A Tool to Support Formal Reasoning about Computer Languages*

Richard J. Boulton

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, United Kingdom

November 1996

Abstract

A tool to support formal reasoning about computer languages and specific language texts is described. The intention is to provide a tool that can build a formal reasoning system in a mechanical theorem prover from two specifications, one for the syntax of the language and one for the semantics. A parser, pretty-printer and internal representations are generated from the former. Logical representations of syntax and semantics, and associated theorem proving tools, are generated from the combination of the two specifications. The main aim is to eliminate tedious work from the task of prototyping a reasoning tool for a computer language, but the abstract specifications of the language also assist the automation of proof.

1 Introduction

For several decades theorem proving systems have been used to reason about computer languages. A common approach has been to define the semantics of a language in the logic of the theorem prover. Properties of particular language texts can then be proved. With some approaches it is also possible to prove general properties about the language itself. The technique has been referred to as *embedding*.

Examples of embedding include an early compiler correctness proof for a simple ALGOL-like language [MW72] mechanised in Stanford LCF [Mil72], Moore's

*Research supported by the Engineering and Physical Sciences Research Council of Great Britain under grant GR/J42236.

work on the Piton assembly language [Moo89] using an operational semantics in the Boyer-Moore logic [BM79], Gordon's axiomatic semantics for a simple imperative language [Gor89] in the HOL system [GM93], embeddings of various hardware description languages [Goo91, BHY92, BGG+92, Ree95], a compiler correctness proof for a simple functional programming language in the LF logical framework [HP92], and reasoning about the core of Standard ML [Sym93, VG93] and its module system [GM95]. Formalisms such as temporal logics and process algebras have also been embedded in theorem proving systems.

Increasingly the embedding technique is being applied to industrial-strength computer languages. This creates problems akin to those arising when trying to write a large program in an assembly language — the level of description is too low. Generating an embedding is tedious and error-prone. Furthermore, changes to the syntax of the language (or more likely the subset of the language being considered) may require changes to the abstract syntax representation, the parser, the pretty-printer, the definition of semantics, and the associated theorem proving tools. Keeping all these entities consistent is difficult and time-consuming. However, the real information content of the parser, etc., is simply the syntax and semantics of the language. It should be possible, therefore, to generate an embedding from high-level specifications of syntax and semantics. This would not only reduce development and maintenance times but would also allow the embedding to be retargeted to a different theorem prover much as compilers allow a program in a high-level language to be retargeted to different architectures and operating systems.

This paper is an overview of a suite of tools for generating embeddings from high-level specifications of syntax and semantics. The tools for syntax are fairly mature and have been used in formal reasoning projects for the C programming language and the hardware description languages VHDL, Verilog, and ELLA. The language for specifying syntax is unusual in allowing the form of the abstract syntax trees (ASTs), the lexical analysis, the parsing, and pretty-printing information, all to be given in a single non-redundant formalism. Details of the language can be found in a separate paper [Bou96]. The tools for semantics are still under development. Collectively the tools are called "CLaReT" which is an abbreviation for "Computer Language Reasoning Tool". CLaReT has been developed within the framework of a wider project. This project aims to provide formal methods support for the design of application-specific integrated circuits (ASICs) using multiple hardware description languages at various levels of abstraction.

2 How CLaReT Might Be Used

CLaReT is designed to generate code for a theorem proving system that has both an object logic and a meta-language, ML[1]. From high-level specifications of the syntax and semantics of a language $\mathcal{L}$, the following can be generated:

- representations of the abstract syntax in ML and in logic;
- functions to map between these two representations;
- a parser and a pretty-printer;
- logical definitions for the semantics;
- ML functions and logical inference rules to animate the semantics.

To see how these might be used, suppose that we want to verify a program $\mathcal{P}$ written in $\mathcal{L}$. We first parse it to obtain an internal representation in ML. The program can then be tested on various data by applying the fast ML animation functions. This testing is with respect to the formal semantics and could equally well be used to test the semantics. After one or more cycles of modification and animation, we are happy with the results. We might then wish to formally verify $\mathcal{P}$. To achieve this the ML representation is converted to logic and the property $\mathcal{S}$ we wish to prove is specified in the logic. The theorem prover is used (to attempt) to prove that $\mathcal{P}$ satisfies $\mathcal{S}$ with respect to the semantics. The proof may require that the semantics be 'executed', which can be achieved using the animation inference rules. These are not used for the initial testing because they are much slower than the ML functions.

3 An Overview of CLaReT

CLaReT is implemented in Standard ML [MTH90], a functional programming language, and currently also has ML as its target language. The Standard ML of New Jersey implementation is used because it provides the lexer and parser generating tools ML-Lex and ML-Yacc. These tools generate Standard ML code in much the same way as the Lex and Yacc tools do for the C programming language. A somewhat simplified view of the architecture of CLaReT is shown in Fig. 1. The software around which CLaReT has been built is indicated by dotted lines.

