MAG version 2.11: User Manual

Ganesh Sittampalam and Oege de Moor

September 3, 2003

Contents

1 About this document

This manual is intended to provide up-to-date usage instructions and a syntax description of the syntax for MAG version 2.11, but is not intended to serve as a tutorial in writing MAG programs. See section 9 for a description of other documents that may help with this. Some familiarity with Haskell is recommended before using MAG.

MAG is a joint work by Ganesh Sittampalam and Oege de Moor, but all blame for the brevity of this manual should be directed at Ganesh, who will try to respond to requests for additional help: `ganesh@comlab.ox.ac.uk`.

2 Introduction

MAG is a program transformation system for a small functional language similar to Haskell. The main features it provides are:

- The ability to write *active source*: source code that actively takes part in the compilation process by providing instructions to the compiler on how to optimise it.
- An implementation of *higher-order conditional rewriting* which makes use of certain *higher-order matching* algorithms developed by the authors.

In particular, MAG is able to automate *fusion* (sometimes also known as *promotion*) transformations, which provide a framework for safely transforming certain kinds of recursive programs. For example, the well-known *fast reverse* transformation can be expressed by the following MAG source code:

```
{- reverse.p -}

reverse [] = [];
reverse (x:xs) = reverse xs ++ [x];

TRANSFORM fastrev
REDEFINE reverse xs = fastreverse xs []
SPECIFYING fastreverse xs ys = reverse (foldr (:) [] xs) ++ ys
USING
      DEFINITION reverse,(++);

      catassoc: (xs ++ ys) ++ zs = xs ++ (ys ++ zs);

      fusion: f (foldr plusl e xs) = foldr crossl e' xs,
                if {f e = e';
                    \ x y -> f (plusl x y)
                       = \ x y -> crossl x (f y)}
END;

foldr f e [] = e;
foldr f e (x:xs) = f x (foldr f e xs);
```

Essentially, this source file provides the slow program for `reverse` together with instructions for mechanising the fast reverse optimisation. The syntax used is explained in section 6.

3 Compiling

Binary distributions of MAG are available for Linux/x86, Cygwin/x86, FreeBSD/x86, Solaris/Sparc. If you use one of these distributions, you can ignore the rest of this section.

MAG is entirely Haskell 98 code. The recommended environment for using MAG is GHC, for performance reasons, but it can also be used under Hugs. It should also build with NHC, but this has not been tested recently. Whichever environment is used, a reasonably recent version of GHC is required for dependency calculation.

Most recently, MAG has been tested with the February 2001 release of Hugs, and the following versions of GHC:

- Linux/i386 : 5.04.3, 6.0.1
- Cygwin/i686 : 5.02

4 Installing

If MAG is being used by a single user, it is recommended to run it from the directory it is untarred into. If being installed by the system administrator for multiple users, the `mag` script and `magdcalc` binaries from the `bin` directory should be placed in `/usr/local/bin` or `/usr/bin`, and the `.p` files from the `examples` directory should be placed in `/usr/local/share/mag` or `/usr/share/mag`. Appropriate files from the `doc` directory should go into whatever directory is mandated by local policy – often `/usr/local/doc/mag-2.11` or `/usr/share/doc/mag-2.11`.

5 Quick usage guide

The above example `reverse.p` can be run by changing to the `examples` directory and running the following command (change the paths appropriately if the installation is system-wide):

```
../bin/mag --program reverse --transform fastrev showprogram
```

6 Syntax of MAG programs

A MAG program consists of a list of *definitions* and *transformations*. A definition can be a (possibly partial) function declaration, such as:

```
reverse [] = [];
```

It can also be a data declaration, for example:

```
data Tree a = Leaf a | Node (Tree a) (Tree a);
```

There is no "layout" rule, so all definitions and transformations must be terminated with a semicolon.

The left-hand side of a function declaration is a name followed by a list of arguments, some or all of which may be patterns (such as `[]`). The right-hand side is an expression built up from variables, constants, λ-abstractions, applications and `let` expressions, which have the same syntax as in Haskell.

A transformation is introduced by the keyword `TRANSFORM`, together with a name for the transformation. This name can optionally be followed by a colon together with a comma-separated list of transformations on which this transformation depends.

