Yet Another Language Extension Scheme

Anya Helene Bagge

Bergen Language Design Laboratory
Dept. of Informatics, University of Bergen, Norway
anya@ii.uib.no

Abstract. Magnolia is an experimental programming language designed to try out novel language features. For a language to be a flexible basis for new constructs and language extensions, it will need a flexible compiler, one where new features can be prototyped with a minimum of effort. This paper proposes a scheme for compilation by transformation, in which the compilation process can be extended by the program being compiled. We achieve this by making a domain-specific transformation language for processing Magnolia programs, and embedding it into Magnolia itself.

1 Introduction

Implementing a compiler for a new programming language is a challenging but exciting task. As the language design evolves, the compiler must be updated to support the new design or to prototype the design of new features. Magnolia is both an experimental programming language, and a language for language experiments. We therefore need a compiler flexible enough to keep up with changes in the language design, and with features that make implementation of experimental features easy.

Use cases for a language extension facility include experimental features such as data-dependency based loop statements, embedding of domain-specific languages, restriction to sub-languages with stricter semantics and language implementation using a simple core language, and building the rest as extensions.

In Magnolia, the programmer can express extra knowledge about abstractions as *axioms*. In the compiler, we would therefore like to preserve abstractions for as long as possible, in order to take advantage of axioms. Language extensions also provide abstractions, with knowledge we may also want to take advantage of. Desugaring extensions to lower-level language constructs at an early stage, as is done with syntax macros, discards any special meaning associated with the constructs, which could have been used for optimisation and extension-specific error checking.

The Magnolia compiler is implemented in Stratego/XT [1], using compilation by transformation, where a sequence of transformation steps transform code in the source language to a target language (object code, or another programming

M. van den Brand, D. Gašević, J. Gray (Eds.): SLE 2009, LNCS 5969, pp. 123–132, 2010.

language). It is therefore natural to make use of transformation techniques for describing language extension. This paper presents an extension of the Magnolia language with transformation-based meta-programming features, so that extensions to the Magnolia language can be made in Magnolia itself, rather than by extending the Stratego code of the compiler. This gives more independence from the underlying compiler implementation.

The rest of this paper is organised as follows. First, we give a brief introduction to the Magnolia language, before we look at how to add language extension to it (Section 3). We have two extension facilities, macro-like *operation patterns* (Section 3.1) and low-level *transforms* (Section 3.2). We provide an example of two extensions, before discussing related work and concluding (Section 4).

2 The Magnolia Language

We will start by briefly introducing the parts of Magnolia that are necessary to understand the rest of the paper. Magnolia is designed as a general-purpose language, with an emphasis on abstraction and specification. Abstractions are described by *concepts*, which consist of abstract types, operations on the types, and axioms specifying the behaviour of the operations algebraically. Multiple *implementations* may be provided for each concept, and *signature morphisms* may be used to map between differences in concept and implementation.

Operations can be either *procedures* or *functions*. Procedures are allowed to update their parameters, and have no return values. Pure procedures only interact with the world through their parameters (e.g., no I/O or global data). Functions may not change their parameters, and are always pure – the only effect a function has is its return value, and it will always produce the same return value for the same arguments. Function applications form expressions, while procedure calls are statements. In addition, Magnolia has regular control-flow statements like `if` and `while`.

A novel feature (detailed in a previous paper [2]) is the special relationship between pure procedures and functions. Procedures may be called as if they were functions – the process of *mutification* turns expressions with calls to functionalised procedures into procedure call statements. An expression-oriented coding style is encouraged. Procedures are often preferred for performance reasons, while expressions with pure functions are easier to reason about, and is also the preferred way of writing axioms.

3 Extending Magnolia

At least four types of useful extensions spring to mind:

1. Adding new operation-like constructs, that look like normal functions or procedures, but for some reason cannot or should not be implemented that way – for example, because we need to bypass normal argument evaluation, or because some of the computation should be done at compile time. This

type of change has a local effect on the particular expressions or statements where the new constructs are used, and is similar to syntax macros in other systems.
2. Adding new syntax to the language, in order to make it more convenient to work with. We may also consider removing some of the default syntax. In Magnolia, this can be handled by extending the SDF2 grammar of the language.
3. Disabling features or adding extra semantic checks to existing language constructs. This can be used to enforce a particular coding style, to disable general-purpose features when making a DSL embedding, or to ensure that certain assumptions for aggressive optimisation holds.
4. Making non-local changes to the language – features requiring global analysis, or touching a wide selection of code. Cross-cutting concerns in aspect orientation are an example of this. We can implement this by extending the compiler with new transformations and storing context information across transformations.

