Semantics-Directed Compiler Generation*

Michael J A Smith, University of Cambridge
<msmith@lanther.co.uk>

July 21, 2005

Abstract

This dissertation concerns the design and implementation of SemCom, an interpreter generator for arbitrary languages, which compiles a two-level operational semantics (in either big- or small-step style). Input to SemCom is expressed in a purpose-designed Semantic Description Language (SDL). An SDL file contains the definition of an evaluation and typing relation, and can express type systems up to polymorphism. The SemCom compiler performs a sequence of analyses on this input, and generates a lexer, parser, and interpreter (in both OCaml and F#) for the user's language. SDL itself is typed, and a number of consistency checks are performed, producing comprehensive debugging output and useful error messages. Non-deterministic semantics are supported.

1 Introduction

In essence, computation has always been concerned with automation, and this takes place at many different levels. Rather than building systems directly in hardware, we have a processor that provides a convenient *abstraction* – that is to say, we can concern ourselves with the movement and processing of binary data, rather than the control of electrical signals. Furthermore, we do not write general purpose programs in assembler, but use an *abstracted* language in which we can talk about more meaningful concepts, such as objects, functions, and so forth.

The concept of abstraction does not end here, however. A programming language is not just a shorthand for sequences of assembler instructions – it has an underlying meaning, which is in some sense separate from the implementation. Over the course of time, our concept of the *semantics* of a language has developed, so that we now have an underlying mathematical framework with which to describe the execution of a program. In particular, an *operational* semantics is a specification of an abstract machine (an inductively defined transition relation), that explains the execution of a program as a sequence of syntactic transformations. Given this formal description of a programming language, is it not natural to ask whether a compiler could be generated automatically?

This is not as far-fetched as it sounds. Consider the humble parser. Before the theory of formal context-free grammars was understood, the programmer had to code their lexing and parsing routines by hand. Now, we have tools such as `yacc`, which will generate an efficient parser from a BNF-style description of a grammar. Although this is a considerably more constrained problem than the one just posed, it shows that useful tools can arise from the

*In this document, the term *compiler generator* is used to refer more generally to the automated generation of compilers, interpreters and abstract machines.

application of theoretical Computer Science. Therefore, it seems worthwhile to investigate the extent to which a compiler or interpreter may be generated from a semantics.

In undertaking this project, the most important step was to develop a metalanguage for describing these semantics. My primary aim was to be pragmatic, yet not sacrifice the expressivity of the semantics to too great an extent. Looking at the existing work[1], there is a strong tendency towards one extreme or the other. The Relational MetaLanguage RML [4] certainly succeeds at generating an efficient interpreter (in C), but its language is not truly expressive – it is constrained to *directional* relations that operate over a user-defined abstract syntax. The work by Diehl [1], on the other hand, focuses on provably correct generation of a compiler and abstract machine, and is not concerned with efficiency.

From the beginning, I wanted to produce a system that would require minimal work on the part of the user, other than the design of the semantics itself. For example, in the case of a simple language, it is much more intuitive to write down rules concerning *concrete* rather than *abstract* syntax. If there is a simple mapping between the two, why not automate this? And if the user provides syntax-directed typing rules, this gives us enough information[2] to generate a parser and lexer, so why not do so automatically?

Alongside these goals, I did not want to over-constrain the metalanguage for the semantics. Yet, given the timescale of my work, I had to make some compromises. In the end, I decided to allow only two relations to be defined; namely an evaluation relation '$\Rightarrow$' and a typing relation ':'. While this disallowed some language features (e.g. subtyping), it is easy to see how to generalise, and it allowed me to concentrate on my primary proof of concept. Other than that, I tried not to constrain too much else – I wanted to support polymorphism, hence I needed type variables and type schemes (abstraction), and I did not want the user to have to worry about unimportant matters such as the ordering of premises (as RML forces them to).

