ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Gerard O'Regan and Sharon Flynn (Eds)

1st Irish Workshop on Formal Methods

Proceedings of the 1st Irish Workshop on Formal Methods, Dublin 3-4 July 1997

Paper:

A Simple Algorithm Specification Language and its Application

John Herbert

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

A Simple Algorithm Specification Language and its Application

John Herbert

Cambridge Computer Science Research Centre, SRI International
Cambridge, England

Abstract

A pragmatic approach to algorithm specification and verification is presented. The language AL provides a level of abstraction between a mathematical specification notation and a programming language, supporting compact but expressive algorithm description. Proofs of correctness about algorithms written in AL can be done via an embedding of the semantics of the language in a proof system; implementations of algorithms can be done through translation to standard programming languages. The proofs of correctness are more tractable than direct verification of programming language code; descriptions in AL are more easily related to executable programs than standard mathematical specifications.

AL provides an independent, portable description which can be related to different proof systems and different programming languages. Several interfaces have been explored and tools for fully automatic translation of AL specifications into the HOL logic and Standard ML executable code have been implemented.

A substantial case study uses AL as the common specification language from which both the formal proofs of correctness and executable code have been produced.

1 Introduction

While algorithms may be described in both standard mathematical notations and standard programming languages, both of these have disadvantages. Mathematical notations may be more complex than is necessary for algorithm description and may not be executable or easily related to programming languages. Programming languages, while providing an executable and probably straightforward description of the algorithm, may not provide an abstract description and may not provide a good basis for proof. The AL notation is an effort to find a pragmatic compromise through the use of a very compact but expressive functional specification language, smaller than both standard specification logics and programming languages.

1.1 Motivation

Software verification is difficult: few substantial pieces of software written in a conventional programming language have been formally specified and verified by mechanical proof. Techniques for verifying imperative programs are based largely on the seminal work of Floyd [4], Hoare [11], and Dijkstra [3]. Ongoing research demonstrates the subtleties involved [5, 9, 10, 12, 13, 15, 19, 27], and the verification of large imperative programs remains a daunting task. The functional programming language Standard ML offers a better option since not only does it resemble a mathematical logic but it has a formal definition [18, 17]. However, the present Standard ML (SML) is very much a real, fully featured programming language and constructing a full semantics of the language is non-trivial as is conducting mechanised proofs of correctness [24, 22, 23]. While it is clear that a programming language might be given a semantics, even a mechanised semantics, the important practical issue is whether such a semantics provides a tractable basis for proofs of correctness of reasonably sized programs.

The approach presented here is very much a pragmatic one, and was motivated by a desire to verify a non-trivial program. AL provides a compact notation that approximates a subset of standard mathematical logics and standard functional programming languages. The language is appropriate for an algorithm description lying between an abstract specification and an implementation, thus making the task of relating implementation to specification more tractable.

The goal is to have in the notation just the essentials needed — thus, for example, the parts of a mathematical notation dealing with quantifiers are omitted and the parts of a functional language dealing with evaluation and exceptions are omitted. In addition, it is intended that AL provide a portable, neutral description of the algorithm. By ‘neutral’ we mean that the description should be compatible with a number of mathematical notations so that proofs can be done in different systems, and compatible with a number of programming languages so that the algorithm can have various implementations.

1.2 Design

The language, AL, that has been developed and used is very simple: it just consists of type and function declarations. So it doesn’t actually do anything but maps out how the computation can be achieved. Datatypes describe the problem space and solution space and the functions describe the algorithm for computing the required solutions. The complexity of needing an execution model in the semantics is avoided and since one can reason from the declaration of a function about the relationship between any possible result and the corresponding function parameters, one loses very little.

The language looks simpler than mathematical notations since it does not have any quantifiers or other mathematical symbols. It is simpler than most programming languages due to the small number of constructs. AL corresponds to a subset of a functional language like Standard ML (SML).

The important design features of AL are:

- a small number of primitives
- expressive primitives
- a declarative model
- a simple function definition mechanism
- lack of IO primitives

The most important design decision is that a small set of primitives be used and that these have a simple relationship to the corresponding translated forms. This means that the interfaces between AL and the formal and programming systems can be correspondingly small, thus increasing the security of translation and minimising the effort involved. Minimising the interfaces means that one can have more confidence in any correspondence between formal specifications and programs.

Although AL has a small number of primitives the use of rich datatypes and higher order functions makes the language very expressive. A declarative model means that one avoids the need for an execution semantics which makes the model simpler and more neutral with respect to various programming language models.

The function definition mechanism is simple and liberal even allowing malformed function expressions. The important point is that the semantic model devised (employing the choice operator) ensures that no inconsistencies are introduced by malformed function definitions. Any function definition can be translated into an expression in the semantics but the transformation of this into a usable function definition is done securely within the formal system and will fail if the function definition is not well-formed.

The absence of IO means that the often subtle aspects of IO behaviour are avoided in the semantic model.

Figure 1 illustrates the intended role of AL as a common intermediate language. The solid lines denote automated interfaces that have been constructed and used in the initial case study; the dashed lines denote other interfaces which have not been fully developed. In the first implementation of AL the main programming language target has been SML and the main mathematical notation for representing the semantics has been HOL [6]. SML is an appropriate choice because it is close to AL, a semantics of the language has been defined, and research on the embedding of the semantics of SML in HOL has been done [22, 23]. AL has also been translated into another higher-order logic, that of PVS [20], and how AL might be translated to C has also been investigated.

AL has been used to specify an algorithm and thus provide the basis for proofs of correctness and executable code on a project dealing with proof checking. This case study is based on work on proof script generation and proof checking for the HOL system [26, 7] and provides a substantial challenge for program verification.

