Language Description for Frontend Implementation

Anya Helene Bagge
Bergen Language Design Laboratory
Dept. of Informatics,
University of Bergen, Norway
http://www.ii.uib.no/~anya

ABSTRACT

For a language to be useful, it requires a robust and reliable implementation. Writing and maintaining such an implementation is a hard task, particularly for experimental or domain-specific language projects where resources are limited. This paper describes an implementation approach based on modular specifications of syntax and static semantics. Specification is done in a language description DSL, which serves both as a specification, and as code from which compiler front ends can be automatically generated.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax, Semantics*; D.3.4 [**Programming Languages**]: Processors—*Translator writing systems and compiler generators*

General Terms

Languages

1. INTRODUCTION

The semantics of programming languages are commonly divided into static semantics, specifying the class of well-typed, valid programs; and dynamic semantics, specifying the behaviour or meaning of programs. A compiler has to consider both kinds, with the compiler front end being responsible for rejecting programs that are invalid according to the static semantics, and delivering valid programs to the rest of the compiler in a suitable internal representation.

While language syntax has traditionally been formally specified, specifications of formal semantics are much rarer, with Standard ML [9] being the most famous example. Defining semantics is generally seen as much harder than syntax definition, and the payoff is less – while production-quality parsers can be generated from a grammar, compilers are still mostly built by hand, with compiler generation mostly being used as a prototyping tool.

Formal descriptions of semantics have several advantages though, since it opens up the possibility of formal reasoning about the semantics, and gives a foundation for reasoning about programs in the language.

LDTA '10, March 28-29, 2010, Paphos, Cyprus

In this paper, we will introduce *MetaMagnolia*, a domain-specific language used for specifying the syntax and static semantics of the Magnolia Programming Language [2, 3]. Our goal is to provide for formal, yet pragmatic specification of compiler front ends, so that we may generate high-quality, usable compiler components, rather than just prototypes. The specification serves a dual role as both a formal language specification and a front end implementation. Eventually, we would also like a formal specification of dynamic semantics, but that is beyond the scope of this work.

Being able to specify other languages than Magnolia is of secondary concern for now, we would rather have a useful formalism and tool for Magnolia, than a half-finished general tool. However, we hope that the same approach can be used for other languages, and possibly be extended to more general applicability.

Magnolia is a statically typed, modular language with support for overload – a feature that typically complicates specification of static semantics. The overload resolution code is one of the most complicated parts of the old Magnolia compiler,[1] so being able to specify this in a concise, easy to grasp manner is important.

Magnolia is also designed to support experimentation and language extension [2], so being able to make custom language definitions with extra syntax and extended semantics is desirable. In particular, we would like to be able to tweak the existing semantics when creating language variants – for example, changing overloading preferences.

We will start by discussing syntax description in the next section, then move on to our static semantics formalism in Section 3, error handling in Section 4, before discussing related work, future directions and conclusion in Section 5. Some auxiliary definitions and rules are provided in the appendix.

2. SYNTAX AND SUGAR

The Magnolia language presented to the user is layered on top of a simplified core language (*Magnolia Core*), where unnecessary variation has been eliminated. This has the benefit of reducing the number of constructs that must be dealt with in the language

[1] With the main overload rules having eight parameters, and relying on several pages of supporting rules. Comparatively, the C++98 standard [6] spends 25 pages on overloading (plus several pages dealing with template functions), which is exactly half the size of the entire Scheme report [7].

specification and in the compiler. In order to support language experimentation and evolution, Magnolia is designed as an extensible language [2] – it is thus natural to design the extension facility in such a way that it can also be used to implement the default user-level language (*Magnolia Base*).

The compiler front end translates between the layered language, and the core language. The optimisation and code generation phases of the compiler work solely on the core language. Magnolia Core differs from Magnolia Base in the following respects:

- All names are fully qualified and unique (i.e., no overloading).
- Variables in Core are always explicitly typed (e.g., `x:int` instead of just `x`).
- Language constructs have no optional parts (e.g., the `else` clause of an `if` must be specified, even when empty).
- Superfluous constructs are removed (e.g., operator calls are mapped to function calls)

Thus, translation to Core consists of

- Qualifying all names and resolving overloaded names.
- Qualifying all variables with type information.
- Removing optional clauses and other syntactic sugar.

