22.4.6 Generation of Efficient Compilers and Interpreters from Natural Semantics Specifications

Peter Fritzson, Mikael Pettersson

Compiler generation from Denotational Semantics was mentioned in the previous section. However, there are still some problems associated with Denotational Semantics, e.g. concerning modularity and when specifying concurrent languages. For these and other reasons, we have now focused on the more recent Natural Semantics formalism, and developed a first version of a system called RML (Relational Meta Language and system) for generating efficient implementations from Natural Semantics specifications. The following sections give more details of this work.

Background

Since the early eighties, a formalism known as Natural Semantics has become increasingly popular among programming language researchers. Natural Semantics is often used to specify type systems for the static semantics of programming languages, or the dynamic semantics, or both, and it has even been used to specify translations from abstract syntax to intermediate code. Lately, there has been a trend to use augmented type systems and translations, all specified in Natural Semantics, to do static analysis and code-improving transformations.

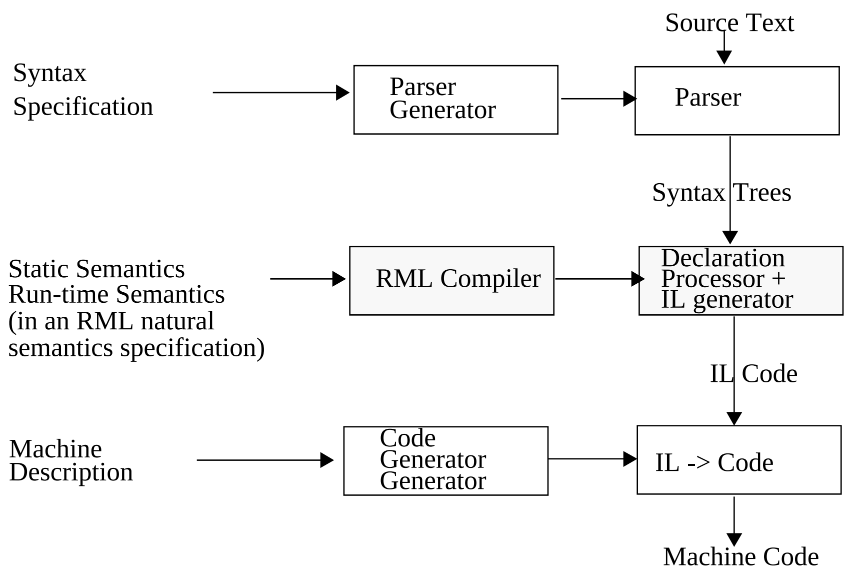

Figure 22-6. *The semantics analysis part of compilers is generated by the RML system from natural semantics specifications.*

Natural Semantics is based on Plotkin's Structural Operational Semantics (SOS) and further developed at INRIA by Kahn. Specifications consist of data type declarations (abstract syntax, environments, run-time values, types, etc.) and sets of inference rules. The inference rules specify relations between objects, in a style akin to Gentzen's Sequent Calculus for Natural Deduction. (Hence the name 'Natural' Semantics.) In a rule like

$$\frac{H_1 \vdash T_1 : R_1 \ \ldots \ H_n \vdash T_n : R_n}{H \vdash T : R} \quad \text{if } \langle\text{cond}\rangle$$

the H_i are hypotheses (typically environments containing bindings of source-level names to semantic objects), the T_i are terms (typically pieces of abstract syntax), and the R_i are results (typically types, run-time values, or augmented environments). An instance $H_j \vdash T_j : R_j$ is called a sequent. The sequents above the line are the premises, and the sequent below the line is the conclusion. The rule may be interpreted as follows: in order to prove a sequent $H \vdash T : R$, one must first prove the sequents $H_1 \vdash T_1 : R_1 \ldots H_n \vdash T_n : R_n$. The side condition, if present, must also be satisfied.

Natural Semantics offers several advantages over classical Denotational Semantics:

- All objects are finite terms, which means that the complicated domain theory of Denotational Semantics is not needed.
- More than one inference rule may be applicable at any given time, which means that some non-deterministic features are easy to model. For instance, the evaluation order of binary expressions in an imperative language can be left unspecified.
- Modern type systems involving polymorphic type inference are much easier to specify in Natural Semantics. A specification in Denotational Semantics would tend to resemble a type inference algorithm expressed as a functional program.