In a syntax macro system, new constructs are introduced by giving a syntax pattern and a replacement (or *expansion*). In languages like Lisp or Scheme, the full power of the language itself is available to construct the expansion. For Magnolia, things are a bit more complicated, since the extension may pass through several stages of the compiler before it is replaced by lower level constructs. We must therefore provide the various compiler stages with a description of how to deal with the language extension.

To provide syntax extensibility of the kind found in languages like Dylan, one could provide Magnolia syntax for syntax definition, then extract and compile the syntax definitions to SDF2, as used in the compiler. We will not consider this here, however. A full treatment of compiler extension in Magnolia is also beyond the scope of this paper, we will therefore focus the macro-like *operation patterns* and briefly sketch the *transform* interface to compiler extension.

3.1 Operation Patterns

An *operation pattern* is a simple interface to language extension, similar to macros in Lisp or Scheme. Patterns are used in the same way as a normal procedure or function, but is implemented using instantiation with arbitrary code transformation. They are useful for things that need to process arguments differently from normal semantics.

The implementation of an operation pattern looks like a procedure or function definition, except that one or more of its parameters are meta-variables that take expression or statement terms, rather than values or variables. The argument terms and pattern body may be rewritten as desired by applying transforms to them (see examples below). When the operation pattern is instantiated, meta-variables in the body are substituted, and any transformations are applied. The resulting code is inlined at the call site.

Meta-variables are typed and are distinguished from normal variables through the type system, thus it is not necessary to use anti-quotation to indicate where

meta-variables should be substituted. Operation patterns introduce a local scope, so local variables will not interfere with the call context.

The semantic properties (typing rules, data-flow rules, etc.) of an operation pattern are handled automatically by the compiler, and calls to operation patterns are treated the same as normal operation calls during type checking and overload resolution. This means that they can be overloaded alongside normal operations, and follow normal module scoping and visibility rules. Processing code with operation pattern calls requires some extra care, so that arguments that should be treated as code terms won't get rewritten or lifted out of the call.

Operation patterns can also conveniently serve as implementations of syntax extensions, by desugaring the syntax extension into a call to the pattern.

For example, the following operation pattern implements a simple way to substitute a default value when an expression yields some error value:

```
forall type T procedure default(T e, T f, expr T d, out T ret) {
  ret = e;
  if(ret == f)
    ret = d;
}
```

The `f` is the failure value (null, for example), `d` is the default replacement, and `e` is the expression to be tested.

Magnolia will automatically provide a function version of it:

```
forall type T function T default(T e, T f, expr T d);
```

which we can use like:

```
name = default(lookup(db,key), "", "Lucy");
```

We can describe the behaviour of `default` by axioms, for example:

```
forall type T axiom default1(T e, T f, T d) {
  if(e == f) assert default(e, f, d) <-> d;
  if(e != f) assert default(e, f, d) <-> e;
  if(f == d) assert default(e, f, d) <-> e;
  if(f != d) assert default(e, f, d) <!-> f;
}
```

3.2 Transforms

For further processing of language extension, we add a new meta-programming operation to Magnolia – the `transform` – corresponding to a rule or strategy in Stratego. Transforms work on the term representation of a program, taking at least one term plus possibly other values as arguments, and returning a replacement term. Provided semantic analysis has been done, term pattern matching in transforms are sensitive to typing, overloading and name scoping rules.

A transform may call other transforms and operations, and may also manipulate symbol tables and other compiler state. Several transforms can share the same name; when applied they are tried in arbitrary order until one succeeds. In addition to explicit calling, transforms can also be controlled through

Table 1. Transform classes: Topdown and bottomup traversals can be modified by repeat, once or frontier. The phase classes can be used to apply a transform before, during or after a particular compiler phase, or to trigger application of a compiler phase. Transforms can also be classified by use – for example, simplification transforms may be marked as such and used many places in the compiler. The `ac` class can be used to reorder expressions for associative-commutative matching.

