Making Haskell .NET Compatible

Department of Computer Science
University of Canterbury
Christchurch, New Zealand

Liam O'Boyle
lab63@student.canterbury.ac.nz
Supervisor: Dr. Nigel Perry

Abstract

Microsoft's new .NET platform makes use of tools, built in a wide variety of languages, all running on one virtual machine, the .NET Common Language Runtime (CLR). Many existing languages are to be made compatible with the .NET CLR, in order to make the platform more appealing to experienced software developers, and new languages are also being developed specifically for the system. As one of the most popular functional languages, it is intended that Haskell 98 be made .NET compatible. The approach taken in this project involves a translation of the Glasgow Haskell Compiler's internal language, GHC Core, to Mondrian Core, and internal language of the Mondrian compiler, to achieve Haskell compatibility with the CLR. A detailed description of the development of a tool to achieve this translation is described also.

Contents

List of Tables

Chapter 1

Background

1.1 Purpose of the Project

Microsoft's new .NET platform (see Section 1.3, Page 4) incorporates a variety of languages with which to develop software executable on the .NET Common Language Runtime (CLR). To be .NET compatible, a language must compile to CLR byte code; this requires either modifications to existing compilers or the creation of entirely new compilers for these languages. Microsoft would like to have Haskell, one of the most popular functional languages, compatible with the .NET platform. Modification of an existing compiler has been done before (see Section 1.2), but these modifications are difficult to update when a new version of the compiler is released, or when a language is updated. The alternative, a new compiler for Haskell, would be a significant task. Fortunately several tools exist already that can greatly simplify the development of this compiler, specifically the Glasgow Haskell Compiler and the Mondrian compiler, which are discussed further on.

1.2 Previous Work

1.3 Microsoft .NET and the Common Language Runtime

The .NET platform is intended to facilitate the development of interactive, web based systems, bringing together remote systems using intelligent clients, servers and services communicating through the internet. Information about the goals of the .NET project can be found at http://www.microsoft.com/net/. All of the software required to provide these services will be run on the .NET virtual machine, known as the Common Language Runtime. A large variety of languages are planned to run on the CLR, to make it as simple as possible for experienced developers to create .NET compatible software. There will also be several new languages, such as C# and Mondrian, designed specifically for the .NET platform.

One of the major aims of the .NET platform is to facilitate communication among programs, regardless of language (see Levy (2002)). By compiling languages to run on the same platform, all languages, by default, will use the same type system, exceptions, calling conventions etc., making interoperability much more simple.

There are many advantages in compiling code to an intermediate form, executable on a virtual machine (VM), rather than native code. Meijer & Gough (2002) lists several, including

- Portability - much fewer translators are required; whereas compilation to native code requires

one translator per language per platform, compilation to an intermediate code require one translator to intermediate code per language plus one translator from intermediate code to native for each platform.

- Compactness - Intermediate code is often much more compact than source code.
- Efficiency - the execution platform can use its knowledge of the underlying platform to improve software performance.
- Security - Higher level intermediate code is more amenable to security and type constraints than low level binaries.
- Interoperability - By sharing a type system and a runtime environment providing common services such as garbage collection, threading and security, interoperability between languages is greatly improved.
- Flexibility - High level code is more amenable to analysis such as reflection, serialisation, type browsing etc.

There are several similarities between the Java Virtual Machine (JVM) (see http://java.sun.com), the most popular VM in use at the moment, and the CLR. Similarities include a design intended to work well with OO languages, a similar code density in compiled code and use of a byte code (1 byte per instruction). The most significant difference between the two systems is that the CLR was designed to support multiple languages, while the JVM was intended to to support only Java; the JVM lacks the primitive type support necessary for the efficient implementation of many languages, and provision for unsafe type features[1] (Meijer & Gough 2002), as they are not necessary for Java interpretation. The current version of the CLR, 1.0, contains some support for a wider range of primitives, and support for unsafe types and their conversion, making it a much better candidate for a multi-language VM. CLR 2.0 is currently under development and is expected to provide improvements, although exactly what it will contain is not publicly available yet. Another advantage of the CLR over the JVM will be its intended support of polymorphic types, where groups of types can be treated in the same way (Kennedy & Syme 2002); it is not clear when this will be included, however. The concept of polymorphism is explained more fully in the following section, Haskell 98.

For a detailed comparison of the two VMs see Gough (2002), and see (Meijer & Gough 2002, Kennedy & Syme 2002) for more detailed information about the CLR itself.

1.4 Haskell 98

Haskell is a polymorphically typed, lazy, purely functional language, the current version of which is Haskell 98. Haskell is one of the most popular functional programming languages, with usage in industry and versions of it are commonly used for teaching functional programming concepts (most notably Hugs - see http://www.haskell.org/hugs). Functional programming has several advantages over both procedural and Object Oriented programming. They are, on the whole, much more concise than their procedural and OO counterparts, as many complex algorithms can be specified in a much more simple manner.

Complex algorithms written in functional languages also tend to be much clearer; an extreme example of this is the quicksort algorithm, which sorts a list of objects, by splitting the list in to three

[1] Such as pointers, intermediate descriptors and unsafe type conversions

smaller lists, one containing a “pivot” value, one containing all values less than that value and one containing all values greater than that value, and then applying itself to each of the resulting lists. The source for a Haskell version of quicksort is shown here.

```
qsort []     = []
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
 where
    elts_lt_x   = [y | y <- xs, y < x]
    elts_greq_x = [y | y <- xs, y >= x]
```

The C source code is shown in the appendix A, page 17, as it is much longer (see the first advantage of functional programming!). The Haskell code is reasonably simple to understand with a very basic understanding of Haskell; the ++ operator concatenates the three lists together, the elts_lt_x is the list of values that are in list xs, but less than the pivot value x (this definition is much the same as one would see in a lambda calculus, upon which Haskell is based) and elts_greq_x is those values greater than or equal to x. The C example does exactly the same thing, but in expressing the idea the programmer also has to handle exactly how the list is traversed, how the pivot is chosen, and maintain pointers to their current location in the list. This simplicity in expressing complex algorithms means that there is much less room for programmer error as well as more easily understood code.

Functional languages also provide higher order functions, where functions can be treated in the same way as variables in a procedural language; they can be passed as arguments to functions, returned from functions stored and so forth.

Haskell’s polymorphic typing allows constraints on the type of arguments upon which a function can operate, instead of the specific type of an argument. For example, the Haskell quicksort shown above will operate on any type which can be compared with the $>$ and $<$ operators, while the C version can only operate on integers. This makes the code shorter again and much more reusable, as one algorithm can work with many types.

Another advantage of Haskell is lazy evaluation, where only evaluation required to produce the final result is performed, which can avoid errors, and evaluation is not performed until it is needed, which can reduce memory requirements. For example, a function could use an infinite list, but only the elements of the list that were used would ever be evaluated; in most procedural or OO languages, even the definition of such a list would be impossible.