The semantic analysis part of the front end, discussed in Section 3, takes care of name and type qualification, while the desugaring part, discussed here, deals with removing syntactic sugar.

The syntax of Magnolia Core is specified in SDF2 [14], and is split into multiple independent files, so that new language variants can be built by combining fragments of the Core language. Syntax extensions can be done by adding SDF definitions, or directly as MetaMagnolia syntax rules.

2.1 Language Definitions

Language variants are specified in language definition files, as shown in Figure 1. External SDF syntax definitions are pulled in and combined by the `syntax mix` declaration. The new language (hereafter referred to as the 'sugar language') is related to the Core syntax by syntax rules, which define a translation based on the *banana algebra* approach of Andersen and Brabrand [1]. Syntax rules quoted by (+...+) introduce new syntax, the others use externally defined syntax, from the `syntax mix` declaration.

The syntax rules form a *constructive catamorphism* (or *banana*) from the sugar language to the core language. Catamorphisms [8] are a generalisation of list folding from functional programming, working on any recursive data type – in this case, abstract syntax trees. For every term constructor in the input language, there is a rule that maps to a term in the same sort in the output language. Subterms are processed inductively, by implicit recursion. While this translation scheme is less powerful than one based on unrestricted program transformation or attribute grammars, it offers several benefits:

Guaranteed Completeness: It is possible to statically verify that every construct in the input language is mapped to the output language.

Safety: The translation will always terminate, and always maps between syntactically valid terms.

Efficiency: The translation can be implemented very efficiently.

As we gain more experience, we may make desugaring rules slightly more powerful, if needed, as long as we can still ensure the safety and efficiency of the desugaring process.

2.2 Syntax Rules

The left hand sides contain patterns in the sugar language (enclosed in colon quotes, `(:...:)`), while right hand sides contain patterns in the core language, possibly with quoted sugar language subterms to be processed recursively. For example,

```
(+ if e then s1* end +)
     <->  (; if (: e :) then (: s1* :) else end ;)
```

Variables are typed according to a fixed naming scheme (see Figure 2), and are displayed here in italics. Variable names are constructed from an initial letter, optionally followed an *s*, a series of digits and a series of primes. Identifiers that do not match this scheme are literal identifiers. A star or plus indicates list matching. Quotes around variables on the right hand sided are optional, as the variables are already known to be in the input language. The type (non-terminal sort) of the right hand side is required to be the same as on the left hand side (e.g., mapping expressions in the input language to expressions in the output language).

Rules can be left-to-right (*desugaring rules*), right-to-left (*ensugaring rules*) or both ways (as above). A plus in front of a pattern indicates that the grammar should be extended according to that pattern – this is done by mapping variables to non-terminals, and non-variables to literal tokens. Grammar productions too complicated to fit into this simple scheme, such as infix operators with precedence, must be specified externally in SDF (for now). Rules without a plus apply to syntax defined in SDF.

It would be perfectly reasonable to define all the syntax directly in SDF; the main benefit of internal syntax definitions is that some boilerplate SDF code is generated automatically, like follow restrictions on keywords, for example. Also, it would be fairly straightforward to check that the new syntax against language style rules.

Ensugaring rules are intended to be used to pretty-print code, or to generate specialised versions of semantic rules, tailored to an specific input language.

2.3 Parsing and Desugaring

Based on the language definition, the MetaMagnolia compiler will generate an SDF syntax definition, importing any syntax definitions referenced by the language definition, and adding productions for any syntax defined in rules.

The SDF syntax definition is then used to generate a parse table and a regular tree grammar for the parse tree, which can later be used to perform format checking on parse trees.