Taking all this into consideration, I designed the Semantic Description Language (SDL). Along with the compiler, SemCom, the result of my project has been a working interpreter generator. SemCom compiles an SDL description of a language's semantics into OCaml code for a lexer, parser and interpreter. In the spirit of generality, it supports not only Kahn's natural semantics [2], but also Plotkin's structured operational semantics [5]. Non-deterministic rules are allowed, and the interpreter performs Prolog-style backtracking to give the option of attempting all possible executions. I have been able to test SemCom with a variety of semantics for a subset of ML, as well as a version of Milner's CCS.

2 Semantic Description Language (SDL)

I decided to keep my language close to the spirit of `yacc`, in that an SDL (`.sem`) file is split into a number of sections, as follows:

⟨*style_directive*⟩
⟨*eval_rules*⟩
%%
⟨*type_rules*⟩
%%
⟨*precedence_info*⟩

The user must provide precedence information in the usual `yacc` format, and the evaluation

[1] The few examples here are not representative of the field. Consult [3] for a more thorough (though slightly old) treatment.

[2] We also require information about operator precedence and associativity, but then again, so does `yacc`.

Type Relation	$\Gamma \vdash e : T$	`G_ |- e : T`
Eval Relation	$\langle e, s \rangle \Rightarrow \langle e', s' \rangle$	`<e,s> => <e',s'>`
Expressions	`fn` x `=>` e	`[[fn \x => \e]]`
	`(fn` x `=>` e_1`)` e_2	`[[\( fn \x => \e1 \) \e2]]`
	e	`[[\e]]` $\equiv$ `e`
Conditions	$x = y \wedge y = z$	`x = y /\ y = z`
	$x \in S_1 \vee x \in S_2$	`x IN S1 \/ x IN S2`
	$\neg(x = add(y, z))$	$\sim$`(x = add(y,z))`
Objects	$\{a \mapsto b\}f$	`{a |-> b}f`
	$f[a]$	`f[a]`
	$\{1, 2, 3\} \cup (S_1 \cap S_2)$	`{1,2,3} UNION (S1 INTERSECT S2)`
	$ftv(T) \backslash ftv(\Gamma)$	`ftv(T) \ ftv(G_)`
Type Scheme	$\forall A.(T)$	`!{A} T`

Figure 1: SDL syntax: mathematical (typeset) notation and concrete syntax

```
G_ |- e1 :  T;
{x |-> !{A}T}G_ |- e2 :  T'
 ----------------------------------------
G_ |- [[let val \x = \e1 in \e2 end]] :  T'
==IF==
A = ftv(T) \ ftv(G_) /\ ~(x IN dom(G_))
```

Figure 2: An SDL type rule for `let`-polymorphism

and typing relations must be given as a sequence of inductive inference rules. Following the work of Diehl, I decided to require a *two-level* semantics. In other words, all mathematical operations are defined separately from the main semantics (hence the 'second' level). From the user's perspective, this means that they must define these operations as a library of OCaml functions.

I was inspired by RML for the overall syntax, and this is best illustrated by example. Figure 2 shows an example typing rule for `let`-polymorphism, and Figure 1 compares SDL syntax to the corresponding mathematical notation. SDL supports a number of useful mathematical constructs, such as partial finite maps, and sets. Since metavariables may range over any of these constructs, including terms in the target language, I designed SDL as a typed language. The type inference system is non-trivial, since there are subtyping relations on maps and sets, for example, but it avoids explicit type declarations on the part of the user.

Each rule consists of a conclusion, zero or more premises, and a side condition. The ordering of the premises and conditions is not important, since SemCom performs an internal sequentialisation to resolve dependencies. Since rules are entirely relational, conditions act as constraints, limiting the applicability of a rule.