Traversals/modifiers		**Compiler Phases**		**Uses**
`repeat`	Can be used repeatedly	`during`(p)	apply during p	`typecheck`
`once`	In traversal: Apply only once	`before`(p)	apply before p	`simplify`
`frontier`	In traversal: Stop on success	`after`(p)	apply after p	`mutify`
`topdown`	Traversal type	`requires`(p)	run p first	`ac`
`bottomup`	Traversal type	`triggers`(p)	run p after	
`innermost`	Innermost reduction			
`outermost`	Outermost reduction			

transform classes, which describe how and (possibly) when transforms should be applied. For example, a transform may have the classes *innermost* and *during(desugar)*, signifying that it should be applied using an innermost strategy during the desugaring phase of the compiler.

A sample transform is:

```
forall int i1, int i2, int i3
transform example(expr i1 * i2 + i3 * i2) [simplify,repeat]
  = (i1 + i3) * i2;
```

This example has a pattern with three meta-variables, `i1`, `i2`, `i3`, all of which will match only integer expressions. The expression pattern in the argument list will be matched against the code the transform is applied to, and will only match the integer versions of `+` and `*`. If the match is successful, the code is transformed to `(i1 + i3) * i2`. The transform classes *simplify* and *repeat* tell the compiler that this rule can be applied during program simplification, and that it will terminate if applied repeatedly. Table 1 shows a few different transform classes. Axioms, when used as rewrite rules, can also have classes assigned to them, making them usable as transforms [3].

Transforms can be applied directly in program code (most useful inside operation patterns). For example,

```
var x = example(a * b + c * b);
```

will apply the above transform (the expression to the left is implicitly passed as the first parameter) and rewrite the code to:

```
var x = (a + c) * b;
```

The double-bracket operator `[[...]]` can be used to apply inline rewrite rules, and to specify traversals – we'll see examples of this later.

3.3 Semantic Rules

Semantic analysis rules are described by the *typecheck* transform, which takes a statement, expression or declaration as argument, and returns a resolved version

of its argument – and its type, in the case of an expression. Resolving means annotating each use of an abstraction with a unique identifier that leads back to its declaration – this is typically taken care of internally in the compiler. Type checking of a declaration will typically involve adding declarations to the symbol table; type checking other constructs is typically a simple case of recursively type checking sub-constructs. A (simplified) typecheck rule for assignment statements is:

```
forall name x, expr e
transform typecheck(stat{x = e;}) = stat{x = e';}
where {  var (e', t) = typecheck(e);
         if(!compatible(typeof(x), t))
           call fail("Incompatible types in assignment"); }
```

Note that typechecking may be better described as more formal semantic rules which can be used as a basis for reasoning about typechecking and programs. This is an option we are exploring.

Axioms [3] can describe the abstract semantics of a construct. This is only applicable to expression-like constructs at the moment, we should also have a way of describing other constructs.

Implementation rules are used to compile constructs to lower-level code. *Instantiation rules* are triggered during semantic analysis, and receive the unique id of the abstraction and the use case, and produce an instantiated version. Other implementation rules are free-form and should be tied to a program traversal strategy and compiler phase. No effort is made on the part of the compiler to ensure that implementation rules don't leave behind uncompiled constructs, though we are looking at techniques that can handle this [4].

Other compiler phases may also need rules – for example, doing data-flow analysis and program slicing requires information about which variables are read and written in a statement – the *readset* and *writeset* transforms are used for this purpose. Transforms may also be provided for mapping between statement and expression forms.

By keeping track of semantic information, we can make more powerful extensions. For example, with the following extended version of `default` a failure value is no longer needed – it is obtained automatically from a function declaration attribute:

```
forall type T
function T default(expr T e, expr T d) =
  default(e, getAttr("fail_value", e), d);
```

3.4 Module-Level and Global Extensions

Language extension should normally be done at the module level, so that some modules in your program may use the extension, and others won't. For example, if your extension defines a restricted subset of Magnolia with some DSL features, you probably still want the compiler to process Magnolia libraries as if they were written in normal Magnolia. Therefore, Magnolia extensions have scope:

- The *names* of transforms and operation patterns are accessible in the module in which they are defined and in modules that import them, just as with other operations.
- Transforms are normally applied to the whole program. Semantically aware term pattern matching ensure that only relevant parts of the code are touched, not code that merely looks similar to what is described by the pattern.
- For syntax extensions and language-changing transforms that should only be applied to certain modules, there is a `language` declaration in the module header that can be used to import extension modules. Transforms imported via `language` are only applied to the local module.

3.5 Example Extensions

We will give two example extensions, one which uses transforms to enforce a restriction on the language, and one which uses operation patterns to add a `map` construct.

