A Tool to Support Formal Reasoning about Computer Languages*

Richard J. Boulton[†]

University of Cambridge Computer Laboratory

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. This may be done by defining new constants in the logic for each language construct, e.g. the assignment statement `x:=e` of an imperative programming language might be defined as:

$$\texttt{ASSIGN}\ (x\texttt{:string})\ e\ s\ \texttt{=}\ \lambda y.\ \texttt{if}\ (y\ \texttt{=}\ x)\ \texttt{then}\ e\ s\ \texttt{else}\ s\ y$$

`ASSIGN` is a higher-order function that takes the logical representations of `x` and `e` as its first and second arguments, and a state as its third argument. The state is itself a function from type `string` to the type of expression values. `ASSIGN` returns a new state in which `x` is bound to the value of `e` but all other variables are bound as in the original state. This technique is known as *shallow embedding.*

The use of constants for each language construct makes parsing a text into the logic straightforward and properties of the text can then be proved. However, this approach makes it difficult to express the static semantics in the logic and does not allow general properties about the language itself to be proved. These limitations are overcome by an alternative approach, known as *deep embedding,* in which the abstract syntax of the language is defined as a type in the logic

[†] Author's address from 1st January 1997: Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, United Kingdom.

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

and the semantics is defined over this type. The semantics may be defined as recursive functions or by making inductive definitions (e.g. structural operational semantics) or in other ways.

Examples of embedding include an early compiler correctness proof for a simple ALGOL-like language [19] mechanised in Stanford LCF, Gordon's axiomatic semantics for a simple imperative language [9] in the HOL system, embeddings of various hardware description languages (e.g. [4]), and reasoning about the Standard ML module system [12].

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 [5]. 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

[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 of the LCF theorem prover.

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 [18], 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 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 [5], is at a higher

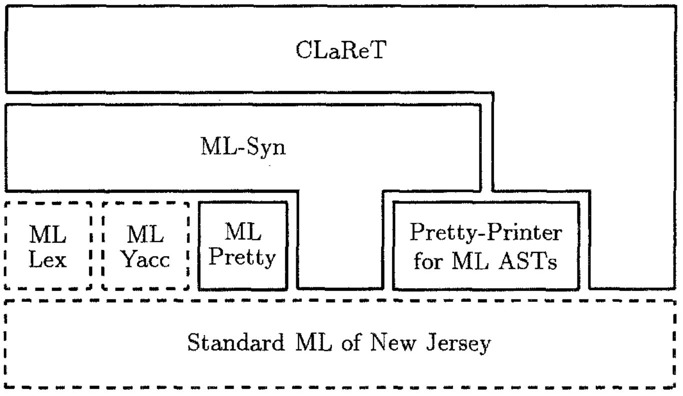

Fig. 1. Simplified architecture of CLaReT

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 [10]. 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. Structural 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* [7] 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 [13] in HOL.

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.

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 effects on the environment 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.

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 [25]. 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 [20].

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 [7]. 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 should produce a fast evaluator that also allows some symbolic entities. Kaufmann and Moore take a different approach in the ACL2 theorem prover [15]; 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 [21] 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
```

where **one** is the unique element of the type that has only one element, equivalent to () in ML. 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. Evaluation continues by attempting beta-reduction, a general strict evaluation using library functions, and finally 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.

[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.

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 Proofs

The denotational specifications for semantics and the generator of logical definitions from them are designed to produce definitions close to those that would be written by hand. Thus, proofs about these definitions should not be significantly less tractable than the numerous significant proofs that have already been done in the HOL system and other theorem provers. The syntactic specification language, Syn, of CLaReT does impose some constraints on the abstract syntax, so the semantic definitions may not be as optimal as hand-written ones, but Syn includes features, such as repetitions, that enable a reasonable abstract syntax to be produced. Where this is not sufficient, one possible approach would be to give two specifications, $\mathcal{C}$ and $\mathcal{A}$, of the language's syntax. $\mathcal{C}$ includes the concrete syntax while $\mathcal{A}$ specifies abstract syntax only. Specification $\mathcal{A}$ is then not constrained by the requirements of the concrete syntax and so an abstract syntax that is optimal for proof can be produced. It is then simply a matter of writing ML functions that map between the data types for abstract syntax that CLaReT generates from $\mathcal{C}$ and $\mathcal{A}$. This technique also allows static and dynamic semantics to be given separately: the static semantics maps a $\mathcal{C}$ tree to an $\mathcal{A}$ tree provided the $\mathcal{C}$ tree is well-formed, while the dynamic semantics is given over the $\mathcal{A}$ tree.

A simple proof involving our example language follows as an illustration. The theorem states that a conditional statement with the null statement `skip` as its branch is equivalent to `skip`, i.e.:

```
if <<bexp>> then skip <-> skip
```

In HOL, the theorem is[3]:

```
⊢ ∀bexp state.
     den_of_com (If bexp Skip NONE) state = den_of_com Skip state
```

The proof proceeds as follows:

```
den_of_com (If bexp Skip NONE) state =
BIND_S (den_of_bexp bexp)
   (λb. COND b (den_of_com Skip) (UNIT_S one)) state =
BIND_S (den_of_bexp bexp)
   (λb. COND b (UNIT_S one) (UNIT_S one)) state =
BIND_S (den_of_bexp bexp) (λb. UNIT_S one) state =
(λ(x,s1). (λb. UNIT_S one) x s1) (den_of_bexp bexp state) =
(λ(x,s1). UNIT_S one s1) (den_of_bexp bexp state) =
(λ(x,s1). UNIT_S one s1) (...,state) =
UNIT_S one state
```

The penultimate step requires a lemma stating that evaluating boolean expressions has no effect on the state.

7 Comparison with Other Systems

7.1 The Reetz/Kropf Embedding Generator

The idea of automatically generating embeddings is not new. Reetz and Kropf [23] 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.

7.2 Tools for Operational Semantics

For many languages one style of semantics is more appropriate than others but the choice may also be a matter of personal taste. A number of systems have been developed to support reasoning with operational semantics rather than

[3] The abstract syntax (type) constructors in HOL's logic are curried. Also, the '`Imp_`' prefix has been omitted from names for brevity.

denotational semantics. These include the SMG system [11] and the Process Algebra Compiler (PAC) [8]. SMG supports temporal logic model checking for languages by transforming programs to suitable finite state models. PAC, as its name suggests, is dedicated to reasoning about process algebras, and is essentially a front-end generator for existing process algebra tools. It produces a parser and functions for computing the semantics of programs.

PAC uses separate specifications of abstract and concrete syntax, plus an additional grammar for specifying the extension of the syntax with structural operational semantics rules. There is no specification of pretty-printing. The specifications of concrete syntax are similar to Yacc, but, as in CLaReT, there is special support for repetitive syntax. On the whole, though, the PAC's syntactic specifications are not as sophisticated as CLaReT's.

7.3 Software Development Environments

There are a number of language-independent software development environments that provide similar features to CLaReT, e.g. CENTAUR [3], the Ergo Support System (ESS) [16], and the programming system generator PSG [1]. 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. A more detailed comparison follows.

CENTAUR has two collections of specification languages: ASF+SDF [2] (a combination of ASF and SDF) and Metal/PPML/Typol. SDF and Metal are specification languages for concrete and abstract syntax. Neither of these specify formatting but PPML (a pretty-printing specification language) may be used with Metal, and recently van den Brand and Visser [24] 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 [5].

ASF is an algebraic specification language and Typol implements Kahn's natural semantics. 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.

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.

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.

7.4 Semantics-Directed Compiler Generators

There have been a number of attempts to generate compilers from denotational semantic specifications including early work by Mosses, and later by Paulson, Wand and Lee. More recently, Pettersson and Fritzson [22] have used a superset of Standard ML to specify denotational semantics with the aim of producing efficient compilers. The extensions include allowing concrete syntax within semantic brackets in place of ML pattern matching. All these systems use denotational specifications similar to those of CLaReT, but their aim is compiler generation not formal reasoning.

A related tool, Actress [6], is a semantics-directed compiler generator for Mosses' action semantics. This uses ML-Lex and ML-Yacc to generate a parser, so its syntactic specification is at a lower level than in CLaReT. 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 yet present in current provers.

8 Summary

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.

Acknowledgements

Thanks to Matthias Mutz (Kiel), 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) and anonymous referees gave helpful comments on a draft of this paper, and Peter Homeier (UCLA) provided pointers to the literature.

References

1. 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.
2. J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press in co-operation with Addison-Wesley, 1989.
3. P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In Henderson [14], pages 14–24.
4. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In 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*, pages 129–156, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.
5. 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, March 1996.
6. 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.
7. J. Camilleri and V. Zammit. Symbolic animation as a proof tool. In Melham and Camilleri [17], pages 113–127.
8. R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In E. Brinksma, et al., editors, *Selected papers from the First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 153–173, Aarhus, Denmark, May 1995. Springer.

9. M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.
10. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.
11. 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.
12. E. Gunter and S. Maharaj. Studying the ML module system in HOL. *The Computer Journal*, 38(2):142–151, 1995.
13. E. L. Gunter. A broader class of trees for recursive type definitions for HOL. In 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*, pages 141–154, Vancouver, B.C., Canada, August 1993. Springer-Verlag, 1994.
14. 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.
15. 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.
16. P. Lee, F. Pfenning, G. Rollins, and W. Scherlis. The Ergo support system: An integrated set of tools for prototyping integrated environments. In Henderson [14], pages 25–34.
17. 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, 1994. Springer-Verlag.
18. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
19. 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.
20. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.
21. L. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.
22. M. Pettersson and P. Fritzson. DML — a meta-language and system for the generation of practical and efficient compilers from denotational specifications. In *Proceedings of the 4th International Conference on Computer Languages (ICCL'92)*, pages 127–136. IEEE, April 1992.
23. R. Reetz and T. Kropf. Simplifying deep embedding: A formalised code generator. In Melham and Camilleri [17], pages 378–390.
24. M. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Trans. on Software Engineering and Methodology*, 5(1):1–41, 1996.
25. 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.