The first component to be built was a pretty-printer for the abstract syntax of ML. This provided a code generator for all the tools that have ML as their target

[1] In this section 'ML' refers to any meta-language but, as described later, the current implementation uses the programming language of the same name. This is not a coincidence; the ML programming language evolved from the meta-language [GMM+78] of the LCF theorem prover [GMW79].

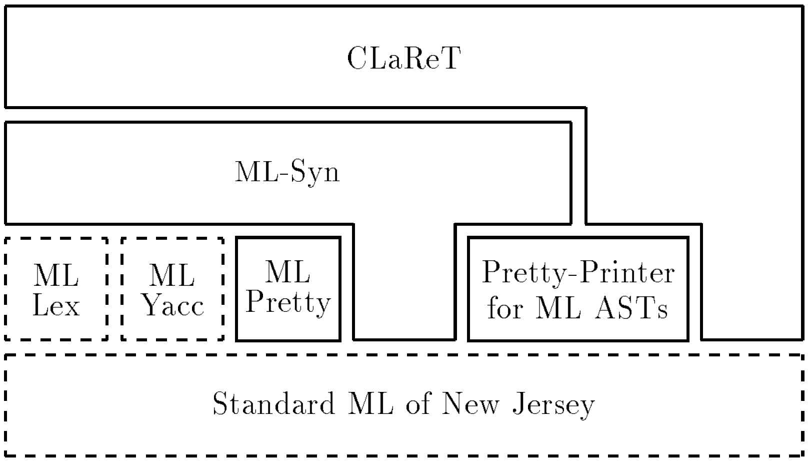

Figure 1: Simplified architecture of CLaReT

language. Each such tool generates an ML AST and passes it to the pretty-printer to produce an output file. So, the tools do not have to be concerned with the concrete syntax of ML, and because pretty-printing is used the output can easily be read by the user.

The second component is the ML-Pretty program. This is a pretty-printer generator. It takes a specification language as input and produces ML as output. It is a self-contained program that should be of general use to people developing systems in ML. The pretty-printers generated by ML-Pretty can maintain a link between positions in the generated text and the AST being printed. This allows them to be used in a graphical user interface.

The next level of the system is called ML-Syn. It takes a single specification for syntax (an extended BNF grammar) and produces input for ML-Lex, ML-Yacc, and ML-Pretty. The specification language, called Syn [Bou96], is at a higher level than the languages used by ML-Lex, etc., and other syntax-specification languages could be generated from it. ML-Syn overcomes the problem of maintaining consistency between the abstract syntax representations used by the parser, pretty-printer and other tools by generating them from one specification.

Like ML-Pretty, ML-Syn is self-contained, so it can be used by people who have no interest in semantics or formal reasoning. ML-Syn generates its output as ASTs which can either be pretty-printed to files or, in the case of ML-Pretty, be fed directly into the ML-Pretty compiler, bypassing its parser. Pretty-printing the files allows them to be read easily by the user and, if necessary, to be modified. Thus, ML-Syn can be used to rapidly generate a parser and pretty-printer which can then be fine-tuned manually, the original Syn specification being discarded.

Finally, CLaReT uses ML-Syn to handle concrete syntax and to obtain the form of the abstract syntax. CLaReT produces additional code for use with a version of the HOL theorem proving system. This version of HOL is implemented on top of Standard ML, so the code for concrete syntax can be used with it. The abstract syntax information is used to generate definitions of types in the HOL logic (higher-order logic), and ML functions to map between the ML representation of ASTs and the representation in logic.

CLaReT also takes a specification of semantics as input. Currently, it has to be in a denotational style or as attributation and translation rules. Structured operational semantics may be supported in the future.

From a denotational specification CLaReT generates definitions of logical functions over the abstract syntax and, if required, analogous ML functions. The latter allow rapid 'execution' of the semantics. For rigorous execution a *symbolic evaluator* [CZ94] is also generated. This uses logical inference rules to ensure the correctness of the evaluation. Such symbolic evaluators can be implemented as a brute-force application of the semantic functions as rewrite rules. However, evaluators written in this fashion are notoriously slow. It is better to make use of the abstract syntax specification to selectively apply the semantic functions at only the points at which they are applicable.

The remainder of this paper goes into more detail about the various features of CLaReT. A fragment of a simple imperative programming language is used as an example.