Impure procedures are ones that violate the assumption that two calls with equivalent inputs give equivalent results. I/O is typically impure, a random generator that keeps track of the seed would also be impure. Since pure code is easier to reason about, we might want to have a sub-language of Magnolia where calls to impure code is forbidden. We implement this in a module `pure`, which is used by putting `language pure` in the module header of pure modules. Our language module contains the following transform:

```
transform purity(stat{call p(_*)}) [after(typecheck)]
where if(getAttr("impure", p))
      call error("In call to ", p, " -- impure calls forbidden");
```

The transform `purity` will be applied to the code in all `language pure` modules after type checking is done (since the type checker might be used to infer impurity), and will match procedure calls. If the called procedure has the `impure` attribute, a compiler error is triggered.

The *map* operation applies an operation element-wise to the elements of one or more indexable data structures (arrays, for example). Our map works on multiple indexables at the same time (like Lisp's `mapcar`), without the overhead of dealing with a list of indexables at runtime. For example,

```
A = map(@A * @B + @C); // map *,+ over elements of A, B, C
A = map(@A * 5);       // multiply all elements of A by 5
A = map(@A * V + @C);  // V is indexable, but used as-is
```

While map in Lisp and functional languages traditionally takes a function (or lambda expression) and one or more lists as arguments – we will instead integrate everything as one argument, making it look more like a list comprehension. Indexables marked with an `@`-sign are those that should have element-wise. The `@` is just a dummy operator, defined as:

```
forall type A, type I, type E where Indexable(A, I, E)
function E @_(A a);
```

This function is generic in `E` (element type), `A` (indexable/array type) and `I` (index type) – together, these must satisfy the *Indexable* concept. Applying the `@`-operator outside a `map` operation will lead to a compilation error – this should ideally be checked for and reported in a user-friendly manner.

A generic implementation of map is:

```
forall type A, type I, type E where Indexable(A, I, E)
procedure map(expr E e, out A a) {
  // define index space as minimum of input index spaces
  var idxSpace = min(e[[collect,frontier: @x:A -> indexes(x)]]);
  call create(a,idxSpace); // create output array
  for i in indexes(a) { // do computation
    a[i] = e[[topdown,frontier: @x:A -> x[i]]];
} }
```

The implementation accepts an expression `e` (of the element type) and an output array `a`. The body of `map` is the pattern for doing maps, and this will be instantiated for each expression it is called with by substituting meta-variables and optionally performing transformations. Note that the statements in the pattern are not meta-level code, but templates to be instantiated. The `[[...]]` code are transformations which are applied to `e` – the result is integrated into the code, as if it had been written by hand. The first transformation uses a `collect` traversal, which collects a list of the indexables, rewriting them to expressions which compute their index spaces on the way. This is used in creating the output array. The computation itself is done by iterating over the index space, and computing the expressions while indexing the `@`-marked indexables of type `A`. The `frontier` traversal modifier prevents the traversal from recursing into an expression marked with `@` – in case we have nested maps.

As an example of `map`, consider the following:

```
Z = map(@X * 5 + @Y);
```

where `X` and `Y` are of type `array(int)`. Here map is used as a function – the compiler will *mutify* the expression, obtaining:

```
call map(@X * 5 + @Y, Z);
```

At this point we can instantiate it and replace the `call`, giving

```
var idxSpace = min([indexes(X), indexes(Y)]);
call create(Z,idxSpace);
for i in indexes(Z) {
  Z[i] = X[i] * 5 + Y[i];
}
```

which will be inlined directly at the call site.

Now that we have gone to the trouble of creating an abstraction for element-wise operations, we would expect there to be some benefit to it, over just writing for-loop code. Apart from the code simplification at the call site, and the fact that we can use `map` in expressions, we can also give the compiler more information about it. For example, the following axiom neatly sums up the behaviour of `map`:

```
forall type A, type I, type E where Indexable(A, I, E)
axiom mapidx(expr E e, I i) {
  map(e)[i] <-> e[[topdown,frontier: @x:A -> x[i]]];
}
```

applying `map` and then indexing the result is the same as just indexing the indexables directly and computing the map expression. Furthermore, we can also easily do optimisations like map/map fusion and map/fold fusion, without the analysis needed to perform loop fusion.

4 Conclusion

There is a wealth of existing research in language extension [5,6,7] and extensible compilers [8,9], and little space for a comprehensive discussion here.