Figure 1: Interfaces to AL

2 AL Syntax

An AL text consists of a series of declarations, each declaration is either a type declaration or a function declaration.¹

```
al_text      := {tydec | fundec}*
```

A type declaration declares a new type; this type may be a type operator if type variables are arguments of its constructors. The type declaration states the name of the new type, the constructors, the types of the arguments of the constructors, the accessers for the arguments of the constructors, and recognisers for the different constructors.

```

tydec       := TYPE type_dec END_TYPE
type_dec    := new_ty = dec { | dec } *
dec         := con_fun con_args :: disc_fun
con_args    := ( acc_fun: atype ) { ( acc_fun: atype ) } *
new_ty      := ty_name
disc_fun    := name
con_fun     := name
acc_fun     := name
atype      := ty_name { tyop_name } *
```

A type argument must either be a valid type expression in the language or a recursive reference to the type being defined. The recursive reference may be either a simple reference to the type or a reference to a list of that type.

At least one constructor must have type arguments with no recursive references. The new type being defined is either a constant type, if there are no type variables, or an n-ary type operator if there are n distinct type variables among the type arguments.

The valid strings for name, ty_name and tyop_name are essentially any beginning with an alphanumeric and including alphanumeric or underscore characters. In addition, by convention, types which denote type variables begin with one or more asterisk characters, and several non-alphanumeric type operators are allowed for readability e.g. # and -> for pair and function types.

So, the general template for type declarations looks like the following:

¹In providing a description of the constructs of AL a simple grammar is given, where items in capitals are terminal tokens in the language, the operator | denotes alternation, { and } group items, and * denotes repetition (possibly zero).

```

TYPE
ntype = C1 (acc1: ty1) ... (accN: tyN) :: is_C1 |
        C2 .... |
        Cn (accX: tyX) ... (accZ: tyZ) :: is_Cn
END_TYPE

```

ntype is the new type being declared;
C1 ... Cn are the type constructors;
ty1 .. tyN are the types of the arguments of type constructor C1;
acc1 ... accX are the corresponding accessers for the arguments of C1;
is_C1 ... is_CN are the recognisers for the different constructors.

For example, declaration of the type operator list in AL might be:

```

TYPE
list = null :: is_null |
        cons (car: *ty) (cdr: list) :: is_cons
END_TYPE

```

where the constructors are null and cons; car and cdr return the head and tail of the list respectively; is_null and is_cons act as recognisers for empty and non-empty lists.

A two place constructor may be used as an infix.

2.1 Function Declaration

A function declaration declares a new function. The declaration states the name of the new function, its arguments, the type of the result, and the expression corresponding to the body of the function. There may be a recursive call to the function being declared in the body. There are no restrictions in the language on the form of recursion, but if the recursion is not well founded then the semantics of the function will not yield anything useful.

```

fundec      := FUN fun_dec END_FUN
fun_dec     := new_fun_name fun_args : atype = expr
fun_args    := (arg_name: atype) {(arg_name: atype)}*
expr        := const_name |
              var_name |
              fun_name expr
new_fun_name := name
const_name  := name
var_name    := name
fun_name    := name
arg_name    := name
atype       := ty_name {tyop_name} *

```

The general template for function declarations is as follows:

```

FUN fn_name (arg1: type1) ... (argn: typen) : type =
  body
END_FUN

```

An example of a function declaration is the following for computing the length of a list:

```

FUN length (lst: * list) : num =
  IF (null lst) THEN 0
  ELSE (1 + (length(cdr lst)))
END_FUN

```

The above definition is recursive. A recursive function declaration delivers a well-defined function only if the declaration can be shown to describe a unique function. The declaration mechanism is kept simple so that recursive, non-recursive and invalid recursive declarations all look similar; the semantic model, however, ensures that no inconsistencies are introduced.

A two place function may be used as an infix.

2.2 Built-in Types and Functions

As well as the facilities for declaring types and functions AL has a small number of built-in types and functions. The types `bool`, `ind`, type operator `->`, polymorphic equality `=` and conditional are built in. (We also experimented with a pattern matching version of AL where even less is built-in.) Also built-in is a polymorphic constant `?` which denotes a fixed, unknown constant value of the appropriate type.

In the main case study some additional basic types and functions have been assumed: the types `num`, `list`, `set` and `string` and their associated functions. These are all built-in to proof systems such as HOL and PVS and to have them as user-defined types and functions would have been confusing and inefficient. (All of these could have been defined from the primitive types, as indeed they are in systems such as HOL.)

3 Semantics and Executable Translation

AL is intended to provide an intermediate level description which can be related to both a specification in a formal logic and an implementation in a programming language. While AL provides a common interface to a number of proof systems and programming languages, the prime candidates for interfaces were the HOL proof system and the SML programming language, sharing a similar mathematical core and providing well-established tools for automated proof and execution, respectively.

3.1 Semantics of AL

An AL text consists of a series of type and function declarations so a straightforward way of expressing the semantics of the text is through corresponding type and function declarations in a proof system such as HOL or PVS supporting similar higher-order declarations. The mapping from AL text to semantics may be done via a shallow embedding, where the parse tree is in the meta-language and a meta-language function defines the mapping from text to logical terms, or a deep embedding where the parse tree is in the object language and a function in the object language defines the mapping.

The semantics has been defined in HOL using a shallow embedding. A shallow embedding is appropriate because HOL type and function declarations are implemented by functions in the meta-language and doing such declarations in the object language would require an embedding of the corresponding infra-structure. It is more straightforward to implement the embedding by a direct use of HOL type and function definitions. The following outlines the semantic translation of AL into the logic of HOL.