4 Specification of Syntax

Here is a Syn specification for the syntactic category of commands in a simple imperative programming language:

```
com ::= (Skip) "skip"
      | (Assign) [<hv 1,3,0> [<h 1> name ":="] iexp]
      | (If) [<hov 1,0,0> [<h 1> "if" bexp]
                          [<h 1> "then" com]
                          [<h 1> "else" com]?]
      | (While) [<hov 1,3,0> [<h 1> "while" bexp "do"] com]
      | (Block) [<hov 1,3,0>
                     "begin" ([<h 0> com ";"]* com) <1,0,0> "end"];
```

Some features to note are:

- Optional and repeated syntactic elements can be specified directly using the notation `[...]?` and `[...]*` respectively.
- One notation acts as both a means of specifying options and repetitions, and as a specification of layout for pretty-printing. The `<...>` notation is formatting information.
- The names of the nodes to be used in the ASTs are given in parentheses at the start of each line. The number and type of the subtrees are deduced from the non-terminals.

- The precedence (binding strength) of terminals is specified implicitly by textual ordering with the aid of dependency analysis on the non-terminals.

In addition, but not illustrated here, high-level constructs are available to specify lexical features such as character strings and comments which cannot always be adequately expressed as regular expressions.

The ML datatype generated to represent the abstract syntax is:

```
datatype com
   = Skip
   | Assign of name * iexp
   | If of bexp * com * com option
   | While of bexp * com
   | Block of com list
```

Notice the use of an option type and lists for the optional and repeated non-terminals. The option type is defined in ML by:

```
datatype 'a option = NONE | SOME of 'a
```

The logical types have much the same form as the ML types and are generated using one of the automatic type definition packages in HOL [Mel89, Gun93].

5 Denotational Semantics

It is the author's intention that the denotational semantics specification language should look similar to the non-mechanised semantics one encounters in research papers, though the ASCII character set is obviously a constraint. Thus, `[|...|]` is used for semantic bracketing and `<<...>>` denotes a meta-variable which ranges over a syntactic category. The specification for commands given below is written over the abstract syntax:

```
[| Skip |] == ();;
[| Assign(<<name>>,<<iexp>>) |] == !<<name>> <- [|<<iexp>>|];;
[| If(<<bexp>>,<<com.1>>,{}) |] ==
   if [|<<bexp>>|] then [|<<com.1>>|] else ();;
[| If(<<bexp>>,<<com.1>>,{<<com.2>>}) |] ==
   if [|<<bexp>>|] then [|<<com.1>>|] else [|<<com.2>>|];;
[| While(<<bexp>>,<<com>>) |] ==
   if [|<<bexp>>|]
   then ([|<<com>>|]; [| While(<<bexp>>,<<com>>) |])
   else ();;
[| Block(<<[coms]>>) |] == ([|<<coms>>|]; ());;
```

The right-hand sides of the definitions are written in a simple ML-like language. The intention is that it should be compilable to both ML and logical function definitions. The similarity between ML and the HOL logic makes this requirement easier to achieve than if a much less ML-like logic were being used. Nevertheless, there are some difficulties:

- ML has a call-by-value semantics whereas the logic of HOL is inherently lazy — evaluation has to be forced by applying inference rules. The term-traversal strategy for rule application determines the 'evaluation' order.

- Properties can be specified abstractly in the HOL logic whereas everything must be 'implemented' in ML, e.g. the existential quantifier '$\exists$' is directly admissible in HOL but needs to be implemented as a function in ML. It is not clear that this can be done in general (at least not efficiently). Practical experience is required to determine the extent to which quantifiers, etc., should be allowed in the specification language. Camilleri [Cam88] and, more recently, Rajan [Raj92] have investigated the execution of logical formulas in the HOL system.

5.1 Denotation Language Features

The specification language has built-in support for environments (or states, as appropriate). The intention is that these be implicit wherever possible to avoid verbosity. Thus it is assumed that the first denotation (`[|...|]`) on the right-hand side is 'evaluated' in the incoming environment, the second in the environment resulting from the first evaluation, and so on. Mechanisms are included to override this default behaviour.

When the value of the first denotation is to be discarded the sequencing notation (`...;...`) may be used, as illustrated in the semantics for `While` and `Block`. The components of a sequence are processed from left to right for their side effects and the value of the last component becomes the value of the entire sequence expression. For the `Block` construct the denotation of a list of commands is a list of null values plus a side effect on the state. For the semantics to be correctly typed a single null value must be returned in place of the list.

The conditional `if ... then ... else ...` expression has a lazy semantics (as in ML); only one of the branches is 'evaluated'.

Special notation is provided for obtaining values from the environment and for updating it. '`!<<name>>`' denotes the value bound to the name `<<name>>` in the current environment, and

```
!<<name>> <- x
```

binds the value of `x` to `<<name>>`. In more complex examples, the environment may have to have several components because values of more than one type have to be bound. Constructs for this and other extensions will be provided in the future.

