Executing Formal Specifications by Translation to Higher Order Logic Programming

James H. Andrews

Dept. of Computer Science
University of British Columbia
Vancouver, BC, Canada V6T 1Z4

Abstract. We describe the construction and use of a system for translating higher order logic-based specifications into programs in the higher order logic programming language Lambda Prolog. The translation improves on previous work in the field of executing specifications by allowing formulas with quantifiers to be executed, and by permitting users to pose Prolog-style queries with free variables to be instantiated by the system. We also discuss various alternative target languages and design decisions in implementing the translator.

1 Introduction

One of the early goals of formal specification in software engineering was to provide a formal model against which an implementation of the software, or a more detailed model, could be checked. This goal has not been fully realized in practice. But even where no tools exist to check an implementation against a specification, the process of writing a formal specification (FS) from an informal specification (IS) can still be very useful. For instance, since formal specification languages are less ambiguous than natural language, writing the FS forces the writer to deal with ambiguous passages in the IS by asking the writers of the IS to clarify them. The FS also casts the IS in the form of more detailed requirements which can be tracked during the course of system development.

Regardless of how the FS is used, however, a problem remains: how do we know when it is correct? Any contradictory statements in the IS which are not directly ambiguous may be translated directly into the FS. Subtleties of the specification language or unstated assumptions made in the translation from IS to FS may introduce other problems. Thus, just as we check the soundness of a software project by writing a FS, we must somehow check the soundness of the FS before using it as the basis of a project.

To check the FS, we can of course prove properties and consequences of it using a general-purpose theorem prover such as HOL. But this may be needlessly time-consuming, and we may find errors in the FS only when we have already done a great deal of manually intensive verification based on the faulty version. To make an initial check of our FS, it is sometimes good enough simply to perform steps like unfolding definitions and checking for the existence of terms which satisfy given formulas; in other words, to execute the specification. Thus, ironically, we can test the feasibility of a program by writing a formal model, and test the feasibility of the formal model by treating it as a program.

How should a specification be executed? Building a custom environment in which to execute it is an obvious choice, but needlessly duplicates much of the effort that goes into building programming languages. A more expeditious approach is to implement a program which translates from the specification language to an existing programming language. This approach has the further advantage of giving the user access to the run-time environment and whatever static analysis tools have been developed for the programming language.

This paper discusses the implementation of a translator from the specification language S to the programming language Lambda Prolog. S is a higher order logic-based specification language developed at the University of British Columbia by Jeffrey Joyce, Nancy Day and Michael Donat [JDD94]. It is a central focus of the FormalWare project, an industry-funded technology transfer project concerning formal methods in software engineering. Lambda Prolog, developed initially by Dale Miller and Gopalan Nadathur at the University of Pennsylvania [MN86, MNPS91], is an elegant higher order logic extension of the Prolog programming language.

This work extends earlier work by Camilleri [Cam88], Murthy [Mur91] and Rajan [Raj92], who translated HOL to ML. Translating to Lambda Prolog allows us to use the backtracking and higher order unification capabilities of the target language to execute such constructs as the Hilbert epsilon operator and disjunctions inside quantifiers. The translation of quantified variables is more straightforward, since the target language also has a notion of quantification. It also allows the user to ask Prolog-style queries with free variables, to be instantiated by the system. The results are cast in terms of S, but are applicable to any higher order specification language.

This work can also been seen as extending work by Kahn [Kah87] and Hannan and Miller [HM90] on operational semantics and logic programming systems. These authors studied specifically translating operational semantics into Prolog or expressing them in Lambda Prolog; in contrast, this paper deals with the translation of general specifications, possibly involving higher order features, into Lambda Prolog.

The structure of this paper is as follows. Section 2 briefly describes the S specification language. Section 3 discusses Lambda Prolog as a target language for the translation, comparing it to S and to some alternate target languages. Section 4 describes the translation scheme and the translated program in greater detail. Section 5 discusses some of the design decisions taken for the implementation, mainly to deal with the problems inherent in trying to execute formulas without crossing into full theorem-proving. Section 6 presents an extended example and experiences with the translation system. Finally, Section 7 gives conclusions and describes future work.