3.2 Semantics of Type Declarations

The type declarations of AL fit the following pattern:

```
TYPE
ntype = C1 (acc1: ty1) ... (accN: tyN) :: is_C1 |
        C2 ... |
        Cn (accX: tyX) ... (accZ: tyZ) :: is_Cn
END_TYPE
```

This declaration introduces a new type, its type constructors and a number of accessor and recogniser functions. For a valid declaration which will produce a corresponding definition in the HOL logic, there must be at least one

constructor with non-recursive arguments and recursive references must be either to the type or a list of the type. The admissible types therefore correspond to those accepted by the extended HOL type definition package [8]. The new types are constructed in HOL using the technique devised for the standard type definition package [16], namely that all types can be represented by an appropriate labelled tree. This means that all the type definitions are conservative extensions of the HOL logic since they can be represented by pre-existing type operators and types.

In the core HOL system there are the primitive types `bool` and `ind`, representing the booleans and individuals respectively, and a primitive type operator, `->`, denoting the function type. All other types are defined and have representations using the primitive types. Types such as `num`, `string`, `list` and `set` are defined in the basic HOL system and so have not been redefined in AL. This is done for efficiency since it is straightforward to define these types in AL.

As well as defining a new type, a type declaration in AL introduces several functions for accessing the arguments of constructors and recognising each constructor of a type. The semantics of the constructor and recogniser functions are represented in a straightforward way in the HOL logic by HOL functions. The result of the application of an accessor to an object with the appropriate constructor returns the required argument; the application of an accessor to an object with an incorrect constructor returns a value in the logic which represents AL's unknown constant value `?` (of the appropriate type). These functions are not recursive so the general model of functions described in the next section is not used. If the more general model were used exactly the same functions would be derived.

3.3 Semantics of Function Declarations

A function declaration in AL looks like this:

```
FUN fn_name (arg1: type1) ... (argn: typen) : type =
  body
END_FUN
```

This declaration introduces a new function `fn_name`. The semantics of the function declaration is represented in HOL by a new function `fn_name` of type `(type1 -> type2 ... -> typen) -> type` which is defined to be the function:

$$\varepsilon \text{ fn_name}'. \text{ fn_name}' \text{ arg1} \dots \text{ argn} = \text{ body}$$

i.e. any function that satisfies the equation `fn_name' arg1 ... argn = body` (ε is the Hilbert choice operator of higher-order logic.²)

Since the body may contain recursive calls to `fn_name'` one must prove that a function exists which satisfies the equation, and that this function is unique. One can then eliminate the choice operator and derive an equation for the function in the logic which corresponds directly to the AL text:

$$\text{fn_name} (\text{arg1: type1}) \dots (\text{argn: typen}) = \text{ body}$$

The use of the choice operator in the semantics of functions has the effect of providing a unified model for recursive, non-recursive, and malformed recursive functions. By doing secure transformation of the logical terms in HOL one can simplify the non-recursive and recursive definitions to obtain the appropriate logical representation of functions. Malformed recursive functions do not introduce any inconsistencies as it is not possible to eliminate the choice operator in such cases to obtain functions in the logic corresponding to the AL text. The use of this model also means that there is just a single method of defining functions, non-recursive or recursive, in AL and the different proof obligations needed are taken care of in the semantic model.

In the core HOL system the only primitive functions are equality, implication and choice. In AL the primitive functions are equality and the conditional, `IF ... THEN ... ELSE ... END_IF`.

² $\varepsilon x. Px$ is read as choose an x such that predicate Px holds.

3.4 Executable Translation of AL

The main design goal of AL is to provide a simple description of an algorithm in a language that maps to different formal notations and programming languages. In particular AL is declarative and so has no execution model. It is of course useful to have an executable implementation corresponding to the functions of AL and since AL matches closely a subset of SML, SML is an obvious candidate for implementations. The following outlines the translation of AL into a set of datatype definitions and executable functions in SML.

3.5 Translation of Type Declarations

Type declarations in AL are translated into datatype and associated function declarations in SML. The datatype declaration closely matches AL except that in AL higher order constructors are allowed, whereas in SML each constructor has just a single argument. In SML one can use tuples or records to gather together multiple arguments. In translating higher-order constructors of AL into SML, a representation using tuples and associated constructors is used. The higher-order constructors are defined in terms of this intermediate SML representation.

An AL type declaration has the general pattern:

```
TYPE
ntype = C1 (acc1: ty1) ... (accN: tyN) :: is_C1 |
      C2 .... |
      Cn (accX: tyX) ... (accZ: tyZ) :: is_Cn
END_TYPE
```

The corresponding SML datatype declaration is:

```
datatype ntype = C1_int of (ty1 * ty2 .... * tyn) |
                C2_int of ..... |
                Cn_int of (tyX * tyY .... * tyZ) ;
```

along with the functions mapping back to the higher-order constructors:

```
fun C1 arg1 arg2 ... argn = C1_int (arg1 * arg2 .... * argn);
```

accessor and recogniser functions are defined along with these declarations. The following definitions for the first constructor of the type show the pattern of these:

```
fun acc1 (C1 arg1 arg2 ... argn) = arg1 |
  acc1 (C2 ....)                = raise access_ex |
  ...
  acc1 (Cn ....)                = raise access_ex;

fun is_C1 (C1 arg1 arg2 ... argn) = true  |
  is_C1 (C2 ....)                = false |
  ...
  is_C1 (Cn ....)                = false ;
```

It should be noted that datatypes in SML can be recursive but that no constraints are imposed on the recursion, thus non-wellfounded recursive types can be declared. In AL a recursive type must be one of the class of recursive types defined earlier. An invalid type will not produce a corresponding definition in the logic representing the semantics.