Similar denotational specifications can be found in Lee's doctoral dissertation [Lee89] and in earlier work. Lee's specifications do not appear to have special support for environments. However, they do use an ML-like language which suggests that ML is well suited to the task of formalising a computer language. Lee's work is directed towards generation of efficient compilers rather than formal reasoning systems.

5.2 Generated ML Functions

To illustrate some of the above points, here is the ML function declaration that might result from compiling the specification:

```
fun den_of_com Skip = unitS ()
  | den_of_com (Assign (name,iexp)) =
   bindS (den_of_iexp iexp,fn i1 => unitS () o set name i1)
  | den_of_com (If (bexp,com1,NONE)) =
   bindS (den_of_bexp bexp,
          fn b1 => if b1 then den_of_com com1 else unitS ())
  | den_of_com (If (bexp,com1,SOME com2)) =
   bindS (den_of_bexp bexp,
          fn b1 =>
             if b1 then den_of_com com1 else den_of_com com2)
  | den_of_com (While (bexp,com)) =
   bindS (den_of_bexp bexp,
          fn b1 =>
             if b1
             then bindS (den_of_com com,
                         fn c1 => den_of_com (While (bexp,com)))
             else unitS ())
  | den_of_com (Block coms) =
   bindS (den_of_list den_of_com coms,fn z1 => unitS ());
```

The functions `unitS` and `bindS` are used to 'thread' the environment through the evaluations. They are based on the *monad of state transformers* used in the functional programming community [Wad92]. Their ML definitions are:

```
fun unitS x s0 = (x,s0);
fun bindS (m,f) s0 = (fn (x,s1) => f x s1) (m s0);
```

Monads are similar to continuation-passing style which was invented for use with denotational semantics. The use of monads in denotational semantics was proposed by Moggi [Mog91].

The function `set` binds a key to a value in the environment. The functions for manipulating bindings and environments are provided as modules implemented as both Standard ML structures and HOL theories (see Sect. 5.5).

Since environments do not have to be mentioned explicitly in the ML code that is generated from the specification, one might ask why the ML is not used directly. The primary reason for not doing so is the desire to generate other things from the specification including logical inference rules (Sect. 5.4) that have structures that do not so closely follow that of the specification.

5.3 Generated HOL Definitions

The HOL definitions generated from the denotational specification are quite similar to the ML code. The logical counterparts of `unitS` and `bindS` are used, and the specification language is deliberately restricted to constructs that can be readily represented in higher-order logic. Even so, the functions to be defined may be mutually recursive. The HOL theorem prover has a tool for making mutually recursive definitions as do a number of other provers.

The example at the beginning of Sect. 5 involves a recursion that is not well-founded: The denotation of the `While` construct is defined in terms of itself. This can easily be implemented in ML (possibly resulting in a non-terminating program) but is problematic in HOL. The use of fixpoints for this is being investigated. The difficulty is not in defining the recursion but in doing it in a way that facilitates symbolic evaluation. In any event, other styles of semantics can be used that avoid the problem.

5.4 Generated Inference Rules

The ML version of the denotational semantics can be evaluated by simply applying the denotation functions to the abstract syntax and the initial environment. However, this does not allow parts of the syntax or the environment to be 'symbolic', i.e. a meta-variable, as is allowed in the logic of the theorem prover [CZ94]. On the other hand, evaluation in the logic requires the definitions of the denotation functions (and any auxiliary functions used) to be applied as rewrite rules. Writing such an evaluator by hand is straightforward but time-consuming and error-prone. CLaReT generates the evaluator automatically. A further advantage is that the generator can be programmed to produce an efficient rewriter, a skill that casual users of the HOL system are unacquainted with.

Another option is to produce a hybrid evaluator. The idea is to perform the environment manipulations, etc., directly in ML, while the basic values being manipulated are logical terms. This approach produces a fast evaluator that also allows some symbolic entities. Rajan [Raj92] describes the necessary translation between terms and ML programs, and the current author has described a theorem proving framework in which the use of a hybrid evaluator can be mixed with normal logical inferences [Bou94]. Kaufmann and Moore take a different approach in the ACL2 theorem prover [KM94]; their logic is an applicative sublanguage of

Common Lisp, so their terms are inherently executable. The drawback is that this language lacks the expressive power of higher-order logic.