2 The S Specification Language

S is a specification language developed at the University of British Columbia. It was designed to be more readable than other formal specification languages such as Z or the script language used as input to HOL, while still retaining the abstracting power of uninterpreted constants and higher order logic. There is a typechecker for S input files, called Fuss [JDD96]; getting an S input file to be accepted by Fuss is comparable to getting an ML program to typecheck. Various other tools associated with S are under development.

S construct	Meaning	Equivalent Lambda Prolog construct
`: process;`	Type declaration	`kind process type.`
`: (A, B) array;`	Parameterized type declaration	`kind array type -> type -> type.`
`version_number: num;`	Constant declaration	`type version_number num.`
`: (A) tree := leaf :A | branch :(A) tree :(A) tree;`	Type definition	`kind tree type -> type.` `type leaf A -> (tree A).` `type branch (tree A) -> (tree A) -> (tree A).`
`: style == colour # size;`	Type abbreviation	No equivalent
`mother X := (parent X) & (female X);`	Boolean function definition	`type mother person -> o.` `mother X :- parent X, female X.`
`reverse X := rev_aux X [];`	Non-boolean function definition	No direct equivalent
`~(separation a1 a2 < 300);`	Boolean constraint	No general equivalent; Horn clause constraints declared with Prolog-style clauses
`function X . (g X X)`	Lambda abstraction	`X\ (g X X)`

Fig. 1. Syntactic constructs of S and Lambda Prolog.

The syntactic constructs of S are summarized in Figure 1, along with a comparison to Lambda Prolog features which will be discussed later. In S, one can declare the names of new types (possibly parameterized), and declare new constants to be of those types. Types and "term constructor" constants can be declared together in ML-style “type definitions”, and one can abbreviate one type by a new name. A type expression in S is a declared or built-in type name, or the type of functions from one type to another (written `T1 -> T2`), or the cross product of two types (written `T1 # T2`). The built-in types of S are `num` (the type of numbers) and `bool` (the Boolean values “true” and “false”).

A constant can also be defined as a function with a given meaning, by constructs such as “`reverse X := rev_aux X []`”. ML-style multi-clause function definitions are also available. For such function definitions, the Fuss typechecker will infer the type of the arguments and the range of the function.

Expressions are largely as they are in ML. The lambda-expression which would be written $\lambda X.(f\ X)$ in lambda notation is written as `function X . (f X)` in S; the formulas $\exists X (p\ X)$ and $\forall X (p\ X)$ are written “`exists X . (p X)`” and “`forall X . (p X)`” respectively. The expression $\epsilon X.(p\ X)$, where ϵ is Hilbert’s epsilon operator, is written “`select X . (p X)`”. (The meaning is “some individual X such that $(p\ X)$ is true.”) Constraints – that is, the actual formulas constituting the formal specification – are written as expressions of type `bool` terminated by semicolons, on the same level as definitions in an S source file.

An example of a full S specification is contained in Figure 2. It is a specification of a simplified version of CCS, Milner’s Calculus of Concurrent Systems [Mil80]. In the example, we define a process as being of one of four forms:

- The “null process” `nullprocess`, which can do nothing;
- An `(andthen L P)` process, which can perform the action indicated by its label `L` and then become process `P`;
- A `(plus P1 P2)` process, which can choose between becoming `P1` or `P2`; or
- A `(par P1 P2)` process, which represents processes `P1` and `P2` running in parallel and communicating via synchronized labels.

The relation `can_do Process Label Newprocess` holds if `Process` can do the action indicated by `Label`, becoming `Newprocess`. The function `trace` is intended to return a possible trace (sequence of actions) of its argument. The last few lines define some example processes.