The core SML system includes the types `bool`, `int`, `string` and `list` along with the constants and functions associated with these types. The built-in SML type `int` is used to represent `num` and the other types are all used for the corresponding AL types. An efficient package for implementing sets is available as part of the distributed SML of New Jersey but to keep the mapping to SML as simple as possible an implementation of sets based on lists is defined. This is included in the SML prologue file used in the translation.

3.6 Translation of Function Declarations

Function declarations in AL are translated into function declarations in SML. A function declaration in AL has the general pattern:

```
FUN fn_name (arg1: type1) ... (argn: typen) : type =
  body
END_FUN
```

and results in a corresponding SML function declaration:

```
fun fn_name (arg1: type1) ... (argn: typen) : type =
  body
```

SML, like AL, does not enforce any restrictions on recursive function definitions, so it is possible to declare and translate into SML a non-wellfounded function. While this might result in the user defining and trying to use functions which may be impossible to evaluate, the semantic model for functions in AL means that this will not lead to any false correctness results. A future, better engineered version of the AL toolkit would issue a warning to the user when non well-founded definitions were encountered.

4 AL Support Tools

The present implementation of AL is a prototype system which has allowed the investigation of AL as a convenient notation for describing algorithms and providing the basis for implementation and proof. A future, more robust, implementation will take advantage of generic tools to support embeddings.

Being a prototype system means that there are a number of dangers that one must be aware of. For example, the parsing and interfacing of AL involves lisp and ML code, and the parser of HOL. Issues like precedence of operators have not been made explicit or defined. Care has been taken to avoid any errors, for example by liberal use of parentheses to avoid precedence issues, but the basic issue still exists. A project to engineer a robust tool-set for AL would try to address these issues.

4.1 Generation of HOL

The programs that have been written to generate the HOL theory are mostly coded in classic ML, the ML of the present HOL88 system. The user uses the tools by either running an interactive session and executing certain functions, or by running a batch job using a file with the appropriate function invocations.

The user calls an ML function `HOL_eval` which has as its argument a string corresponding to the stem of the name of the file containing the AL description. `HOL_eval` reads the named file with extension `.al` and produces the output file with extension `.ml`. The function does two things: it firstly calls the program that translates the AL into HOL, and secondly it loads that file and uses predefined functions which implement the actual type and function declarations which are defined by the HOL translation. For example, for the AL datatype declaration a string corresponding to the body of the declaration is parsed and the corresponding HOL type and associated function definitions are generated from this.

As mentioned elsewhere, the translation is a simple one and the use of the choice function in the model means that incorrect function definitions will not introduce inconsistencies. There are a number of tools which may be used to simplify these initial translations. The main task is the proof for each function that it does implement a valid definition and that the choice operator can be eliminated. This involves a number of tasks including finding the argument (if any) on which a definition recurses; verification that there exists a function satisfying the recursion equation and that this function is unique; and transforming the definition into one which is more like a cases enumeration for the function. For example the following is the AL description of how one checks that a certain type is well-formed:

```

FUN Type_OK (tyl:(string # num)list) (ty:Type) :bool =
CASE
(Is_Tyvar ty, true);
(Is_Tyop ty, (mem1 (Tyop_nam ty) tyl) AND
              (LENGTH (Tyop tyl ty) = corrl (Tyop_nam ty) tyl) AND
              EVERY (Type_OK tyl) (Tyop tyl ty) )
END_CASE
END_FUN

```

When this is initially transformed into HOL and loaded into the system, the following definition is made:

```

Type_OK =
|- Type_OK =
  (ε Type_OK'.
    ∀ tyl ty.
      Type_OK' tyl ty =
        CASE_FN
        [Is_Tyvar ty,T;
         Is_Tyop ty,
         (mem1(Tyop_nam ty)tyl) AND
         ((LENGTH(Tyop tyl ty) = corrl(Tyop_nam ty)tyl) AND
          (EVERY(Type_OK' tyl)(Tyop tyl ty)))]])

```

The above definition is not very useful, so the tools are used to automatically prove that the equation contained within the choice expression uniquely defines a function, and then to generate a more useful defining equation for `Type_OK`. The following is the resultant theorem in HOL which has been proved automatically from the initial definition:

```

|- (∀ (tyl:(string # num)list) (s:string).
    Type_OK tyl(Tyvar s) = T) ∧
  (∀ (tyl:(string # num)list) (s:string) (Tlst:(Type)list).
    Type_OK tyl(Tyop s Tlst) =
      mem1 s tyl ∧
      (LENGTH Tlst = corrl s tyl) ∧
      EVERY(Type_OK tyl)Tlst)

```

4.2 Generation of SML

The generation of SML is done in a similar way to the generation of HOL. The user calls an ML function `SML_eval` which has as its argument a string corresponding to the stem of the name of the file containing the AL description. This command generates an SML file with the extension `.sml`, and this file can then be loaded directly into an SML session using the `use` command.

The generation of SML does not involve any proofs but some extra work is needed because of the use of curried arguments in AL and the non-curried form in SML. The system generates some extra internal functions for constructing and accessing curried datatype declarations and then defines the other accessers and constructors in terms of these. For example, the following is the AL text describing the datatype representing HOL types:

```

TYPE
Type = Tyvar (Tyvar_nam: string) :: Is_Tyvar |
      Tyop (Tyop_nam: string) (Tyop tyl: Type list) :: Is_Tyop
END_TYPE

```

The automatic translation of this into SML generates the following code:

```

datatype Type = Tyvar of (( string)) |
               Tyop_tup of (( string))*(( Type list)) ;
fun Is_Tyvar(Tyvar _) = true |
    Is_Tyvar(Tyop_tup _) = false ;
fun Is_Tyop(Tyvar _) = false |
    Is_Tyop(Tyop_tup _) = true ;
exception de_Tyvar_ex;
exception de_Tyop_tup_ex;
fun de_Tyvar (Tyvar x) = x |
    de_Tyvar (Tyop_tup _) = raise de_Tyvar_ex ;
fun de_Tyop_tup (Tyvar _) = raise de_Tyop_tup_ex |
    de_Tyop_tup (Tyop_tup x) = x ;
val Tyvar_nam = (tup1_1 o de_Tyvar) ;
val Tyop_nam = (tup2_1 o de_Tyop_tup) ;
val Tyop_ty1 = (tup2_2 o de_Tyop_tup) ;
fun Tyop a b = Tyop_tup (a,b) ;

```

The above datatype declaration and associated function declarations include some extra generated constructors, such as `Tyop_tup` and `de_Tyvar`. The actual constructors such as `Tyop` and `Tyvar_nam` are then defined in terms of these.