Lisp dialects like Common Lisp [10] and Scheme [11] come with powerful macro facilities that are used effectively by programmers. The simple syntax give macros a feel of being part of the language, and avoids issues with syntactic extensions.

C++ templates are often used for meta-programming, where techniques such as expression templates [12] allow for features such as the map operation described in Section 3.5 (though the implementation is a lot more complicated).

Template Haskell [13] provides meta-programming for Haskell. Code can be turned into an abstract syntax tree using *quasi-quotation* and processed by Haskell code before being *spliced* back into the program and compiled normally. Template Haskell also supports querying the compiler's symbol tables.

MetaBorg [14] provides syntax extensions based on Stratego/XT. Syntax extension is done with the modular SDF2 system, and the extensions are desugared ("assimilated") into the base language using concrete syntax rules in Stratego.

Andersen and Brabrand [4] describe a safe and efficient way of implementing some types of language extensions using catamorphisms that map to simpler language constructs, and an algebra for composing languages. We have started implementing this as a way of desugaring syntax extensions.

We aim to deal with semantic extension rather than just syntactic extension provided by macros. We do this by ensuring that transformations obey overloading and name resolution, by allowing extension of arbitrary compiler phases, and allowing the abstract semantics of new abstractions to be described by axioms. The language XL [15] provide a type macro-like facility with access to static semantic information – somewhat similar to operation patterns in Magnolia.

In this paper we have discussed how to describe language extensions and presented extension facilities for the Magnolia language extensions, with support for static semantic checking and scoping. The facilities include macro-like *operation patterns*, and *transforms* can perform arbitrary transformations of code. Transforms can be linked into the compiler at different stages in order to implement extensions by transforming extended code to lower-level code. Static semantics of extensions can be given by hooking transforms into the semantic analysis phase of the compiler.

A natural next step is to try and implement as much of Magnolia as possible as extensions to a simple core language. This will give a good feel for what abstractions are needed to implement full-featured extensions, and also entails building a mature implementation of the extension facility – currently we are more in the prototype stage. There are also many details to be worked out, such as a clearer separation between code patterns, variables and transformation code, name capture / hygiene issues, and so on.

The Magnolia compiler is available at `http://magnolia-lang.org/`.

Acknowledgements. Thanks to Magne Haveraaen and Valentin David for input on the Magnolia compiler, and to Karl Trygve Kalleberg and Eelco Visser for inspiration and many discussions in the early phases of this research.

References

1. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming 72(1-2), 52–70 (2008)
2. Bagge, A.H., Haveraaen, M.: Interfacing concepts: Why declaration style shouldn't matter. In: LDTA 2009. ENTCS, York, UK (March 2009)
3. Bagge, A.H., Haveraaen, M.: Axiom-based transformations: Optimisation and testing. In: LDTA 2008, Budapest. ENTCS, vol. 238, pp. 17–33. Elsevier, Amsterdam (2009)
4. Andersen, J., Brabrand, C.: Syntactic language extension via an algebra of languages and transformations. In: LDTA 2009. ENTCS, York, UK (March 2009)
5. Brabrand, C., Schwartzbach, M.I.: Growing languages with metamorphic syntax macros. In: PEPM 2002, pp. 31–40. ACM, New York (2002)
6. Standish, T.A.: Extensibility in programming language design. SIGPLAN Not. 10(7), 18–21 (1975)
7. Wilson, G.V.: Extensible programming for the 21st century. Queue 2(9), 48–57 (2005)
8. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for Java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
9. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: OOPSLA 2007, pp. 1–18. ACM, New York (2007)
10. Graham, P.: Common LISP macros. AI Expert 3(3), 42–53 (1987)
11. Dybvig, R.K., Hieb, R., Bruggeman, C.: Syntactic abstraction in scheme. Lisp Symb. Comput. 5(4), 295–326 (1992)
12. Veldhuizen, T.L.: Expression templates. C++ Report 7(5), 26–31 (1995); Reprinted in C++ Gems, ed. Stanley Lippman
13. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Haskell 2002, pp. 1–16. ACM, New York (2002)
14. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: OOPSLA 2004, pp. 365–383. ACM Press, New York (2004)
15. Maddox, W.: Semantically-sensitive macroprocessing. Technical Report UCB/CSD 89/545, Computer Science Division (EECS), University of California, Berkeley, CA (1989)