Syntax rules are straight-forwardly translated to Stratego [4] rules for desugaring and ensugaring, and a custom front end module is

```
language module Magnolia
syntax mix Magnolia/Core + Magnolia/BaseExt
expression syntax
  (+ n +)          <-> (; n : ? ;)            // Variables
  (: e1 o e2 :)    <-> (; _o_(e1, e2) ;)      // Infix operators
  (: o e1 :)       <-> (; o_(e1) ;)           // Prefix operators
  (: e[es*] :)     <-> (; index(e, es*) ;)    // Indexing operator
statement syntax
  (+ if e then s1* end +)  <-> (; if (: e :) then (: s1* :) else end ;)
```

Figure 1: Fragment of MetaMagnolia language definition module for Magnolia Base.

Starting with	Non-terminal sort	Starting with	Non-terminal sort
d	Declaration	*o*	Operator
e	Expression	*p*	Procedure
f	Function	*s*	Statement
m	Parameter mode	*t*	Type
n	Name / Identifier	*v*	Variable

Figure 2: Meta variables and their types.

built which parses input programs, applies desugaring and feeds the code to the next step in the compiler pipeline.

3. SPECIFYING STATIC SEMANTICS

Static semantics encompasses specification of type checking, and of other compile time checks to determine the validity of an input program. Classic specification of type checking, e.g., as presented by Pierce [12], consists of providing *type judgements*, that succeed or fail depending on whether a term is well-typed or not. For example, the following type judgement states that, given an environment Γ, if f has the type $T \to U$ and e has the type T then $f(e)$ is well-typed, with type U.

$$\frac{\Gamma \vdash f : T \to U \qquad \Gamma \vdash e : T}{\Gamma \vdash f(e) : U} \tag{1}$$

Such rules are unsuitable for direct use for semantic analysis in a compiler front end for a language with overloading and named scopes; we also need information from name and overload resolution – we would need to know which particular f the programmer is referring to. Type checking and semantic analysis should therefore yield both a decision on program validity, and an annotated program tree. While name resolution could be done separately from type checking, overload resolution is tied in with typing, as the type of an expression will depend on the result of overload resolution, and overload resolution will depend on the types of arguments. It therefore makes sense to combine these aspects of static semantics into a single set of rules.

We have the following requirements for a static semantics formalism for Magnolia:

Pragmatic: Should support the generation of a real, high-quality compiler front end with good performance and good error reporting – not just serve as a rapid prototyping tool.

Formal: Should have a formal basis, so properties of the language can be verified, and so semantic analysis can be statically checked for completeness and correctness.

Modular: Should support the addition of new language constructs, and modification of existing constructs.

Friendly: The formalism should be concise and be familiar to people with previous exposure to semantic formalisms.

3.1 Static Semantic Rules

We use a formalism based on the familiar *structural operational semantics* (SOS) [13], but with some changes and extensions to make it more convenient for complex programming languages.

Resolution rules and patterns: Static semantics is described by *resolution rules* of the form $pat \Rightarrow pat'$, where each rule may have zero or more *premises*, written above a dividing horizontal line. Patterns can use pattern abstractions, or simply be terms over Magnolia Core, specified in concrete syntax within quotes, using the same variable naming scheme as before (Figure 2). All resolution rules yield a result of the same sort as the input. The hash symbol can be used to introduce an explicit context in the patterns: *pat#ctx* ⇒ *pat'#ctx*.

Quotes: We use the same quote symbols as in desugaring, with `(:...:)` being used for unresolved code, and `(;...;)` for resolved code.[2]

Premises: A premise can be either a *resolution* `pat ==> pat'` (“*pat resolves to pat'*”, invoking another semantic rule); a *transition* `pat --X-> pat'` (“*X transforms pat to pat'*” or “*pat X-es to pat'*” – multiple transformations can be applied in the same transition); an *operator call* `op(args) = pat'`; or a *predicate* `pred(pat*)`. For resolutions, operations can be applied to the resolution context: `pat ==declare(...)=> pat'`, for example, for resolving with a new declaration added to the environment. A rule succeeds if its left hand side pattern matches and all its premises succeed.

[2]Experience has shown that it's easy to get confused by which quotes to use, so we may change this aspect of the design.