4.3 Translation to PVS

One of the main goals of AL is to provide a simple neutral description of an algorithm which can be translated into different programming languages and can be modelled in different formal systems. Translation of AL to PVS [20] was done for all the datatypes representing the hierarchy of HOL types, terms, sequents and proofs and many of the functions over these types in the case study. A full automated embedding of the theory of HOL proofs was not done but would be straightforward.

4.3.1 Examples

The following examples show some of the datatype and function declarations in AL and the corresponding ones in PVS. The PVS declaration corresponding to the AL one for `Type` is:

```

HType [sty:TYPE] : DATATYPE
  BEGIN
    Tyvar(Tyvar_nam:sty): Is_Tyvar
    Tyop (Tyop_nam:sty, Tyop_ty1:list[HType]): Is_Tyop
  END HType

```

The declaration looks quite like AL since the AL notation is modelled on that of PVS and `nqthm`[2]. Differences to note are that the constructors are not curried and that the type operator is prefix rather than postfix. In the version of PVS being used the type `string` was not built in so the datatype is parameterised by a type variable which can be instantiated to a particular type implementing strings. This style of type parameterisation is an important feature of PVS and there are subtle differences between the use of such polymorphic variables in PVS and HOL. In HOL a polymorphic variable has essentially global scope and can be instantiated at will. In PVS theories are parameterised by their type variables and these may need to be explicitly instantiated when a theory is imported, or one may need to import two different instantiations of a theory.

The following is the declaration in PVS of the datatype representing HOL terms:

```

Pterm [sty:TYPE] : DATATYPE
  BEGIN

```

```

IMPORTING HType
  Const(Const_nam: sty, Const_ty:HType[sty]) : Is_Const
  PVar (Var_var_st: sty, Var_var_t:HType[sty]) : Is_Var
  App (App_fun: Pterm, App_arg: Pterm) : Is_App
  PLam (Lam_var_st:sty,
        Lam_var_t:HType[sty], Lam_bod: Pterm) : Is_Lam
END Pterm

```

The `IMPORTING` clause is used since `HType` is needed in this definition. This theory is again parameterised on the string type and this is also used in the instantiation `HType[sty]`.

4.4 Translation to Imperative Languages

The translation of AL to C was also investigated. The main difficulties that were identified were the use of higher-order constructors, higher-order functions and the use of polymorphism. While rewriting the algorithm in AL in a style suitable for translation to C would have been possible, it would have meant a departure from the compact style used as the basis for the formal proofs in HOL. The absence of similar rich datatypes and higher order functions in programming languages such as C is not such a huge barrier however, since one just needs an ability to implement these features in the programming language rather than actually requiring their presence in the language. Thus, for example, although C may not have as rich a datatype declaration facility, one could produce records and functions in C which would implement an AL datatype.

One would have to consider carefully the semantics of even a subset of C corresponding to AL. In contrast to SML there was no effort to provide a semantics when the language was being defined and no full semantics of the language is available. An alternative would be to transform a functional program in AL into a simple imperative language using a theory of refinement in a mechanised prover following [25]. One could therefore do the refinement safely at the source text level, and the small imperative language could then be mapped to a subset of C or other conventional imperative language.

5 Case Study

As mechanically-produced proofs about computer systems become more common, and as the proof procedures used become less easily inspected, there is a need for a simple independent test of the trustworthiness of claimed proofs. The case study consists of specifying an algorithm for proof checking a proof in formal logic as might be output from a mechanised proof system. If a proof system (which might be large and complex computer program using non-inference decision procedures) claims to have proved a certain theorem one could demand that the system produce a proof log of primitive inference steps. An independent check could then be made to verify this proof by running it through a carefully designed proof checker (cf. figure 2). There is no ultimate checking algorithm that one can appeal to: all one can do is specify formally the proof checking algorithm and show by formal analysis that it has all the appropriate properties that one would expect.

The proof checker to be specified is for proofs in the higher-order logic of the HOL system, and is based directly on the theory of HOL proofs by von Wright [26]. It happens that the proof system generating the proof log is HOL and the proof system being used to check the algorithm is also HOL but this is a coincidence rather than a necessity. The specification of the proof checker will be stating precisely what a proof in the HOL logic consists of, and therefore one needs to build up a description of what HOL types, terms, sequents and inferences are. In addition, one needs to formalise the various concepts needed for describing properties relevant to proof, for example type instantiation and free and bound variables.

The aim of the formalisation is to be able to check proofs, not to generate them; therefore one does not need to model inference rules as functions which return theorems. Instead, it is enough that we are able to recognise a correct inference, once the result is given. This means that one does not have to deal with HOL's intricate procedures for variable renaming used in the primitive inference rules `INST_TYPE` and `SUBST`. The formalisation permits arbitrary renaming schemes, and the particular one used by HOL is just one instance of such a scheme.