For a complete description of the Haskell language, see Jones, Hghes, Augustsson, Barton, Boutel, Burton, Fasel, Hammond, Hinza, Hudak, Johnsson, Jones, Launchbury, Meijer, Peterson, Reid, Runciman & Wadler (1999).

1.5 The Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) is a robust, fully-featured, optimising compiler and interactive environment for Haskell 98. GHC compiles Haskell to either native code or C. It also implements numerous experimental language extensions to Haskell 98, but this is not of great concern to this project. Additionally, the GHC was designed with the intention that it be easy to make use of portions of the compiler, with user developed extensions replacing certain components; in this case, we will be replacing the final compilation process with a translation and the Mondrian compiler.

1.6 The GHC Core Language

GHC Core is an internal language of the GHC. It is somewhat similar to a subset of standard Haskell, but with explicit type information added. The GHC parses Haskell and creates GHC Core code, with full type information. This is then rewritten by the optimiser, before eventually being written to either C or executable code. The GHC can be forced to output core by including the `-fext-core` parameter when compiling code.

This means that we can use the GHC to type check, parse and optimise the Haskell code, as well as allowing it to reduce the Haskell to GHC Core, which is lower level, so more similar to Mondrian Core (see Section 1.8, Page 7).

The details of GHC Core will be discussed in more depth in Section 2.3, Page 10.

1.7 Mondrian

Mondrian is a new functional language, currently being developed at Canterbury University, New Zealand. It shares many of the advantages of common functional languages, such as higher order functions and lazy evaluation. Additionally, as it is being specifically designed to run on the .NET CLR, it is much more adept at dealing with OO classes, threads and exceptions (see Section 1.3, Page 4) than any other functional language currently available. The .NET support for interoperability makes Mondrian very useful; with its ability to handle OO types, it can easily be used to write the complex algorithms for other programs, while leaving areas like GUI implementation to other languages, such as C#, which are better suited to such tasks. The .NET CLR also makes it possible to implement and use libraries, so that the language in which the library, and the language which is calling the library don't matter.

The Mondrian compiler compiles Mondrian to Mondrian Core (See Section 1.8, Page 7), which is then compiled into C#. The .NET C# compiler is finally used to create .NET byte code. At present Mondrian does not require explicit definitions of type; instead they are inferred from the context. This simplifies the task of translation, as it allows us to more or less ignore types in the translation from GHC Core to Mondrian Core.

1.8 The Mondrian Core Language

Like the GHC, the Mondrian compiler also makes use of an internal language, Mondrian Core, which it can accept from external files instead of pure Mondrian. Mondrian Core is, by design, more simple than the GHC Core language, as Haskell is a higher level language than Mondrian. Mondrian Core itself is quite similar to the original Mondrian source, and some understanding of it can be gained by reading the documentation available on the Mondrian website (Perry 2001). Also provided in Appendix B.1, page 19, is a grammar for Mondrian Core, which is derived from the current source of the Mondrian compiler; as Mondrian is still in development, no formal definition of the grammar is currently available, and this grammar may not be valid for long. Please see the Mondrian web site, http://www.mondrian-script.org, for the most up to date version of the Mondrian system. As with the GHC Core language, further explanation of Mondrian Core will be given in Section 2.3, Page 10.

Chapter 2

Approach

2.1 Overview

The plan is to use the GHC to parse, type check and optimise the code (see Section 1.6, Page 7), and output the result in GHC Core format. This output will be piped into a tool which will parse the GHC Core code and convert it into the Mondrian Core. The generated Mondrian Core will be passed on to the Mondrian Compiler, which will then compile it to C#, which can be fed to the .NET C# compiler and finally be compiled into .NET CLR byte code. This process is shown in Figure 2.1.

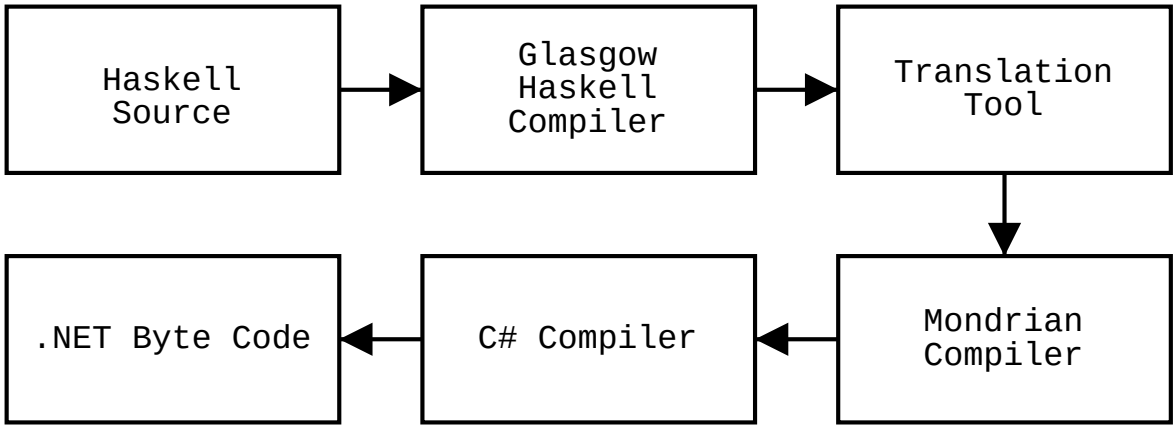

Figure 2.1: Compilation Steps in the Proposed System

2.2 Parsers

Before implementing a parser for the GHC Core, it was necessary to look into some of the available parser generators; it was a given that a parser generator would make a more efficient and less error prone parser than myself. There are many available parser generators, of which several were considered, including Happy, Yacc/Bison, JavaCC, and ANTLR as these were well known and I have had some experience using similar tools before.

2.2.1 Yacc and Bison

Bison and yacc are both C/C++ LALR(1) parser generators, available on most Unix and linux systems. Bison is simply the GNU implementation of yacc, and shares most of the same features. As the GHC Core language is describable by a LL(1) grammar (although the grammar supplied in Tolmach (2000) is not, for the purposes of clarity), a LALR parser is not necessary; there seems to be no clear verdict of which is the better parser to use in this situation. Two factors put bison/yacc out of consideration First, the lack of a a tree parser, allowing a more intelligent parsing of GHC Core; this feature is available in both JavaCC and ANTLR, and its lack would have made the task of translation much more difficult in a bison/yacc solution. Second, there was no support for a lexical analyser[1] within the parser generator; while it is possible to add this functionality through other devices (the most obvious being lex/flex, see http://www.gnu.org/manual/flex-2.5.4/html_mono/flex.html), this was another complication. More information about bison/yacc can be found using the Unix/linux man pages (`man yacc` and `man bison`) or at http://campuscgi.princeton.edu/man?yacc or http://www.gnu.org/manual/bison-1.35/bison.html for yacc and bison respectively.

