Michael J. A. Smith

Semantics-Directed Compiler Generation[1]

Part II Computer Science, 2005

Robinson College

May 17, 2005

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

Proforma

Name:	**Michael James Andrew Smith**
College:	**Robinson College**
Project Title:	**Semantics-Directed Compiler Generation**
Examination:	**Part II Computer Science, 2005**
Word Count:	$\sim$ **11,900**
Project Originator:	M. J. A. Smith
Supervisor:	T. Stuart

Original Aims of the Project

To design and implement an interpreter generator, that will accept as input a simple operational semantics of a language, and a simple type system definition, and generate code for an interpreter for said language.

Work Completed

An interpreter generator, SemCom, for arbitrary languages, based on a two-level natural semantics style definition. The metalanguage SDL allows specification of type systems up to polymorphism, and the execution engine is general enough to allow small-step dynamic semantics as well. Numerous analyses are performed to transform the semantics (including sequentialisation of premises and conditions) and provide consistency checks, and a lexer and parser are automatically generated from the semantics. We also produce comprehensive debugging output and useful error messages.

Special Difficulties

None.

Declaration

I, Michael J. A. Smith of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date: May 17, 2005

Contents

List of Figures

Chapter 1

Introduction

The essence of computation is automation, and this is manifest on many different levels. A program is a specified computation, which is then executed, or automated, by a computer. To help the programmer, higher level languages were introduced, along with the concept of compilation. Programs can then be written more *abstractly*, without worrying about low-level details such as the target architecture. Tools have even been written to help the compiler-writer – for example `yacc`, which will generate an efficient parser from a BNF-style description of a grammar.

Given this trend for automation, it is interesting to consider how far we can go. For example, is it possible to generate an *entire* compiler automatically? It is a question that many computer scientists have attempted to answer, and to differing degrees of success. In this project, we will consider another approach, much closer to the specification style of `yacc`, and based on *operational semantics*.

1.1 Semantics and Compilers

A programming language may be specified formally in a number of different ways:

- *Axiomatic semantics* – the behaviour of a program is defined by an inductive set of axioms over the syntax of the language, describing the properties that hold before and after execution.
- *Denotational semantics* – a program is mapped onto a mathematical object by an inductively defined interpretation function, over the syntax of the language.
- *Operational semantics* – a transition relation defines an evaluation step from one term in the target language to another. There are two notable approaches to this:
  - *Structural Operational Semantics*[1] (Plotkin [12]) – the transition relation defines one execution step, equivalent to one transition in an abstract machine whose states are target language expressions. The result of evaluating an expression is the final state (that can make no more progress) in a sequence of transitions from it.
  - *Natural Semantics* (Kahn [4]) – the transition relation defines the execution of an expression to a value in just one step. This is expressed in the style of Gentzen's natural deduction system, hence the name 'natural' semantics.

[1]Structural operational semantics is often called *small-step*, in contrast to the *big-step* natural semantics.

Automated tools have been investigated for each of these paradigms. Axiomatic schemes such as Floyd-Hoare logic may be embedded in an automated theorem prover, so that we may prove properties of programs in a given language. The problem with this approach, from the perspective of a compiler, is that we abstract away from the implementation – the semantics describes what can be *proven* about a program, not how it is *executed*.

Similarly, a denotational semantics tells us little about execution strategy – a program's denotation is a mathematical object corresponding to the *result* of execution. Work has been done to automatically generate compilers from such semantics, e.g. Paulson [8] uses attribute grammars, specifying compilation to lambda calculus formulae over domains, which are executed on an SECD-style machine. Whilst this is a viable approach to compiler generation, the use of denotational semantics depends on a knowledge of domain theory, and is not as accessible as one might like. This, and its problems in dealing with concurrency, has resulted in a decline in popularity.

In this project, we will focus entirely on operational semantics. They provide a much better description of execution strategy, and are fairly widespread in use.

1.2 Existing Work

The existing techniques for compiler generation vary greatly, depending on the aim of the work[2]. In the context of operational semantics, one of the earliest attempts was the Typol language [2] (for natural semantics specifications) coupled with the Centaur system [1] for compilation to Prolog. This is quite natural, since we can express inference rules as Horn clauses, and so Prolog can determine the valuations that satisfy a query using its resolution-based search strategy. It is also nice and flexible, since type inference and type checking reduce to the same problem, but the system is very slow, and specifications are constrained by Prolog's left-to-right and depth-first execution strategy.

The more recent Relational Meta-Language (RML) [9], attempts to solve these efficiency problems. RML is essentially a declarative language, in the style of natural semantics, and the `rml2c` compiler translates this into C (in a continuation-passing style). This means that relations are *directional* (unlike Typol) so that they can be implemented as procedures[3]. Great claims have been made about the applicability of RML, for example to generate a `javac`-style compiler from its semantics. Such a 'semantics', however, simply describes a syntactic translation into bytecodes, and therefore does not capture the meaning of *executing* a Java program.

A quite different approach is taken by Diehl [3], whose emphasis is on provably correct (as opposed to efficient) compiler generation. He uses a pass separation-based approach to generate both an abstract machine and a compiler from a natural semantics specification. This undergoes a number of provably correct transformations, for example *(i)* augmenting states in the abstract machine with a 'temporary variable' stack, and *(ii)* sequentialising rules by adding transitions between the start and end states of successive predicates. The result is that each rule is a sequence of named transitions on an abstract machine, and the compiler is just a (recursively defined) syntactic mapping from terms to these transitions.

Pierce's TinkerType [6] illustrates yet another perspective to the problem. Instead of attempting to produce an entirely automated system, as with other work, the aim is primarily

[2]Only a few notable examples of such work are presented here; a good, though somewhat old, survey is given in [5], and in Chapter 3 of [3].

[3]An inductively defined relation corresponds to a recursive procedure.

to provide a modular and extensible system for organising the implementation and typesetting of inference rules. It is left to the user to provide the ML code for the body of a rule, or the TEX source to typeset it. The TinkerType system also carries out some consistency checks, and allows easy maintenance, generation of working implementations, and pretty-printing of rules.

Despite the many different approaches, we still do not know how to generate an efficient compiler without sacrificing the generality and flexibility of natural semantics. In this project, our aim was to address some of the issues in the above techniques, and we feel that we have succeeded in doing so. Let us now turn to this approach.

1.3 SemCom – The Semantics Compiler

Throughout the rest of this document, we will present our attempt at this problem – by no means do we claim to have solved it, but we do believe that our work shows some new insight, and is an interesting approach to compiler generation. The result of this work is the SemCom compiler, which translates a specification in our Semantic Description Language (SDL) to an interpreter in OCaml or F#. The main features of SDL are:

1. **Syntax-directed specification** – rules are given in terms of *concrete syntax* and a lexer and parser are automatically generated for the language. The syntax of SDL reflects this, being in many ways an extension of `yacc`.

2. **Close to natural semantics** – unlike RML, SDL does not impose ordering constraints on the premises of rules, and performs its own internal sequentialisation. Constructs such as conditionals that require more than one rule are handled implicitly, using a backtracking engine to ensure that all possible rules are attempted.

3. **Expressive power** – SDL is powerful enough to express many language features, including polymorphic type systems, capture-avoiding substitution, and mutable stores. The system is general enough to accept either small- or big-step semantics. SDL has been used to express a reasonable subset of SML.

In the following chapters, we will discuss the design and implementation of SemCom. As a rough guide to the structure:

- *Chapter 2* discusses the design of SDL and its type system, along with the framework for analysing semantics for language features.

- *Chapter 3* describes the implementation of SemCom, from the module structure and segregation of functionality, to the algorithms and optimisations we have implemented.

- *Chapter 4* provides examples of SemCom's functionality, and we evaluate its features and performance.

- *Chapter 5* concludes by reflecting on what we have achieved.

Chapter 2

Preparation

Designing a semantics compiler requires careful consideration, and it is important to decide on its capabilities at an early stage. Initially, we restricted ourselves to implementing *two-level big-step semantics* (see §2.1.6), but since the resulting system is general enough to implement small-step semantics too (see §3.7), our design is relevant to both. In this chapter, we will examine:

1. The design of the meta-language SDL.
2. The type system of SDL.
3. Analyses for inferring higher-level operation of semantics.
4. The execution engine for type checking and interpreting/compiling.

2.1 Design of the Meta-Language

SemCom takes as input the semantics of a programming language, and so we need a *meta-language* for expressing this. To have the full power and expressibility of natural semantics we could use a higher-order logic, but then SemCom would need to be a general-purpose theorem prover. Clearly this is too general, so we *restrict* the meta-language to make it manageable, but still expressive enough to describe useful languages. The result is the language SDL, which we describe below.

E	:	Target language expressions. Target language *values* are a subset of expressions.
S	:	Semantic objects (see §2.1.3).
Γ	:	Type environments (a form of semantic object S).
T	:	Target language types (a form of semantic object S).

Figure 2.1: Notation

2.1.1 Evaluation and Typing Relations

SDL requires the definition of separate static and dynamic semantics, which is a standard design choice (e.g. Milner [7]). This means that the user must define two distinct relations. Using the notation in Figure 2.1:

1. An **evaluation relation**, '$\Rightarrow$' $\subseteq (S_1 \times \ldots \times S_n \times E) \times (S_1 \times \ldots \times S_n \times E)$. Since we treat values as a subset of expressions, this definition applies to both big-step and small-step semantics.

2. A **typing relation**, ':' $\subseteq \Gamma \times E \times T$.

For simplicity, SDL limits us to defining *only* these relations, since they correspond to the type checking and evaluation phases of an interpreter.

These definitions are fine mathematically, but not very useful for actually defining the relations. We do so by an unordered set of *rules*, defined inductively over the abstract syntax of the target language. Here, $\Gamma \vdash \sigma\ R\ \pi$ is a *sequent*, meaning that the property π holds of the state σ (i.e. $(\sigma, \pi) \in R$) given the set of assumptions Γ. These rules have the general form (for arbitrary relation R):

$$\frac{\Gamma_1 \vdash \sigma_1\ R\ \pi_1 \qquad \ldots \qquad \Gamma_n \vdash \sigma_n\ R\ \pi_n}{\Gamma \vdash \sigma\ R\ \pi}\ \text{if } c_1 \wedge \ldots \wedge c_m$$

The rule itself states that the *conclusion*, $\Gamma \vdash \sigma\ R\ \pi$, holds if and only if each of the *premises*, $\Gamma_i \vdash \sigma_i\ R\ \pi_i$, and *conditions*, c_j, hold.

More specifically, for the evaluation and typing relations, we have:

1. Evaluation Relation:

   - Assumptions Γ are the empty set, since nothing is assumed of variables in expressions[1]. We omit writing $\{\} \vdash$ for evaluation rules.
   - States σ are terms in the target language, augmented with a list of *semantic objects* (see §2.1.3), for example a mutable store.
   - Properties π are the final results of evaluating σ, and must have the same type[2] as σ.

2. Typing Relation:

   - Assumptions Γ are type environments (partial finite maps from variables to types).
   - States σ are expressions in the target language.
   - Properties π are types in the target language.

Conditions take the same form for both relations, and are propositions in quantifier-free first-order logic. They restrict the applicability of the rule, by providing constraints on the variables in the sequents. Their syntax is:

$$\boxed{c ::= \mathbf{T} \mid \mathbf{F} \mid c \wedge c \mid c \vee c \mid \neg\ c \mid A(x_1, \ldots, x_n)}$$

Here, A are predicates over metavariables x_i.

[1]One might like the use of such assumptions to maintain, for instance, the assignment of free variables to values. However, it is more conventional to do this either by substitution, or use of an immutable store, so we are not losing generality.

[2]Specifically, this is a type in the meta-language and not the target language, so that the evaluation relation maps states onto states. See §2.2.

2.1.2 Target Language Syntax and Values

An important constraint imposed by SDL is that there must be precisely one typing rule for every term in the target language's syntax (the conclusion of each rule corresponds to a syntactic production). In particular, there must be a bijection between the concrete and abstract syntax of the language, and SemCom uses this knowledge to automatically generate a parser[3] (see §3.4).

Whilst the typing rules are used to infer the language's *syntax*, we 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 a function abstraction, for languages like ML. On the other hand, if the expression matches a rule, but subsequently fails (e.g. a condition is not satisfied), then we treat this as a *stuck* state, resulting in evaluation failure if we are unable to backtrack (see §2.4). 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[4], whereas we have a sequence of transitions for a small-step semantics. In this case, we apply the relation until we reach a fixed point (a normal form), such that we cannot reduce the expression any further.

2.1.3 Semantic Objects and Expressions

The phrase 'semantic object' is somewhat ambiguous, so let us begin by clarifying this. In general, we are dealing with two varieties of object:

1. *Syntactic objects* – fragments of target language syntax, possibly containing metavariables that range over all possible expressions. For example, the expression '`if` e_1 `then` e_2 `else` e_3' will match any expression with a conditional as the root of its abstract syntax tree. The capture-avoiding substitution of the expression e_1 for the identifier x in the expression e_2 is written as $\{e_1/x\}e_2$.

2. *Semantic objects* – mathematical, set-theoretic objects. A semantic object may be one of the following:

   (a) **Partial finite map** – an ordered relation $f : A \rightarrow B$, used for stores, type environments etc. SDL restricts A to be the set of target language identifiers (since this is how maps are usually used), and allows the following map operations:

   - *Update* – $\{a \mapsto b\}f$ returns a new map, f', with the same mappings as f, except that a now maps to b instead of $f(a)$.
   - *Lookup* – $f[a]$ returns the object to which a maps, in f.
   - *Domain* – $dom(f)$ returns the domain of f as a set.

   (b) **Set** – an unordered collection of values, used for sets of free type variables, enumeration types etc. SDL supports the standard set operations (union, intersection, and difference). The syntax of these *user-defined* sets is:

   $$\boxed{S ::= \{v_1, \ldots, v_n\} \mid S \cup S \mid S \cap S \mid S \backslash S}$$

   where v_i are values.

[3]This may be a serious limitation for more complex languages/type systems, but it was felt that, for simple languages, this decision greatly eases their specification.

[4]We perform an analysis (see §2.3.2) to check that this is the case.

$$\boxed{\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{e_1 \div e_2 \Rightarrow v} \quad \text{if } \neg(v_2 = 0) \wedge v = \mathit{divide}(v_1, v_2)}$$

Figure 2.2: An example evaluation rule, illustrating the overloading of '='

(c) **Pair** – a pair of semantic objects constructed by $P = (S_1, S_2)$. The projection operators #1 and #2 extract the elements of a pair, such that $S_1 = \#1(P)$ and $S_2 = \#2(P)$.