The ML functions to evaluate the semantic definitions in the logic are functions that map a logical term to an equational theorem between that term and a new term. These *conversions* are built up from applications of rewrite rules using combinators for congruence rules, sequencing, etc. This approach was suggested by Paulson [Pau83] and is heavily used in the HOL system. Part of the conversion for commands in the example language is illustrated below.

```
fun den_of_com_CONV key_eq_conv tm =
   (case constructor_of_den_app (rator tm)
    of ...
     | "Imp_Block" =>
       RATOR_CONV Block_REWR THENC
       TRY_CONV (BIND_S_CONV (den_of_coms_CONV key_eq_conv)) THENC
       TRY_CONV (RATOR_CONV BETA_CONV) THENC
       TRY_CONV
          (RATOR_CONV
              (RAND_CONV
                  (STRICT_EVAL_CONV
                      (LIBRARY_OP_CONV key_eq_conv []))))) THENC
       TRY_CONV UNIT_S_CONV)
```

If the term to which the denotation function is applied has `Imp_Block`[2] at its head then the definition of the semantic function for that constructor is used as a rewrite rule. This is implemented by the conversion `Block_REWR`. The combinator `RATOR_CONV` applies the conversion to the operator of an application. It is used because the term will be of the form:

```
(Imp_den_of_com (Imp_Block coms)) state
```

The result is an equational theorem with the following term as its right-hand side:

```
((BIND_S (Imp_den_of_coms coms)) (λz1. UNIT_S one)) state
```

The infix combinator `THENC` sequences conversions. It arranges for the next conversion to be applied to this new term. The next conversion tries to evaluate the `BIND_S` function. The result is:

```
(λ(x,s1). ((λz1. UNIT_S one) x) s1) (Imp_den_of_coms coms state)
```

If the application of `Imp_den_of_coms` succeeds to yield a value/state pair `(y,s)` then this evaluates further to:

[2]The names generated by CLaReT for use with HOL are prefixed by the language name (`Imp`) because HOL has a global name space for logical constants. For the ML version, ML's structures (modules) are used to avoid naming conflicts.

```
((λz1. UNIT_S one) y) s
```

The next conversion tries to beta-reduce the operator:

```
(UNIT_S one) s
```

The remaining conversions in the sequence attempt a general strict evaluation using library functions (which is actually unnecessary in this case because `one` is already fully evaluated) followed by evaluation of the `UNIT_S` function. The form of the whole conversion is derived mechanically from the denotational specification.

The state binds keys (e.g. variable names) to values. A means of computing whether two keys are equal is required in order to symbolically evaluate. The parameter `key_eq_conv` is a conversion that does this.

5.5 A Library of Modules

As can be seen from the preceding sections, the ML functions and the HOL inference rules differ in structure. The names used for particular functions also differ between the targets. For this reason, CLaReT includes a library mechanism. Functions are grouped together in modules, e.g. for standard types like the integers. For each module the library contains a specification file and implementation files. The specification file is used to map names occurring in the denotation language to names to be used in the generated code. In some cases the target names will be built-in functions of ML or HOL. In other cases the definitions are stored in the implementation files. The denotational semantics specification language includes a construct that allows users to specify which modules they wish to use.

With the library mechanism it would be easy to add implementation files for theorem provers other than HOL. It is also possible for a specialist user to implement modules for a particular application area, such as semantics of structural hardware description languages, which can then be used by someone unfamiliar with the intricacies of the theorem prover in order to specify the semantics of a language. Since the libraries include proof procedures for the functions, it may be possible in this way to provide a high degree of proof automation without the language specifier needing to know how to implement proof procedures.

6 Comparison with Other Systems

6.1 The Reetz/Kropf Embedding Generator

The idea of automatically generating embeddings is not new. Reetz and Kropf [RK94, Ree95] have produced a system that generates an embedding in the HOL theorem prover from specifications of the grammar of the language and attributation and translation rules for attributed abstract syntax trees (derivation trees).

The semantic information is stored in the attributes rather than in the environment argument used in the denotational style (Sect. 5).

The Reetz/Kropf embedding generator does not deal with concrete syntax, i.e. it does not generate parsers or pretty-printers. For realistically-sized language texts these are important; entering an abstract syntax tree for such a text is tedious and error-prone. CLaReT has been interfaced to their system to provide support for concrete syntax. Since CLaReT also supports a different style of semantics it is complementary to the work of Reetz and Kropf.

6.2 Software Development Environments

There are a number of language-independent software development environments that provide similar features to CLaReT, e.g. CENTAUR [BCD+88], the Ergo Support System (ESS) [LPRS88], the programming system generator PSG [BS86], and the Synthesizer Generator [RT89]. These systems have not been used by researchers who embed computer languages in interactive theorem provers. This suggests that the effort involved in integrating such systems to theorem provers for one-off embeddings is prohibitive. An alternative to developing CLaReT would have been to provide a generic interface between such a system and HOL. However, CLaReT has the advantage that it produces code in a general purpose programming language with all the flexibility that provides. For example, the code produced for syntactic support can be integrated into any system implemented in ML, and where appropriate the code fragments produced are independent. Thus, CLaReT can be (and has been) used by people who simply wish to generate a parser for an ML program. In contrast, software development environments tend to be stand-alone and heavyweight.