2.2.2 Happy

Happy is a LALR(1) parser generator written in Haskell, which has the advantage of being .NET compatible by default (given the eventual completion of this project). It shares most of the same features and flaws as yacc/bison, including the lack of a tree parser and lexical analyser. For these reasons, Happy was not chosen for the development of the parser. For more information about Happy, please see http://www.haskell.org/happy.

2.2.3 JavaCC

JavaCC is a very thorough set of tools for parser generation. It develops LL(k) parsers in Java, which will be supported by .NET. It can also create complete parse trees, with support for tree manipulation, such as putting nodes back on to the stack to be scanned again, renaming nodes and suppressing nodes for some non-terminals, which is useful when translating GHC Core to Mondrian Core. There is also very good error handling support, although this should not be a problem, as the code produced by the GHC will always be syntactically correct; however, it will be useful to include until full Haskell support is implemented. Also, the documentation provided is very thorough and a repository of real language examples, including grammars of common languages such as Java, is also available at http://www.cobase.cs.ucla.edu/pub/javacc/. Finally, JavaCC is capable of creating a lexical analyser along with the parser. These advantages meant that the final choice was to develop the parser using JavaCC. For more information about the JavaCC system, see http://www.webgain.com/products/java_cc/.

2.2.4 ANTLR

ANTLR, ANother Tool for Language Recognition, is capable of generating parsers in both C++ and Java. Like JavaCC, it creates LL(k) parsers, which are sufficient to parse the GHC Core language, it can create tree parsers, has some support for error handling, and can also create a lexical analyser within the parser. The level of documentation available is not as good at that available for JavaCC,

[1] A lexical analyser breaks down the input into tokens, rather than a simple stream of characters, and is necessary in almost all language parsers.

and the number of examples available is very limited. Overall, the capabilities of ANTLR seem to be much the same as those of JavaCC, but clarity in the documentation and the availability of example grammars gave the edge to JavaCC. For more information about ANTLR, see http://www.antlr.org/.

2.3 Translation

2.3.1 Modifications to the Original GHC Core Grammar

The choice of JavaCC for parser development required that the grammar be converted to LL(k), preferably LL(1) for the purposes of speed and computational efficiency. While most of the grammar available in the formal definition of GHC Core, given in Tolmach (2000), was LL(1) there were some areas that required alteration to allow either LL(k) or LL(1) parsing. For a full listing of the grammar, please see Tolmach (2000), as only the relevant sections will be reproduced here. A complete listing of the modified grammar can be found in Appendix B.2 on page 21. Please note that the grammar here uses the same rules as the Mondrian Core grammar in Appendix B.1; please refer to the description there.

There was only one case where left recursion[2] in the original grammar; this was for basic types (Table 2.1).

Original Version
bty = *aty* | *bty aty*
Modified Version
bty = { *aty* }+

Table 2.1: The Basic Type (*bty*) Non-Terminal

The removal of the left recursion was simple; by inspection it is possible to see that the original grammar will produce one or more *aty*s, and this can be expressed in a non-left recursive way.

A significant number of changes were required to reduce the rest of the grammar to LL(1) though; changes were made to atomic expressions, expressions, literals and kinds.

The expansion of the *lit* (Table 2.2) was modified, so as to remove the first open bracket from it; this was mainly for convenience in the alterations to the atomic expressions, where it was causing lookahead difficulties. By removing it from the *lit*, this could be avoided. A second change avoided lookahead problems within the *lit* expansion itself. For clarity, the definitions of integers and rational numbers both began the same way; a simple change makes the . { *digit* }+ optional. The common end, "`::` *ty* `)`", was only expressed once, for the sake of brevity.

The atomic expression (Table 2.3) had lookahead problems for two reasons. First, the qualified data constructor and the qualified variables sometimes have the same start, a *mident*, although this was optional for the variable. This was solved by moving the *mident* from the *qdcon* and *qvar* expansions into the *aexp* expansion, and allowing it to be followed by an unqualified data constructor or variable. Note that no modifications were made to the definitions of *qvar*s and *qdcon*s; the grammar just uses unqualified cases in the *aexp* expansion instead. To cover the situation where a *qvar* does not have a qualifying *mident*, the *aexp* was also allowed to expand simply to a *var*. The second problem was caused by *lit*s having a bracket at the start or each expansion, and the nested expression expansion of

[2]Where a non-terminal character's expansion has itself as the first non-terminal, e.g. A -> A B | B

Original Version
lit = ([−] { *digit* }+ :: *ty*) | ([−] { *digit* }+ . { *digit* }+ :: *ty*) | (' *char* ' :: *ty*) | ('' { *char* } '' :: *ty*)
Modified Version
lit = ([−] { *digit* }+ [. { *digit* }+] | ' *char* ' | '' { *char* } '') :: *ty*)

Table 2.2: The Literal (*lit*) Non-Terminal

aexp having brackets also. This was fixed by modifying *lit* as described above, and moving it into the brackets as shown below.

Original Version
aexp = *qvar* | *qdcon* | *lit* | (*expr*)
Modified Version
aexp = *mident* . (*dcon* | *var*) | *var* | ((*lit* | *expr*))

Table 2.3: The Atomic Expression (*aexp*) Non-Terminal

The modification to expressions (Table 2.4) was very minor, so only a partial listing of the expansion is given below. Each of the other alternatives has a unique, terminal character beginning its expansion, causing no lookahead problems.

Original Version
exp = *aexp* | *aexp* { *arg* }+
Modified Version
exp = *aexp* { *arg* }

Table 2.4: The Expression (*exp*) Non-Terminal

The original version shows an atomic expression, by itself, and an application, which is an atomic expression followed by one or more arguments. As far as a parser is concerned, this could be represented as an atomic expression followed by zero or more arguments instead; this is only one choice instead of two.

The final change was made to kinds (Table 2.5). The change was similar to that needed in literals

to combine integers and floating point numbers. Kind can expand to *akind* or and *akind* followed by a "-> *kind*". Again, this appears to have been done for the sake of clarity. The fix for this is simple, combining the two expansions together and making the arrow kind optional.

Original Version
kind = *akind* | *akind* -> *kind*
Modified Version
kind = *akind* [-> *kind*]

Table 2.5: The Kind (*kind*) Non-Terminal

2.3.2 Parser Implementation

The initial version of the parser was developed to parse the grammar as it is described in **?**. It was in developing this parser that the necessity for changes to the grammar became apparent; this was done in conjunction with development, testing the effects of any changes to the grammar by testing the performance of the parser on GHC Core files.

The design of the first successful parser did not take into account any requirements for output; it attempted nothing more than to successfully parse a valid GHC Core file. It consisted of one JavaCC Compilation unit for each non terminal token in the modified grammar in Appendix B.2, with the tokens being based on those used in the MondrianFE.jjt component of the Mondrian compiler.