Transformation rules: Auxiliary transformation rules have the form `pat --X(args)-> pat'`, and may have premises just as resolution rules do. Although multiple transformations can be chained when used as a transition in a premise, each transformation must be defined separately.

Context: Rules are evaluated in a *context*, containing information about name bindings (i.e., the environment in a traditional SOS setting) and other useful information. The context is an abstract data type, only accessed through operations such as name lookup, name binding, scoping, etc. The context is implicitly propagated to all premises of a rule (Appendix, PROP1–PROP3), and may be modified only upon applying a resolution rule. Context changes in resolutions are usually local to that transition, though certain operations may make non-local changes to the context (e.g., when processing global or top-level declarations, or updating a global table of definitions).

For example, the non-overloaded function application rule from (1) would look like this in our scheme:

```
construct funapp:
    f --?-> (; fun f'(_:t) : t';)    e ==> e':t
    ----------------------------------------
              (: f(e) :) ==> (; f'(e') ;)
```

In the first premise, the lookup operation `?` is applied to the function name f, yielding a function declaration; the second resolves e to an annotated expression e' and a type t'. The overall transition resolves $f(e)$ to an annotated expression $f'(e')$ and its type t'. The `construct` keyword introduces rules relevant to a particular construct.

The context can also be dealt with explicitly when desired:

```
construct funapp:
    f # ctx --?-> (; fun f'(_:t) : t';) # ctx
          e # ctx ==> e':t # ctx
    ---------------------------------------
     (: f(e) :) # ctx ==> (; f'(e') ;) # ctx
```

Note that here, the context is not updated, so there is no real need to deal with it explicitly.

Constructs: Rules are placed in a language definition module, with the rules for each construct included in a `construct` clause. The construct clause may also contain syntax definition and sugar rules, documentation, and other information relating to a particular language construct. By naming the constructs and analysis rules, we make it possible to refer to them later on, for example to replace the rules for a particular construct in an language extension, or provide additional rules, etc. Other than that, grouping in construct declarations do not affect the semantics of the rules – for example, the semantics of `if` could be split over multiple `construct`s, if desired. The corresponding grouping clause for transformation rule declaration is `operator`.

Pattern abstraction: In addition to concrete syntax patterns, Meta-Magnolia supports pattern abstractions, which can be defined for each construct. For example, the abstract pattern `e:t` allows us to pick the type from any expression. Similarly, given the following definitions;

```
construct call statement:
  syntax     (: call p(as*); :)
  patterns   call(p, as*)   args(as*)    name(p)
```

`args(as*)` would match any procedure call, and bind `as*` to the argument list – with a similar definition for function calls, we could use the same pattern for both procedure and function calls.

Lists: In most cases, resolution/transformation on single items are lifted automatically to operations on lists. If special behaviour is needed, the list operations can be defined manually (automatic lifting only occurs if there are no appropriately typed operations available).

Limitations: In keeping with our pragmatic approach, we place the following restrictions on premises, enabling semantic rules to be easily translated to rewrite rules:

- The left hand side of a premise may not refer to unbound variables. Unbound variables occurring on the right hand side get bound if the premise holds (already bound variables must match the result). Variables occurring on the left-hand side of conclusions are always bound.
- All variables in predicate and operator arguments must be bound.
- Premises are processed in sequence.

The rules presented here are somewhat simplified compared to the rules for Magnolia. The statement rules are unchanged.

3.2 Resolving Expressions

For resolving expressions, we have the following considerations:

- Magnolia allows function overloading, so we need access to the types of subexpressions when resolving an expression.
- Any names used must be updated to the unique identifier of the relevant function, type or variable.

An expression's type can be matched with the pattern `e:t`.

Let's start with a simple rule for variables:

```
construct variable expression:
  syntax       (: n:t :)
  patterns     var(n,t)    n:t    name(t)    type(t)
  static semantics
    n --?-unique-> (; var n':t' = _; ;)
    ---------------------------
        (: n:? :) ==> (; n':t' ;)
```

In Core syntax, a variable consists of a name and a type. For an unresolved variable, the type is unknown, hence `n:?` in the pattern above. Name resolution is performed by a transition, applying first the lookup operator `?`, which yields a list of declarations. In the case of variables, we don't allow overloading, so we apply `unique` which accepts exactly one declaration, failing otherwise. The result is matched against a variable declaration, picking out the resolved name and type.