We refer to variables in SDL as *metavariables*, and these could potentially be any of the objects above. To avoid excessive user-annotation, SEMCOM performs type inference – we describe the type system in §2.2.

2.1.4 Conditions and Constraints

All the real work done by a rule (i.e. computations, store updates etc.) is contained in its side conditions. These are constructed from predicates, which may be user-defined (see §2.1.6), or one of the following:

1. $v \in S$ – tests whether the value v is an element of the set S.

2. $x = e$ – tests whether the metavariable x is equal to the object e (which may be the result of a user-defined function, described in §2.1.6).

3. $\mathit{is\_value}(v)$ – tests whether the expression v is a value in the target language (see §2.1.2).

The above definition of equality is consistent with the view of these rules as relations. However, remember that SEMCOM must implement these, and so the conditions must be tested in some order. Although, from the user's perspective, equality is *relational*, in the implementation we consider it to be overloaded – that is to say, the first time a metavariable is assigned to it is *instantiated*, and any subsequent conditions are equality *tests*:

$$\mathtt{x} = \mathtt{e} =_{\mathit{def}} \begin{cases} \mathtt{x} := \mathtt{e} & \text{if } x \text{ is not defined} \\ \mathtt{assert}(\mathtt{x} = \mathtt{e}) & \text{otherwise} \end{cases}$$

An example of such overloading of '=' is given in Figure 2.2.

This leads us to consider, from an implementation perspective, that:

1. A metavariable is *defined* if it appears in the initial state σ of the conclusion, the final state π of a premise, or on the left-hand side of an equality condition.

2. A metavariable is *used* if it appears in the final state π of the conclusion, the initial state σ of a premise, or anywhere else in a condition.

From what we have just said, it is tempting to believe that SDL is a dressed-up declarative language, like RML. It is true that we consider relations to be *directed*, i.e. a rule is 'called', its argument being the initial state of the conclusion, and its return value the final state. Unlike RML, however, the order does not have to be defined by the user – all that we require is that there is *some* order, such that all metavariables are defined before they are used. This well-formedness requirement is enforced by SEMCOM when it sequentialises the rules (see §2.3.1).

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`
	$\neg(x = add(y, z))$	`~(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 2.3: SDL syntax: mathematical (typeset) notation and concrete syntax

2.1.5 A Note About Types

In SDL, types are represented as sets[5] (described above), but we also provide some in-built syntactic types (corresponding to the primitive types of OCaml), so that we have the syntax:

$$\boxed{T ::= S \mid \textbf{int} \mid \textbf{float} \mid \textbf{bool} \mid \textbf{char} \mid \textbf{string} \mid \textbf{unit} \mid T_{var} \mid T\ T_{cons} \mid T\ T_{cons}\ T \mid \forall A.(T)}$$

User-defined sets, S, are enumeration types in this context – since they are represented differently (see §3.6.2), we don't allow the standard set operations on arbitrary types. Type constructors T_{cons} form labelled types in the unary case, and labelled Cartesian products in the binary case.

Since SDL allows polymorphic type systems, type variables T_{var} (implemented via mutable variables), and type schemes $\forall(A)T$ (where A is a set of type variables) are allowed. In fact, SDL makes dealing with type variables very easy, since these are just metavariables – unconstrained 'type' metavariables are treated as type variables, and when they *are* constrained, we implicitly perform *unification*. As an example, consider this SDL rule for function abstraction (the syntax of SDL is described in Figure 2.3):

```
{x |-> T}G_ |- e : T'
-------------------------------[fn]
G_ |- [[fn \x => \e]] : T -> T'
```

Here, the type T is implicitly instantiated to a type variable in the premise, and will be unified if the type of x is constrained in the expression e.

Type schemes are also easy to deal with, since *specialisation* (i.e. loss of quantifiers and fresh instantiation of type variables) is an implicit operation – if we try to constrain a type to a type scheme, the type scheme will be specialised. We also need to construct sets of free type variables (see, for example, Figure 2.4), and SDL provides an in-built function for doing so:

$$ftv(S) =_{def} \begin{cases} \text{free type variables in } S, \text{ if } S \text{ is a type} \\ \text{free type variables in all types in the range of } S, \text{ if } S \text{ is a type environment} \end{cases}$$

[5]It would be more accurate to represent types as domains, since the value that an expression evaluates to may be undefined (for example, a non-terminating loop). Since we are dealing with evaluation operationally, however, sets are an adequate representation.

```
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 2.4: An SDL type rule for `let`-polymorphism

2.1.6 Two-Level Semantics

In the spirit of Diehl [3], SDL requires a *two-level* semantics. This means that we abstract away from the definition of auxiliary operations in the semantics, by placing them in a *second-level library.* We do this because there are two interpretations of such operations – let us consider $add(x, y, z)$ as an example. The mathematical (predicate calculus) interpretation of *add* is the set $\{(x, y, z) \mid x = y + z\}$. However, our *implementation* interprets this as an OCaml function, taking two of the arguments, and computing the third. We abstract in this way to avoid having these implementation details in the semantics.

These *second-level functions* are implemented in a user-defined library, though SemCom provides a library of standard routines. They may be used as predicates, if they return a boolean. We refer to the application of such functions as *second-level calls.*

2.1.7 The Semantic Description Language – SDL Syntax

We have so far described SDL fairly completely, with its syntax given by example (see Figure 2.3, and the example rule (for `let`-polymorphism) in Figure 2.4). The full grammar is described in BNF, in Appendix B. An SDL (`.sem`) file has the general form:

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

The style directive is optional, and may be `%big` (the default) or `%small` to specify big- or small-step semantics respectively. To generate a parser, we also need to know the precedence (and associativity) of tokens in the language, presented in the usual `yacc` style, and this is placed at the end of the file.

2.2 Type System of SDL

Metavariables in SDL can describe many different sorts of object (see §2.1.3), and this makes type inference quite a complex task. It is a necessary task, however, since it provides vital information to the code generator (see §3.6), as well as catching many errors at compile-time. To cope with this complexity, we define two hierarchies of types:

1. *Top-level types* – whether an object is a map, or a set, etc.

2. *Specific types* – whether a map is from identifiers to values, or to types, etc.

The reason for this distinction will become apparent, as we examine SDL's type system in more detail.

2.2.1 Top-Level Types

All SDL expressions must have a top-level type (`ttype`), and it will cause a type error if SemCom is not able to infer it. These types are the constructors of:

```
type ttype  =  Expr_t of etype                      (* expressions *)
            |  Map_t of ttype                       (* maps (from ids) *)
            |  Set_t of stype                       (* sets *)
            |  Pair_t of ttype * ttype              (* pairs *)
            |  TypeCon_t of int                     (* type constructors *)
            |  Call_t of valtype list * valtype     (* second-level calls *)
```

For the most part, the type of a metavariable is clear from its context, and this is always the case for type constructors and second-level calls. Since different top-level types do not unify, we can type-check at this level in the standard way (as described in §3.3). The exception is if a metavariable is not used within a rule, but forms part of the relation, in which case global information about the relation's type (see §2.2.6) can be used.

2.2.2 Expression Types

For implementation purposes, there are two special cases of expression metavariables – those that are SDL values, and those that are target language identifiers. In other words, the expression types are:

```
type etype  =  Value_t of valtype   (* values *)
            |  Id_t                 (* identifiers *)
            |  Expr_t               (* general expressions *)
```

These form a simple sub-typing relation, with `id` <: `expr` and `value` <: `expr`. We assume that an expression metavariable is of type `expr` unless its context restricts this – for example, only metavariables of type `value` may be arguments of a second-level call.

The types of SDL values are:

```
type valtype  =  Int_t
              |  Float_t
              |  Bool_t
              |  Char_t
              |  String_t
              |  Unit_t
              |  ValVar_t of int
```

This allows the target language to interact with SDL – they share the same primitive values. Thus if a rule contains the side condition '$b \in$ `bool`', this restricts the values that b can take *in the target language* to booleans.

As far as type inference goes, we allow polymorphism in second-level functions (see §2.2.5), and therefore use type variables for unification (`VarVal_t`). If a metavariable still has variable type after unification, it is allowed to be any SDL value.

2.2.3 Map Types

The domain of a partial finite map in SDL must be the set of target language identifiers, and so we are left to infer the type of its range. These elements must unify in their top-level type, but otherwise they may differ. For example, a store will likely map to values of all types, and even other expressions (such as function-values). In this case, we want the smallest common *super-type* of the elements to be the type of the map's range. We therefore perform *anti-unification* (see §3.3) on the elements of the map, to find the least-general anti-unifier, i.e. the most specific type that encompasses the types of all the elements.

2.2.4 Set Types

Although we have seen that target language types have a more specific representation than arbitrary user-defined sets, the SDL type system makes no such distinction. The type of a set is characterised by the type of the elements it contains. Thus the set $\{1, 2, 3\}$ has the type `int set`, for example. We also allow the types of these elements to differ, so long as they have a common super-type, so {1,'1',"*one*"} would be given the type α `set` (where α ranges over values). The built-in types are assigned SDL types similarly, except that we flag them with a *set qualifier* to distinguish them from user-defined sets. So the type of `int` becomes $\texttt{int}_\infty$ `set`.

This may seem quite confusing, but remember that '`int`' is just a syntactic type, and is *not* represented internally as a set. We treat it as a set of integers when type checking (at the SDL level), so that type variables can range over both syntactic types and 'real' sets (enumeration types) – i.e. they have a common supertype. In this way, all the target language types are thought of as sets by the SDL type checker, so that they have a uniform representation:

```
type stype  =  Set0_t of valtype * set_qual    (* base set *)
            |  Set1_t of int * stype           (* unary type cons *)
            |  Set2_t of int * stype * stype   (* binary type cons *)
            |  IdSet_t                         (* identifiers *)
            |  TypeIdSet_t                     (* type identifiers *)
            |  TypeVar of int                  (* type variables *)
```

Here, we see that differing type constructors result in differing SDL types, which allows the type system to distinguish between them. In addition, we have types for sets of target language identifiers and type variables. SDL type variables are used for unification, but are not necessarily resolved.

As in the case of maps (see §2.2.3), we must allow for the type of a set to become less specific (e.g. if we union a set of integers with a set of characters to get a set of *values*). In other words, type checking a set proceeds by anti-unification of the types of its elements.

2.2.5 Call Types

The type of a second-level call is inferred from the OCaml interface file (`.mli`) of the library (see §3.2.2), and therefore this is largely straightforward on the part of the type checker. The only special case is for polymorphic SDL functions in the library, and we discuss this in §3.3.

2.2.6 Relation Types

The type of the evaluation and typing relations must be consistent across all the rules. We therefore perform global anti-unification across each, to ensure consistency and to determine what the type is. We *anti*-unify because if a given rule makes a more specific typing assumption than the entire relation, it may fail to evaluate when presented with some result of one of its premises. As usual, top-level types must unify across the entire relation.

2.3 Analysis of Semantics

To generate an interpreter we not only need to *transform* the semantics into an executable form, but also to *analyse* it for the higher-level language features we are implementing. For instance, which terms are *conditionally* executed, which variables are *bound* in an expression, and which terms are *values*? In this section, we describe how to infer this information – consult §3.5 for implementation details.

2.3.1 Constraint Analysis

The premises and conditions of an SDL rule are unordered, but their execution is necessarily ordered, so we need to *sequentialise*[6] them. But how do we decide on a 'correct' ordering? Recall from §2.1.4 that rules have a well-formedness requirement – there must be some ordering of premises and conditions such that all metavariables are defined before they are used. To find this ordering, we perform a *constraint-based analysis.*

The analysis itself is quite simple – starting with the set of metavariables in the initial state of the conclusion, we simply examine all the premises and conditions until we find one in which all used metavariables are already defined. This is emitted, and the metavariables *it* defined are added to our set. This is repeated, until all premises and conditions have been emitted (and are therefore sequentialised), or we can make no further progress, in which case we report an error in the semantics.

As an example, consider a mis-typed SDL rule for sequencing:

```
<e1,s''> => <[[skip]],s'>;
<e2,s'> => <v,s''>
----------------------------[seq]
<[[\e1 ; \e2]],s> => <v,s''>
```

Here, the constraint analysis will fail, since s'' depends on s' and vice-versa, so we cannot find a suitable ordering of the premises. We can therefore catch this sort of typographical error.

[6]This is similar in style to Diehl [3], except that we transform into a continuation-passing form.

2.3.2 Value-Reduction Analysis

We have seen, in §2.1.2, how values in the target language are implicitly defined by the evaluation relation in SDL. For big-step semantics, evaluation takes place in a single step, so if the user makes a mistake, and some rule does not reduce to a value, some expressions will not evaluate completely. Luckily, we can perform a simple analysis to detect this in most cases, and emit a warning if so.

We examine the result of each rule's conclusion, and test whether it *could* match another evaluation rule. If the conclusion is a metavariable, and is not obviously a value, we see whether it is the result of a premise – based on the inductive hypothesis that all other rules reduce to values, this one must also. If any of these tests fail, we emit a warning.

2.3.3 Multiple Match Detection

We can detect conditional execution by looking for two or more rules that match the same expression. For example, an `if`-statement has two rules (in big-step semantics), for whether the guard evaluates to `true` or `false`. When we implement these rules, we cannot rely on OCaml's pattern matching, otherwise the second alternative will never be attempted. We therefore combine these into a single clause, and have a backtracking mechanism (see §2.4) to attempt each one in turn.

2.3.4 Binder Detection

To implement capture-avoiding substitution, we need to know what the *binders* are. We use a fairly accurate heuristic for this, by looking at type environment updates in the typing rules. If a variable is *bound* within an expression (i.e. its scope is limited), then it gets updated in the expression's type environment. This is best illustrated by example – consider the following SDL rule for recursion:

```
{x |-> T1 -> T2, y |-> T1}G_ |- e1 : T2;
{x |-> T1 -> T2}G_ |- e2 : T
---------------------------------------------------------[letrec]
G_ |- [[let val rec \x = ( fn \y => \e1 ) in \e2 end]] : T
```

Here, by looking at the Γ (`G_`) updates, we see that x and y are in scope in e_1, but only x is in scope in e_2. Since we identify these instances of binders, we can construct a substitution function that only substitutes for *free* instances of an identifier in an expression.

2.4 Execution Engine

As we have mentioned, SemCom considers the evaluation and typing relations to be directed. This means that the type checker and evaluation engine can execute recursively over the abstract syntax of the target language. One approach would therefore be to implement the evaluation relation as a recursive `eval` function, which calls itself to evaluate the premises of a rule. However, because of the limited OCaml runtime stack, and issues concerning backtracking, SemCom uses its own execution stack. In fact, we have two stacks, as illustrated in Figure 2.5:

1. *Evaluation stack*, ϕ – when a rule is fired (i.e. it matches the expression we are evaluating), we place its premises and conditions (ordered as per §2.3.1) onto this stack. The top element of ϕ is the next one to execute.

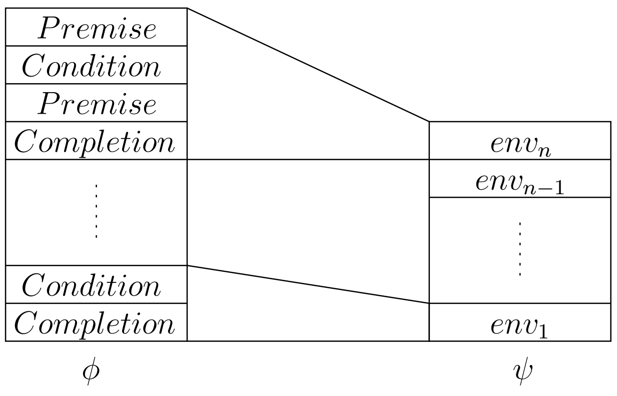

Figure 2.5: The evaluation and environment stacks

2. *Environment stack*, ψ – the top element of this stack is the current environment; a mapping from metavariables to semantic and syntactic objects. A new environment is pushed onto ψ whenever a premise is executed, since this is a new instance of a rule. When a rule completes, it removes its environment from ψ, and updates the previous environment with its result. This is done by a *completion function* (described below); a form of continuation.

The elements of ϕ fall into three categories:

1. **Premise**(m,f) – evaluate the expression m, using the completion function f, which updates the previous environment with the result of the premise.

2. **Condition**(f) – evaluate the function f, which returns successfully if the condition is true, and raises an exception if not (i.e. the rule cannot be applied).

3. **Completion**(m,f) – execute the completion function f, with the constructed expression m.

Note that 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)[7]. From a theorem-proving perspective, we can also think of the evaluation stack as a set of goals to prove.

Backtracking is implemented with a third stack, β, for expressions that match multiple rules (see §2.3.3). When such an expression is executed, we keep a record of how many elements to discard from ϕ and ψ to return to this state, on β. So, if the rule fails, we backtrack and attempt the next rule in sequence. If all rules fail, we consider execution to have failed, and the interpreter aborts with an error. If no rules match an expression, we treat it as a value, and apply its completion function immediately.

The interpreter iteratively pops and executes the top element of ϕ, until it contains just one element (a completion), and ψ just one environment, from which the final result of evaluation can be constructed.

[7]More precisely, the evaluation stack is a stack of local stacks, separated by completions.

2.5 Languages And Tools

From the outset, the language OCaml was chosen for the development of SemCom. Due to its primitive support for pattern matching and higher-order functions (in combination with imperative features), it is the perfect language for such syntax-directed work. Lexers and parsers (for both SemCom itself, and for those that it generates) are written using `ocamllex` and `ocamlyacc`, provided with the OCaml distribution. The recent F# programming language, developed at Microsoft® Research, is also supported by SemCom, with the option to generate code for either platform.

Initially, as stated in the proposal (see Appendix G) it appeared that Fresh OCaml would be a more suitable language, due to its primitive support for abstraction and α-conversion. However, we did not need these facilities, since we implement capture-avoiding substitution ourselves (see §2.3.4).

A project of this scale requires some degree of management, and this is done so using the standard Unix tools – compilation is controlled by makefiles and the `make` utility (indeed, SemCom provides the option to generate a makefile automatically), and version control is provided by `cvs`, with the repository on a remote server for additional safety.

Chapter 3

Implementation

SEMCOM's implementation is in many respects similar to that of a standard programming-language compiler; the difference being that we compile an operational semantics into an interpreter, rather than a program into assembler. The main stages of processing an SDL file are shown in Figure 3.1, and these may be described approximately as follows:

1. *Lexing and parsing* – the SDL (`.sem`) file, and the interface of the second-level library (`.mli`) file are tokenised and parsed into an abstract syntax tree, with the option of generating TEX output (a `.tex` file).

2. *type checking* – the abstract syntax tree of the semantics is type checked, and the inferred type environment is attached to each rule, with debug information placed in a `.out` file.

3. *Syntax-directed transformation* – the lexer, parser and header files for the interpreter are generated.

4. *Semantics-directed transformation* – the analyses described in §2.3 take place, transforming the semantics through a number of intermediate representations.

5. *Code generation* – executable code for the interpreter is emitted, for either OCaml (`.ml`) or F# (`.fs`).

The resulting code, when compiled, is an interpreter for the target language.

In this chapter, we will examine each of these phases in more detail, but let us begin by considering the module structure.

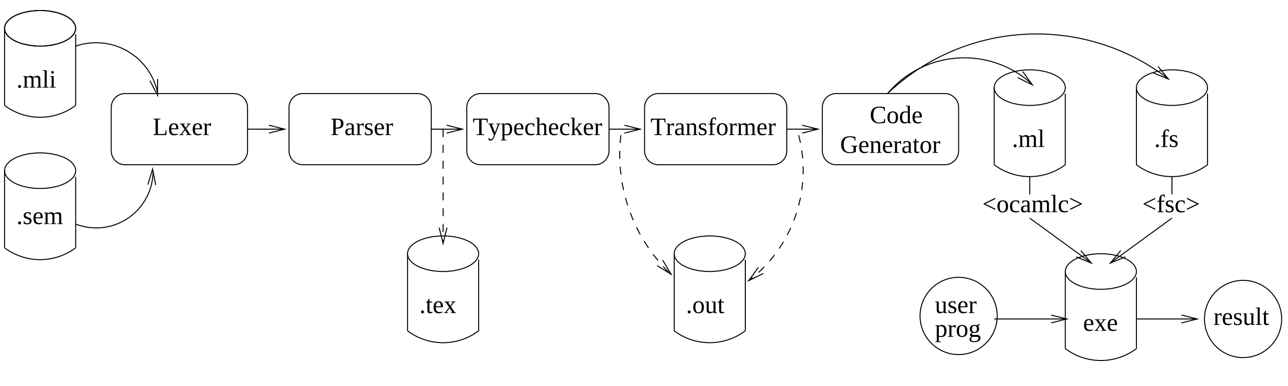

Figure 3.1: The main execution stages of the SEMCOM compiler

3.1 SemCom – From Design to Realisation

A compiler consists of a number of phases, each dependent on the previous, and so its implementation is naturally linear. The same is true for SemCom, and so we followed the *waterfall model* of development, breaking down the design into separate modules (see §3.1.1). Each module was then implemented and tested in isolation (see §3.1.2), before being integrated and tested as a complete system.

After implementing SemCom in this way, we were then able to add additional features in an iterative fashion, using the *spiral model* – for example, the ability to emit F# code, and to deal with small-step semantics (as well as big-step). For each extension, we analysed the new requirements, and the design changes to SemCom, before implementing and testing the system as a whole.

3.1.1 The Module Structure

We present the modules of SemCom in Figure 3.2 – an arrow indicates that a module uses another, except for `semcom_strings` and `semcom_types`, which are used by nearly all. The function of each module is described below:

- *Header files*
  - `semcom_strings` – string pools and general functions for string creation.
  - `semcom_types` – definitions of datatypes used by SemCom, including the type system, abstract syntax, and intermediate representations.
- *SDL parsing and type checking*
  - `semcom_lexer` – lexer for SDL (`.sem`) file.
  - `semcom_parser` – parser for tokenised SDL file.
  - `semcom_typechecker` – SDL type checker.
- *Library parsing*
  - `semcom_lib_lexer` – lexer for library interface (`.mli`) file.
  - `semcom_lib_parser` – parser for tokenised library interface file.
- *Syntax-directed generation*
  - `semcom_headergen` – header file (abstract syntax and pretty-printing) for target language.
  - `semcom_lexgen` – lexer generation for target language.
  - `semcom_parsegen` – parser generation for target language.
- *Utility generation*
  - `semcom_latex` – latexifier for parsed semantics.
  - `semcom_makegen` – makefile generation for emitted files.

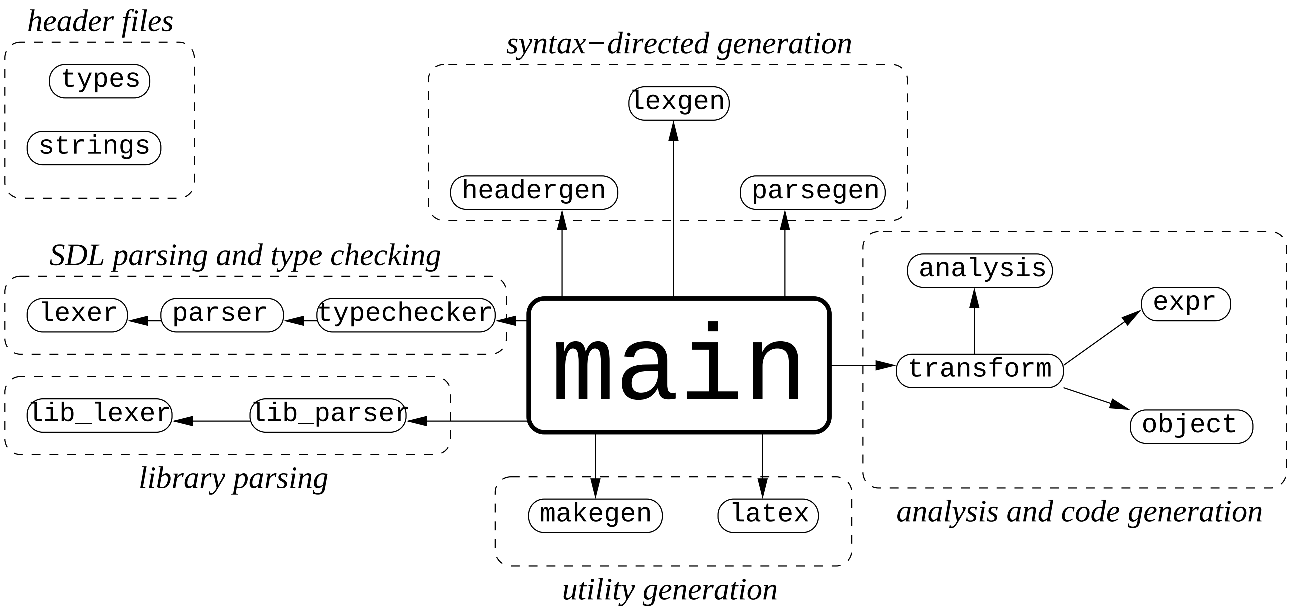

Figure 3.2: The modules of SemCom

- *Analysis and code generation*
  - `semcom_analysis` – analysis of semantics as per §2.3.
  - `semcom_expr` – internal parsing of target language expressions.
  - `semcom_object` – code generation for semantic objects and conditions.
  - `semcom_transform` – integration of analyses, and generation of overall interpreter.
- *Integration*
  - `semcom_main` – integration of the above modules, and command-line interface (described in Appendix A).

3.1.2 Testing Infrastructure

To allow testing of each phase of SemCom as soon as it is finished, we have taken great care to generate debugging output along the way. Of the output files shown in Figure 3.1, we have:

1. *LaTeX output* (`.tex`) – the output from the lexer/parser is typeset, to confirm that the source file has been correctly parsed (see §3.2.4).
2. *Debug output* (`.out`) – debug information from various intermediate phases is placed into this file, when the `-v` command-line option is specified:
   (a) Output from the library lexer/parser, showing the types of second-level functions.
   (b) Output from the type checker, showing the inferred types of metavariables.
   (c) Output from the internal parser, showing the parsings of target language expressions.

3. *Code output* (`.mll`,`.mly`,`.ml`) – the parser, lexer, and header-file generation modules produce independent output, so may be independently tested. The main code generation is more complex, but this has been tested with the language SiMpL (see Appendix C), and we have run a variety of programs through the generated interpreter.

In addition to this, SemCom produces useful error messages by remembering the position of tokens in the source file (see §3.2.3), so that we may tell the user precisely where an error occurred. This makes it easy to test how SemCom responds to incorrect input.

3.2 Lexing and Parsing

SemCom contains two lexers/parsers, both written in `ocamllex` and `ocamlyacc` respectively. The main parser processes the SDL (`.sem`) file, and a smaller auxiliary parser the library's interface (`.mli`) file.

3.2.1 The SDL Parser

The syntax of SDL is not trivial, but it is not overly complex, since most of the difficulties are handled by the type checker. We previously described the main syntactic features of SDL (see §2.1.7), and the parser largely reflects its BNF grammar, given in Appendix B.

The lexer, in addition to tokenising the character stream of the input file[1], maintains token labels (see §3.2.3), and populates a *string pool* and a *keyword pool*. Internally, we represent metavariables and target language keywords as integers (since this is more efficient), so these pools allow us to convert them back to strings. The lexer uses a separate mode[2] to read target language expressions, which is entered when the characters '`[[`' are seen, and exited on '`]]`'.

3.2.2 The Library Parser

The parser for the library interface (`.mli`) file is much simpler, and presents the SDL type checker with the types of the second-level functions. Since these functions are restricted to primitive types (though they may be polymorphic), this only has to be able to parse declarations like the following:

```
val geq : 'a -> 'a -> bool
```

We store this set of type declarations as a hash table (from function names to SDL types), and this is printed in the `.out` file – the entry for the above `geq` function is:

```
geq |-> call(var_val(-1) * var_val(-1) -> bool)
```

In OCaml, the type variables in top-level functions are implicitly polymorphic, so the SDL type variables (`var_val`) have negative arguments to avoid conflict with the non-bound type variables we use for unification (see 3.3).

[1]As is standard practice, we use a hash table of SDL keywords, to reduce the state space of the lexer.

[2]There are also two additional modes, for comments (to disregard them) and for errors (to continue to read to the end of the line).

3.2.3 Token Labels

To generate useful error messages, the SDL lexer labels every token with a *tag* – an index into a table containing its line number, and start/end characters, in the input file. The parser, when constructing the abstract syntax tree, labels each with a *pair* of tags, so that the start and end position of a term can be reconstructed in an error message. As an example, here is an error message when a parenthesis is mismatched:

```
L2.sem:
Parse error on line 98 at character 6:
<e1,(s$
       > => <[[fn \x => \e]],s'>;
```

3.2.4 Latexification

SemCom has the option of automatically typesetting SDL into LaTeX, using the `-t` command-line option. For example, the rule for `let`-polymorphism presented in Figure 2.4, when transformed into LaTeX, produces:

$$(\text{LET})\ \frac{\Gamma \vdash e_1\ : T \qquad \{x \mapsto \forall A.(T)\}\Gamma \vdash e_2\ : T'}{\Gamma \vdash \texttt{let val}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}\ : T'}\ \text{if } A = \text{ftv}(T)/\text{ftv}(\Gamma) \wedge \neg(x \in \text{dom}(\Gamma))$$

As well as being a useful feature, we can use this to test the SDL parser at an early stage. The semantics of the language SiMpL, presented in Appendix C, have been automatically typeset in this way.

3.3 Type Checking

We have described the type system of SDL in §2.2, and the implementation of this is for the most part fairly standard. The end result of type checking is (assuming the type checker succeeds) an abstract syntax tree, annotated with a type environment[3] for each rule, which assigns types to metavariables. These environments are printed in the `.out` file, with the example for our `let`-polymorphism rule (see Figure 2.4) shown below:

```
(type) let
e1  |-> expr
e2  |-> expr
x   |-> id
G_  |-> map(id,set(typevar_2))
T   |-> set(typevar_1)
T'  |-> set(typevar_0)
A   |-> set(t_id)
```

The type checker recurses over the abstract syntax tree of each rule, propagating the type environment into each call, and returning the updated environment afterwards. There are four main auxiliary functions that it uses:

1. `unify` – this takes a type environment and two types, t_1 and t_2, and attempts to *unify* them. We find the most general unifier – the smallest substitution θ, on type variables, such that $\theta t_1 = \theta t_2$. The algorithm jointly recurses into t_1 and t_2, such that:

[3]Implemented using the OCaml `Map` module.

- If neither t_1 nor t_2 are type variables, and they do not match, then they fail to unify.
- If t_1 is a type variable, and t_2 is not, then substitute t_1 for t_2, unless t_1 *occurs* in t_2. The converse is also true.
- If both t_1 and t_2 are type variables, substitute t_1 for t_2, without loss of generality.

The type environment is updated by applying the resulting substitution, taking care not to substitute for bound type variables (in type schemes).

2. `antiunify` – this is required in order to support the sub-typing of some semantic objects in SDL (notably partial finite maps and sets). The *anti-unifier* of two types, t_1 and t_2, is the smallest common super-type (see §2.2) of the two. If this does not exist, then anti-unification fails. Note that the top-level types (see §2.2.1) have no super-type, so a map and a set will fail to anti-unify, for example.

3. `constrain` – this constrains a metavariable x to a type t in a type environment, by extracting the type of x from the environment and unifying it with t. If x is not in the domain of the environment, we insert a mapping from x to t.

4. `anticonstrain` – this assigns a metavariable x to the *most general* super-type of a given type t, in a type environment.

Under some circumstances, unification is the correct course of action, for example if a metavariable of type *bool* is passed as an argument to a second-level call expecting an *int*, then a typing error should occur, since they do not unify. However, if an *int* and a *bool* are elements of a set, then they should anti-unify, so that we have a set of arbitrary values.

The types of the second-level functions are obtained from the library parser (see §3.2.2). If a polymorphic function is used, it is implicitly *specialised* – i.e. we create a new instance of the function's type, with fresh variables. The function can then retain its polymorphic type elsewhere in the SDL file.

As we described in §2.2.6, the type of the evaluation and typing relations must be consistent across all the rules. This is achieved by global anti-unification, but we may require several passes to anti-constrain the type environment of each rule (so that it will cope with all instances of the relation). The effective algorithm generates the least fixed point of this type, for each relation:

1. type-check every rule in the relation, initialising the global type to the first result. For every instance of the relation's type, anti-unify with the global type, recording whether it changed.

2. Repeat this process, anti-unifying every rule with the global type, until the global type doesn't change.

3.4 Syntax-Directed Transformation

In the previous chapter (see §2.1.7), we described how the concrete and abstract syntax of the language is inferred from the typing rules. More specifically, SemCom generates a header file (⟨*filename*⟩`_header.ml`), defining the abstract syntax of the target language, and a parser, to define how the concrete syntax maps to this. In this section, we will examine how these files are generated.

3.4.1 Header File Generation

The header file contains a few utility functions, such as dealing with string pools, but its main content is the definition of the abstract syntax of the target language, and a function for pretty-printing expressions. For each typing rule, the generator looks at the expression in the conclusion, which is represented as a simple list of metavariables and keywords at that stage. To generate the abstract syntax, we ignore the keywords and produce a numbered constructor on expressions. The pretty-printer is defined recursively over these constructors, returning a string with the original keywords.

As an example, let us once again use the rule for `let`-polymorphism (see Figure 2.4), which has the following expression in its conclusion:

```
[[let val \x = \e1 in \e2 end]]
```

The term in the generated abstract syntax definition is:

```
| Expr14 of id * expr * expr * label
```

And the generated clause of the pretty-printer is:

```
| Expr14(i1,i2,i3,_) -> "let val " ^ print_id env i1 ^ " = " ^
                                      print_expr env i2 ^ " in " ^
                                      print_expr env i3 ^ " end"
```

3.4.2 Automated Parser Generation

Generating a lexer for the target language is a simple task, since we already have a table of the language's keywords (returned by the SDL lexer; see §3.2.1). We treat identifier-like keywords[4] differently, placing them in a hash table. If the lexer finds an identifier, it looks it up in this hash table – if not a keyword, the identifier is placed in string pool, and subsequently manipulated as an integer. Other keywords are detected by separate states. The generated lexer (like SemCom itself) labels tokens with their source file position, so we can generate useful error messages.

The parser is similarly straightforward to generate. The basic grammar is constructed in the same way as the header file generator (basically, the inverse of the pretty-printer), and the precedence of keywords is provided by the user (see §2.1.7). If this information is not sufficient, SemCom will fail with an error, due to its *internal* parser generator, described in §3.5.2.

To generate the lexer and parser, the `-p` command-line option must be used. The user is then free to customise them (so long as they conform to the abstract syntax in the header file), and can choose that the files not be overwritten every time their semantics is compiled.

3.4.3 Makefile Generation

As a simple addition to SemCom, the `-m` command-line option generates a makefile for the emitted files. This is trivial to implement, but is very useful when compiling the generated interpreter.

[4]Matching the regular expression of identifiers: `([A-Z]|[a-z])([A-Z]|[a-z]|[0-9]|'_')*`.

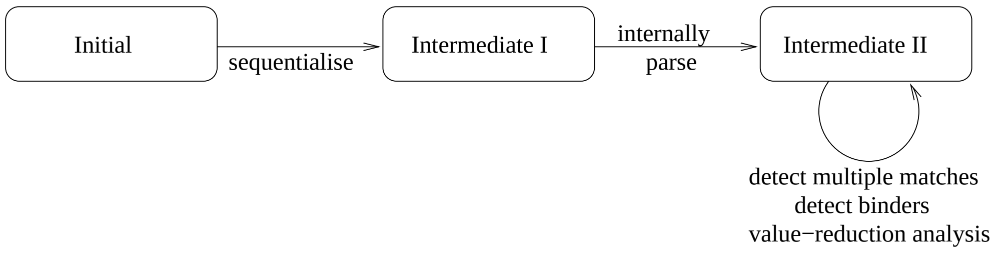

Figure 3.3: Intermediate representations of semantics

3.5 Semantics-Directed Transformation

Between the output of the type checker, and the final code generation phase, we need to transform our semantics, according to the analyses in §2.3. SemCom uses two intermediate representations (*Intermediate I* and *Intermediate II*), and the transformations between them are shown diagrammatically in Figure 3.3. We describe them in more detail below.

3.5.1 Sequentialisation: *Initial → Intermediate I*

We described the constraint analysis in §2.3.1, and this algorithm for sequentialisation (determining instantiation order) is straightforward to implement. Initially, we must split the side-condition (which is one big conjunction of conditions) into a list of conditions, splitting on conjunctions. The bulk of the work is then to generate sets of defined and required metavariables for each premise or condition so that we may compare them to those already defined. The algorithm itself is at worst $O(n^2)$, on the number of premises/conditions, since in the worst case we need to search to the end of the list each time to find one to emit.

The result of this transformation is the *Intermediate I* form, for which each rule contains an *ordered* list of premises and conditions.

3.5.2 Internal Parsing: *Intermediate I → Intermediate II*

Given that an SDL file contains numerous fragments of target language syntax (with metavariables as placeholders), we need some way to deal with these internally. We have described how an `ocamlyacc` parser may be automatically generated (see §3.4.2), but we need to *internally parse* these syntax fragments if we are to generate an interpreter that takes as input the *abstract* syntax of the language.

To do this, we cannot simply use the `ocamlyacc` parser we have generated, since our expressions contain *metavariables*, which should be treated as placeholders accordingly. To solve this problem, we implement a standard LR(1) parser generator and parser, in the module `semcom_expr`. As well as allowing arbitrarily complex expressions in evaluation rules, this has the advantage that SemCom itself can detect conflicts that would occur in the generated parser, and fail with an appropriate error. The result of this transformation is the *Intermediate II* representation, for which expressions have the form `i2_expr`:

```
type i2_expr =
    I2Meta of int * idtype
  | I2Expr of int * (i2_expr list)
```

Metavariables (represented as integers) are labelled with their type, and an expression is an integer with a list of sub-expressions.

3.5.3 Other Analyses and Transformations

The other analyses described in §2.3 operate on the *Intermediate II* form, though the order in which they take place is important. In particular, value-reduction analysis must take place before multiple match detection, as the latter alters the structure of the semantics. Value-reduction analysis (see §2.3.2) does not alter the semantics, but will output a warning if it detects a rule that possibly does not reduce expressions to values. For example, a rule for β-reduction in the λ-calculus ($(\lambda x.e_1)e_2 \rightarrow \{e_2/x\}e_1$) will produce a warning, since this is a small-step reduction.

Multiple match detection, described in §2.3.3, reorganises the rules for the code generator, so that rules with the same matching are combined. In the *Intermediate II* form, each rule contains a list of other rules with the same matching, and this analysis populates these lists.

Finally, binder detection (see §2.3.4) is purely an examination of the typing rules, returning a *binding table* for each rule. This is a map from metavariables to the sets of identifiers that are bound (in scope) within them, and is used to generate the capture-avoiding substitution function, described in §3.6.3.

3.6 Code Generation

The final phase of SemCom is to generate the actual code for an interpreter. We described the execution mechanism in §2.4; the task of generating code for this is quite large, but we may conveniently divide the work into simpler tasks, namely:

1. Code generation for semantic objects and conditions.
2. Code generation for expressions.
3. Generation of header and utility functions.
4. Integration of execution mechanism.

We describe each of these below, but first we will consider the datatypes manipulated by the interpreter.

3.6.1 Representation of Runtime Objects

As we have observed on numerous occasions, SemCom uses a variety of objects, both syntactic and semantic, and the generated interpreter must also deal with these. Since the execution mechanism (see §2.4) uses an environment for each rule, mapping metavariables to objects, the OCaml type system requires that all these objects be the same type (if we are to use OCaml `Maps` for environments). This means that we are forced to *box* all objects into a top-level type, which we call `semobj` (identifiers are represented as integers):

```
type semobj  =  MapObj of semobj IdMap.t
             |  SetObj of setobj
             |  PairObj of semobj * semobj
             |  IdObj of int
             |  ExprObj of ⟨filename⟩_header.expr
             |  ValueObj of ⟨filename⟩_header.value
```

We describe the map, set and pair objects in §3.6.2, and the identifier, value and expression objects in §3.6.3.

3.6.2 Implementing Semantic Objects and Conditions

The semantic objects described in §2.1.3 are implemented using the standard OCaml libraries, since these provide much of the functionality we need:

1. *Partial finite maps* are implemented using OCaml's `Map` module. Since we represent identifiers as integers, we construct our own `IntMap` module for this, using the `Map.Make` functor. The built-in functions `add` and `find` are used for updates and lookups respectively, and we can construct any map as a series of updates to the empty map (`IntMap.empty`).

2. *Sets* are implemented using OCaml's `Set` module. Again, this provides a functor interface, and we create a `ValueSet` module for sets of target language values, and an `IntSet` module for sets of identifiers (or type identifiers). We can then use the built-in operations `union`, `inter` and `diff`, as well as `add` for constructing sets, and `mem` for testing set membership.

   We represent target language types syntactically, which requires a few additional constructors, as shown below:

```
type setobj  =  SUser of ValueSet.t
             |  SId of IntSet.t
             |  STypeId of IntSet.t
             |  SDefined of typeset
             |  SAbst of int * setobj
             |  SVar of varobj ref
             |  SCons1 of int * setobj
             |  SCons2 of int * setobj * setobj

and varobj   =  VarId of int
             |  VarObj of setobj
```

   Furthermore, the built-in types are:

```
type typeset  =  Int_t
              |  Float_t
              |  Bool_t
              |  Char_t
              |  String_t
              |  Unit_t
```

We represent abstraction (for type schemes) and type constructors as one might expect, but our implementation of type variables requires some explanation. Since a type variable may be instantiated in one rule, and only unified to a concrete type many rule instantiations later, we represent it as a *reference* (to a `varobj`). This enables us to back-propagate an update to a type variable – since the field is mutable, updating it in one place updates it everywhere[5].

3. *Pairs* are trivially implemented as OCaml pairs, with the `fst` and `snd` functions for projecting elements.

The code for a semantic object is constructed by recursing over its structure, emitting code in the manner described above. We also implement a few auxiliary functions, for SDL operations such as `ftv` (free type variables), `dom` (domain of a map), and constructing abstractions from a *set* of type variables. These are emitted in the header section of the code.

Conditions are implemented in a similar way, except that we are performing logic tests, i.e. using the OCaml boolean operators (`&&`, || etc.). Thus, a condition is converted into an OCaml expression of type `bool`, and we generate code that raises an exception if the condition evaluates to `false`.

3.6.3 Implementing Expressions

Since we have already internally parsed expressions (see §3.5.2), the remaining code generation is quite simple. Essentially, the interpreter has two uses for expressions:

1. *Pattern matching* – the metavariables in the expression are given place-holder variables, so that we may use it in a pattern-matching position (the variables will be instantiated at runtime). Examples of this are in the initial state of the conclusion, or the final state of a premise.

2. *Construction* – the metavariables in the expression are constructed, i.e. code is generated to lookup their values in the environment[6]. Examples of this are in the final state of the conclusion, or the initial state of a premise.

The main auxiliary function left to describe is that of capture-avoiding substitution. Using the results of the binder detection analysis (see §2.3.4), we know which variables are bound in each rule, and so we can ensure that we only substitute for *free* occurrences of the identifier. As an example, the generated clause in the substitution function for the recursion rule shown in §2.3.4 is:

```
| Expr15(i0,i1,i2,i3,lab) ->
    Expr15((i0),
           (i1),
           (if (id i1 = i || id i0 = i) then i2 else sub_ i2),
           (if (id i0 = i) then i3 else sub_ i3),
           lab)
```

where `i` is the identifier we are substituting for. Notice the test against the binders `i0` and `i1`, and if they match, to not substitute in the expression.

[5]Since SDL requires the typing rules to be syntax-directed (with a unique rule for each term in the syntax), we never need to backtrack – hence we never need to undo a type variable update.

[6]Sequentialisation (see §3.5.1) ensures that the metavariables will all be defined at runtime

```
let lib_add lab i0 i1 =
  (match i0 with
       ValueObj(Int(x)) -> let i0' = x in
         (match i1 with
              ValueObj(Int(x)) -> let i1' = x in
                ValueObj(Int(Semcom_lib.add i0' i1' ))
            | _ -> no_match lab)
     | _ -> no_match lab)
```

Figure 3.4: Generated code for the library function `add`

3.6.4 Utility Functions

In the generated code, we use a number of functions that are invariant to specifics of the semantics. There are a number of such *utility* functions:

1. type checking functionality is provided by a collection of standard functions, including unification (with an occurs-check) of types in the target language, specialisation of type schemes, and the ability to create fresh type variables.

2. Library calls must be interfaced using a wrapper function, since we must unbox the arguments, and box the result. Polymorphic functions require more pattern matching, since we must allow any value to be unboxed. A simple example, for an addition library function (`add`) is shown in Figure 3.4.

3. Shorthand (notational) functions are used for common actions in the generated code, since it has a tendency to be quite large otherwise. If we consider the code in Figure 3.6, the function `id_create`, for example, is used very often in typing rules, and returns the object corresponding to the identifier in the given environment, creating a new type variable if there is no such mapping:

```
let id_create env i =
  try
    IntMap.find i (!env)
  with
    Not_found ->
      let t = new_typevar() in
        env := IntMap.add i t (!env); t
```

3.6.5 Integration

Our final task is to integrate these snippets of generated code into a complete interpreter, that correctly implements the semantics. This is largely an implementation of the algorithm described in §2.4. Each rule, when fired, pushes a new environment onto the environment stack, and zero or more premises and conditions, followed by a completion, onto the evaluation stack.

The polymorphism of these elements, shown in Figure 3.5, allows us to perform polymorphic stack manipulations. It is important to note that the evaluation function also takes a *multimatch*

```
type ('a,'b) stack_element =
  Condition of ((semobj IntMap.t ref) -> unit)
| Premise of ((semobj IntMap.t ref) -> 'a) *
            ((semobj IntMap.t ref) -> 'b -> unit)
| Completion of ((semobj IntMap.t ref) -> 'b) *
               ((semobj IntMap.t ref) -> 'b -> unit)
```

Figure 3.5: The type of evaluation stack elements

index as argument, such that if more than one rule may fire, we may select which one. This is only changed if a rule fails to fire, and we have to backtrack, in which case we try the next possible rule (if there is one).

Execution proceeds iteratively, by popping the top element of the stack and executing it. A premise is evaluated by pattern matching its initial state with that of all the rules in turn – this function is called `match_type` for the typing relation, and `match_eval` for the evaluation relation. As an example, the clause generated for the `let`-polymorphism rule of Figure 2.4 is shown in Figure 3.6.

Since these stacks may grow quite large for iterative or recursive rules, we implement an optimisation here – tail-recursion. If the final state of the last premise is the same as the final state of the conclusion of a rule, we do not need to retain the environment and completion of that rule whilst the premise is being executed. We therefore refrain from pushing the completion of such rules onto the stack, and pop their environment *before* evaluating the final premise. The completion of the final premise will do the update for us, so we can nicely avoid having stacks that grow when we loop.[7]

It is instructive to view the code for the iteratively-called functions described above, namely `eval_step` and `type_step`, and we present these in Appendix D.

3.7 Implementation of Small-Step Semantics

We end this chapter with a discussion of how we generalised SemCom to handle small-step semantics. The difference between the two forms (see §1.1) is in whether the evaluation relation reduces expressions directly to values (big-step) or to another intermediate expression (small-step), so that a chain of transitions will terminate with its value. The only major modification to SemCom is to iteratively apply the evaluation procedure on the expression, until we cannot reduce it any further (i.e. it does not change). Of particular use for small-step semantics is the built-in relation *is_value(e)*, which tests whether e is a value (defined in the same way as for big-step semantics, as described in §2.3.2). Other than this, the ability to cope with small-step semantics essentially comes for free.

To handle non-deterministic semantics, little extra work is needed – we describe this, and present an SDL semantics of Milner's CCS in Appendix E.

[7]We need to be careful to update the backtracking stack as well, but this is not difficult.

```
| (s,Expr14(e2,e1,e0,lab)) ->
  let env = ref IntMap.empty in (* let *)
    test lab ((match_id env 27 e2) && (match_expr env 0 e1) &&
              (match_expr env 4 e0) && (unify env 52 s));
    type_push_env env;
    type_push (completion (fun env ->
                 id_create env 58) f_completion);
    type_push (premise
                 (fun env ->
                    (MapObj(IdMap.add (extract_id env 27)
                                      (abs_cons !env 61 (id_create env 53))
                                      (extract_map (IntMap.find 52 !env))),
                            make ((get_expr env lab 4))))
                 (fun env -> fun s ->
                    test lab (unify_val env (id_create env 58) s)));
    type_push (condition (fun env ->
                 test lab (unify env 61
                            (difference (ftv (id_create env 53),
                                         ftv (IntMap.find 52 !env))))));
    type_push (premise
                 (fun env ->
                    (IntMap.find 52 !env,make ((get_expr env lab 0))))
                 (fun env -> fun s ->
                    test lab (unify_val env (id_create env 53) s)));
    type_push (condition (fun env ->
                 test lab (not ((id_member env 27
                                  (map_dom (IntMap.find 52 !env))))))
```

Figure 3.6: Generated code for the `let`-polymorphism rule

Chapter 4

Evaluation

Overall, the implementation of SemCom (6000 lines of OCaml, across 18 modules) has been a great success, meeting the success criterion and exceeding it with various extensions. In this chapter, we will demonstrate evidence of this, and examine the performance of generated interpreters. Naturally, given the nature of this project, there are many improvements that could be made to SemCom, and we discuss these in §4.3 below.

4.1 Example Executions

To test the capabilities of SemCom, the semantics of a simple functional language was written in SDL, such that it supports a variety of features, including recursion, loops, conditionals, `let`-polymorphism and reference types. This language, SiMpL (described in Appendix C), is based on an amalgamation and modification of Sewell's L2 [13] and Pitts' Midi-ML [11]. We have also written a small-step version of this semantics, and have verified that the execution behaviour is the same for the two generated interpreters.

Here, we present a number of examples, executing SiMpL programs with the interpreter that SemCom generates.

4.1.1 Call-by-Value vs Call-by-Name

A good example of SemCom's usefulness is to observe the effects of small changes in the semantics of a language. The semantics of SiMpL shown in Appendix C uses call-by-value function application – i.e. we evaluate the argument *before* substituting it into the function. The rule for this is shown in Figure 4.1.

The interpreter that SemCom generates for us is roughly 900 lines of OCaml – the code for the `let`-polymorphism rule is given on page 30, and we give the code for the iterative execution engine in Appendix D. To test this interpreter, we will use it to evaluate the following SiMpL expression:

```
let val s = ref 0 in
  let val x = (fn x => s := !s + 2) in
    let val y = (fn y => (y; y)) in
      y(x(0)); !s
    end
  end
end
```

```
<e1,s> => <[[fn \x => \e]],s'>;
<e2,s'> => <v,s''>;
<{v/x}e,s''> => <v2,s'''>
---------------------------------[app]
<[[\e1 ( \e2 )]],s> => <v2,s'''>
```

Figure 4.1: An SDL rule for call-by-value function application

```
<e1,s> => <[[fn \x => \e]],s'>;
<{e2/x}e,s'> => <v,s''>
--------------------------------[app]
<[[\e1 ( \e2 )]],s> => <v,s''>
```

Figure 4.2: An SDL rule for call-by-name function application

The function `x` increments a mutable variable `s` whenever it is called, and `y` is a function that executes its argument twice. Using this call-by-value semantics, `s` is only incremented once, and indeed the interpreter returns:

```
: int =  2
```

Let us now modify the semantics of SiMpL, by replacing the function application rule with the one shown in Figure 4.2, for call-by-name. In this case, we substitute the *unevaluated* argument into the function, and this is evaluated every time it is used (possibly zero times). Evaluating the same expression, we now get (since `s` is incremented twice):

```
: int =  4
```

This is evidence to support the claim that the interpreter generated by SemCom is a correct implementation of the semantics.

4.1.2 Reference Types and Polymorphism

The semantics of SiMpL, as stated in Appendix C, has a deliberate flaw, and it is interesting to observe how SemCom copes with this. Specifically, the language has both reference types and polymorphism, which clash[1] in an interesting way in the following SiMpL expression:

```
let val r = ref (fn x => x) in
  let val u = (r := (fn y => ref !y)) in
    (!r)(skip)
  end
end
```

Here, the type of `r` is $\forall\alpha.(\alpha \to \alpha)\mathrm{ref}$, which is retained, even when we assign to `r`, due to polymorphism. According to our semantics, the above expression is therefore typeable, but will fail when we try to evaluate it, since we cannot dereference `skip`. To test SemCom, we generate an interpreter from these semantics, and use it to execute this expression. The interpreter returns:

[1]See Chapter 3 of [11]

```
: unit = Error on line 2, characters 33 to 35:

Evaluation failure
```

This is what we expected, and the interpreter is even kind enough to point out where the evaluation failed - namely the expression '`!y`', which could not be evaluated when `y` was substituted for `skip`.

This problem is resolved in ML by value-restricting the `let`-polymorphism rule[2], but is a nice example of SemCom precisely implementing the semantics it is given. It would be nice if we could automatically prove that the type system is sound with respect to evaluation, but this is beyond the scope of this project.

4.2 Performance Analysis

The SemCom compiler itself is impressively fast, considering the amount of work being done. On all the platforms that we have tested SemCom, it takes considerably less than a second to compile the reasonably sized semantics of SiMpL. To examine the performance of the generated interpreter, we use an inefficient (exponential time) function for calculating the nth Fibonacci number, which is implemented in SiMpL as (for $n = 15$):