Of course, this is only an example for the present purposes; it is possible to specify CCS in other formalisms as well. Readers may wish to compare this treatment with Nesi’s specification of full value-passing CCS in HOL [Nes93].

```
%include startup.s

: label;
: process := andthen :label :process
           | plus :process :process
           | par :process :process
           | nullprocess;
tau: label;
prime: label -> label;

can_do (andthen Label Process) Donelabel Newprocess :=
  (Donelabel = Label) /\ (Newprocess = Process) |
can_do (plus Process1 Process2) Label Newprocess :=
  can_do Process1 Label Newprocess \/
  can_do Process2 Label Newprocess |
can_do (par Process1 Process2) Label Newprocess :=
  ( exists Newprocess1 .
    (  can_do Process1 Label Newprocess1
    /\ (Newprocess = (par Newprocess1 Process2)))) \/
  ( exists Newprocess2 .
    (  can_do Process2 Label Newprocess2
    /\ (Newprocess = (par Process1 Newprocess2)))) \/
  ( exists Newprocess1 Newprocess2 Handshakelabel .
    ( ( can_do Process1 Handshakelabel Newprocess1 /\
        can_do Process2 (prime Handshakelabel) Newprocess2 ) \/
      ( can_do Process2 Handshakelabel Newprocess2 /\
        can_do Process1 (prime Handshakelabel) Newprocess1 ) )
    /\ (Newprocess = (par Newprocess1 Newprocess2))
    /\ (Label = tau));

trace Process :=
  if (Process == nullprocess) then []
  else ( select Trace .
         ( exists Label Newprocess .
           (  (can_do Process Label Newprocess)
           /\ (Trace = (CONS Label (trace Newprocess))))));

% Example

a, b, c: label;
process1 := (andthen a (andthen b nullprocess));
process2 := (andthen a (plus (andthen b nullprocess)
                             (andthen c nullprocess)));
process3 := (plus (andthen (prime a) (andthen (prime b) nullprocess))
                  (andthen (prime a) (andthen (prime c) nullprocess)));
```

Fig. 2. A sample specification in S: The CCS formalism.

3 Lambda Prolog as a Target Language

Lambda Prolog was chosen as the target language for the translation because of the relatively large number of features it shares with S, and its ability to reason both about logical connectives and quantifiers and about higher-order constructs.

3.1 A Comparison of S and Lambda Prolog

S and Lambda Prolog are both based on typed higher order logic. Both languages allow type declarations; parameterized type declarations; declarations of new uninterpreted constants of arbitrary types; and definitions of the meanings of functions with boolean range. Figure 1 shows the correspondences between the syntax of these constructions in the two languages.

The most important notational difference between the languages is that Lambda Prolog has an explicit notion of "kind". Kinds form a third level of objects above individuals and types; just as each individual belongs to some type, each type belongs to some kind. Thus, for example, `list` is of kind `type -> type` in Lambda Prolog, because it takes a type (the type of its elements) and returns another type (the type of lists of that element type). In S, the notion of kind is implicit in type expressions.

Some of the most important features which S (or its typechecker Fuss) has but which Lambda Prolog[1] does not have are: type inference on constant definitions; type abbreviations; the ability to define functions with arbitrary range; the "select" (Hilbert epsilon) operator; and constraint paragraphs.

As we shall see, the only one of these that causes a significant problem in the translation is the absence of function definitions. We must translate functions into predicate clauses in order to achieve the same effect. However, these features help make S a more useful specification language. Because of their absence, Lambda Prolog itself cannot be used effectively as a specification language.

The main features which Lambda Prolog has but S/Fuss does not have are, obviously, its explicit evaluation semantics, runtime environment and so on. Lambda Prolog also has a slightly richer type structure, theoretically allowing objects to be of kind `(type -> type) -> type` whereas in S we are implicitly restricted to "linear" kinds such as `type -> (type -> (type -> type))`; however, this added richness is not usually important in practice.

[1] Unless described otherwise, all references to Lambda Prolog in this paper are to the Terzo implementation available from the University of Pennsylvania [Mil96].

3.2 Alternative Target Languages

What other languages could be used as the target language for the translation from S? It can be assumed that we do not want to consider languages that involve explicit memory allocation and pointer manipulation for building dynamic data structures, since the task of building a translator to such a language would be of similar complexity to that of building an interpreter for S. There are several other programming languages with features which could make them useful as the translation target.