The aim is for CLaReT to minimise the amount of effort required to prototype a reasoning tool for a language. To this end, CLaReT trades off some flexibility in the specifications for simplicity, maintainability and automation. We conjecture that constraining the form of specification is a key to better automation, especially of the theorem proving aspects. The challenge is to strike a good balance between expressive power and simplicity of use.

CLaReT is also designed to be open. The data structures that abstractly represent the information required as input to parser generators, etc., are available and documented. The intention is to make it easy to retarget the output for other tools and other programming languages.

6.2.1 Centaur and ASF+SDF

CENTAUR has two collections of specification languages: ASF+SDF [BHK89] (a combination of ASF and SDF) and Metal/PPML/Typol. SDF [HHKR89] and Metal [KLMM83] are specification languages for concrete and abstract syntax. Neither of these specify formatting but PPML [MC86] (a pretty-printing specification language) may be used with Metal, and recently van den Brand and Visser [vdBV96] have shown how default pretty-printers can be generated from SDF. Both the latter work and the formatting in CLaReT's Syn language are based on PPML. For a discussion of how Syn compares with the syntactic specification languages of CENTAUR and the other software development environments, see the paper on Syn [Bou96].

ASF is an algebraic specification language and Typol [Des84] implements Kahn's natural semantics [Kah87]. ASF specifies semantics by means of conditional equations. The equations can be written over a concrete syntax specified in SDF. Currently, CLaReT is limited to specifications over the abstract syntax. The use of SDF also allows the syntax of (meta-)variables to be specified so that the ASF specifications are not restricted to a fixed syntax such as the `<<...>>` used in CLaReT. Typol is particularly suited to static semantics, e.g. type checking, and there exists a means of translating natural semantics in Typol to inductive definitions in the Coq theorem prover [Coq94].

It is questionable whether the ASF language would be accepted by researchers who use theorem provers to reason about languages, since it limits them to using equations for specification and rewriting for proof. The expressive power of higher-order logics and the extensive library of commonly required theories found in the HOL system are often exploited. This is not to deny the usefulness of ASF for other purposes; it is simply suggesting that a formal semantics that is suitable as a specification for compiler writers may not be practical as a medium for proving correctness of programs.

6.2.2 The Ergo Support System

ESS has very similar aims and facilities to CLaReT. Like CLaReT, ESS has a deduction (theorem proving) aspect, though this is perhaps more the focus of attention in CLaReT. Syntax is specified in ESS using a single language that like CLaReT has iterators, unparsing annotations, and high-level lexical specification. However, CLaReT's Syn language has more implicit features such as inferring precedence and the form of the ASTs. ESS makes use of attributes and higher-order abstract syntax which CLaReT currently does not. In terms of semantic specification the difference between the two systems is more significant: CLaReT has declarative specifications for semantics while in ESS the semantic aspects of a language have to be implemented as programs.

6.2.3 The Programming System Generator

PSG generates interactive programming environments from specifications of syntax, context conditions, and dynamic semantics. The various aspects of syntax are specified separately, so there is a lot of redundancy, which is something CLaReT tries to avoid. In contrast to ESS but like ASF and CLaReT, PSG allows semantics to be specified non-procedurally. The dynamic semantics of a language is defined in a denotational style using a functional language based on the lambda calculus. This is very similar to CLaReT but whereas CLaReT's primary concern is to support formal proof using the semantics, in PSG the semantics is used only to execute program fragments. Execution, if the semantics permits it, is a secondary concern in CLaReT. Another difference is that, in PSG, states and environments used in the semantics apparently have to be mentioned explicitly.

In PSG, program fragments are compiled to terms of the functional language which are then executed by an interpreter. In contrast, the ML functions generated by CLaReT take the abstract syntax as a parameter. Partial evaluation [JGS93] could be used to achieve the same effect as in PSG. For the logical definitions generated by CLaReT it is important to maintain parameterisation over the abstract syntax so that general properties of the language can be proved.

Execution of program fragments with CLaReT can be done in both ML and in logic. In the latter case the fragments may contain meta-variables (place holders for pieces of syntax) resulting in an expression involving these variables instead of a constant value. PSG, on the other hand, requires that place holders be instantiated before execution can proceed.

6.3 A Generic State Machine Generator

The SMG system [GB88] supports temporal logic model checking for languages by transforming programs to suitable finite state models. The languages are specified by syntax and structured operational semantics, and the tool interfaces to various temporal logic model checkers. This differs from CLaReT in the style of semantics and in targeting model checkers rather than theorem provers.

6.4 Semantics-Directed Compiler Generators

There have been a number of attempts to generate compilers from denotational semantic specifications including early work by Mosses [Mos79], and later by Paulson [Pau82], Wand [Wan84], and Lee [Lee89]. One tool in current use is Actress [BMW92], a semantics-directed compiler generator for Mosses' action semantics [Mos92]. This uses ML-Lex and ML-Yacc to generate a parser, so its syntactic specification is at a lower level than in CLaReT, but the big difference is that Actress is not intended to support formal reasoning. We are not aware of any language embeddings in theorem provers that are based on action semantics, possibly because the theoretical underpinnings required for action semantics are not present in current provers.