```
let val rec f = (fn n =>
   if n == 0 then 0 else
   if n == 1 then 1 else
   f(n - 1) + (f(n - 2)) )
in
   f(15)
end
```

Using a simple OCaml script, we measure the time taken to execute this function, for varying n, using both the big-step and small-step semantics of our test language. As a base for comparison, we also execute a similar function with the native OCaml interpreter (`ocaml`). Figure 4.3 shows a graph[3] of these execution times, against n.

The performance of the interpreter is not fantastic, but we must remember the incredible amount of overhead involved, compared to running the code natively. It is apparent that the big-step interpreter is approximately three orders of magnitude slower than OCaml, and the small-step interpreter is a further order of magnitude slower than this. This is because small-step semantics have more rules for each term in the abstract syntax, and therefore more backtracking takes place.

Comparing this to the figures quoted in Chapter 8 of Diehl [3], we are approximately 25 times faster than his abstract machine in SML. Considering the increase in processor speeds since this was written (approximately 50-fold), the performance of SemCom is comparable.

4.3 Evaluation of Work Undertaken

Given how ambitious the project was, the resulting SemCom interpreter generator is a great success. Not only have we produced a system that will deal with arbitrary, user-defined, syntax;

[2]If an expression is not a value (e.g. it is a reference), its type should not become polymorphic.

[3]Generated by `gnuplot`, using a logarithmic time scale for easier comparison.

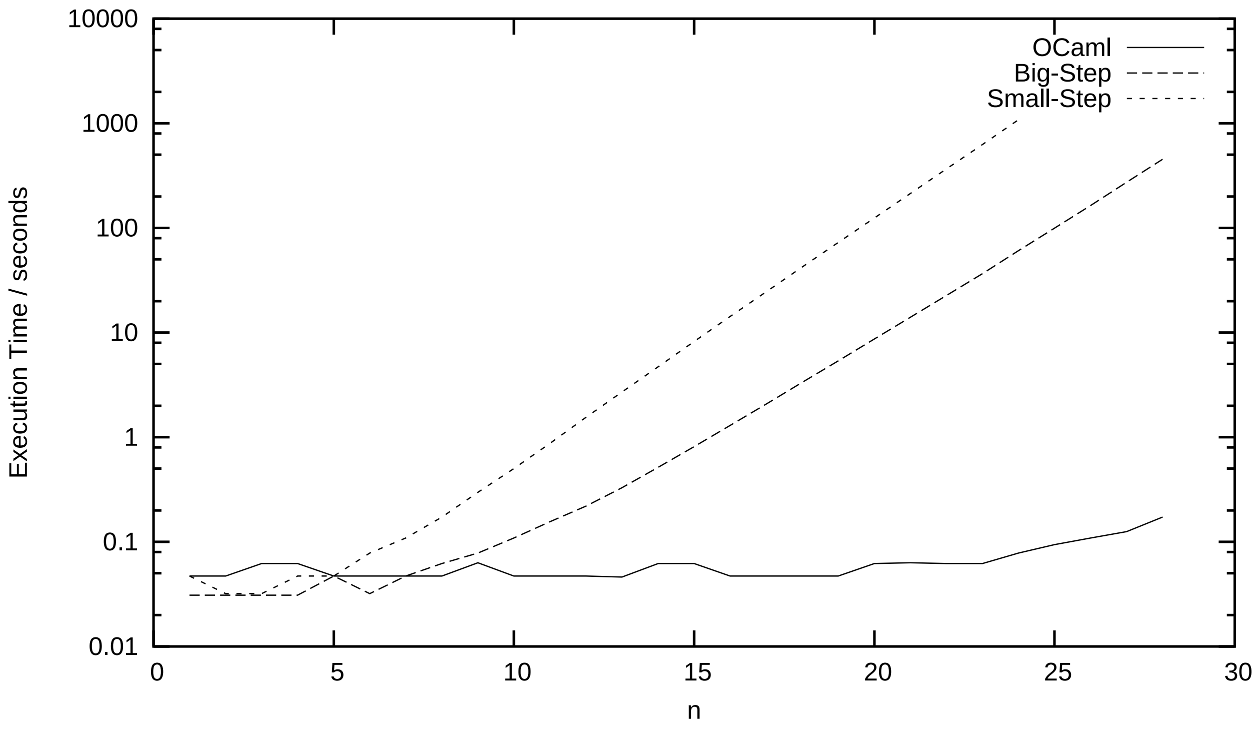

Figure 4.3: Performance comparison of SemCom with native OCaml

we have made life as simple as possible for the user. Given a semantics, the generation of an interpreter is *completely* automatic, including a lexer and parser for the language. The specification follows more closely Kahn's natural semantics [4] than systems such as RML [9], since we allow the premises and conditions of a rule to be unordered, and perform sequentialisation ourselves. Type systems up to polymorphism are supported. There are only really two general concerns that we have:

1. *Generality* – SemCom is, out of necessity, restrictive in the forms of semantics it can take, although we have seen that it is general enough to cope with both big-step and small-step styles. In fact, the core project concerned various semantics for a *fixed* language, and we have gone far beyond this. Whilst such a system could be made more general, it is an ongoing question as to how far we can go, and how useful it would be compared to the complexity of implementation.

2. *Efficiency* – to provide a compiler generator that people will actually use, efficiency is a key consideration. RML is efficient, but sacrifices expressibility and abstraction from implementation details. SemCom is much less efficient, but it only generates an *interpreter*. Unfortunately, we did not have time to implement, as an extension, a more efficient *compiler* generator, but we give an overview of how to do this in Appendix F.

On a more immediate level, there are a number of improvements that we would like to make to SemCom, but did not have time to in the course of this project:

1. Allow user-defined relations (other than for evaluation and typing), so that features like subtyping could be implemented. This would not be difficult to support, since these relations would follow the framework that is already in place, but we would require a more general execution engine to allow a rule to have premises from other relations.

2. Have the option of explicitly defined abstract syntax, to allow more complex typing and execution schemes to be expressed. This would be a purely syntactic extension, since we would be passing responsibility to the user.

3. Support type annotations in the target language. Again, this is mainly syntactic, since we would need to parse types from target language programs.

Despite this small wish-list, it must be said that the project has been a great success overall – in particular, we have met the success criterion set out in the project proposal (see Appendix G). Furthermore, the extensions we have implemented go far beyond what we initially set out to do, and the progress that we have made in such a short space of time is very impressive. SemCom has illustrated the sort of work that can be done, and has been great fun, and an interesting challenge, to design and implement.

Chapter 5

Conclusion

The challenge of automatically generating a compiler has captivated many computer scientists, and a number of different approaches have been taken. The result of this project, SemCom, is just one. Looking back at our initial aims, we have passed our success criterion, and built several extensions that go well beyond it, along with implementing a number of semantic definitions. Particularly pleasing is the generality of our system, supporting type systems up to polymorphism, and both big-step and small-step semantics. Although we have concentrated our discussion on a functional language (SiMpL – see Appendix C), we can also describe languages such as Milner's CCS in SDL – see Appendix E for more detail. We have even improved upon some features of existing work, such as RML [9], by sequentialising rules internally, and automatically generating a parser from a semantics.

Were we to design and build SemCom again, with the knowledge we have gained, we would still take largely the same approach. The main features that the current system lacks are the improvements listed in the previous chapter, such as generating a compiler, but this was simply a matter of limited time. One notable thing that we would improve, is the SDL type system – the way we deal with 'types of types' at present is too complicated for our needs, and could have been more simply dealt with using *kinds*[1]. On the whole, however, the design has been a great success, and has achieved more than we anticipated, in many respects.

Automated compiler generation, if we want to be both general *and* efficient, is hard, and we have yet to see what the end result of research in this area will be. The success that we have experienced in our attempt is encouraging, however, and only goes to show that there are many new ideas, just waiting to be investigated.

[1]See Chapter 29 of [10].

Bibliography

[1] P. Borras, D. Clement, T. Despeyrouz, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, volume 24, pages 14–24, New York, NY, 1989. ACM Press.

[2] T. Despeyroux. TYPOL: a formalism to implement natural semantics. Technical Report 94, INRIA, March 1988.

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

[4] 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.

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

[6] M. Y. Levin and B. C. Pierce. Tinkertype: A language for playing with formal systems. *Journal of Functional Programming*, 13(2), March 2003. A preliminary version appeared as an invited paper at the *Logical Frameworks and Metalanguages Workshop (LFM)*, June 2000.

[7] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[8] L. C. Paulson. A semantics-directed compiler generator. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–233, New York, NY, USA, 1982. ACM Press.

[9] 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.

[10] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[11] A. M. Pitts. Types (Computer Science Tripos Part II lecture notes), 2004.

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

[13] P. E. Sewell. Semantics of programming languages (Computer Science Tripos Part IB lecture notes), 2004.

Appendix A

Command Line Usage

```
semcom: semantics-directed compiler generator, version 1.0
==========================================================
Semcom and the Semantic Description Language (SDL) are
copyright (c) 2004-2005 Michael J A Smith

Usage: semcom [options] filename
-c comp    Specify the compatibility of generated code (ocaml|fs)
             Default: OCaml
-l lib     Specify a custom location of the second level library
             Default: semcom_lib in current dir
-m         Generate a Makefile for the generated files
-p         Generate an OCaml lexer/parser for the source language
-t         Generate a latexified version of the semantics
-v         Verbose mode - output directed to filename.out
```

Appendix B

BNF Grammar of SDL

⟨*main*⟩	`::=`	[⟨*step*⟩] ⟨*eval_rule*⟩ {“`;;`” ⟨*eval_rule*⟩} “`%%`” ⟨*type_rule*⟩ {“`;;`” ⟨*type_rule*⟩} [“`%%`” {⟨*prec_element*⟩}$^{+}$]
⟨*step*⟩	`::=`	“`%big`”
	|	“`%small`”
⟨*prec_element*⟩	`::=`	⟨*precedence*⟩ ⟨*source_token*⟩
⟨*precedence*⟩	`::=`	“`%left`”
	|	“`%right`”
	|	“`%nonassoc`”
⟨*eval_rule*⟩	`::=`	[⟨*eval_reduction*⟩ {“`;`” ⟨*eval_reduction*⟩}] ⟨*infer*⟩ ⟨*eval_reduction*⟩ [“`==IF==`” ⟨*condition*⟩]
⟨*type_rule*⟩	`::=`	[⟨*type_reduction*⟩ {“`;`” ⟨*type_reduction*⟩}] ⟨*infer*⟩ ⟨*type_reduction*⟩ [“`==IF==`” ⟨*condition*⟩]
⟨*infer*⟩	`::=`	{“`-`”}$^{+}$ [“`[`” ⟨*identifier*⟩ “`]`”]
⟨*eval_reduction*⟩	`::=`	⟨*meta_expr_br*⟩ “`=>`” ⟨*meta_expr_br*⟩
⟨*type_reduction*⟩	`::=`	⟨*semantic_object*⟩ “`|-`” ⟨*expr*⟩ “`:`” ⟨*semantic_object*⟩
⟨*condition*⟩	`::=`	⟨*second_level_call*⟩
	|	⟨*identifier*⟩ “`=`” ⟨*condition_expr*⟩
	|	“`is_value`” “`(`” ⟨*identifier*⟩ “`)`”
	|	⟨*identifier*⟩ “`IN`” ⟨*semantic_object*⟩
	|	⟨*condition*⟩ “`\/`” ⟨*condition*⟩
	|	⟨*condition*⟩ “`/\`” ⟨*condition*⟩
	|	“`~`” ⟨*condition*⟩
	|	“`(`” ⟨*condition*⟩ “`)`”
⟨*condition_expr*⟩	`::=`	⟨*second_level_call*⟩
	|	⟨*semantic_object*⟩

```
                      |    ⟨value⟩
                      |    “freshid” “(” “)”

 ⟨second_level_call⟩ ::=   ⟨identifier⟩ “(” [⟨argument⟩ {“,” ⟨argument⟩}] “)”

          ⟨argument⟩ ::=   ⟨identifier⟩
                      |    ⟨value⟩

      ⟨meta_expr_br⟩ ::=   ⟨meta_expr⟩
                      |    “<” ⟨meta_expr⟩ “>”

         ⟨meta_expr⟩ ::=   ⟨meta_base_expr⟩ {“,” ⟨semantic_object⟩}

    ⟨meta_base_expr⟩ ::=   [“{” ⟨substitution⟩ {“,” ⟨substitution⟩} “}”]⟨expr⟩

      ⟨substitution⟩ ::=   ⟨expr⟩ “|->” ⟨identifier⟩

              ⟨expr⟩ ::=   ⟨identifier⟩
                      |    “[[” {⟨source_element⟩}+ “]]”

   ⟨semantic_object⟩ ::=   ⟨identifier⟩
                      |    ⟨typeset⟩
                      |    “!” “{” {⟨identifier⟩}+ “}” ⟨semantic_object⟩
                      |    “{” ⟨map_update⟩ {“,” ⟨map_update⟩} “}” [⟨semantic_object⟩]
                      |    “{” ⟨value⟩ {“,” ⟨value⟩} “}”
                      |    ⟨semantic_object⟩ ⟨type_cons⟩ [⟨semantic_object⟩]
                      |    ⟨operation⟩ “(” ⟨semantic_object⟩ “)”
                      |    “(” ⟨semantic_object⟩ “,” ⟨semantic_object⟩ “)”
                      |    ⟨semantic_object⟩ “[” ⟨identifier⟩ “]”
                      |    ⟨semantic_object⟩ ⟨infix_operation⟩ ⟨semantic_object⟩
                      |    “(” ⟨semantic_object⟩ “)”

           ⟨typeset⟩ ::=   “int”
                      |    “float”
                      |    “char”
                      |    “string”
                      |    “bool”
                      |    “unit”

         ⟨operation⟩ ::=   “ftv”
                      |    “dom”
                      |    “#1”
                      |    “#2”

   ⟨infix_operation⟩ ::=   “UNION”
                      |    “INTERSECT”
                      |    “\”
      ⟨map_update⟩  ::=  ⟨identifier⟩ “|->” ⟨value⟩
                      |   ⟨identifier⟩ “|->” ⟨semantic_object⟩

            ⟨value⟩  ::=  ⟨integer⟩
                      |   ⟨float⟩
                      |   ⟨string_literal⟩
                      |   ⟨char_literal⟩
                      |   ⟨boolean⟩

           ⟨integer⟩  ::=  [“-”]{⟨digit⟩}+

            ⟨float⟩  ::=  [“-”]{⟨digit⟩}+ “.” {⟨digit⟩}+

   ⟨string_literal⟩  ::=  “"” {⟨any_character⟩}+ “"”

     ⟨char_literal⟩  ::=  “'” ⟨any_character⟩ “'”

           ⟨boolean⟩  ::=  “true”
                      |   “false”

   ⟨source_element⟩  ::=  ⟨source_token⟩
                      |   “\” ⟨identifier⟩

        ⟨type_cons⟩  ::=  “+”
                      |   “*”
                      |   “->”
                      |   “~” ⟨identifier⟩

     ⟨source_token⟩  ::=  {⟨any_character⟩}+

      ⟨identifier⟩  ::=  (⟨ucase_char⟩ | ⟨lcase_char⟩)
                         {⟨ucase_char⟩ | ⟨lcase_char⟩ | ⟨digit⟩ | “'” | “_”}

              ⟨digit⟩  ::=  “0” | ... | “9”

      ⟨ucase_char⟩  ::=  “A” | ... | “Z”