The Centaur programming environment, developed within the Esprit GIPE and GIPE-II projects, contains a meta-language for Natural Semantics called TYPOL. Until recently, this was the only available implementation of a language intended specifically for Natural Semantics. The default implementation uses a simple translation from TYPOL to the Centaur mu-Prolog sub-language for execution. It has also been shown that a restricted class of Natural Semantics specifications is equivalent to a certain kind of attribute grammars that can be executed by a functional evaluator.

We see several problems with the current state of affairs:

- The TYPOL implementation is very inefficient. We also feel that the Centaur system does not lend itself to the use of Natural Semantics in stand-alone applications.
- Coding Natural Semantics specifications in Prolog is not attractive, due to the lack of a decent type system in Prolog. We also believe that a compiler for a special-purpose Natural Semantics language can generate much better code than a Prolog compiler can for Natural Semantics specifications translated to Prolog.
- Some prefer to use the higher-order λProlog language. We feel that this language is too complicated, both for users and implementors alike.

Objectives

In the long run, we want to see Natural Semantics being as useful in programming language research and implementation, as are Context-Free Grammars and parser generators today.

Results

In the short run, we have defined a meta-language for Natural Semantics, and studied its implementability and practical usefulness. We have identified statically determinable properties of Natural Semantics specifications that allow (or disallow) interesting optimizations to be applied to the implementation of Natural Semantics specifications. In particular, the following results have recently been obtained:

- The Relational Meta-Language (RML) has been defined. It is strongly typed with a type system very much like that in Standard ML, has type-safe separate compilation and modules, and supports Natural Semantics-style inference rules. It has fewer non-declarative constructs than Prolog. The SML-like data types directly support structural-induction style specifications, which are central to Natural Semantics.
- The operational properties of RML were investigated and used to derive the initial implementation. A key component is the use of a Continuation-Passing Style (CPS) intermediate representation. CPS is easy to optimize, due to its declarative nature, but is also easily translated to low-level code, due to its simple operational semantics.
- Further observations lead to a refinement, whereby RML specifications are first translated to a First-Order Logic. High-level equivalences are used to rewrite this representation in order to reduce the amount of unnecessary non-determinism. This phase has proven to be essential for the practicality of the generated code.
- A compiler generating portable ANSI-C code has been implemented. The code runs unchanged on several different 32 and 64-bit architectures. Performance measurements indicate that this code runs several times faster than that generated by commercial Prolog compilers, and several orders of magnitude faster than TYPOL.
- Recent work has concentrated on the mapping of the control flow aspects of high-level languages to C. Results indicate that significant performance improvements can be made.

Performance Figures

We have a standard benchmark consisting of a NS for the dynamic semantics of a call-by-name functional language ‘Mini-Freja’. From this, we generate a compiled interpreter for the same language. Finally, we invoke the interpreter on a Mini-Freja program computing prime numbers.

Comparing the performance of TYPOL (T) and RML2C (R) for this specification on a Sun 10/41, gives the following results:

#primes	T	R	T/R
3	13s	0.0026s	5000
4	72s	0.0037s	19459
5	1130s	0.0063s	179365

The Mini-Freja specification was rewritten in Prolog to allow comparisons to be made with commercial native-code Prolog compilers: SICStus Prolog (S) and Quintus Prolog (Q). On a Sun 4/470, we have the following results:

#primes	S	Q	R	S/R	Q/R
18	5.0s	4.5s	0.45s	11.1	10.0

Further work

The pragmatic aspects of the generated code need to be improved, especially for interoperability with ‘foreign’ code. This is mostly a matter of design and engineering. There is much room for improvement in the compiler. Static analysis should be used to reduce the inefficiencies introduced by the language itself (e.g. unnecessary dereferencing), and those pertinent to certain classes of Natural Semantics specifications. For instance, dynamic semantics involving states are likely to benefit from an Natural Semantics analogy of the single-threadedness analysis of denotational semantics and lazy purely functional programming languages.