7 Summary and Future Work

This research gathers together a number of technologies that have previously been used manually or in isolation and makes them readily available to anyone interested in formal reasoning about computer languages. By targeting a commonly used theorem proving system, users who wish to prove properties about their languages and programs have the tools to do so at their disposal, and the support of a substantial user community. Some key points of the research are:

- There is only one specification for syntax and one for semantics.
- CLaReT attempts to hide logic and theorem proving to make the tools more accessible to software engineers, hardware designers, etc.
- Limiting the expressive power of the specification languages allows greater automation. Modules of functions are provided so that the system may know how to reason about them without user intervention.
- The similarity between ML and higher-order logic makes it easier to exploit both the meta-language and the logic of the theorem proving system.

Future work may include:

- extension of the denotational semantics specification language to allow compound environments, auxiliary functions, etc.;
- support for semantic specifications over concrete syntax;
- providing a means of specifying transformations on language texts and automating proof of the resulting transformation theorems;
- enhancements to the syntactic specification language, e.g. support for sublanguages;
- other styles of semantics, e.g. rule-based operational semantics using one of the packages in HOL for making inductive definitions [Mel91, Har95];
- retargeting of the embedding to other theorem proving systems (a number of which provide the means to define new recursive types in the logic similar to the data types of Standard ML).

Acknowledgements

Thanks to Matthias Mutz (Passau), Ralf Reetz (Karlsruhe), Michael Norrish (Cambridge) and especially Daryl Stewart (Cambridge) for feedback on CLaReT. Discussions with Juanito Camilleri (Malta) inspired part of this research. Mike

Gordon (Cambridge) gave helpful comments on a draft of this paper, and he and Peter Homeier (UCLA) provided pointers to the literature. Thanks also to the anonymous referees for their comments.

References

[BCD+88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In Henderson [Hen88], pages 14–24.

[BGG+92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In Stavridou et al. [SMB92], pages 129–156.

[BHK89] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press in co-operation with Addison-Wesley, 1989.

[BHY92] B. C. Brock, W. A. Hunt, Jr., and W. D. Young. Introduction to a formally defined hardware description language. In Stavridou et al. [SMB92], pages 3–35.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, New York, 1979.

[BMW92] D. F. Brown, H. Moura, and D. A. Watt. Actress: an action semantics directed compiler generator. In U. Kastens and P. Pfahler, editors, *Proceedings of the 4th International Conference on Compiler Construction (CC'92)*, volume 641 of *Lecture Notes in Computer Science*, pages 95–109, Paderborn, FRG, October 1992. Springer-Verlag.

[Bou94] R. J. Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, U.K., May 1994. Technical Report 337.

[Bou96] R. J. Boulton. Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical Report 390, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK, March 1996.

[BS86] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[BS89] G. Birtwistle and P. A. Subrahmanyam, editors. *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[Cam88] A. J. Camilleri. *Executing Behavioural Definitions in Higher Order Logic*. PhD thesis, University of Cambridge, July 1988. Available as University of Cambridge Computer Laboratory Technical Report 140.

[Coq94] INRIA Rocquencourt and ENS Lyon. *The Coq Proof Assistant Reference Manual*, version 5.10 edition, 1994.

[CZ94] J. Camilleri and V. Zammit. Symbolic animation as a proof tool. In Melham and Camilleri [MC94], pages 113–127.

[Des84] T. Despeyroux. Executable specification of static semantics. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Proceedings of the International Symposium on the Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, Sophia-Antipolis, France, June 1984. Springer-Verlag.

[GB88] G. D. Gough and H. Barringer. A semantics driven temporal verification system. In H. Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming (ESOP'88)*, volume 300 of *Lecture Notes in Computer Science*, pages 21–33, Nancy, France, March 1988. Springer-Verlag.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[GM95] E. Gunter and S. Maharaj. Studying the ML module system in HOL. *The Computer Journal*, 38(2):142–151, 1995.

[GMM+78] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 119–130, Tucson, Arizona, January 1978. Also issued as Report CSR-16-77, Department of Computer Science, University of Edinburgh, 1977.

[GMW79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Goo91] K. G. W. Goossens. Embedding a CHDDL in a proof system. In *Proceedings of IFIP TC 10/WG 10.2 Advanced Research Workshop on Correct Hardware Design Methodologies (CHARME'91)*, pages 369–386, Turin, Italy, June 1991. Also available as University of Edinburgh LFCS, Technical Report ECS-LFCS-91-155, May 1991.