The Core2MC.jjt file contains the JavaCC grammar specification, along with the definition of tokens and some node renaming and suppression details. The file described in Appendix C.1, page 24, is the Core2MC.jjt file at its current state, not the state described above. The main task of the Core2MC.jjt section is to build a parse tree from the input core file. This parse tree is the output to file using the Visitor pattern, where each node is "visited" and output. The details on how a certain type of node is output is specified in Core2MCDumpVisitor.java (in Appendix C.2, Page 37). Also worth noting is that a Modified version of SimpleNode.java, which is automatically generated by JavaCC and is the parent of all other nodes, was used; this is the same version as used in the Mondrian compiler, so the source is not given here.

2.3.3 Translation Details

Please note that the translation itself and implementation of the translation within the parser are not complete, and that the current version of the tool does not work. Almost none of the translation has actually been implemented; this section will describe portions of the conceptual translation and it is likely that in implementation many of the expected translations here will need modification or reassessment.

Note that, for convenience, Mondrian Core has been abbreviated to MC, and GHC Core has been abbreviated to GC for this section.

The top level compilation units are a good place to start; both begin with a a terminal symbol, "Package" and "%module" for MC and GC respectively, followed by an identifier. This is translated simply by changing the symbol and keeping the identifier the same, or split into separate strings if required. The MC then follows with a list of 0 or more declarations; while this has four possible

GHC Core Literal Type[3]	Mondrian Core Literal Type
Intzh and Int32zh and Int64zh	IntLit[4]
Floatzh and Doublezh	DoubleLit
Charzh	CharLit
Addrzh	StringLit

Table 2.6: GHC Core Literals to Mondrian Core Literals

expansions, only two of the are required for the GC translation. The ImportDecl expansion is unnecessary, as the GC output includes any imported functions already. The SigDecl is optional in Mondrian (Perry 2001), as Mondrian infers the types, so it is ignored here. The other two Decl expansions are ClassDecl, which corresponds to tdefs in the GC grammar, and VarDecl, which corresponds to the GC vdefg.

In the ClassDecl, the first list of names can indicate an "extends" relationship; this is equivalent to the optional tbinds following the data and newtype expansions in tdef; as type information can be inferred, it is easiest to ignore this and leave the list empty. The list of Decl following this name corresponds to the cdefs which can follow in a data expansion. The list should be left empty in the case of a newtype expansion.

The VarDecl expansion contains the name and the expression that it refers to. This is identical to the vdef expansion of vdefg, except that the GC version also has type information; again, this may be ignored. In the case of a recursive vdefg instead, this should be expanded to several VarDecl statements, one for each vdef contained, as the VarDecl does not have a recursive option.

GC literals can be in one of four formats, corresponding to the MC IntLit, DoubleLit, CharLit and StringLit literals. GC does not contain boolean or null literals. The GC literal looks something like "(value::type)", which translates to "type value" in MC. The value can remain the same, but the name of the the type must be translated; while in GC it is possible, according to the grammar, that this could be anything in the definition of the ty non-terminal, in practice this does not occur. This is documented in (Tolmach 2000), Section 4.2, page 12. Instead, the following translation can be used.

Finally, the core of any functional language is the expressions it is capable of. Both languages contain a large number of expressions; we will begin with the simple ones. Translation has not been done for some of the expressions. Several of the expansions for the GC expression can be ignored entirely, as they are of no use to MC. GC "%external" expressions, which indicate external identifiers in a Haskell module, are not needed in MC, which has no mechanism for specifying external identifiers[5] The GC "%note" expansion contains information relevant only to the GHC (Tolmach 2000) and so can also be ignored. A "%coerce" expression is used for type manipulation; as usual, this is unneeded in MC.

GC "%let" expressions define subexpressions within an expression; have a look at the quicksort code example in Section 1.4, page 5 to see a Haskell example of this. The syntax is much the same in both GC and MC, except that MC has both Let and SimpleLet expansions; however, as SimpleLet is simply the non-recursive binding, all GC let expressions can be translated into the Let, rather than the SimpleLet. The "%in" keyword in the GC version is removed, and the relevant translation is applied to convert GC vdefgs into MC Decls.

GC atomic expressions correspond to the MC Lit Lit for qvar and qdcon expansions and Lit Lit for the lit expansion. I am uncertain how to translate the GC nested expression into MC.

[5]How these are dealt with is not important for the translation.

The “%case” expression is the most complicated of the GC expressions and my translation here is definitely unreliable. It corresponds to the MC HSwitch expansion, which is intended as a target for GHC Core translations, where the “%case” becomes and “HSwitch”, the vdefg is translated into an expression, how I’m unsure, and each alt is translated into a Pattern, again, I’m uncertain how.

2.3.4 Still to be Done

Implementation and testing of the translations described in the previous section. This will involve modifications both to Core2MCDumpVisitor.java, where the output for each node will be modified, and to Core2MC.jjt, where more nodes will have to be suppressed (note that the NODE_DEFAULT_VOID option in the options section forces this as a default) and the way in which many nodes are named will have to be changed to include important information from their children.

Chapter 3

Users Guide

3.1 Guide to Installation

3.1.1 Tools Required

A variety of tools are required in order to compile and run Core2MC. The following table lists the tools required and where they can be obtained from; installation details can be found with each tool. The locations given are valid at November 2002. Note that to make a full installation, the .NET Framework must be installed under Windows NT/2000/XP; Core2MC, the tool developed in this project, will run under any system with a Java Virtual Machine, but Windows and .NET are required to run the Mondrian and C# compilers. If a partial installation in being performed on a Unix/Linux, then the `make`, `tar` and `gzip` tools will be required; under a Windows OS, the WinZip tool will be required.

If a user does not wish to go through the compilation process, a compiled Core2MC class file can also be downloaded from http://www.treshna.com/ liam/Core2MC. This can then be run on the Java Platform without modification.

Tool	Location
Java 2 SDK	http://java.sun.com
JavaCC	http://www.webgain.com/products/java_cc
Glasgow Haskell Compiler 5.02	http://www.haskell.org/ghc
Mondrian	http://www.mondrian-script.org
.NET Framework	http://msdn.microsoft.com/netframework/downloads/howtoget.asp
Core2MC	http://www.treshna.com/ liam/Core2MC
Possibly	
`make`	http://www.gnu.org/software/make/make.html
`tar`	http://www.gnu.org/software/tar
`gzip`	http://www.gzip.org/#exe
WinZip	http://www.winzip.com

Table 3.1: Tools Required for Installation

3.1.2 Installing Core2MC Manually

Installation details here are given for Linux/Unix systems, as this is the platform Core2MC was developed under. A rough outline for installation on other platforms is also given, as a fully functional system will have to be running under Windows^{TM} NT/2000/XP until .NET Frameworks are released for other platforms. This guide is for those who wish to go through the whole process themselves; this is likely to be the case if you are modifying or extending the Core2MC. Otherwise, simply download a precompiled class file from http://www.treshna.com/ liam/Core2MC

Linux/Unix installation

This assumes that the system is fitted with the `make`, `tar` and `gzip/gunzip` applications, which are standard on this platform. See Section 3.1.1 if your system does not have `make` installed.

