairs by passing the parameter definitions and the actual arguments to the function `mapActToForm`. The body of the function definition is then extracted. The function `expandFuncBody` is then used to replace all occurrences of the parameter identifiers in the body with the appropriate values of the parameters. The result is a body that can then be executed using `execTree`.

5.5 TypeCheckAST.hs

The main entry point into the type checking code is the function `typeCheck`. This function takes, as arguments, a function that is capable of executing `AST` structures, a table of definitions and a piece of abstract syntax to type check. This function returns a `Bool` which is `True` if the abstract syntax type checks correctly and `False` otherwise.

This function only type checks `Assign` nodes. These nodes were chosen because they have a clear left hand side and right hand side, which must be of the same type. `Assign` nodes get type checked, but *only* if they are part of an assignment. All other nodes are assumed to type check.

The function that is passed as an argument to the `typeCheck` function is used by the type checking code to evaluate some of the abstract syntax during type checking. For instance, the code

```
idType(T:Type):Type == T;
a:idType(Integer) := 4;
```

needs to have `idType(Integer)` evaluated before the assignment can be type checked. The type checking code cannot call the `execTree` function from the `ExecuteAST.hs` module directly because the `ExecuteAST.hs` module imports the `TypeCheckAST.hs` module (to be able to use the `typeCheck` function). Hence, the `TypeCheckAST.hs` module cannot import the `ExecuteAST.hs` module, as this would cause a recursive dependency in the two modules. Because of this, the execution function is passed as an argument to all the type checking code that may need it.

The essence of type checking is to build a list of possible types for each side of an expression, then compare the two lists to see if there is a common type. If there is no common type, the type check has failed. If there is more than one common type, the expression is ambiguous. The expression passes the type check when there is exactly one common type.

Perhaps the most important function in this module is the function `matches` which takes two lists of types as arguments and returns `True` if there are any type matches in them. This function is the heart of the type checker. All other functions are used to build the lists of possible types.

To build a list of possible types of an expression, the function `getTypeList` is used. This function has the types of the literals (`LitInt`,`LitFloat`, etc) builtin. For `Declare` nodes, the right hand side of the declaration (the right hand side of the colon) is returned. For `Ident` nodes, if the identifier is in the table of definitions, the function `getUserIdType` is used to build the list of types. Otherwise the `Ident` node is assumed to be a type and thus the type `Type` is returned. The final node type for which this function produces a list of types is `Apply` nodes. If the operation being applied is defined in the table of definitions, the function `getUserFuncType` is used to get the type list. If the operation is one of the builtin operations, the function `getBuiltInType` is used.

The function `getUserIdType` is used to return the list of possible types for a user defined identifier. This involves looking up the identifier and retrieving the appropriate entry. From this entry the type of the identifier is obtained, which is then returned as a list.

The function `getUserFuncType` returns the list of possible types for a user defined function. This starts by looking up the definition for the function. When it is found, the stored type of the function is retrieved. This type, however, may have type variables, so these must be replaced with their actual values. To facilitate this, the type (which is represented by a single `AST`) is broken up into a list of types, corresponding to the arguments and return type of the function. The function `getInstanceType` is then used to convert the list of types into the list of *actual* types. That is, any type arguments are replaced by the appropriate value from the parameters of the function. The expected types of the parameters are then determined, along with the actual types of the arguments. These types are compared, and, if they are the same, are returned as the type of the function.

The function `getBuiltInType` is used to determine the types of builtin operations such as `+`. This is achieved by looking up the operation in a hard coded table of builtin operations and their types. This lookup will return a list of types, because the builtin operations work on many types. To determine which type is the correct type, the types of the arguments to the operation must be determined. Once the types of the arguments are determined, the type of the operation is chosen and returned.