- ML was the target language for both Murthy's [Mur91] and Rajan's [Raj92] translations from HOL. It also has parameterized types, polymorphic functions, and lambda-expressions. Its main advantages over Lambda Prolog are that it is more widely known, used and supported, and that one can make explicit function declarations in it. However, it does not have built-in support for logical variables or explicit backtracking over disjunctions, as Lambda Prolog does. Thus the translation would have to use, or build, ML code to simulate these features, somewhat defeating the purpose of translating rather than custom-building an interpreter.
- Standard Prolog has logical variables and explicit backtracking, and is even more widely used and supported than ML. However, it does not handle lambda-expressions and has no built-in capabilities for checking the well-typedness of queries. Again, these things would have to be built.
- Aït-Kaci's language LIFE [AKP93] would be another interesting choice, insofar as it allows both function and predicate definitions, and can handle function calls containing uninstantiated variables. (This issue is an important one, as we will later see.) However, LIFE also lacks lambda-abstraction capabilities, and its type system has an entirely different foundation from that of the higher order logic type system.

Various other logic programming systems exist (see Section 5) which could provide other useful features in a target language. However, the crucial combination of lambda notation, unification and backtracking, which makes the execution of higher order logical constructs possible, is available only in Lambda Prolog.

4 The Translation Scheme

There are two main components to the translation scheme: a program called `s2lp` and a Lambda Prolog source file called `s2lp_common.lp`. `s2lp` acts as a Unix filter, translating an S file on its standard input into a Lambda Prolog file on its standard output.

It is an adaptation of the S typechecking program Fuss [JDD96], and typechecks its input before translation. `s2lp_common.lp` contains declarations supporting the translated specifications, and is included in every translated file. Here we will look at the most important features of `s2lp` and `s2lp_common.lp`.

4.1 Translation of Type Declarations and Function Definitions

`s2lp` translates S type declarations into Lambda Prolog in the straightforward way; for instance, the declaration of the parameterized type `array` in Figure 1 is translated into the equivalent Lambda Prolog construct in Figure 1.

The main duty of `s2lp` is to generate Lambda Prolog clauses of the predicate `eval` for every constant declaration and definition in the S input file. `eval` is a predicate which takes two arguments of the same type, and (in its usual mode of use) instantiates its second argument to the "value" (according to the S input) of its first argument. The queries we will pose to the Lambda Prolog program will usually be of the form `eval` *expr* `Result`, where *expr* is some expression to be evaluated and `Result` is a variable which will be bound to its value.

Thus the S polymorphic function declaration

```
  (:Element_type)
  reverse (X: (Element_type) list) := rev_aux X [];
```

will (assuming `rev_aux` has been declared) produce the expected Lambda Prolog type declaration[2]

```
type    reverse ((list Element_type) -> (list Element_type)).
```

but also the `eval` clause

```
eval (reverse X) Result$ :-
  eval ((rev_aux X) 'NIL') Result$.
```

Because of this clause, when `eval` evaluates a call to `reverse`, it does so by immediately evaluating the corresponding call to `rev_aux`.

4.2 Translation of Constant Declarations and Type Definitions

The recursion of the `eval` predicate in the translated program "bottoms out" on declared constants, in keeping with the view of these constants as "uninterpreted". Thus the S constant declaration `version: num` will produce the Lambda Prolog declarations

[2] Note that in Lambda Prolog, variable names start with an upper case letter and constant names start with a lower case letter, whereas case is not significant in S. This paper largely glosses over the difference by choosing names consistent with Lambda Prolog, but `s2lp` does do the required translation.

```
type  version  num.
eval (version) (version).
```

This indicates that the value of the expression `version` is the term `version` itself. Similarly, the type definition

```
: (A) tree := leaf :A | branch :(A)tree :(A)tree;
```

which creates constructors `leaf` and `branch`, will produce the declarations

```
kind    tree    type -> type.
type    leaf    (A -> (tree A)).
type    branch  ((tree A) -> ((tree A) -> (tree A))).
```

but also the clauses

```
eval (leaf X$1) (leaf Y$1) :-
  eval X$1 Y$1.
eval (branch X$1 X$2) (branch Y$1 Y$2) :-
  eval X$1 Y$1,
  eval X$2 Y$2.
```

These clauses ensure that, for instance, if an expression like `(branch Tree1 Tree2)` is evaluated, the arguments will be evaluated and the results will be assembled into a new `branch` structure as the value.