This concept of constraint is one that needs further explanation. A rule contains, in its conclusion, a fragment of target language syntax; for example '`if` x `then` y `else` z'. This is a *pattern*. That is to say, the rule can *match* any program that has appropriate (i.e. grammatical) expressions for x, y and z. If a side condition turns out to be false, or a premise cannot be matched, the rule application is said to become *stuck*. We can then backtrack from such a state to attempt other possible rules. This gives some insight into the execution mechanism I use,

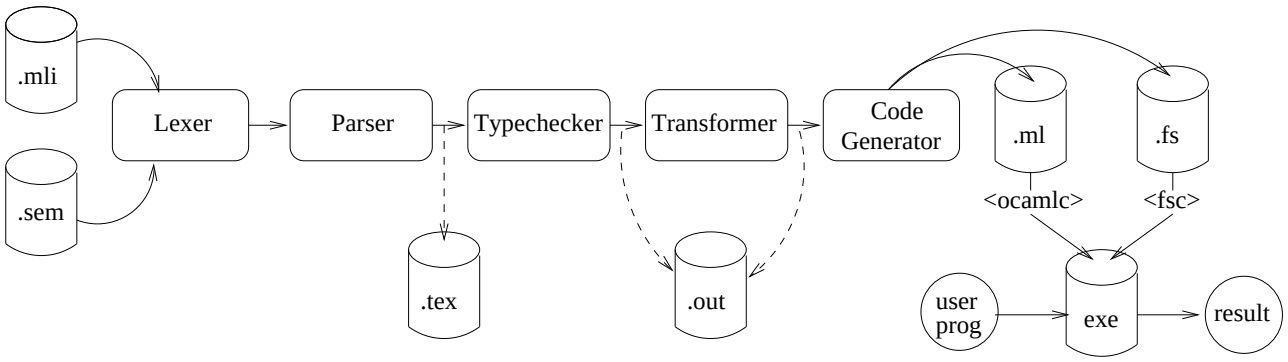

Figure 3: The main execution stages of the SemCom compiler

as explained in the next section.

An important constraint that I imposed on SDL is that there must be precisely one typing rule for every term in the target language's syntax. Why is this? It allows SemCom to infer the syntax of the target language, and therefore to generate a parser for it. It also assumes that there is a bijection between the language's concrete and abstract syntax, so that the interpreter can operate over an abstract syntax tree, without the user explicitly specifying it.

Whilst the typing rules are used to infer the language's *syntax*, I use the evaluation rules to infer its *values*. A target language expression is a *value* if it does not match any evaluation rule (or it is a primitive value, such as an integer). A good example is function abstraction, for languages like ML. Contrast this with finding a match for the expression, but getting stuck on all possible rules, which is treated as an evaluation *failure*. A good example is division by zero.

For a big-step semantics, a single transition of the evaluation relation reduces an expression to a value, but we have a sequence of transitions for a small-step semantics. Here, we apply the relation until we reach a fixed point (a normal form), at which there are no more reductions.

3 Implementation

Like any compiler, SemCom has a number of phases of operation, as illustrated in Figure 3. An SDL (`.sem`) file is passed through a lexer and parser, before being type-checked (to ease human readability, there is the option here of typesetting the semantics (`.tex` output)). The semantics then goes through a series of analyses and corresponding transformations, before being passed to the code generator (outputting OCaml or F# by a command-line option). The most interesting analyses are:

1. *Constraint analysis* – the premises and conditions of a rule may depend on metavariables that have not yet been instantiated. This detects these dependencies, and performs a sequentialisation transformation.

2. *Multiple match detection* – if two rules can match the same expression (e.g. for conditional execution), we need to know about it to generate the interpreter's backtracking engine.

3. *Binder detection* – to implement capture-avoiding substitution, we need to know which identifiers are bound. This looks at the typing rules to determine which identifiers are updated in the typing environment, hence determine which are binders.