In the directory containing Core2MC.tar.gz type

```
tar xzf Core2MC.tar.gz
cd Core2MC
make
```

This will compile the Core2MC class file.

Non Linux OS

Unzip Core2MC.zip using WinZip tool. All required files will be placed in a Core2MC folder in the directory that Core2MC.zip was unzipped to. Run the following commands.

```
jjtree Core2MC.jjt
javacc Core2MC.jj
javac Core2MC.java
```

This will compile the Core2MC class file.

3.2 Using Core2MC

Compile the Haskell source file using the GHC with the `-fext-core` parameter. This will cause it to output GHC Core into a `.hcr` file, named the same as the source file. To translate the GHC Core file, run `java Core2MC <core file name>`. The output from this will be placed in a file `.mc` file, which can then be compiled by the Mondrian compiler. See the documentation available with the Mondrian installation on how to do this.

Note that, as the translation *is not complete*, that any `.mc` files generated will not successfully compile.

Appendix A

C Quicksort

```
void quicksort( int array[], int first, int last )
{
 int piv_index;
 int partition( int *, int, int );

 if ( first < last )
 {
    piv_index = partition( array, first, last );
    quicksort( array, first, piv_index - 1 );
    quicksort( array, piv_index, last );
 }
}

int partition( int array[], int first, int last )
{
 int pivot, temp;

 pivot = array[ (first + last) / 2 ];

 while ( first <= last )
 {
   while ( array[first] < pivot )
      ++first;

   while ( array[last] > pivot )
      --last;

   if ( first <= last )
   {
     temp = array[first];
     array[first] = array[last];
     array[last] = temp;
```

```
      ++first;
      --last;
    }
  }
  return (first);
}
```

Appendix B

Grammars for Mondrian Core and GHC Core

B.1 Grammar for Mondrian Core

This grammar was derived from Mondrian.hs, part of the Mondrian package available at http://www.mondrian-script.org.

The following rules apply to this grammar

- Non-terminal symbols appear like *this*
- Terminal symbols appear like `this`
- Symbols enclosed within square brackets are optional
- { pattern } is 0 or more occurences
- Note the difference between (and `(` characters; the second one is a non-terminal, the first is symbol of the grammar.
- $\text{pattern}_1 \mid \text{pattern}_2$ is a choice

CompilationUnit = `Package` *Name* `[` { Decl } `]`
Decl = *ClassDecl Name* `[` { *Name* } `]` `[` { Decl } `]` | *ImportDecl Name* | *VarDecl Name Expr* | *SigDecl Name Expr*
Expr = *Lit Lit* | `Var` *Name* | `Switch` *Expr* `[` (*Pattern*, *Expr*)`]` | `HSwitch` *Expr Name* `[` { `(` Pattern, *Expr*`)` } `]` | `Iff` *Expr Expr Expr* | `HIff` *Expr Name Expr Expr* | `Let` `[` { Decl } `]` *Expr* | `SimpleLet` [Decl] *Expr* | `Lambda` `[` { *Expr* } `]` *Expr* | `App` *Expr Expr* | `New` *Name* `[` { Decl } `]` | `Box` *Name Name Expr* | `Do` `[` { Statement } `]` | `CustomDo` `[` { Statement } `]` | `TryExpr` *Expr* `[` { `(` *Name*, *Name*, *Expr* `)` } `]` (Maybe *Expr*) | `Throw` *Expr* | `Sync` *Name Expr* | `Tuple` `[` { *Expr* } `]` | `Invoke` *Name* `[` { *Name* } `]` | `InvokeStatic` *Name* `[` { *Name* } `]` | `Create` *Name* `[` { *Name* } `]` | `Set` *Name* | `SetStatic` *Name* | `Get` *Name* | `GetStatic` *Name*

Table B.1: Mondrian Core Grammar pt 1

Statement = *Expr Expr* | `Generator` *Name Expr*
Pattern = *Pattern Name* `[` { Decl } `]` | `UnBox` *Name Name Name* | `Value` *Lit* | `Default`
Lit = `IntLit` Int | `DoubleLit` Double | `CharLit` Char | `StringLit` String | `BoolLit` Bool | `NullLit`
Name = `[` String `]`

Table B.2: Mondrian Core Grammar pt 2

B.2 Modified Grammar for GHC Core

The following rules apply to this grammar

- Non-terminal symbols appear like *this*
- Terminal symbols appear like `this`
- Symbols enclosed within square brackets are optional
- { pattern } is 0 or more occurences
- Note the difference between (and `(` characters; the second one is a non-terminal, the first is symbol of the grammar.
- pattern_1 | pattern_2 is a choice

Module = `%module` *mident* { *tdef* `;` } { [`%local`] *vdefg* `;` }
tdef = `%data` *qtycon* { *tbind* } = { *cdef* { `;` *cdef* } } | `%newtype` *qtycon* { *tbind* } [`=` *ty*]
cdef = *qdcon* { `@` *tbind* } { *aty* }
vdefg = `%rec` { *vdef* { `;` *vdef* } } | *vdef*
vdef = *qvar* `::` *ty* = *exp*
aexp = *mident* `.` (*dcon* | *var*) | *var* | ((*lit* | *expr*))
exp = *aexp* { *arg* } | `note: put backslash here, somehow!` { *binder* }+ `->` *exp* | `%let` *vdefg* `%in` *exp* | `%case` *exp* `%of` *vbind* { *alt* { `;` *alt* } } | `%coerce` *aty exp* | `%note` ‘‘ { *char* } ’’*exp* | `%external` “ { *char* } “ *aty*
arg = `@` *aty* | *aexp*
alt = *qdcon* { `@` *tbind* } { *vbind* } `->` *exp* | *lit* `->` *exp* | `%_` `->` *exp*
binder = *tyvar* | (*tyvar* `::` *kind*)
vbind = (*var* `::` *ty*)
lit = ([`-`] { *digit* }+ [`.` { *digit* }+] | ’ *char* ’ | ’’ { *char* } ’’) `::` *ty*)
char = Any ASCII character in range 0x20-0x7E except 0x22,0x27,0x5c | `backslash x` *hex hex*
hex = `0`|...|`9`|`a`|...|`f`
aty = *tyvar* | *qtycon* | (*tt*)
bty = { *aty* }+
ty = *bty* | `%forall` { *tbind* }+ `.` *ty* | *bty* `->`
akind = `*` | `#` | `?` | (*kind*)
kind = *akind* [`->` *kind*]

Table B.3: GHC Core Grammar pt 1

mident = *uname*
tycon = *uname*
qtycon = *mident* `.` *tycon*
tyvar = *lname*
dcon = *uname*
qdcon = *mident* `.` *dcon*
var = *lname*
qvar = [*mident* `.`] *var*
lname = *lower* { *namechar* }
unmae = *upper* { *namechar* }
namechar = *lower* | *upper* | *digit* | ‘
lower = `a` | ... | `z` | _
upper = `A` | ... | `Z`
digit = `0` | `9`