A more complicated example is function application:

```
construct apply expression:
                e* ==> e'*:t'*
    n --?-applicables(t'*)-best-unique-> f
    ------------------------------------------
            n(e*) ==> f(e'*)
```

Here we have a function name `n` and a list of arguments `e*` (we will concentrate on the semantic definitions from now on, and omit the pattern definitions). We start by resolving the arguments, getting a list of resolved arguments and a list of their types. Then we do a name lookup of `n`, giving us a list of declarations. We now need to find the declarations that match the particular argument types – this is done by the `applicables` filter, which yields a list of weighted pairs of identifiers and return types. Finally, since we may be faced with more than one applicable candidate, we find the `best` candidate according to the weights, and ensure that we have a `unique` result.

Applicability is checked by a filter, applying `applicable?` to each element of the declaration list, and removing those for which it fails. A function f with formal argument types t_1* is applicable with respect to an actual argument list t_2* if t_1* is compatible with t_2*:

```
operator applicable?:
          compatible(t1*, t2*) = x
    ------------------------------------
    (: function f(n1*:t1*):t = e; :)
          --applicable?(t2*)-> f weight x
```

The compatibility check yields a weight, x, that indicates how 'good' the match is (number of type conversions, generic type matching, etc.). Note the syntax abstraction for argument list matching – $n* : t*$ matches a list of name/type pairs $n_1 : t_1 \cdots n_k : t_k$. The definition of the filter is not shown.

Now, type compatibility can get interesting, if we go beyond just checking for equality. For instance, assume we allow implicit conversions from one type to another, by defining a function with the same name as the target type. In this case, we would like the implicit conversion made explicit, so we should let the compatibility check yield a modified argument list. The applicability and function application rules would have to be changed accordingly. The rule for checking a single argument would look like (with 0 and -10 being the weights given for no conversion and implicit conversion, respectively):

```
compatible?(t, t, e) = e weight 0

   not(equal(t1, t2))    nameOf(t1) = n
          (: n(e2) :) ==> e2':t2
-----------------------------------------
compatible?(t1, t2, e2) = e2' weight -10
```

The definition of `compatible` applies `compatible?` to all arguments, and combine the weight using some scheme (addition, for example).

Similarly, to support generics, we would have `compatible` yield the list of type argument bindings that was required for argument list compatibility.

3.3 Resolving Statements

Statement resolution consists mostly of processing the constituent parts, and combining the result. Here we can use an implicit recursion scheme, similar to how we did with desugaring, by marking parts in need of resolution with colon quotes. For example,

```
construct if statement:
                    isPredicate((: e :))
    ------------------------------------------------
    (: if e then s1* else s2* end :)
    ==> (; if (: e :) then (: s1* :) else (: s2* :) end ;)
```

The `if` statement is resolved by resolving is parts recursively. We also checking that the condition has a truth value type. The two occurrences of `(: e :)` will combined to a single resolution.

Procedure calls follow the same pattern as function application; resolve the arguments, lookup the procedure name and find the best candidate that matches the argument list.

```
construct call statement:
                e* ==> e'*:t'*
     n --?-callables(e'*)-best-unique-> p
    ---------------------------------------
    (: call n(e*); :) ==> (; call p(e'*); ;)
```

The `let` construct is more interesting, as it involves modifying the context:

```
construct let statement:
                    e ==> e':t'
              s* ==declare(n, t)=> s'*
    ------------------------------------------------
    (: let var n : ? = e; in s* end :)
          ==> (; let var n : t'= e'; in s'* end ;)
```

The `declare` operation (defined in an accompanying library) adds a declaration to the context used to resolve the statement list `s*`.