Then next element of a transformation is the keyword `REDEFINE`, followed by a semicolon-separated list of definitions, and then the keyword `SPECIFYING` and another semicolon-separated list of definitions. The final part is the keyword `USING` and a semicolon-separated list of *rewrite rules*, and the transformation is terminated by the keyword `END` (and a final semicolon).

A rewrite rule is either the keyword `DEFINITION` followed by a comma-separated list of function names, or a name, a colon, an expression denoting the left-hand side of the rule, an equals sign, and an expression denoting the right-hand side. In the latter case it can optionally be followed by a comma, the keyword `if`, and a comma-separated list of *side-conditions*, enclosed in curly braces. Each side-condition consists of two expressions separated by an equals sign.

See section 2 for an example of a transformation.

7 Using MAG

To run MAG in interactive mode, execute the `mag` script from the `bin` directory. You will see the following banner:

```
MAG version 2.11 (built on Wed Sep 3 13:51:37 BST 2003)
```

This will be followed by the MAG command prompt:

```
Mag>
```

A number of commands can be executed from this prompt. After each command completes, MAG returns to this prompt. Some commands can also be specified from the command-line, in one of two ways:

- Some commands have command-line versions. Any number of these commands can be specified, and they will be executed in the order they occur.
- The final element of the command line can be any MAG command except `quit`. If a command is specified in this way MAG will execute any commands specified by `--` options, then this command, and then exit without offering a command prompt.

`loadprogram`

- Syntax: `loadprogram` *filename*
- Command line option: `--program` *filename*

Loads, parses and type-checks *filename*`.p`.

`applytransform`

- Syntax: `applytransform` *transformation*
- Command line option: `--transform` *transformation*

Applies *transformation* to the currently loaded program, and leaves the transformed program loaded.

`showprogram`

- Syntax: `showprogram`

Prints out the currently loaded program.

`setwidth`

- Syntax: `setwidth` *width*
- Command line option: `--width` *width*

Set the output width (in characters) for calculations and programs. If you set this too low some output may be replaced by `*`s if it cannot be formatted to fit. The initial value for this setting is 80.

`setpath`

- Syntax: `setpath` *path*
- Command line option: `--path` *path*

Set the base path from which programs, theory files and scripts are loaded. The path should be enclosed in quotes. The initial value is the empty string.

`runscript`

- Syntax: `runscript` *filename*
- Command line option: `--run` *filename*

Load and run the specified script. The filename should be enclosed in quotes.

`skip`

- Syntax: `skip`

Do nothing. Intended for scripts, etc.

`quit`

- Syntax: `quit`

Exit the interactive loop or the current script. An EOF will have the same effect, so there is no need to end scripts with this command, and `^D` can be used to exit the interactive loop.

8 Known Bugs

- Many error conditions cause MAG to exit, often with a cryptic error message, instead of catching it and returning to the command prompt.
- The `mag` script in the `bin` directory does not work when accessing a cygwin machine via ssh.

9 Other information

The most up-to-date account of the ideas underlying MAG can be found in [5]. A more detailed account of the underlying theory can be found in [3]. Other descriptions, which have in the main been superseded by these documents, can be found in [1, 2, 4].

References

[1] O. de Moor and G. Sittampalam. Generic program transformation. In *Third International Summer School on Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 116–149. Springer-Verlag, 1998. Available from URL: `http://web.comlab.ox.ac.uk/oucl/work/ganesh.sittampalam/`. 6

[2] O. de Moor and G. Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269:135–162, 2001. Available from URL: `http://www.comlab.ox.ac.uk/oucl/work/oege.demoor/pubs.htm`. 6

[3] G. Sittampalam. *Higher-order matching for program transformation.* PhD thesis, University of Oxford, 2001. Available from URL: `http://web.comlab.ox.ac.uk/oucl/work/ganesh.sittampalam/`. 6

[4] Ganesh Sittampalam and Oege de Moor. Higher-order pattern matching for automatically applying fusion transformations. In O. Danvy and A. Filinski, editors, *Proceedings of 2nd Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 198–217. Springer-Verlag, 2001. Available from URL: `http://www.comlab.ox.ac.uk/oucl/work/oege.demoor/pubs.htm`. 6

[5] Ganesh Sittampalam and Oege de Moor. Mechanising fusion. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, chapter 5, pages 79–104. Palgrave, 2003. 6