Table B.4: GHC Core Grammar pt 2

Appendix C

Project Source Code

C.1 Core2MC.jjt

```
/**
 *
 * Core2MC - A tool to convert GHC Core to Mondrian Core
 *
 * Parsing information is contained within this file.
 * Core2MCDumpVisitor.java contains most of the translation functionality.
 *
 * Author: Liam O'Boyle
 * Date: August - November 2002
 *
 * Copyright (C) 2002, Liam O'Boyle
 *
 *
 *=================================================================================
 * Parts of this grammar have been taken from the MondrianFE grammar produced
 * by Nigel Perry.  The following disclaimer was included in this grammar, and
 * has been included here for completeness.
 *=================================================================================
 *
 * Parts of this grammar are taken from a sample Java grammar from Sun:
 *
 * Copyright (C) 1996, 1997 Sun Microsystems Inc.
 *
 * Use of this file and the system it is part of is constrained by the
 * file COPYRIGHT in the root directory of this system.  You may, however,
 * make any modifications you wish to this file.
 *
 * Java files generated by running JavaCC on this file (or modified versions
 * of this file) may be used in exactly the same manner as Java files
 * generated from any grammar developed by you.
```

```
 *
 * Author: Sriram Sankar
 * Date: 3/5/97
 *
 */

options {
  JAVA_UNICODE_ESCAPE = true;
  MULTI               = true;
  FORCE_LA_CHECK      = true;
  VISITOR             = true;
  NODE_DEFAULT_VOID   = true;
}

PARSER_BEGIN(Core2MC)

import java.awt.*;
import java.io.*;

/* Most of this class is identical to that in MondrianFE.jjt */
public class Core2MC {

// global error flag
static boolean HadError = false;

// dummy AsterixIO, sort of...
static class Console
{ static PrintWriter out = new PrintWriter(System.out, true);
static Frame getFrame() { return new Frame(); }
}
static class TextFileOutputStream
{ static OutputStream open(String outFile) throws java.io.IOException
             { return new FileOutputStream(outFile); }
}

public static void main(String args[])
{ try
{ parseFile(args);
}
catch (TokenMgrError lex)
{   Console.out.println(lex);
Console.out.println("Errors, no output produced");
}
catch (ParseException parse)
{ Console.out.println(parse);
Console.out.println("Errors, no output produced");
}
```

```
catch (Exception e)
{ Console.out.println(e);
}
}

public static void parseFile(String args[]) throws TokenMgrError,
                                        ParseException, Exception
    {   Core2MC parser;
String inFile, outFile;

Console.out.println("Core2MC v 0.01");

if (args.length == 0)
{ FileDialog fd = new FileDialog(Console.getFrame());
fd.show();
if( fd.getFile() == null ) return;
inFile = fd.getDirectory() + fd.getFile();
}
else

if (args.length == 1)
{ inFile = args[0];
}
else
{ Console.out.println("Usage: java Core2MC <inputfile>");
return;
}

try
{ parser = new Core2MC(new FileInputStream(inFile));
}
catch (Exception e)
{ Console.out.println("Opening " + inFile + ": " + e);
return;
}

try
{ outFile = inFile.substring(0, inFile.lastIndexOf('.')) + ".mc";
PrintWriter pw = new PrintWriter(TextFileOutputStream.open(outFile));

ASTModule n = parser.Module();
if(HadError)
{ Console.out.println("Errors, no output produced");
}
else
{ Console.out.println("GHC core file parsed successfully.");
n.dump(""); // debug
```

```
Core2MCVisitor jpv = new Core2MCDumpVisitor(pw);
n.jjtAccept(jpv, null);
Console.out.println("Mondrian Core (.mc) output in " + outFile);
}
pw.close();
}
catch (TokenMgrError lex)
{ Console.out.println(lex);
Console.out.println("Errors, no output produced");
}
catch (ParseException parse)
{ Console.out.println(parse);
Console.out.println("Errors, no output produced");
}
catch (Exception e)
{ Console.out.println(e);
}
}

}

PARSER_END(Core2MC)

/* WHITE SPACE */

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

/* Haskell identifiers beginning with upper and lowercase characters */

TOKEN :
{
   < UNAME: <UPPER> ( <NAMECHAR> )* >
|  < LNAME: <LOWER> ( <NAMECHAR> )* >
|  < #NAMECHAR: ( <UPPER> | <LOWER> | <NAMEDIGIT> | "'" ) >
|  < #UPPER: [ "A"-"Z" ] >
|  < #LOWER: [ "a"-"z", "_" ] >
|  < #NAMEDIGIT: [ "0"-"9" ] >
}

/* LITERALS */
```

```
/* Adapted from MondrianFE.jjt
 * Some of this is still more than needed
 */

TOKEN :
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL>
      | <HEX_LITERAL>
      | <OCTAL_LITERAL>
  >
|
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
|
  < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
|
  < #OCTAL_LITERAL: "0" (["0"-"7"])* >
|
  < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
  >
|
  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
| /* Added \x characters for char and string literals. */
  < CHARACTER_LITERAL:
      "'"
      (   (~["'","\\","\n","\r"])
        | ("\\"
            ( ["\\","'","\""]
            | "x" ["0"-"9", "a"-"f"] ["0"-"9", "a"-"f"]
            )
          )
      )
      "'"
  >
|
  < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["x","n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
```

```
          )
      )*
      "\""
  >
}

/* IDENTIFIERS */
/* These are used instead of the hexidecimal codes given in the
 * core grammar; testing has indicated that these will suffice,
 * and there were difficulties using a jjt file that had a \x
 * in it.
 */
TOKEN :
{
  < LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
|
  < DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
```

```
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
  >
}

/* Reserved Words and Symbols */

TOKEN:
{
   < MODULE  : "%module"   >
|  < DATA    : "%data"     >
|  < NEWTYPE : "%newtype"  >
|  < REC     : "%rec"      >
|  < LET     : "%let"      >
|  < CASE    : "%case"     >
|  < COERCE  : "%coerce"   >
|  < NOTE    : "%note"     >
|  < EXTERNAL: "%external" >
|  < LOCAL   : "%local"    >
|  < IN      : "%in"       >
|  < OF      : "%of"       >
|  < P_UNDER : "%_"        >
|  < FORALL  : "%forall"   >
}

TOKEN:
{
   < SEMICOLON : ";"  >
|  < EQUALS    : "="  >
|  < AT        : "@"  >
|  < DCOLON    : "::" >
|  < LBRACE    : "(" >
|  < RBRACE    : ")" >
|  < LCBRACE   : "{" >
|  < RCBRACE   : "}" >
|  < BACKSLASH : "\\" >
|  < DASHARROW : "->" >
|  < SPEECHMARK: "\"" >
|  < DOT       : "." >
|  < QUOTEMARK : "'" >
|  < STAR      : "*" >
|  < HASH      : "#" >
|  < QMARK     : "?" >
}

/* Module Definition */
```

```
ASTModule Module() #Module :
{
 Token t;
}
{
   <MODULE> MIdent() ( TDef() <SEMICOLON> )* ( [ <LOCAL> ] VDefG() <SEMICOLON> )*
   {
     return jjtThis;
   }
}


/* Type Definition */

void TDef() #TDef : {}
{
   <DATA> QTyCon() ( TBind() )* <EQUALS> <LCBRACE> CDef() ( <SEMICOLON> CDef() )* "}"
|  <NEWTYPE> QTyCon() ( TBind() )* [ <EQUALS> Ty() ]
}


/* Constraint Definition */

void CDef() #CDef : {}
{
   QDCon() ( <AT> TBind() )* ( ATy() )*
}


/* Value Definition */

/* There was initially some confusion here - the grammar uses "{ }" symbols
as "[ ]" unfortunately, it appears that the core language includes { symbols
as well, making it difficult to tell which appear in the .hcr files
and which denote grammatical meaning - so it appears that %rec tags are
followed by { symbols - here is the old (commented) version and the new
operational one.

After contacting Andrew Tolmach, it becomes clear that these are meant
to be <LCBRACE> rather than gramatical symbols.  Different font in the file.
*/

void VDefG() #VDefG : {}
{
   <REC> <LCBRACE> VDef() ( <SEMICOLON> VDef() )* <RCBRACE>
|  VDef()
```

```
}

void VDef() #VDef : {}
{
   QVar() <DCOLON> Ty() <EQUALS> Exp()
}

/* Atomic Expression */

void AExp() #AExp : {}
{
( MIdent() <DOT> ( DCon() | Var() ) | Var() )
| <LBRACE> ( Literal() |  Exp() <RBRACE> )
}


/* Expression */

void Exp() #Exp : {}
{
   AExp() ( Arg() )*
|  <BACKSLASH> ( Binder() )+ <DASHARROW> Exp()
|  <LET> VDefG() <IN> Exp()
|  <CASE> Exp() <OF> VBind() <LCBRACE> Alt() ( <SEMICOLON> Alt() )* <RCBRACE>
|  <COERCE> ATy() Exp()
|  <NOTE> <STRING_LITERAL> Exp()
|  <EXTERNAL> <STRING_LITERAL> ATy()
}


/* Argument */

void Arg() #Arg : {}
{
   <AT> ATy()
|  AExp()
}


/* Case alt. */

void Alt() #Alt : {}
{
   QDCon() ( <AT> TBind() )* ( VBind() )* <DASHARROW> Exp()
|  <LBRACE> Literal() <DASHARROW> Exp()
|  <P_UNDER> <DASHARROW> Exp()
}
```

```
/* Binder */

void Binder() #Binder : {}
{
   <AT> TBind()
|  VBind()
}




/* Type Binder */

void TBind() #TBind : {}
{
   TyVar()
|  <LBRACE> TyVar() <DCOLON> Kind() <RBRACE>
}


/* Value Binder */

void VBind() #VBind : {}
{
   <LBRACE> Var() <DCOLON> Ty() <RBRACE>
}

/* Literal */

void Literal() #Literal : {}
{
    ( <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> | <STRING_LITERAL>
      | <CHARACTER_LITERAL>
    ) <DCOLON> Ty() <RBRACE>
}

/* Character */
/* Note that this uses the Mondrian definitions here, as JavaCC did not want
   to allow any \x characters to appear within the file.

void Char() #Char : {}
{
   <LETTER>
|  <DIGIT>
}
```

```
/* Atomic Type */

void ATy() #ATy : {}
{
   TyVar()
|  QTyCon()
|  <LBRACE> Ty() <RBRACE>
}


/* Basic Type */

void BTy() : {}
{
   ( ATy() )+
}


/* Type */

void Ty() #Ty : {}
{
   BTy()
|  <FORALL> ( TBind() )+ <DOT> Ty()
}


/* Atomic Kind */

void AKind() #AKind : {}
{
   <STAR>
|  <HASH>
|  <QMARK>
|  <LBRACE> Kind() <RBRACE>
}


/* Kind */

void Kind() #Kind : {}
{
   AKind() [ <DASHARROW> Kind() ]
}
```

```
/* Identifier */

Token MIdent() #String :
{
 Token t;
}
{
   t=<UNAME>
   {
    jjtThis.setName(t.image);
    return t; // passed back for qualified names
   }
}

Token TyCon() #String :
{
 Token t;
}
{
   t=<UNAME>
   {
    jjtThis.setName(t.image);
    return t;
   }
}

void QTyCon() #String :
{
 Token m, t;
}
{
   m=MIdent() <DOT> t=TyCon()
   {
    jjtThis.setName(m.image+"\", \".\", \""+t.image);
   } // note dodgy hack to create a list of strings for MC format
}

void TyVar() #String :
{
 Token t;
}
{
   t=<LNAME>
   {
    jjtThis.setName(t.image);
   }
```

```
}

Token DCon() #String :
{
 Token t;
}
{
   t=<UNAME>
   {
    jjtThis.setName(t.image);
    return t;
   }
}

void QDCon() #String :
{
 Token m, d;
}
{
   m=MIdent() <DOT> d=DCon()
   {
    jjtThis.setName(m.image+"\", \".\", \""+d.image);
   }
}

Token Var() #String :
{
 Token t;
}
{
   t=<LNAME>
   {
    jjtThis.setName(t.image);
    return t;
   }
}

void QVar() #String :
{
 Token m = null, v;
}
{
   [ m=MIdent() <DOT> ] v=Var()
   {
    if(m==null)
     jjtThis.setName(v.image);
    else
```

```
    jjtThis.setName(m.image+"\", \".\", \""+v.image);
   }
}
```