3.4 Resolving Declarations and Modules

Declarations should be mapped to resolved declarations (i.e., with constituent parts resolved, and the declaration name made unique). For example,

```
construct function declaration:
                t ==> t'          t1* ==> t1'*
                  n --uniqueId(t1'*)-> f
               e ==declareArgs(n1*,t1*)=> e'
    ------------------------------------------------------
    (: function n(n1*:t1*) : t = e; :)
          ==> (; function f(n1*:t1'*) : t' = e'; ;)
```

where `uniqueId` constructs a unique function identifier based on the name and the parameter list, and `declareArgs` adds declarations for all the parameters to the context used to resolve the function body:

```
operator declareArgs:
    ctx --declareArgs([],[])-> ctx

    ctx --declare(n,t)-declareArgs(n'*,t'*)-> ctx'
    ------------------------------------------------
      ctx --declareArgs([n|n'*],[t|t'*])-> ctx'
```

This resolution style disallows recursive functions – to allow recursion, the function declaration would also have to be added to the context of the body.

This specification style also means that all operations must be declared before use – a more user-friendly style could be achieved by splitting the declaration rules into multiple passes, one dealing solely with declarations, and the next dealing with declaration bodies.

A *module* is a named list of declarations. We add the name of the current module to the resolution context, for use in creating unique names for declarations.

```
construct module declaration:
  n --uniqueId(n')-> n' d'* ==set(currentModule,n')=> d'*
  ------------------------------------------------------
              (: module n d'* :) ==> (; module n'd'* ;)
```

Declaration lists are resolved such that a declaration is visible in the context of the following declarations. Importing of modules is dealt with by adding all the declarations of the imported module to the context.

Processing of a program starts with loading a main module, and then resolving that.

4. HANDLING ERRORS

Failing with just a "typechecking failed" message is clearly not a satisfactory solution for a compiler front end. We would like error messages to inform the user of *what* went wrong, *where* it went wrong (location and code sample), and provide related information like a list of candidates for function overloading.

One possibility for dealing with errors is to introduce fallback rules, that are tried when the main resolution rules fail. For example, we could have a rule that reported failure in variable lookup (with `/fail` in the construct name indicating a fallback rule):

```
construct variable expression/fail:
                n --?-> []
    ---------------------------------------
    n:? ==> fail("Undeclared variable (; n ;)")
```

Of course, plenty of other things could go wrong when resolving a variable – we might find that the name refers to something which is not a variable, for example. Making one rule for each case is tedious, and gives us a specification where most of the rules are actually concerned with error handling. On the other hand, a single error message for all variable errors will provide the user with poor feedback:

```
construct variable expression/fail:
    n:? ==> fail("Error resolving variable (; n ;)")
```

Another way to deal with error reporting is to trigger error messages as soon as possible. This can be done by extending the lookup transition chain with more steps, that check for errors and report as necessary. This allows us to differentiate between "nothing found by that name" and "no variable found by that name". Adding information about the current construct to the context would allow us to write generic error messages that are automatically tailored to each case.

4.1 How to Treat Failure

While we now have a fair idea of how to construct meaningful error messages, there is still the problem of what to do when an error is encountered. Just writing out the message and aborting the whole resolution process is a possibility, but this is sub-optimal for programmers, who will have to discover and fix type errors one at a time. It is better to find all errors in the program first, before aborting the compilation.

Also, there may be cases where we are just checking if something works or not, and failure is acceptable. In this case we don't want the system to output an error message. What we'll do instead is produce a default/unchanged result, and annotate it with the error messages. If it turns out to be actually used somewhere, we can output the error (i.e., when it reaches the declaration level).

We can avoid cascading errors by simply withholding error messages when any sub-resolution gives an 'unknown' result – or by just printing the innermost error messages.

5. DISCUSSION

In our current implementation, desugaring is done in a separate pass before type checking. However, the restricted format of the desugaring rules means that it should be possible to combing desugaring and resolving into a single pass. This would also be advantageous from the user's perspective as error messages will be related directly to the input code rather than code in the Core language. The idea here is to use the ensugaring rules on the static semantic rules to obtain rules tailored to a particular input language. An simpler approach would be to attempt ensugaring before outputting error messages. The main use of ensugaring rules would be in constructing error messages, and in outputting whole, transformed programs – though as of now, they remain unused.