Figure 2: Overview of the case study

5.1 Example of Formalisation of HOL Types

For the formalisation of proofs in the logic of HOL one must build up a set of definitions for the types, terms, sequents and inferences of HOL along with all the associated functions, such as those for checking well-formedness of terms and checking the application of primitive inference rules. The following example of the definition of the datatype representing a type in the HOL logic and of an associated function are meant to give a flavour of what is involved in the specification.

The HOL logic has four different kinds of types: type variables, type constants, function types and n -ary type operators. In the formalisation, type constants and the function type are taken to be special cases of type operators. HOL types are then represented by the following data type declaration in AL:

```
TYPE
Type = Tyvar (Tyvar_nam: string) :: Is_Tyvar |
      Tyop (Tyop_nam: string) (Tyop tyl: Type list) :: Is_Tyop
END_TYPE
```

This states that a type is either a type variable or is built from a type operator along with a list of argument types.

One of the items that characterise a *theory* in HOL is a type structure. In the formalisation, a type structure is represented by a list of pairs (op, n) of type `string#num`, where n is the arity of the type operator op . The following function states that a Type is OK if it is a type variable or it is composed from OK types by a permitted type operator under the given type structure:

```
FUN Type_OK (tyl:(string # num)list) (ty:Type) :bool =
CASE
(Is_Tyvar ty, true);
(Is_Tyop ty, (mem1 (Tyop_nam ty) tyl) AND
              (LENGTH (Tyop tyl ty) = corrl (Tyop_nam ty) tyl) AND
              EVERY (Type_OK tyl) (Tyop tyl ty) )
END_CASE
END_FUN
```

`mem1 s l` holds if s is the first component of some pair in the list l ;
`corrl s l` returns the second component corresponding to s in list l ;
`LENGTH` returns the length of a list, and
`EVERY` holds of a predicate and a list if the predicate holds of each element of the list.

Thus a type constructed from a type operator is defined to be valid in the current theory if it is a type variable or constructed from a type operator, where the type operator is a member of the type structure, the length of its argument list is the same as the arity defined for it in the type structure, and all of the argument types are in turn valid types.

The specification builds on the definition of types, terms, sequents and inference rules and ultimately defines a predicate `Is_proof` which checks that a sequence of inferences constitutes a proof. For a more comprehensive account see the description by von Wright [26]. To illustrate the AL specification a set of definitions associated with the formalisation of HOL types in AL along with the corresponding semantics and executable code is given in Appendices A, B and C.

5.2 Case Study Results

The proof checking algorithm was completely specified in AL, and from that the semantic description in HOL and the executable code in SML were generated automatically. Figure 2 shows the different components in the case study and how the AL specification acts as the common basis for both the formal analysis and the executable code.

The total AL specification has four datatype definitions and sixty four function definitions, and consists of 678 lines (18.5kb) of AL source. The corresponding HOL source has the same number of lines but when loaded into the HOL system generates all the extra accessor and recogniser functions for datatypes which are implicit in the AL source.

The SML generated from the AL specification is 1099 lines (30kb) and again the extra size is mostly due to the extra functions associated with datatypes. (There are in total 80 implicit functions associated with the four datatypes.)

The scripts which perform the proofs of correctness in HOL of the algorithm specification amount to over 5000 lines of meta-language source code. The proof scripts subject the specification to 293 tests of correctness devised originally by von Wright [26]. The specification is shown to be correct with respect to all these tests.

The executable SML code produced from the specification is incorporated into a larger proof checking program as the core checking mechanism. The 1099 lines of SML generated from AL forms part of a program of 4623 lines of code (100kb). The rest of the SML code is due to Wong (following from [28]) and implements the many auxiliary functions required: two of the largest modules are those dealing with parsing and io. An important concern when checking real proofs is the huge size of the generated proof scripts. A medium-sized proof, such as the multiplier benchmark distributed with the HOL system produces a proof file of 43.6Mb. Consequently the full checker works in two passes and uses compression routines.

The SML code generated from the AL specification is not very efficient. The inefficiency could be improved by a verified refinement of AL into a more efficient algorithm, but for the sort of certification proof checking envisaged speed would not be a critical factor. It should be noted that it is only the core proof checking function, that has been specified in AL and verified; one should be aware of the possibility of errors in the utility functions that surround it. There is always an issue of where one should draw the line when ensuring correctness — can the hardware and operating system be trusted to do what they should? It is certainly of great benefit to have proofs of correctness for the core algorithm, and be aware of the correctness issues in the areas which have not been verified.

6 Discussion

A lightweight, compact, expressive specification language has been implemented and used in a substantial case study. The use of a small language is a common pedagogical device and our design of AL is in the spirit of, for example, Pidgin-Algol [1] used by Ullman et al. in the analysis of algorithms. The important design decisions of AL have been the use of a very restricted set of primitives which makes the interfaces to the notation small and less a source of insecurity, and the decision to keep AL declarative without an execution model.

AL has provided a tractable basis for doing proofs of correctness. The simple translation and automatic, secure transformation in the semantic domain creates a theory similar in complexity and tractability to that created by a direct representation in a proof system (which is the more usual way one might attempt a formal verification of an algorithm e.g. [21]). In contrast, even for the core of Standard ML a very substantial effort is required to encode the semantics in a mechanised prover and conduct proofs of correctness [23].

It should be possible to construct interfaces to other proof systems and programming languages. The interface to the formal system is important for formal proofs and one of the advantages of interfacing to HOL is the ability to define a very safe, simple translation and then use the programmability of HOL to write the tools to do the simplification necessary. The embedding of AL in other proof systems will need some sort of similar support. One should be aware of the subtleties and associated dangers in semantic interfaces: as noted earlier, the logic of PVS looks almost identical to that of HOL but there are differences in polymorphic types and their scope that need to be carefully dealt with.