C.2 Core2MCDumpVisitor.java

```
import java.io.*;

public class Core2MCDumpVisitor implements Core2MCVisitor
{ private PrintWriter output;
private int indent = 0;
// We use the "data" argument to indicate whether the first line
// of output should be preceeded by the current indent and the
// last line followed by a newline. This is used for instance for
// visitList().
static Object INDENT = null;
static Object NO_INDENT = new Object();

public Core2MCDumpVisitor()
{ output = new PrintWriter(System.out); // Console.out;
}

public Core2MCDumpVisitor(PrintWriter outp)
{ output = outp;
}

private String indentString()
{ StringBuffer sb = new StringBuffer();
for (int i = 0; i < indent; ++i)
{ sb.append("  ");
}
return sb.toString();
}

private void need(SimpleNode node, int n)
{ if(node.jjtGetNumChildren() != n)
throw new Error("invalid number of children in node");
}

public Object visit(SimpleNode node, Object data)
{ if(data == INDENT) output.print(indentString());
if((data == INDENT) && (node.jjtGetNumChildren() > 0))
{ output.println(" ( " + node);
indent += 2;
data = node.childrenAccept(this, INDENT);
indent -= 2;
```

```
output.println(indentString() + " )");
}
else
{ output.println(node);
indent++;
data = node.childrenAccept(this, INDENT);
indent--;
}
return data;
}

/* public Object visitMaybe(SimpleNode node, Object data)
{ if(data == INDENT) output.print(indentString());
if(node.jjtGetNumChildren() > 0)
{ need(node, 1);
output.println("(Just");
indent++;
output.print(indentString() + "( ");
indent++;
data = node.childrenAccept(this, NO_INDENT);
indent--;
output.println(indentString() + ")");
indent--;
output.println(indentString() + ")");
}
else
{ output.println("Nothing");
}
return data;
}*/

private Object visitList(SimpleNode node, Object data)
{ return visitSeq(node, data, "[", "]");
}

private Object visitTuple(SimpleNode node, Object data)
{ return visitSeq(node, data, "(", ")");
}

private Object visitSeq(SimpleNode node, Object data, String open, String close)
{ int count = node.jjtGetNumChildren();
String indentStr = indentString();

if(count > 0)
{ if(data == INDENT) output.print(indentStr);
output.print(open + " ");
indent++;
```

```
node.jjtGetChild(0).jjtAccept(this, NO_INDENT);
for(int ix = 1; ix < count; ix++)
{ output.print(indentStr + ", ");
node.jjtGetChild(ix).jjtAccept(this, NO_INDENT);
}
// output.println("");
indent--;
output.println(indentStr + close);
}
else
output.println(indentStr + open + close);
return data;
}

private Object visitTokenList(SimpleNode node, Object data)
{ if(data == INDENT) output.print(indentString());
printTokenList(node);
output.println("");
return data;
}

private void printTokenList(SimpleNode node)
{ int count = node.jjtGetNumChildren();

if(count > 0)
{ output.print("[" + node.jjtGetChild(0));
for(int ix = 1; ix < count; ix++)
{ output.print(", " + node.jjtGetChild(ix));
}
output.print("]");
}
else
output.print("[]");
}

private Object visitName(SimpleNode node, Object data, boolean quote)
{ if(data == INDENT) output.print(indentString());
printName(node, quote);
output.println("");
return data;
}

private void printName(SimpleNode node, boolean quote)
{ if(quote) output.print("\"");
output.print(node.getName());
if(quote) output.print("\"");
}
```

```
private Object visitNameList(SimpleNode node, Object data, boolean quote)
{ if(data == INDENT) output.print(indentString());
printNameList(node, quote);
output.println("");
return data;
}

private void printNameList(SimpleNode node, boolean quote)
{ int count = node.jjtGetNumChildren();

if(count > 0)
{ output.print("[");
printName((SimpleNode)node.jjtGetChild(0), quote);
for(int ix = 1; ix < count; ix++)
{ output.print(", ");
printName((SimpleNode)node.jjtGetChild(ix), quote);
}
output.print("]");
}
else
output.print("[]");
}

   public Object visit(ASTAExp node, Object data)
     {
return visit((SimpleNode) node, data);
     }

   public Object visit(ASTAKind node, Object data)
     {
return visit((SimpleNode) node, data);
     }

   public Object visit(ASTATy node, Object data)
     {
return visit((SimpleNode) node, data);
     }

   public Object visit(ASTAlt node, Object data)
     {
return visit((SimpleNode) node, data);
     }

   public Object visit(ASTArg node, Object data)
     {
return visit((SimpleNode) node, data);
```

```
     }

/* now void in .jjt, unneeded here
   public Object visit(ASTBTy node, Object data)
     {
return visit((SimpleNode) node, data);
     }*/

   public Object visit(ASTBinder node, Object data)
     {
return visit((SimpleNode) node, data);
     }

   public Object visit(ASTCDef node, Object data)
     {
return visit((SimpleNode) node, data);
     }

   public Object visit(ASTChar node, Object data)
     {
return visit((SimpleNode) node, data);
     }

   public Object visit(ASTExp node, Object data)
     {
return visit((SimpleNode) node, data);
     }

   public Object visit(ASTKind node, Object data)
     {
return visit((SimpleNode) node, data);
     }

   public Object visit(ASTLiteral node, Object data)
     {
return visit((SimpleNode) node, data);
     }

   /* tidy the indentation technique here */
   public Object visit(ASTModule node, Object data)
     {
if(data == INDENT) output.print(indentString());
if((data == INDENT) && (node.jjtGetNumChildren() > 0))
  {
     output.println("Package");
     indent += 1;
     data = node.childrenAccept(this, INDENT);
```

```
      indent -= 1;
      output.println();
    }
  else
    {
      output.println(node);
      indent++;
      data = node.childrenAccept(this, INDENT);
      indent--;
    }
  return data;
  /* return visit((SimpleNode) node, data);*/
      }

     public Object visit(ASTTBind node, Object data)
      {
  return visit((SimpleNode) node, data);
      }

     public Object visit(ASTTDef node, Object data)
      {
  return visit((SimpleNode) node, data);
      }

     public Object visit(ASTTy node, Object data)
      {
  return visit((SimpleNode) node, data);
      }

     public Object visit(ASTVBind node, Object data)
      {
  return visit((SimpleNode) node, data);
      }

     public Object visit(ASTVDef node, Object data)
      {
  return visit((SimpleNode) node, data);
      }

     public Object visit(ASTVDefG node, Object data)
      {
  return visit((SimpleNode) node, data);
      }

     public Object visit(ASTString node, Object data)
      {
  if(data == INDENT) output.print(indentString());
```

```
output.println(" [\"" + node.getName() + "\"]");
return data;
     }

}
```