4.3 The Common Declarations

Every translated file contains a directive to include the Lambda Prolog source file `s2lp_common.lp`. This file contains supporting declarations of the built-in types and constants of S (such as `'NIL'`), as well as the `eval` clauses for evaluating logical expressions.

These clauses are crucial to the success and ease of the translation. For example, in S one can write Hilbert-epsilon expressions of the form `select x . `A, where A is any formula. In accordance with higher order logic conventions, these are parsed as `SELECT (`λ`x. `A`)`. The `eval` clause in `s2lp_common.lp` which evaluates such expressions is simply

```
eval ('SELECT' Abstraction) Result :-
  eval (Abstraction Result) 'T'.
```

In other words, if the lambda-abstraction applied to the result is a formula which evaluates to the truth value `'T'`, then the value of the `'SELECT'` expression is the result.

With the declarations in `s2lp_common.lp`, one can even compute goals involving implication and the universal quantifier, to the extent to which this is possible in Lambda Prolog; for instance, one can write boolean functions with bodies of the form `forall Pred . (Defn ==> Goal)`, where `Defn` is a Horn clause defining `Pred` and `Goal` is a boolean expression.

5 Design Decisions

A specification is a collection of logical formulas associated with some notion of what constitutes a proof; since even first order logic is undecidable, a proof cannot always be found by following a predefined strategy. A program, however, is an object associated with a predefined execution strategy. Thus an attempt to "execute a specification" must inevitably come up against the contrast between these two notions. These issues have long been explored in the functional and logic programming communities, and the design decisions made in the `s2lp` translation scheme reflect some of the knowledge that has been gained.

5.1 Negation

In the logic programming community there seems to be a consensus that trying to handle full negation in a sound and complete manner takes one into the realm of theorem proving. Some schemes, such as Loveland's "near-Horn" programming [LR91], provide for a graceful degradation of performance from regular Prolog when the user attempts to work with programs containing a small number of negations; but they are not available with Lambda Prolog.

`s2lp` deals with this issue by providing `eval` clauses for negation which perform the usual Lambda Prolog negation as failure, which is in some circumstances incomplete or unsound. Users should understand that if they wish to evaluate negation completely, they should move to a semi-automated theorem proving system such as HOL.

5.2 Uninstantiated Variables in Function Calls

When working with a combination of function and predicate definitions, the issue arises of what to do with a function call which contains an uninstantiated logical variable. This arises frequently in `s2lp` translations when the source specifications involve both quantification and functional syntax.

For example, consider the expression `trace process1`, with respect to the specification of CCS (Fig. 2); we can evaluate this expression in the translated program by the query `eval (trace process1) Result`. The evaluation will eventually involve the solving of a subgoal of the form

```
eval (can_do process1 Label Newprocess) Result1
```

which in turn will eventually involve the solving of a subgoal of the form

```
eval ('=$' Donelabel a) Result2
```

where '=$' is the prefix binary operator corresponding to the S operator =. But now, if the Lambda Prolog program treats the arguments of '=$' exactly as it would any other expressions, it will try to "evaluate" the uninstantiated variable Donelabel. This results in an infinite recursion as the program matches variables against the existing eval clauses. Clearly this is not acceptable.

Aït-Kaci's language LIFE [AKP93] takes the approach of delaying function calls containing uninstantiated variables until such time as all their arguments are instantiated, a scheme called *residuation*. Again, this scheme is not available in combination with Lambda Prolog.

The scheme adopted in s2lp is to assign a special meaning to the standard "=" operator. Normally, arguments to any function symbol appearing in a definition body are evaluated in that function's eval clause. In contrast, the clauses in s2lp_common.lp handling = evaluate only the right-hand argument, and unify the value with the left-hand argument. Thus s = t will succeed only if s is syntactically identical to the eval value of t, or is some partially-instantiated generalization of that value (including an uninstantiated variable). With care, as in the CCS example, it can be used to instantiate uninstantiated variables in the correct pattern. The operator == is defined with the more expected semantics of evaluating both arguments and comparing the results.