In order to use MetaMagnolia for other languages, one would have to specify the embedding of the language's core syntax into Magnolia, possibly changing the non-terminal sorts known to MetaMagnolia. The implicit propagation and scoping of contexts may also have to be changed, depending on scoping rules, and one would of course need to implement any supporting abstractions such as symbol tables if the existing ones aren't suitable (i.e. operations such as `uniqueId` and `declare`). We have started development in this direction, with the new name of the tool being *MetaXa* (of which MetaMagnolia is now an application). MetaXa will have fewer assumptions built in, and we intend to experiment with applying it to other languages than Magnolia.

Dinesh [5] deals with error handling by having type checking of a term yield *true* if it is type-correct and yielding a structured error message otherwise. These messages propagate up through the program, so that the overall type checking results in either *true* or a message. This assumes of course that we are just dealing with type checking and not overload resolution, but the idea of propagating messages up the program tree instead of outputting directly

is useful, particularly since we may want to create rules like "if this resolves, do something, otherwise do something else", without getting spurious error messages from checking whether something resolves or not.

Mosses and New [11] introduce I-MSOS, an approach to MSOS with implicit propagation of auxiliary entities. Our context-propagation approach can be seen as a limited version of this, though we are not based on I-MSOS in any way. It may turn out that our scheme is too restrictive, and we may have to introduce more entities than just the context, and allow more powerful propagation patterns. In particular, it may be useful to allow semantic rules to insert new declarations into the module top-level list of declarations, something which is not possible in our approach.

The main inspiration for our static semantic rules is SOS, *structural operational semantics* [13], and the modular variant, MSOS [10]. While traditional SOS explicitly deals with environments, stores and other auxiliary entities, MSOS attaches labels to the transitions, indicating how the auxiliary entities are propagated throughout a rule. This means that rules can be reused for different languages which have different sets of auxiliary entities. We have less need for many different kinds of auxiliary entities, preferring to model this using a single *context*. Instead we need to do a wide range of processing, such as name lookup, filtering and overload resolution, which is conveniently specified on transition arrows. Thus labels on arrows in MSOS have an entirely different meaning than operators on arrows in MetaMagnolia.

5.1 Conclusion

MetaMagnolia is a formalism for specifying the syntax and static semantics of programming languages (in particular, the Magnolia language). Syntax is described either in the SDF formalism, or directly as MetaMagnolia rules; desugaring rules are then written to map the syntax into a core language. Static semantic rules for the core language are specified in an SOS-like formalism, which allows complex language features like overloading to be specified using a series of transitions.

We have yet to fully evaluate the approach, but so far we have had success with applying the system to Magnolia. Both desugaring rules and semantic rules turn out shorter and more readable than the corresponding Stratego code, though it may be that an approach based on idiomatic transformation rules would work quite well also. We are also considering whether it would be fruitful to use a syntax which is close to the Magnolia language itself, and thus be able to apply Magnolia's axiom system to the meta-level. Possibly, we may provide a translation between multiple syntaxes, according to whether the reader has training in operational semantics or not.

The syntax and semantic rules are compiled into SDF [14] and Stratego [4] code, giving a custom compiler front end for the particular language definition. We are also developing a C++ back end, which would make the generated code independent of Stratego. This is intended to replace the old hand-written Magnolia front end as soon as the tools are mature enough.

Acknowledgements. Many thanks to the reviewers for very insightful comments. This research is partially funded by the Research Council of Norway. Magnolia is developed at the Bergen Language Design Laboratory.

6. REFERENCES

[1] J. Andersen and C. Brabrand. Syntactic language extension via an algebra of languages and transformations. In T. Ekman and J. J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09)*, Electronic Notes in Theoretical Computer Science, York, UK, March 2009. Elsevier.

[2] A. H. Bagge. Yet another language extension scheme. In M. van den Brand, D. Gašević, and J. Gray, editors, *SLE '09: Proceedings of the Second International Conference on Software Language Engineering*, volume 5969 of *LNCS*, pages 123–132. Springer, March 2010.