Bibliography

Gough, J. (2002), ‘Stacking them up: A comparison of virtual machines’.
URL: *http://docs.msdnaa.net/ark_new/Webfiles/whitepapers.htm#virtual*

Jones, S. P., Hghes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinza, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C. & Wadler, P. (1999), Haskell 98: A non-strict, purely functional language, Technical report.
URL: *http://www.haskell.org/onlinereport/*

Kennedy, A. & Syme, D. (2002), ‘Design and implementation of generics for the .net common language runtime’.
URL: *http://docs.msdnaa.net/ark_new/Webfiles/whitepapers.htm#generics*

Levy, S. (2002), So what’s this .NET thing?, Technical report, www.microsoft.com, http://www.microsoft.com/technet/columns/ednote/en0102p1.asp.

Meijer, E. & Gough, J. (2002), ‘Technical overview of the common language runtime’.
URL: *http://docs.msdnaa.net/ark_new/Webfiles/whitepapers.htm#CLR*

Perry, N. (2001), Introduction to mondrian, Technical report, www.mondrian-script.org.
URL: *http://www.mondrian-script.org/mondrian/doc/mondrian.html*

Tolmach, A. (2000), An external representation for the ghc core language (draft for ghc 5.02), Technical report.
URL: *http://haskell.cs.yale.edu/ghc/docs/papers/core.ps.gz*