5.3 Caller and Callee Evaluation

s2lp adopts a "callee evaluation" scheme (sometimes referred to as "call-by-name"), where the called function is passed unevaluated arguments and evaluates them itself during the course of its computation. A potential alternative is a "caller evaluation" scheme: each eval clause assumes that all its function arguments contain no calls to defined functions (though they may contain uninstantiated variables), and pre-evaluates all the arguments of the functions it calls. For instance, the S declaration

```
stalled Y := ((trace Y) = []);
```

would under s2lp's "callee evaluation" be translated into the form

```
eval (stalled Y) Result :-
  eval ('=$' (trace Y) []) Result.
```

but under a "caller evaluation" scheme be translated into the form

```
eval (stalled Y) Result :-
  eval (trace Y) Trace,
  eval ('=$' Trace []) Result.
```

Caller evaluation allows an uninstantiated variable to be passed through until it reaches an `=` expression and is unified straightforwardly. It therefore solves the problem of uninstantiated variables. However, it has the unfortunate effect of requiring that users similarly evaluate each function call in each query given to the Lambda Prolog interpreter, making queries clumsier.

Moreover, under caller evaluation it is not clear how to translate function definitions of the form `apply X Y := (X Y)` in such a way as to allow the first argument to be either a constant, a defined function, or a lambda expression. On the whole, the callee evaluation scheme seems less problematic for the user.

5.4 Uninterpreted Constants and Interpreted Operators

In specifications, we may occasionally want to declare uninterpreted boolean constants – for instance, `heater_on: bool` – to stand for conditions on the environment of the system under specification. When such constants are declared in S specifications, the `s2lp` translation essentially treats them as false rather than taking them as "new truth values", as we might prefer.

An alternative is for the translated program to take the expression to be evaluated, whether boolean or otherwise, and rewrite it to the most reduced form possible. Donat [Don97] has developed a rewriting package as an extension of Fuss for use in generating test cases from S specifications. This rewriting package is preferable to `s2lp` for this purpose because the truth of quantified formulas does not have to be determined.

Unfortunately, rewriting does not help when it comes to quantified formulas. For example, if `(p 3)` rewrites to `heater_on` and `(p 4)` rewrites to `ac_on`, it is not at all clear what the returned value of `exists X . (p X)` should be. Logic programming with boolean constraints, as is possible in SICStus Prolog [oCS94], essentially allows for such behaviour by allowing variables to be unified with "true" and "false" in whatever way will cause a goal to succeed.

The problem extends to constants of types other than `bool`. For instance, if `temperature: num`, we would like the expression `temperature+2 = temperature+3` to evaluate to `'F'`, but `temperature*2 = temperature*3` to evaluate to `'T'` while unifying `temperature` with 0. In general, the problem of uninterpreted constants and interpreted operators leads us into the realm of constraint logic programming systems [JMSY92], which can process such queries correctly. Again, however, constraint processing is not available in Lambda Prolog.

In conclusion, then, it seems that the ideal target language for translation from a higher order specification language would be a higher order constraint logic programming language with near-Horn processing and residuation. In the absence of such a language, Lambda Prolog seems to be a reasonable choice given the design decisions made in the implementation of s2lp.

6 Working with the Translated Program

The following is an extended example to illustrate how the s2lp translation of an S specification works. After the command "s2lp <ccs.s >ccs.lp", which translates the CCS specification given in Fig. 2 into Lambda Prolog, we invoke the Lambda Prolog interpreter, terzo. Commands given to the Terzo loader start with #.

```
re[1]: /usr/bin/time terzo
loading /isd/local/generic/src/terzo/bin/sun4/.lpsml ...................
..... done
Terzo lambda-Prolog, Version 1.0b, Built Wed Jul 17 22:14:23 EDT 1996
[reading file /isd/local/generic/src/terzo/lib/terzo.rc]
[closed file /isd/local/generic/src/terzo/lib/terzo.rc]
Terzo> #load "ccs.lp".
[reading file ./ccs.lp]
[reading file ./s2lp_common.lp]
GC #0.0.0.0.1.1:   (0 ms.)
module s2lp_common
[closed file ./s2lp_common.lp]
GC #0.0.0.1.2.14:   (80 ms.)
module s2lp
[closed file ./ccs.lp]
Terzo> #query s2lp.
?-
```