[3] A. H. Bagge and M. Haveraaen. Interfacing concepts: Why declaration style shouldn't matter. In T. Ekman and J. J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09)*, Electronic Notes in Theoretical Computer Science, York, UK, March 2009. Elsevier.

[4] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[5] T. Dinesh. Type-checking revisited: Modular error-handling. In D. J. Andrews, J. F. Groote, and C. A. Middelburg, editors, *In Proceedings of the Workshop on Semantics of Specification Languages*, Workshops in Computing, pages 216–231. Springer, 1993.

[6] ISO/IEC JTC1/SC22. *ISO/IEC 14882: Programming languages – C++*, 1998.

[7] R. Kelsey, W. Clinger, J. Rees, H. Abelson, N. I. Adams IV, R. K. Dybvig, C. T. Haynes, G. J. Rozas, D. P. Friedman, D. H. Bartley, R. Halstead, D. Oxley, E. Kohlbecker, G. J. Sussman, G. Brooks, C. Hanson, G. L. Steele, Jr, K. M. Pitman, and M. Wand. Revised^5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76, 1998.

[8] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[9] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, revised edition, 1997.

[10] P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.

[11] P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. *Electron. Notes Theor. Comput. Sci.*, 229(4):49–66, 2009.

[12] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[13] G. D. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[14] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

APPENDIX
A. AUXILIARY RULES

In these rules, π refers to patterns, ρ to contexts, h to premises, χ to transformations, and ω to context transformations. For example, $\pi \rightarrow_\rho \pi$ is a transformation in context ρ. The notation π_ρ corresponds to `pat#ctx` in MetaMagnolia code.

Implicit propagation is desugared according to these rules, with the context being distributed over the rule definition:

$$\frac{h_1 \cdots h_n}{\pi \overset{\omega}{\Rightarrow} \pi'} \quad \rightarrow \quad \frac{h_1 \cdots h_n}{\pi \overset{\omega}{\Rightarrow}_\rho \pi'} \quad \rightarrow \quad \frac{h_{1,\rho_1} \cdots h_{n,\rho_n}}{\pi_{\rho_1} \overset{\omega}{\Rightarrow} \pi'_{\rho_n}} \qquad \text{(Prop1)}$$

$$\frac{h_1 \cdots h_n}{\pi \overset{\chi}{\rightarrow} \pi'} \quad \rightarrow \quad \frac{h_{1,\rho_1} \cdots h_{n,\rho_n}}{\pi_{\rho_1} \overset{\chi}{\rightarrow} \pi'_{\rho_n}} \qquad \text{(Prop2)}$$

$$\pi \overset{\chi}{\rightarrow}_\rho \pi' \quad \rightarrow \quad \pi_\rho \overset{\chi}{\rightarrow} \pi'_{\rho'} \qquad \text{(Prop3)}$$

Resolution is just a particular kind of transition (note that resolution normally doesn't modify the context, so usually $\rho' = \omega(\rho)$):

$$\frac{\pi_{\omega(\rho)} \xrightarrow{\text{resolve}} \pi'_{\rho'}}{\pi_\rho \overset{\omega}{\Rightarrow} \pi'_{\omega^{-1}(\rho')}} \qquad \text{(Resolve)}$$

Operator calls and *predicates* are simply sugar for transformations, with predicates discarding the result:

$$\frac{\pi_1 \xrightarrow{op(\pi*)} \pi'}{op(\pi_1, \pi*) = \pi'} \qquad \text{(Op-Call)}$$

$$\frac{\pi_1 \xrightarrow{pred(\pi*)} \_}{pred(\pi_1, \pi*)} \qquad \text{(Pred-Call)}$$

Transition chains (ρ'' is usually discarded afterwards):

$$\frac{\pi_\rho \overset{\chi 1}{\longrightarrow} \pi'_{\rho'} \qquad \pi'_{\rho'} \overset{\chi*}{\longrightarrow} \pi''_{\rho''}}{\pi_\rho \overset{\chi 1 \chi*}{\longrightarrow} \pi''_{\rho''}} \qquad \text{(Trans-Chain)}$$