Another interesting possibility would be to interface AL to a model checking tool since if one is dealing with a simpler specification over finite data types one might apply model-checking to verify properties of the specification. AL might also prove useful as a common intermediate notation for interfacing between model checkers and proof tools. If one had a problem which partitioned into decidable and non-decidable subsets then one might want to do proofs both in a mechanised proof system and a model checker.

AL has not been interfaced to an imperative programming language but one could use refinement to transform a functional AL specification to an imperative one. A more natural technique might be to support an imperative style through a state-machine based specification where the next-state and output functions can be specified directly in AL.

AL and its tool kit have been developed in a pragmatic way and one could engineer a better set of tools, possibly using generic tool development kits. The notation itself may be improved in a number of ways. The use of curried constructors for datatypes was driven by the desire for compatibility with the previous formalisation of HOL proofs. There is no particular advantage in having this feature and a future version will probably drop it.

The use of pattern matching in SML is one of its attractions and the introduction of this would make AL descriptions easier to read. Pattern matching was avoided in the first version of AL because it was thought to be an obstacle to interfacing to more conventional non pattern matching programming languages. In fact it would be possible to use pattern matching in the AL description and then have a translation to a non pattern matching version where necessary.

Additions to the small set of primitives in AL could be made although it has been instructive how much can be achieved with such a tiny but expressive language. Other uses for the notation could also be explored including algorithm development and refinement, and the computational analysis of algorithms.

Acknowledgements

The work was part of research on proof checking at SRI and the University of Cambridge Computer Laboratory, and benefited from interaction with all those involved. Particular thanks to Dr. Joakim von Wright and Dr. Wai Wong who co-operated with the case study and shared their work on proof checking HOL proofs, to Prof. Mike Gordon who initiated the project, and Dr. Roger Hale at SRI.

References

- [1] J. E. Hopcroft A. V. Aho and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [2] Robert S. Boyer and J Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.
- [3] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [4] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.
- [5] M. J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.
- [6] M. J. C. Gordon. HOL: A proof generating system for higher order logic. In *Current trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.
- [7] Michael J. C. Gordon, Roger Hale, John Herbert, Joakim von Wright, and Wai Wong. Proof checking for the HOL system. In David Basin, Fausto Giunchiglia, and Matt Kaufmann, editors, *Correctness and metatheoretic extensibility of automated reasoning systems*, pages 49–50, INRIA Lorraine, 1994.
- [8] B.T. Graham. `rec_tys_listop` — a contribution to the HOL system. Distributed with HOL system, 1993.
- [9] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
- [10] E. C. R. Hehner. *The Logic of Programming*. Prentice Hall, 1984.
- [11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
- [12] C. A. R. Hoare and J. C. Shepherdson, editors. *Mathematical Logic and Programming Languages*, Prentice-Hall International Series in Computer Science. Prentice-Hall, 1985.
- [13] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.
- [14] Jeffrey J. Joyce and Carl Seger, editors. *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications*, volume 780 of *Lecture Notes in Computer Science*, UBC, Vancouver, Canada, 1993. Springer-Verlag.
- [15] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming*. Addison-Wesley, 1985.

- [16] Thomas F. Melham. Automating recursive type definitions in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.
- [17] Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, 1991.
- [18] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
- [19] C. C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.
- [20] S. Owre N. Shankar and J. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, 1993.
- [21] D. Nazareth and T. Nipkow. Formal verification of algorithm W: The monomorphic case. In *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
- [22] Donald Syme. Reasoning with the formal definition of Standard ML in HOL. In Joyce and Seger [14], pages 43–60.
- [23] Myra VanInwegen. The machine-assisted proof of programming language properties. University of Pennsylvania Computer and Information Science Technical Report MS-CIS-96-31, University of Pennsylvania, 1996(Phd thesis).
- [24] Myra VanInwegen and Elsa Gunter. Hol-ml. In Joyce and Seger [14], pages 61–74.
- [25] Joakim von Wright. RefCalc — a contribution to the HOL system. Distributed with HOL system, 1994.
- [26] Joakim von Wright. Representing higher-order logic proofs in HOL. In Thomas F. Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science*, pages 456–470, Valletta, Malta, 1994. Springer-Verlag.
- [27] G. Winskel. *The Formal Semantics of Programming Languages — An Introduction*. MIT Press, 1993.
- [28] Wai Wong. Recording HOL proofs. University of Cambridge Computer Laboratory Technical Report 306, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, 1993.

Appendix A.

AL specification of HOL types

```

TYPE
Type = Tyvar (Tyvar_nam: string) :: Is_Tyvar |
      Tyop (Tyop_nam: string) (Tyop tyl: Type list) :: Is_Tyop
END_TYPE

FUN Type_OK (tyl:(string # num)list) (ty:Type) :bool =
CASE
(Is_Tyvar ty, true);
(Is_Tyop ty, (mem1 (Tyop_nam ty) tyl) AND
              (LENGTH (Tyop tyl ty) = corr1 (Tyop_nam ty) tyl) AND
              EVERY (Type_OK tyl) (Tyop tyl ty) )
END_CASE
END_FUN

FUN Type_compat (ty1:Type) (ty2:Type) :bool =
CASE
(Is_Tyvar ty2, true);
(Is_Tyop ty2, (Is_Tyop ty1) AND
              ((Tyop_nam ty1) = (Tyop_nam ty2)) AND
              (LENGTH (Tyop tyl ty1) = LENGTH (Tyop tyl ty2)) AND
              (EVERY2 Type_compat (Tyop tyl ty1) (Tyop tyl ty2)) )
END_CASE
END_FUN

FUN Type_occurs (str:string) (ty:Type) :bool =
CASE
(Is_Tyvar ty, (str = Tyvar_nam ty));
(Is_Tyop ty, lor (MAP (Type_occurs str) (Tyop tyl ty)) )
END_CASE
END_FUN

FUN Type_replace (l:(Type#string)list) (ty:Type) :Type =
CASE
(Is_Tyvar ty, IF mem2 (Tyvar_nam ty) l
                THEN corr2 (Tyvar_nam ty) l
                ELSE ty
                END_IF );
(Is_Tyop ty, Tyop (Tyop_nam ty) (MAP(Type_replace l)(Tyop tyl ty)) )
END_CASE
END_FUN

FUN Type_instlx (ty1:Type) (ty2:Type) :(Type#string)list =
CASE
(Is_Tyvar ty1, [(ty2,Tyvar_nam ty1)]);
(Is_Tyop ty1,
  LAPPEND(XMAP2 Type_instlx (Tyop tyl ty1) (Tyop tyl ty2)) )
END_CASE
END_FUN

FUN Type_instl (ty1:Type) (ty2:Type) :(Type#string)list =
  Type_instlx ty2 ty1
END_FUN