[Gor89] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In Birtwistle and Subrahmanyam [BS89]. Also available as University of Cambridge Computer Laboratory Technical Report 145.

[Gun93] E. L. Gunter. A broader class of trees for recursive type definitions for HOL. In Joyce and Seger [JS93], pages 141–154.

[Har95] J. Harrison. Inductive definitions: Automation and application. In Schubert et al. [SWAF95], pages 200–213.

[Hen88] P. Henderson, editor. *ACM SIGSOFT'88: Third Symposium on Software Development Environments* (*ACM SIGSOFT Software Engineering Notes*, 13(5), and, *ACM SIGPLAN Notices*, 24(2)), Boston, Massachusetts, November 1988.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — reference manual —. *ACM SIGPLAN Notices*, 24(11):43–75, November 1989.

[HP92] J. Hannan and F. Pfenning. Compiler verification in LF. In A. Scedrov, editor, *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, 1993.

[JS93] J. J. Joyce and C.-J. H. Seger, editors. *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications (HUG'93)*, volume 780 of *Lecture Notes in Computer Science*, Vancouver, B.C., Canada, August 1993. Springer-Verlag, 1994.

[Kah87] G. Kahn. Natural semantics. In F. J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS 87)*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39, Passau, Federal Republic of Germany, February 1987. Springer-Verlag.

[KLMM83] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: A formalism to specify formalisms. *Science of Computer Programming*, 3(2):151–188, 1983.

[KM94] M. Kaufmann and J S. Moore. Design goals of ACL2. Technical Report 101, Computational Logic, Inc., 1717 West Sixth St., Suite 290, Austin, Texas 78703-4776, USA, August 1994.

[Lee89] P. Lee. *Realistic Compiler Generation*. Foundations of Computing Series. MIT Press, Cambridge, Massachusetts, 1989.

[LPRS88] P. Lee, F. Pfenning, G. Rollins, and W. Scherlis. The Ergo support system: An integrated set of tools for prototyping integrated environments. In Henderson [Hen88], pages 25–34.

[MC86] E. Morcos-Chounet and A. Conchon. PPML: A general formalism to specify pretty-printing. In H.-J. Kugler, editor, *Information Processing 86 (Proceedings of IFIP Congress)*, pages 583–590, Dublin, 1986. IFIP, Elsevier Science Publishers B.V. (North-Holland).

[MC94] T. F. Melham and J. Camilleri, editors. *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, Valletta, Malta, September 1994. Springer-Verlag.

[Mel89] T. F. Melham. Automating recursive type definitions in higher order logic. In Birtwistle and Subrahmanyam [BS89]. Also available as University of Cambridge Computer Laboratory Technical Report 146.

[Mel91] T. F. Melham. A package for inductive relation definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 350–357, Davis, California, USA, August 1991. IEEE Computer Society Press, 1992.

[Mil72] R. Milner. Implementation and application of Scott's logic for computable functions. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, 1972. ACM SIGPLAN Notices 7(1).

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[Moo89] J S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, December 1989.

[Mos79] P. D. Mosses. SIS — semantics implementation system, reference manual and user guide. Departmental Report DAIMI MD-30, Computer Science Department, Aarhus University, Denmark, 1979.

[Mos92] P. D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MW72] R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, chapter 3, pages 51–70. Edinburgh University Press, 1972.

[Pau82] L. Paulson. A semantics-directed compiler generator. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 224–233, Albuquerque, New Mexico, January 1982.

[Pau83] L. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[Raj92] P. S. Rajan. Executing HOL specifications: Towards an evaluation semantics for classical higher order logic. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications: Proceedings of the IFIP TC10/WG10.2 Workshop*, volume A-20 of *IFIP Transactions*, pages 527–536, Leuven, Belgium, September 1992. North-Holland/Elsevier.

[Ree95] R. Reetz. Deep embedding VHDL. In Schubert et al. [SWAF95], pages 277–292.

[RK94] R. Reetz and T. Kropf. Simplifying deep embedding: A formalised code generator. In Melham and Camilleri [MC94], pages 378–390.

[RT89] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[SMB92] V. Stavridou, T. F. Melham, and R. T. Boute, editors. *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.

[SWAF95] E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors. *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, Aspen Grove, UT, USA, September 1995. Springer-Verlag.

[Sym93] D. Syme. Reasoning with the formal definition of Standard ML in HOL. In Joyce and Seger [JS93], pages 43–60.

[vdBV96] M. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996.

[VG93] M. VanInwegen and E. Gunter. HOL-ML. In Joyce and Seger [JS93], pages 61–74.

[Wad92] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, USA, January 1992.

[Wan84] M. Wand. A semantic prototyping system. In *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction* (*ACM SIGPLAN Notices*, 19(6)), pages 213–221, Montreal, Canada, June 1984.