("GC" lines give Terzo garbage collection statistics.) We are now in an interpretive loop for the program, and can give queries. First we ask it to evaluate the simple function call process1 and return the result in the variable Result.

```
?- eval process1 Result.

Result = andthen a (andthen b nullprocess) ;

no more solutions
```

Next we ask it for the possible traces of this process (the sequences of actions which it can perform).

```
?- eval (trace process1) Result.
GC #0.0.0.2.3.81:   (130 ms.)

Result = CONS a (CONS b NIL) ;

no more solutions
```

We do the same for a more complex process.

```
?- eval (trace process2) Result.

Result = CONS a (CONS b NIL) ;

Result = CONS a (CONS c NIL) ;

no more solutions
```

We have obtained two results from the function call, and the `select` expression inside it, because there were two traces which met the given criteria. Next we look at the process made up by placing `process2` and `process3` in parallel; we evaluate the boolean function `can_do` with two uninstantiated variables, to see what possible ways the process can evolve. We should get five solutions.

```
?- eval (can_do (par process2 process3) Label Newp) Boolresult.
GC #0.0.0.3.4.287:   (140 ms.)

Label = a
Newp =
 par (plus (andthen b nullprocess) (andthen c nullprocess))
  (plus (andthen (prime a) (andthen (prime b) nullprocess))
    (andthen (prime a) (andthen (prime c) nullprocess)))
Boolresult = T ;

Label = prime a
Newp =
 par (andthen a (plus (andthen b nullprocess) (andthen c nullprocess)))
  (andthen (prime b) nullprocess)
Boolresult = T ;

Label = prime a
Newp =
 par (andthen a (plus (andthen b nullprocess) (andthen c nullprocess)))
  (andthen (prime c) nullprocess)
Boolresult = T ;

Label = tau
Newp =
 par (plus (andthen b nullprocess) (andthen c nullprocess))
  (andthen (prime b) nullprocess)
Boolresult = T ;
GC #0.0.0.3.5.519:   (50 ms.)

Label = tau
Newp =
 par (plus (andthen b nullprocess) (andthen c nullprocess))
  (andthen (prime c) nullprocess)
Boolresult = T ;
```

```
Label = Label
Newp = Newp
Boolresult = F ;

no more solutions
```

In fact we get a sixth solution, in which the variables have not been instantiated and the system returns a result of "false". This is to be expected as the final alternative, since we are backtracking on the value of a boolean expression and the only other possibility is falsehood. Note that the syntax of the output of Lambda Prolog is very close to that of simple S terms, and can in most cases be input back into Fuss if necessary.

```
?-       241.0 real        59.7 user         1.7 sys
re[2]:
```

After breaking out of the interpreter, we find that all this evaluation has taken about a minute of CPU time. We have been running the Terzo interpreter; on compiled Lambda Prolog systems like Prolog/MALI [BR93] this may be significantly reduced due to such techniques as first argument indexing.

Finally, we re-enter Terzo and try to see the possible traces of the `(par process2 process3)` process.

```
re[229]: terzo
...
Terzo> #query s2lp.
?- eval (trace (par process2 process3)) Result.
GC #0.0.0.2.3.83:   (150 ms.)
GC #0.0.0.3.4.260:   (270 ms.)
GC #0.0.1.4.5.380:   (380 ms.)
GC #0.1.2.5.6.494:   (400 ms.)
GC #0.1.2.5.7.620:   (30 ms.)
^C
interrupt
Terzo>
```

We have found a hole in the specification: there is a specification for the trace of the null process, but no specification for the trace of the process `(par nullprocess nullprocess)`, one of the possible descendents of the process in the query. The system has responded by going into an infinite loop on one of the uninstantiated variables. Now we can return to the specification and patch the hole before basing any further work on it. Future edit-translate-execute cycles will allow our confidence in the correctness of our specification to increase.

`s2lp` has been run on a number of different specifications, including portions of an S specification of a telecommunications network being developed within the FormalWare project. The translated Lambda Prolog code returns results of function calls as expected and finds solutions within its capabilities.