```

Appendix B.

Semantics in HOL for AL specification of type(after automatic simplification)

The access and recogniser functions for the type are omitted from the HOL and SML translations for brevity. The following is the axiom characterising the type Type generated from AL:

```
|- !f0 f1.
    ?! fn.
      (!s. fn(Tyvar s) = f0 s) /\
      (!s Tlst. fn(Tyop s Tlst) = f1(MAP fn Tlst)s Tlst)
```

The following are the semantic definitions corresponding to the AL functions:

```
Type_OK_DEF
|- (!tyl s. Type_OK tyl(Tyvar s) = T) /\
  (!tyl s Tlst.
    Type_OK tyl(Tyop s Tlst) =
    mem1 s tyl /\
    (LENGTH Tlst = corr1 s tyl) /\
    EVERY(Type_OK tyl)Tlst)

Type_compat_DEF
|- (!tyl s. Type_compat tyl(Tyvar s) = T) /\
  (!tyl s Tlst.
    Type_compat tyl(Tyop s Tlst) =
    Is_Tyop tyl /\
    (Tyop_nam tyl = s) /\
    (LENGTH(Tyop tyl tyl) = LENGTH Tlst) /\
    EVERY2 Type_compat(Tyop tyl tyl)Tlst)

Type_occurs_DEF
|- (!str s. Type_occurs str(Tyvar s) = (str = s)) /\
  (!str s Tlst.
    Type_occurs str(Tyop s Tlst) = lor(MAP(Type_occurs str)Tlst))

Type_replace_DEF
|- (!l s.
  Type_replace l(Tyvar s) = (mem2 s l => corr2 s l | Tyvar s)) /\
  (!l s Tlst.
    Type_replace l(Tyop s Tlst) = Tyop s(MAP(Type_replace l)Tlst))

Type_instlx_DEF |- !tyl ty2. Type_instlx ty2 tyl = Type_instl tyl ty2

Type_instl_DEF
|- (!s tyl. Type_instl tyl(Tyvar s) = [tyl,s]) /\
  (!s Tlst tyl.
    Type_instl tyl(Tyop s Tlst) =
    LAPPEND(XMAP2 Type_instlx Tlst(Tyop tyl tyl)))
```

Appendix C.

SML generated from AL specification of type

```

datatype Type = Tyvar of (( string)) |
               Tyop_tup of (( string))*(( Type list)) ;

fun Type_OK (tyl:(string * num)list) (ty:Type) :bool =
CASE_fun [
  (Is_Tyvar ty, fn zz => true),
  (Is_Tyop ty, fn zz => (mem1 (Tyop_nam ty) tyl) AND
    (LENGTH (Tyop_tyl ty) = corr1 (Tyop_nam ty) tyl) AND
    EVERY (Type_OK tyl) (Tyop_tyl ty) )
]
; (* end fun dec *)

fun Type_compat (ty1:Type) (ty2:Type) :bool =
CASE_fun [
  (Is_Tyvar ty2, fn zz => true),
  (Is_Tyop ty2, fn zz => (Is_Tyop ty1) AND
    ((Tyop_nam ty1) = (Tyop_nam ty2)) AND
    (LENGTH (Tyop_tyl ty1) = LENGTH (Tyop_tyl ty2)) AND
    (EVERY2 Type_compat (Tyop_tyl ty1) (Tyop_tyl ty2)) )
]
; (* end fun dec *)

fun Type_occurs (str:string) (ty:Type) :bool =
CASE_fun [
  (Is_Tyvar ty, fn zz => (str = Tyvar_nam ty)),
  (Is_Tyop ty, fn zz => lor (MAP (Type_occurs str) (Tyop_tyl ty)) )
]
; (* end fun dec *)

fun Type_replace (l:(Type * string)list) (ty:Type) :Type =
CASE_fun [
  (Is_Tyvar ty, fn zz => (if mem2 (Tyvar_nam ty) l
    then corr2 (Tyvar_nam ty) l
    else ty
  ) ),
  (Is_Tyop ty, fn zz => Tyop (Tyop_nam ty) (MAP(Type_replace l)(Tyop_tyl ty)) )
]
; (* end fun dec *)

fun Type_instlx (ty1:Type) (ty2:Type) :(Type * string)list =
CASE_fun [
  (Is_Tyvar ty1, fn zz => [(ty2,Tyvar_nam ty1)]),
  (Is_Tyop ty1, fn zz =>
    LAPPEND(XMAP2 Type_instlx (Tyop_tyl ty1) (Tyop_tyl ty2)) )
]
; (* end fun dec *)

fun Type_instl (ty1:Type) (ty2:Type) :(Type * string)list =
  Type_instlx ty2 ty1
; (* end fun dec *)

```