         ⟨lcase_char⟩  ::=  “a” | ... | “z”
```

Appendix C

Semantics of SiMpL

Here, we present the semantics of the test language SiMpL, which is a reasonable subset of SML, used to test SemCom. This semantics was written in SDL, but we present the latexified form here (generated by `semcom -t`) for convenience.

C.1 Dynamic Semantics

$$(\textsc{add})\ \frac{\langle e_1, s\rangle \Rightarrow \langle n_1, s'\rangle \qquad \langle e_2, s'\rangle \Rightarrow \langle n_2, s''\rangle}{\langle e_1 + e_2, s\rangle \Rightarrow \langle n, s''\rangle}\ \text{if } n = \mathit{add}(n_1, n_2)$$

$$(\textsc{sub})\ \frac{\langle e_1, s\rangle \Rightarrow \langle n_1, s'\rangle \qquad \langle e_2, s'\rangle \Rightarrow \langle n_2, s''\rangle}{\langle e_1 - e_2, s\rangle \Rightarrow \langle n, s''\rangle}\ \text{if } n = \mathit{subtract}(n_1, n_2)$$

$$(\textsc{geq})\ \frac{\langle e_1, s\rangle \Rightarrow \langle n_1, s'\rangle \qquad \langle e_2, s'\rangle \Rightarrow \langle n_2, s''\rangle}{\langle e_1 >= e_2, s\rangle \Rightarrow \langle b, s''\rangle}\ \text{if } b = \mathit{geq}(n_1, n_2)$$

$$(\textsc{eq})\ \frac{\langle e_1, s\rangle \Rightarrow \langle n_1, s'\rangle \qquad \langle e_2, s'\rangle \Rightarrow \langle n_2, s''\rangle}{\langle e_1 == e_2, s\rangle \Rightarrow \langle b, s''\rangle}\ \text{if } b = \mathit{eq}(n_1, n_2)$$

$$(\textsc{deref})\ \frac{\langle e, s\rangle \Rightarrow \langle l, s'\rangle}{\langle !\, e, s\rangle \Rightarrow \langle n, s'\rangle}\ \text{if } n = s'(l)$$

$$(\textsc{assign})\ \frac{\langle e, s\rangle \Rightarrow \langle n, s'\rangle}{\langle l := e, s\rangle \Rightarrow \langle \texttt{skip}, \{l \mapsto n\}s'\rangle}$$

$$(\textsc{ref})\ \frac{\langle e, s\rangle \Rightarrow \langle v, s'\rangle}{\langle \texttt{ref}\ e, s\rangle \Rightarrow \langle x, \{x \mapsto v\}s'\rangle}\ \text{if } x = \text{freshid}()$$

$$(\textsc{seq})\ \frac{\langle e_1, s\rangle \Rightarrow \langle \texttt{skip}, s'\rangle \qquad \langle e_2, s'\rangle \Rightarrow \langle v, s''\rangle}{\langle e_1 \,;\, e_2, s\rangle \Rightarrow \langle v, s''\rangle}$$

$$(\textsc{cond1})\ \frac{\langle e_1, s\rangle \Rightarrow \langle b, s'\rangle \qquad \langle e_2, s'\rangle \Rightarrow \langle v, s''\rangle}{\langle \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, s\rangle \Rightarrow \langle v, s''\rangle}\ \text{if } b = \mathit{true}$$

$$(\textsc{cond2})\ \frac{\langle e_1, s\rangle \Rightarrow \langle b, s'\rangle \qquad \langle e_3, s'\rangle \Rightarrow \langle v, s''\rangle}{\langle \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, s\rangle \Rightarrow \langle v, s''\rangle}\ \text{if } b = \mathit{false}$$

$$(\textsc{while1})\ \frac{\langle e_1, s\rangle \Rightarrow \langle b, s'\rangle \qquad \langle e_2, s'\rangle \Rightarrow \langle \texttt{skip}, s''\rangle \qquad \langle \texttt{while}\ e_1\ \texttt{do}\ e_2, s''\rangle \Rightarrow \langle v, s'''\rangle}{\langle \texttt{while}\ e_1\ \texttt{do}\ e_2, s\rangle \Rightarrow \langle v, s'''\rangle}\ \text{if}$$
$$b = \mathit{true}$$

$$(\textsc{while2})\ \frac{\langle e_1, s\rangle \Rightarrow \langle b, s'\rangle}{\langle \texttt{while}\ e_1\ \texttt{do}\ e_2, s\rangle \Rightarrow \langle \texttt{skip}, s'\rangle}\ \text{if } b = \mathit{false}$$

$$(\textsc{app})\ \frac{\langle e_1, s\rangle \Rightarrow \langle \texttt{fn}\ x \texttt{=>}\ e, s'\rangle \qquad \langle e_2, s'\rangle \Rightarrow \langle v, s''\rangle \qquad \langle \{v\,/x\}e, s''\rangle \Rightarrow \langle v_2, s'''\rangle}{\langle e_1\ (\ e_2\ ), s\rangle \Rightarrow \langle v_2, s'''\rangle}$$

$$(\textsc{let})\ \frac{\langle e_1, s\rangle \Rightarrow \langle v_1, s'\rangle \qquad \langle \{v_1\,/x\}e_2, s'\rangle \Rightarrow \langle v_2, s''\rangle}{\langle \texttt{let val}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}, s\rangle \Rightarrow \langle v_2, s''\rangle}$$

$$(\textsc{letrec})\ \frac{\langle \{(\ \texttt{fn}\ y \texttt{=>}\ \texttt{let val rec}\ x = (\ \texttt{fn}\ y \texttt{=>}\ e_1\ )\ \texttt{in}\ e_1\ \texttt{end}\ )\ /x\}e_2, s\rangle \Rightarrow \langle v, s'\rangle}{\langle \texttt{let val rec}\ x = (\ \texttt{fn}\ y \texttt{=>}\ e_1\ )\ \texttt{in}\ e_2\ \texttt{end}, s\rangle \Rightarrow \langle v, s'\rangle}$$

$$(\textsc{brack})\ \frac{\langle e, s\rangle \Rightarrow \langle e', s'\rangle}{\langle (\ e\ ), s\rangle \Rightarrow \langle e', s'\rangle}$$

C.2 Static Semantics

$$(\textsc{add})\ \frac{\Gamma \vdash e_1\ : \text{int} \qquad \Gamma \vdash e_2\ : \text{int}}{\Gamma \vdash e_1 + e_2\ : \text{int}}$$

$$(\textsc{sub})\ \frac{\Gamma \vdash e_1\ : \text{int} \qquad \Gamma \vdash e_2\ : \text{int}}{\Gamma \vdash e_1 - e_2\ : \text{int}}$$

$$(\textsc{geq})\ \frac{\Gamma \vdash e_1\ : \text{int} \qquad \Gamma \vdash e_2\ : \text{int}}{\Gamma \vdash e_1 >= e_2\ : \text{bool}}$$

$$(\textsc{eq})\ \frac{\Gamma \vdash e_1\ : \text{int} \qquad \Gamma \vdash e_2\ : \text{int}}{\Gamma \vdash e_1 == e_2\ : \text{bool}}$$

$$(\textsc{cond})\ \frac{\Gamma \vdash e_1\ : \text{bool} \qquad \Gamma \vdash e_2\ : T \qquad \Gamma \vdash e_3\ : T}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ : T}$$

$$(\textsc{assign})\ \frac{\Gamma \vdash l\ : T\,\text{ref} \qquad \Gamma \vdash e\ : T}{\Gamma \vdash l := e\ : \text{unit}}$$

$$(\textsc{ref})\ \frac{\Gamma \vdash x\ : T}{\Gamma \vdash \texttt{ref}\ x\ : T\,\text{ref}}$$

$$(\textsc{deref})\ \frac{\Gamma \vdash l\ : T\,\text{ref}}{\Gamma \vdash !\,l\ : T}$$

$$(\textsc{id})\ \frac{}{\Gamma \vdash l\ : T}\ \text{ if } T = \Gamma(l)$$

$$(\textsc{skip})\ \frac{}{\Gamma \vdash \texttt{skip}\ : \text{unit}}$$

$$(\textsc{seq})\ \frac{\Gamma \vdash e_1\ : \text{unit} \qquad \Gamma \vdash e_2\ : T}{\Gamma \vdash e_1\, ;\, e_2\ : T}$$

$$(\textsc{while})\ \frac{\Gamma \vdash e_1\ : \text{bool} \qquad \Gamma \vdash e_2\ : \text{unit}}{\Gamma \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2\ : \text{unit}}$$

$$(\textsc{fn})\ \frac{\{x \mapsto T\}\Gamma \vdash e\ : T'}{\Gamma \vdash \texttt{fn}\ x => e\ : T\ \rightarrow\ T'}$$

$$(\textsc{app})\ \frac{\Gamma \vdash e_1\ : T\ \rightarrow\ T' \qquad \Gamma \vdash e_2\ : T}{\Gamma \vdash e_1\ (\ e_2\ )\ : T'}$$

$$(\textsc{let})\ \frac{\Gamma \vdash e_1\ : T \qquad \{x \mapsto \forall A.(T)\}\Gamma \vdash e_2\ : T'}{\Gamma \vdash \texttt{let val}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}\ : T'}\ \text{ if } A = \text{ftv}(T)\backslash\text{ftv}(\Gamma) \wedge \neg(x \in \text{dom}(\Gamma))$$

$$(\textsc{letrec})\ \frac{\{x \mapsto T_1\ \rightarrow\ T_2, y \mapsto T_1\}\Gamma \vdash e_1\ : T_2 \qquad \{x \mapsto T_1\ \rightarrow\ T_2\}\Gamma \vdash e_2\ : T}{\Gamma \vdash \texttt{let val rec}\ x = (\ \texttt{fn}\ y => e_1\ )\ \texttt{in}\ e_2\ \texttt{end}\ : T}$$

$$(\textsc{brack})\ \frac{\Gamma \vdash e\ : T}{\Gamma \vdash (\ e\ )\ : T}$$

Appendix D

Code for the Iterative Execution of Stack Elements

D.1 The Function eval_step

```
let eval_step i x =                              i is the multimatch index.
  try (match x with                              x is the top element of the stack.
    Condition(f) ->
      (try
         f (Stack.top env_eval_stack)            Evaluate the condition, using the
                                                 current environment.
       with
           Stack.Empty -> assert false)
  | Premise(m,f) ->
      let m' = m (Stack.top env_eval_stack) in   Construct the premise's expres-
                                                 sion, using the current environ-
                                                 ment
        if not (match_eval f i m') then          Expand the premise by calling
                                                 match_eval. This returns true if
                                                 this was a success.
          f (Stack.top env_eval_stack) m'        If we fail, then m' was a value, so
                                                 we execute it's completion func-
                                                 tion f.
  | Completion(f1,f2) ->
      (try
         let env = Stack.pop env_eval_stack in   Pop the current environment.
           backtrack_decrement_env();            Update the backtracking stack.
           f2 (Stack.top env_eval_stack) (f1 env)   Use f1 to construct the result ex-
                                                 pression from the current envi-
                                                 ronment, and f2 to update the
                                                 previous environment.
       with
           Stack.Empty -> assert false))
  with NoMatchException l ->                     Execution has failed.
    if Stack.length backtrack_eval_stack > 0 then
      backtrack (Stack.pop backtrack_eval_stack)   Attempt to backtrack.
    else
      no_match l                                 Fail with position l of failure.
```

D.2 The Function `type_step`

```
let type_step = function


    Condition(f) ->
      (try
         f (Stack.top env_type_stack)

       with
           Stack.Empty -> assert false)
  | Premise(m,f) ->
      match_type f (m (Stack.top env_type_stack))




  | Completion(f1,f2) ->
      (try
         let env = Stack.pop env_type_stack in
           f2 (Stack.top env_type_stack) (f1 env)




       with
           Stack.Empty -> assert false)
```

Note that this is simpler than `eval_step`*, since we do not have to catch failures to backtrack.*

Evaluate the condition, using the current environment.

Construct the premise's expression, using the current environment, and expand this by calling `match_type`*.*

Pop the current environment.
Use $f1$ to construct the result expression from the current environment, and $f2$ to update the previous environment.

Appendix E

Non-Determinism and CCS

Non-determinism arises in a small-step semantics when two or more rules could be successfully applied to an expression – in other words, the expression could undergo a number of different transitions. In the initial implementation of SemCom, we simply attempt the rules in the order set out by the user, but this means that the first possible rule is always chosen. Yet with non-deterministic rules, we wish to find all the possible values that an expression reduces to.

As in Prolog, we have a backtracking mechanism, so that we may try alternative rules if one fails (such as in the case of conditionals). Could we not then 'pretend' that we have failed when a solution is found, and continue backtracking to find another solution? In actual fact, there is a little more involved than that. Remember that in a small-step semantics, we have a separate 'proof tree' for every transition, and so we throw away backtracking information when a transition succeeds.

We get around this problem by maintaining a set of *checkpoints* – whenever a rule succeeds in being applied, but we could have tried another, we store a copy of the execution stacks. When an expression is reduced to a value, we output it, but also prompt the user to find another solution. If they select 'yes', we resume from the last checkpoint until we reduce to a *different* value. This is repeated, until we have exhausted all possibilities, and no more solutions can be found.

To illustrate this, we present a semantics for a version of Milner's CCS (the SDL file for which is given below, in §E.1). For simplicity, we require that all channels be restricted (to avoid dealing with input/output on open channels), and we treat synchronisation as the only way to communicate between concurrent processes. We may output a value on a non-synchronised channel, but the value is then lost, and a non-synchronised input is treated as though the process is in a blocked state. Note also that the type system ascribes type τ `process` to a process that terminates with a value of type τ, and type α `channel` to a channel that transmits values of type α.

As an example, we present the interpreter that SemCom generates with the following CCS expression:

```
new a. (a?x -> (a!3 -> val x)) | (a!2 -> (a?x -> val x))
```

This has two possible ways of synchronising, and indeed the output from the interpreter is (where user input is shown in italics):

```
 :  int + int process = new a . a ? x -> ( a ! 3 -> val x ) | a ? x -> val x

 Find another solution?  (y/n)  y
 new a . 2 | a ? x -> val x

 Find another solution?  (y/n)  y
 new a . 2 | 3

 Find another solution?  (y/n)  y
 No more solutions
```

E.1 CCS.sem

```
%small

------------------[tau]
[[tau -> \p]] => p
;;

-----------------------[out0]
[[\a_ ! \n -> \p]] => p
==IF==
is_value(n)
;;

a => a'
-----------------------------------------[out1]
[[\a_ ! \a -> \p]] => [[\a_ ! \a' -> \p]]
;;

---------------[val]
[[val \v]] => v
;;

-----------------------------------[restrict0]
[[new \x . \v]] => [[new \x . \v]]
==IF==
is_value(v)
;;

p => p'
------------------------------------[restrict1]
[[new \x . \p]] => [[new \x . \p']]
;;

p0 => p0'
---------------------[sum0]
[[\p0 + \p1]] => p0'
;;
```

```
p1 => p1'
--------------------[sum1]
[[\p0 + \p1]] => p1'
;;

------------------------------[compval]
[[\v1 | \v2]] => [[\v1 | \v2]]
==IF==
is_value(v1) /\ is_value(v2)
;;

p0 => p0'
-------------------------------[comp0]
[[\p0 | \p1]] => [[\p0' | \p1]]
;;

p1 => p1'
-------------------------------[comp1]
[[\p0 | \p1]] => [[\p0 | \p1']]
;;

---------------------------------------------------------------------------[sync0]
[[\( \a_ ? \x -> \p0' \) | \( \a_ ! \n -> \p1' \)]] => {n/x}[[\p0' | \p1']]
==IF==
is_value(n)
;;

---------------------------------------------------------------------------[sync1]
[[\( \a_ ! \n -> \p0' \) | \( \a_ ? \x -> \p1' \)]] => {n/x}[[\p0' | \p1']]
==IF==
is_value(n)
;;

---------------[brack]
[[( \p )]] => p

%%

-----------[id]
G_ |- x : T
==IF==
T = G_[x]
;;

G_ |- v : T
-----------------------------[val]
G_ |- [[val \v]] : T ~process
==IF==
(T = int) \/ (T = bool) \/ (T = char) \/
(T = string) \/ (T = float)
;;
```

```
-----------------------------[nil]
G_ |- [[nil]] : unit ~process
;;

G_ |- a_ : T ~channel;
G_ |- a : T;
G_ |- p : T' ~process
--------------------------------------[out]
G_ |- [[\a_ ! \a -> \p]] : T' ~process
;;