The basic idea behind the generated interpreter is to define the evaluation and typing relations as recursive functions. In reality, it is a little more complicated, since we need to support backtracking, so SemCom uses its own execution stack. There are two such stacks, as shown in Figure 4:

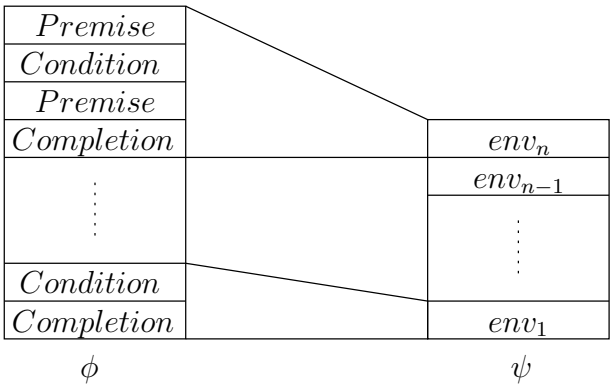

Figure 4: The evaluation and environment stacks

1. The *evaluation stack*, ϕ, maintains a list of 'goals to prove', i.e. the premises and conditions yet to be evaluated.

2. The *environment stack*, ψ, holds a mapping from metavariables to semantic objects (a context) for each rule in the process of being evaluated.

When a rule is matched, it pushes a list of premises and conditions onto the evaluation stack, and also a *completion*. Without this, there would be no way for a premise to update the environment of the rule that 'called' it. As a useful analogy, the environment stack behaves like a *stack frame* (containing local variables and parameters), and the evaluation stack like a *local stack* (on which execution takes place). I implemented an optimisation for tail-recursive rules (such as loops), to prevent these stacks from growing too large.

The final issue is backtracking. This in fact requires a third stack, on which SemCom pushes the point to backtrack to, if an expression matches multiple rules. During execution, the interpreter iteratively pops and executes the top element of ϕ. If it becomes stuck, it attempts to backtrack. Ultimately, either execution fails with an exception, or it completes, so that ϕ and ψ each contain just one element; the final result of evaluation.

4 Conclusion

Automated compiler generation is by no means a new concept. Much research in the area has taken place, but we have yet to see any significant products. There is a constant trade-off between generality and efficiency; while the former leads to inefficient theorem-prover style solutions, the latter leads to a dressed-up language with the emphasis still on the user. In order to get anywhere useful, compromises have to be made. It is simply not feasible to be general enough to cope with the most advanced type systems, nor to be as efficient as a modern, optimised, compiler.

In my work on SemCom, I wanted to see how far *complete* automation is possible. To this end, it has been a great success. By imposing a few simple constraints, such as syntax-directed typing, it is possible to automatically generate a lexer and parser for the target language, in addition to an interpreter. Furthermore, generality is not greatly sacrificed, since SemCom can deal with polymorphic type systems, and both big- and small-step semantics, including non-determinism. Testing semantics of a subset of ML, and of Milner's CCS, I investigated various effects; for example call-by-value versus call-by-name, and the consequences of non-value-restricted `let`-polymorphism.

Whilst I was not aiming for efficiency, it is interesting to note that the generated interpreters execute roughly three orders of magnitude slower than natively compiled OCaml. This is comparable to Diehl [1]. However, it is not difficult to turn an interpreter into a compiler; it simply emits code rather than performing the execution itself[3]. This would offer a significant increase in performance, since the overhead of the generated code is now mostly in the compiler.

Automated compiler generation is hard, but the majority of languages are relatively simple. In fact, many are custom scripting languages, geared towards a specific application, and are often interpreted. This sort of domain could benefit a great deal from automation, and I feel that SemCom illustrates the feasible of this. As a solution, it is not complete, but it serves to show that such tools are not as far away as one might think.

References

[1] S. Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines*. PhD thesis, University of the Saarland, Saarbrücken, Germany, 1996.

[2] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS'87*, volume 247, pages 22–39, London, UK, 1987. Springer-Verlag.

[3] P. Lee. *Realistic Compiler Generation*. MIT Press, Cambridge, MA, USA, 1989.

[4] M. Pettersson. RML - a new language and implementation for natural semantics. In *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94*, volume 844, pages 117–131. Springer-Verlag, 1994.

[5] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[6] M.J.A. Smith. Semantics-directed compiler generation. *Part II Dissertation, University of Cambridge Computer Laboratory,* `http://lanther.co.uk/compsci/semcom/semcom.pdf`, 2005.

[3] Loops need special treatment, but this is not difficult. See [6].