Foreword

When Mikael asked me to write this foreword, I happily agreed. Since the time I wrote my own Ph.D. thesis 1984 on incremental programming environments, compilers, and generator tools for such systems, I have been interested in the problem of automatically generating practical tools from formal semantics language specifications. This problem was unsolved in my own work at that time, which focused on incremental environment architecture, debugging, and code generation aspects, and could generate parts of incremental programming environments from specifications. However, I always wished for a good opportunity to attack the semantics problem in a practical context.

This opportunity came in the fall of 1989, when Mikael Pettersson started as a graduate student in the PELAB research group (Programming Environments Laboratory) at Linköping University, of which I recently had become leader. Mikael shared my interest in efficient programming language implementations and generator tools, and was very research minded from the start. Already during his undergraduate studies he read many research papers on language implementation techniques and formal semantics, at that time primarily Denotational Semantics, and did several experimental designs. This also became the topic of his first 2–3 years of research, on the problem of generating fast compilers that emit efficient code from denotational specifications. Mikael developed the DML system (Denotational Meta Language), which included a subset of Standard ML, concrete syntax notation for pattern matching, and implementation techniques that made generated compilers both execute quickly and emit efficient code.

Around 1990–92, we became increasingly aware of the Natural Semantics specification formalism, which was gaining in popularity and seemed to be rather easy to use, yet providing high abstraction power and good modularity properties. During 1992 my group had just started to cooperate in an Esprit project with Gilles Kahn's group at the Sophia-Antipolis branch of INRIA in France. Kahn proposed the Natural Semantics formalism 1985, and his group had since then developed the Centaur system which includes Typol as the meta-language and tool for Natural Semantics specifications. This system provides a nice interactive environment for prototyping language specifications; however, generated compilers and interpreters execute quite slowly.

Therefore, Mikael, with my support, decided to slightly change his Ph.D. thesis topic. The new goal was to develop techniques and tools for generating very efficient language implementations from Natural Semantics specifications. Generated implementations should be comparable with or better than hand-written ones in performance. This was an ambitious goal that nobody had realized before.

After studying the new area, analyzing possible approaches, and implementation techniques used by logic programming and functional programming languages, Mikael completed the first RML prototype in February 1994. At that time RML was both relational and non-deterministic, similar to most logic programming languages, so the name *Relational Meta Language* was appropriate. Even the first prototype was rather efficient compared to Typol, but better was to come. Mikael observed that the great majority of language specifications in Natural Semantics are deterministic. Only seldom is non-determinism really needed, and in those cases specifications can usually be reformulated in a deterministic way. Mikael decided to make RML deterministic to enable further improvements in efficiency. A year later, with the addition of more sophisticated optimizations in the RML compiler, the generated code had improved another factor of five in performance. You can read about all the details in this book.

I am very proud of Mikael's work. His RML system is the first generator tool for Natural Semantics that can produce really efficient implementations. The measured efficiency of generated example implementations seems to be roughly the same as (or sometimes better than) comparable hand implementations in Pascal or C. Another important property is compatibility and modularity. Generated modules are produced in C, and can be readily integrated with existing frontends and backends.

I feel quite enthusiastic about the future prospects of automatically generating practically useful implementations from formal specifications of programming languages, using tools such as RML. Perhaps we will soon reach the point where ease of use and efficiency of the generated result will make it as attractive and common to generate semantic processing parts of translators from Natural Semantics specifications, as is currently the case for generating scanners and parsers using tools such as Lex and Yacc. Only the future will tell.

Linköping, October 1998
Peter Fritzson

Preface

Abstract

Natural semantics has become a popular tool among programming language researchers. It is used for specifying many aspects of programming languages, including type systems, dynamic semantics, translations between representations, and static analyses. The formalism has so far largely been limited to theoretical applications, due to the absence of practical tools for its implementation. Those who try to use it in applications have had to translate their specifications by hand into existing programming languages, which can be tedious and error-prone. Hence, natural semantics is rarely used in applications.

Compiling high-level languages to correct and efficient code is non-trivial, hence implementing compilers is difficult and time-consuming. It has become customary to specify *parts* of compilers using special-purpose specification languages, and to compile these specifications to executable code. While this has simplified the construction of compiler front-ends, and to some extent their back-ends, little is available to help construct those parts that deal with *semantics* and translations between higher-level and lower-level representations. This is especially true for the Natural Semantics formalism.

In this thesis, we introduce the Relational Meta-Language, RML, which is intended as a practical language for natural semantics specifications. Runtime efficiency is a prerequisite if natural semantics is to be generally accepted as a *practical* tool. Hence, the main parts of this thesis deal with the problem of compiling natural semantics, actually RML, to highly efficient code.

We have designed and implemented a compiler, `rml2c`, that translates RML to efficient low-level C code. The compilation phases are described in detail. High-level transformations are applied to reduce the usually enormous amount of non-determinism present in specifications. The resulting forms are often completely deterministic. Pattern-matching constructs are expanded using a pattern-match compiler, and a translation is made into a continuation-passing style intermediate representation. Intermediate-level CPS optimizations are applied before low-level C code is emitted. A new and efficient technique for mapping *tailcalls* to C has been developed.

We have compared our code with other alternative implementations. Our

benchmarking results show that our code is much faster, sometimes by orders of magnitude. This supports our thesis that the given compilation strategy is suitable for a significant class of specifications.

A natural semantics specification for RML itself is given in the appendix.

Acknowledgements

I thank my thesis supervisor Peter Fritzson for giving me free reins to explore my interests in formal semantics and language implementation technology. I also thank the members of my thesis committee, Reinhard Wilhelm, Isabelle Attali, Tore Risch, and Björn Lisper, for their interest in my work, and my friends and colleagues at the Department of Computer Science at Linköping University. And finally, I thank my family for being there.

Addendum

This book is a revised version of the Ph.D. dissertation I defended in December 1995 at the University of Linköping. The RML system has evolved in several directions since then, and I summarize the main developments here.

In the RML type system, implicit logical variables have been replaced by a polymorphic type `'a lvar` with explicit binding and inspection operators, and the notion of *equality types* has been borrowed from Standard ML.

Top-level declarations are now subject to a dependency analysis and reordering phase before type checking, as in Haskell [129, Section 4.5.11].

More techniques for implementing tailcalls in C have been tested, including one used by two Scheme compilers [63]. However, no real performance improvements have been achieved to date.

The RML compiler has been made much more user-friendly. The type checker now gives accurate and relevant error messages, and a new compiler driver automates the many steps involved in compiling and linking code. Work is underway to support debugging and profiling [139].

Students at Linköping University have used the system to construct compilers for real-world languages, including Java and Modelica [95]. The experience has been positive, but a simplified foreign C code interface, a debugger, and support for more traditional programming are sometimes requested.

The Swedish National Board for Industrial and Technical Development (NUTEK) and the Center for Industrial Information Technology (CENIIT) supported my research at Linköping University. Recent developments where implemented during my postdoc at INRIA Sophia-Antipolis 1997–98, funded by the Swedish Research Council for Engineering Sciences (TFR).

Uppsala, October 1998
Mikael Pettersson