G_ |- a_ : T ~channel;
{a |-> T}G_ |- p : T' ~process
--------------------------------------[in]
G_ |- [[\a_ ? \a -> \p]] : T' ~process
;;

{x |-> T ~channel}G_ |- p : T' ~process
---------------------------------------[restrict]
G_ |- [[new \x . \p]] : T' ~process
;;

G_ |- p : T ~process
--------------------------------[tau]
G_ |- [[tau -> \p]] : T ~process
;;

G_ |- p0 : T ~process;
G_ |- p1 : T ~process
--------------------------------[sum]
G_ |- [[\p0 + \p1]] : T ~process
;;

G_ |- p0 : T1 ~process;
G_ |- p1 : T2 ~process
----------------------------------------[comp]
G_ |- [[\p0 | \p1]] : (T1 + T2) ~process
;;

G_ |- p : T
---------------------[brack]
G_ |- [[( \p )]] : T

%%

%nonassoc .
%left |
%left +
%right ?
%right !
%nonassoc val
```

Appendix F

How to Generate a Compiler

Generating a compiler is not wildly different to an interpreter. In fact, a compiler may in general be thought of as 'interpreting' the input program – emitting equivalent lower-level code, rather than the result of executing the program. In this way, we can see how SemCom may be modified to generate a compiler, given a big-step evaluation relation. More specifically, there are two stages involved:

1. The evaluation relation is compiled into an *intermediate code* generator, which emits a sequence of instructions in a stack-based intermediate language, given a target language expression as input.

2. The intermediate code is translated into code for an abstract machine, such as the JVM, or the .NET CLR. Note that the use of intermediate code allows multiple abstract machines to be supported.

For many expressions in a language's syntax, there will only be one rule to apply, and this will in turn evaluate its sub-expressions. In this case, the generated compiler should proceed in the same way as an interpreter, except that the *side-conditions* are transformed into intermediate code, and the code for each of the premises is pieced together. The key idea is that we don't test the applicability of a rule at compile-time, but emit code to do so at runtime.

Notice that in this case, we are relying on two separate properties – that there is only rule to apply, and that the premises of the rule are over sub-expressions of the conclusion. We need to do things differently if each of these fails to hold:

1. *Conditional execution* – if more than one rule applies, we have a conditional execution. We want to treat this differently to an interpreter however, since we need to emit code for all possibilities. Instead of backtracking, we emit *forward branch* instructions, such that if a condition in one rule fails, we jump to the next. It would be useful to left-factor the rules here, so we can evaluate the guard just once, and use this to jump to the remainder of the appropriate rule.

2. *Iterative or recursive execution* – if, in a rule, the premises are not based on the sub-expressions of the conclusion, we cannot guarantee that the expression evaluated by each premise will be monotonically smaller. For example, the rule for a `while` loop executes the entire loop *again*, if the guard evaluates to true. We therefore cannot simply expand every rule we encounter, as we do when interpreting, since this would mean unrolling all loops in the compiler, and we would be unable to guarantee termination! So, we must

detect such looping rules, and emit *backwards branch* instructions, to loop through the same code at runtime.

What signifies a looping rule? If we consider rules as functions, then the premises of a rule are function calls – the function a premise calls is the rule that its expression would match. Most of the time we only find out which rule is applied at runtime, since this depends on the syntax of a sub-expression. However, in looping or recursive rules, we explicitly construct an expression that will *always* fire the same rule again.

To analyse this, we need to construct a *call graph* of the rules (ignoring the premises that could match any rule), and treat the premises that match specific rules as calls. We can then detect any cycles in this graph, since these correspond to loops. More correctly, these cycles correspond to *recursive rules*, and loops are a special case corresponding to *tail-recursive rules*. For the more general case of recursion, we can implement this using *closures*, so that we recursively call a function at runtime.

Given that we can detect the special cases of rules described above, generating a compiler would not be particularly difficult – in fact most of the work is ensuring that we glue sequences of instructions together correctly. It would still, however, be very interesting to attempt this, both in terms of observing the performance, and providing interoperability with other code using a common abstract machine, such as the .NET CLR.

Appendix G

Project Proposal

Part II Computer Science Project Proposal
Semantics-Directed Compiler Generation[1]

Michael J A Smith, Robinson College
<mjas2@cam.ac.uk>

October 22, 2004

Special Resources Required

The use of my own IBM PC (2.08GHz Athlon, 1GB RAM, 180GB HDD).

Project Supervisor: T. Stuart

Director of Studies: Dr A. Mycroft

Project Overseers: Dr M. Kuhn & Dr G. Titmus

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

1 Introduction

The automated generation of compilers, interpreters and abstract machines is an area that has interested many people. There has been much work undertaken on generation techniques using denotational semantics, though more recent work has placed emphasis on various forms of operational semantics (e.g. natural semantics, structural operational semantics).

2 Core Project Description

I will develop a compiler that will accept a simple operational semantics for a programming language (initially with a fixed abstract syntax), and output an interpreter for programs written in this language. The input will take the form of two-level big-step semantics [1], with certain restrictions, such as determinism[1]. To express such definitions, the metalanguage will provide primitive objects such as partial finite maps, and operations such as capture-avoiding substitution in expressions. Side conditions will have at least the flexibility of first order logic without quantifiers, with set operations as primitives.

The use of a two-level semantics allows the implementation of non-primitive operations to be partitioned from the semantics itself. Operations such as basic arithmetic will be provided in a standard library. Simple types will also be expressible in the (static) semantics, such that the interpreter will perform type-checking on source programs.

3 Core Project Details

3.1 Languages and Tools

I will write the compiler generator and subsequently generated code in the OCaml language, due to its facility for pattern matching and polymorphism. Furthermore, I will use the variant Fresh OCaml due to its primitive support for abstraction and α-conversion. I will use an automated parser generator such as OCamlYacc to write the parser for the semantics metalanguage.

3.2 Metalanguage of Semantics

I will adopt the standard style (used for example in Pierce [4] and Milner [2]) of separating the static semantics (i.e. the type relation, $\vdash$) from the dynamic semantics (i.e. the evaluation relation, $\Rightarrow$). It is yet to be decided the exact metalanguage that will be used, but it should be able to accept evaluation rules in a form such as:

$$\frac{\varphi_1 \qquad \cdots \qquad \varphi_n}{\langle e, S\rangle \Rightarrow \langle e', S'\rangle} \quad \text{if } s_1 \wedge \ldots \wedge s_m$$

Such an evaluation rule consists of a conclusion $\langle e, S\rangle \Rightarrow \langle e', S'\rangle$, a set of premises φ_i, and a set of side conditions s_j. Conditions may refer to functions in the second-level semantics, which will be defined separately (such that the underlying implementation is hidden from the top-level semantics). A standard library will be presented, with the option of augmentation with user-defined functions.

The static semantics define a ternary relation $\Gamma \vdash e : T$, of a similar form to the above, except that Γ is a semantic object describing the assumptions, and T is a type (i.e. an element of the abstract syntax of the type system). In addition to type rules, the static semantics will

[1]Each rule should be of itself deterministic, and the set of rules should together be determinate

describe a mapping from each of the base types to the set of values of that type. The sets of values of the base types of the underlying implementation (for example, integers, booleans etc.) will be made available, in addition to user defined finite sets (corresponding to enumeration types).

The term **semantic object**, used above, refers to 'objects' such as partial finite maps, sets, tuples and records, which will be supported as primitives along with their associated basic operations.

3.3 Structure of the Compiler Generator

The process of generating the interpreter may be split into four main phases:

1. **Parsing** – The static and dynamic semantics will be translated from their textual representation into the abstract syntax of the metalanguage. Expressions within the semantics will be expressed in the abstract syntax of the concrete language, extended with metavariables. User-defined functions (written in OCaml) in the second-layer semantics will also be parsed at this point, and augmented with the standard library.

2. **Analysis** – Each evaluation rule will be analysed by a number of consistency checks. If the set of conditions of a rule form an inconsistency, this will result in a failure for the rule to be applied to any expression. It is not expected that the compiler generator will be capable of inferring this; rather that the resulting interpreter will halt on such inputs. Analysis of the type rules will take place in the same way.

3. **Transformation** – The internal representation of the semantic rules will undergo a number of transformations, resulting in a form that may be used directly by the interpreter. The most important of these will be a continuation passing style (CPS) transformation (RML [3]), as this allows for an iterative (rather than recursive) evaluation function, which will execute using an execution stack of continuations.

4. **Generation** – The transformed rules for the static and dynamic semantics will be integrated with a typing function `type_check()`, and an evaluation function `eval()`. The structure of the interpreter is described below. The compiler generator will then output the OCaml code corresponding to this interpreter.

3.4 Structure of the Generated Interpreter

The interpreter accepts the abstract syntax tree of a program, and returns an expression indicating the final state of the program. The main steps performed are:

1. **Static analysis** – the type checker, using the transformed static semantics, will verify that the program is well typed, and annotate the type of each expression. This requires an implementation of a unification algorithm, in order to unify type variables during the static analysis. If the program is not well typed, the interpreter will exit with a type error.

2. **Evaluation** – the typed program will then be executed in the dynamic semantics by the evaluation function `eval()`. Calls to the routines in the second-level semantics must be facilitated, along with implementations of the supported semantic objects. A symbol table (symtab) data structure will be used to map the string representation of variables to that internal to the interpreter. This is required to handle issues of scope (for example, overriding of names), and also to store the type of the variable (e.g. distinguising between function and value type).

4 Extensions

4.1 Arbitrary Abstract Syntax

On completion of the above core, the first extension will be to allow an arbitrary abstract syntax to be presented to the compiler generator. The grammar for expressions, types, declarations and values would need to be specified by the user, and the terms in the semantics conform to the given abstract syntax.

4.2 Compiler Generator to .NET

One problem in existing compiler generators is the lack of support for integration with other languages. An interesting challenge is therefore to replace the generated interpreter with a compiler to the .NET Common Intermediate Language (CIL). This would entail compiling to a fixed abstract machine, unlike the work of Diehl [1] where both the abstract machine and compiler are generated. Integration with existing code and standard libraries would then be possible through the Common Language Runtime (CLR).

Such a generated compiler would operate by executing the program in the semantics, rather like an interpreter. However, rather than outputting the result of the execution, it would produce CIL instructions. Care would need to be taken when dealing with, for example, looping constructs, as we would wish to output appropriate comparison and branch instructions, rather than unfolding the loop in the compiler. There is much scope here for partial evaluation style optimisations.

5 Starting Point

I have examined some of the current work in this area, and in particular that of Stephan Diehl [1], and of Mikael Pettersson et al [3] with respect to the Relational Metalanguage (RML). I worked as a research intern under Peter Sewell over the summer, working on the TCP semantics project[2], so I am familiar with dealing with operational semantics and labelled transition system (LTS) styles of definition.

I already have a working knowledge of the OCaml language, and have looked a little at Fresh OCaml and at the paper describing FreshML [5].

6 Success Criterion

The success criterion will be a compiler generator that accepts a number of different well-formed[3] semantics for a fixed language (for example, the simply-typed lambda calculus), each producing an interpreter that correctly type-checks and executes programs written in this language.

7 Resources

This project will require no special resources other than my own machine in order to work outside the lab. See my Project Resource Form for more details. In order to prevent loss of data due to computer failure, accidental deletion or other such circumstances, I shall set up

[2]`http://www.cl.cam.ac.uk/users/pes20/Netsem/`

[3]The semantics must be sound, complete and determinate

a CVS repository on my PWF account, to which all work shall be regularly committed. The PWF systems are regularly backed up by the University Computer Service, and I will also have a checked-out copy of the repository on my own machine. In addition, I routinely backup data from my machine to CD on a weekly basis.

8 Timetable

During this project, I will adopt a software engineering approach similar to the spiral model of software development. The core project will be undertaken as a complete cycle in itself, after which I will reenter the design phase, to plan the implementation of the first extension, and similarly for the second. This is the intended purpose of Boehm's spiral model, since the highest priority features (i.e. those of the core project) are designed and implemented first.

The proposed timetable for the project is outlined below:

Preparation *22/10/04 to 29/10/04*

Read through the existing papers in more detail.
Finalise the abstract syntax of the fixed language to be used.
Finalise the features and structure of the metalanguage, including side conditions.
Determine the details of the analysis and transformation stages of the generator.

Metalanguage *30/10/04 to 12/11/04*

Write the parser for the semantics metalanguage.
Create an initial library of functions for the second-level semantics.

Analysis and Transformation *13/11/04 to 26/11/04*

Implement the analysis and transformation stages of the compiler generator.

Integration of Interpreter *27/11/04 to 10/12/04*

Implement a unification algorithm for use with the static semantics.
Implement the overall type-checking function.
Implement the overall evaluation function.

Testing of Generated Interpreters *11/12/04 to 17/12/04*

Write a number of different semantics for the language.
Use these to test the compiler generator, and fix any implementation bugs.

Extension to Arbitrary Abstract Syntax: Preparation *15/01/05 to 29/01/05*

Decide the mechanism for user input of abstract syntax.
Decide how such arbitrary syntax will integrate with the existing generator.
Extend the parser.

Preparation of Progress Report *30/01/05 to 03/02/05*

Write the progress report.
Prepare the progress presentation.

Progress Report Deadline *04/02/05*

Extension to Arbitrary Abstract Syntax: Implementation *05/02/05 to 18/02/05*

Modify the implementation of the compiler generator to accept the abstract syntax.
Test the new implementation with some simple languages and semantics.

Extension to .NET Compiler Generation: Preparation *18/02/05 to 28/02/05*

Read the Microsoft .NET documentation in more detail.
Decide the modifications needed to the interpreter for it to output CIL code.
Decide the structure of generated .NET code (objects required, class structures etc.).
Decide the mechanism for accessing the .NET standard libraries.

Extension to .NET Compiler Generation: Implementation *29/02/05 to 26/03/05*

Modify the compiler generator to output CIL code.
Ensure the correct metadata is emitted, such that the code will compile.
Test the with the same semantics as before, to ensure that the two executions agree.
Modify the compiler generator to facilitate library calls.
Test the calling of library functions in a simple language.

Writing Dissertation *27/03/05 to 27/04/05*

Generate comparison statistics with existing compilers and compiler generators.
Complete writeup of dissertation.

Dissertation Deadline *20/05/05*

References

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

[2] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[3] 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.

[4] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[5] M. Shinwell, A. Pitts, and M. Gabbay. Freshml: Programming with binders made simple, 2003.