7 Conclusions and Future Work

`s2lp` represents an advance over some previous schemes for evaluating higher order specifications, in that it allows a wider range of quantified formulas to be executed using the Prolog backtracking and unification features. With `s2lp`, users can build formal specifications of their systems of interest in S, and translate them into Lambda Prolog in order to verify that desired properties hold before doing more extensive theorem-proving or implementation.

The results of this paper are not specific to S, but generalize to other specification languages. For instance, we should be able to translate HOL into Lambda Prolog via ML functions which traverse the ML internal representation of HOL terms. Future extensions of the scheme could include extensions for partially evaluating arithmetic expressions, or providing the option of "caller evaluation" (see Section 5.3) for situations which require it.

Ideally, one can envision an integrated logic and functional programming and specification language, in which computable functions and predicates can be defined in a natural style, but uninterpreted constants and assertions can be included where required to produce a readable and sufficiently abstract specification. Such a scheme might be built as a generalization of an existing programming language, such as ML or Lambda Prolog, or an existing specification language, such as Z or S.

8 Acknowledgments

Thanks to Jeff Joyce for suggesting the topic of this paper and, together with Nancy Day and Michael Donat, for the development of the Fuss typechecker on which `s2lp` was based, for giving valuable comments on an earlier version of this paper, and for providing a stimulating working environment in which to explore ideas of logic, computation, and software engineering.

The FormalWare project is financially supported by the BC Advanced Systems Institute (BCASI), Hughes Aircraft of Canada Limited Systems Division (HCSD), and MacDonald Dettwiler Limited (MDA). The author derives half his funding from this project and half from the generous support of Dr. Paul Gilmore of UBC Computer Science, via his grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[AKP93] Hassan Ait-Kaci and Andreas Podelski. Towards a meaning of Life. *Journal of Logic Programming*, 16(3/4):195, July 1993.

[BR93] Pascal Brisset and Olivier Ridoux. The compilation of Lambda Prolog and its execution with MALI. Technical Report 1831, INRIA, 1993.

[Cam88] Albert Camilleri. Simulation as an aid to verification using the HOL theorem prover. Technical Report 150, University of Cambridge Computer Laboratory, October 1988.

[Don97] Michael R. Donat. Automating formal specification-based testing. In *TAPSOFT: 7th International Joint Conference on Theory and Practice of Software Engineering*, April 1997.

[HM90] John Hannan and Dale Miller. From operational semantics to abstract machines: Preliminary results. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 323–332, Nice, France, June 1990. ACM Press.

[JDD94] Jeffrey J. Joyce, Nancy A. Day, and Michael R. Donat. S: A machine readable specification notation based on higher order logic. In *Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop*, volume 859 of *LNCS*. Springer-Verlag, 1994.

[JDD96] Jeffrey J. Joyce, Nancy A. Day, and Michael R. Donat. S – a general-purpose specification notation. Draft report, 1996.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[Kah87] G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39, Passau, Federal Republic of Germany, Feb 1987. Springer.

[LR91] Donald W. Loveland and David W. Reed. A near-Horn Prolog for compliation. In *Computational Logic: Essays in Honor of Alan Robinson*, Cambridge, Mass., 1991. MIT Press.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[Mil96] Dale A. Miller. Lambda Prolog home page. `http://www.cis.upenn.edu/ dale/lProlog/index.html/`, 1996.

[MN86] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In *Proceedings of the Third International Logic Programming Conference*, Imperial College, London, July 1986.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Mur91] Chetan R. Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*. IEEE, 1991.

[Nes93] Monica Nesi. Value-passing CCS in HOL. In *HOL Users' Group Workshop*, Vancouver, August 1993.

[oCS94] SICS (Swedish Institute of Computer Science). SICStus Prolog user's manual. Technical report, Swedish Institute of Computer Science, Kista, Sweden, April 1994.

[Raj92] P. Sreeranga Rajan. Executing HOL specifications: Towards an evaluation semantics for classical higher order logic. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, Leuven, Belgium, September 1992. Elsevier.

This article was typeset using the LaTeX macro package with the LLNCS2E class.