Executing Formal Specifications by Translation to Higher Order Logic Programming

—

James H. Andrews

—

Dept. of Computer Science, University of BC
Vancouver, BC, Canada

⇓

Dept. of Computer Science, University of Western Ontario
London, Ont., Canada

- Motivations
- Basic System Structure
- Translation Scheme
- Working With the System
- Design Decisions

Motivations

Why do formal specification?

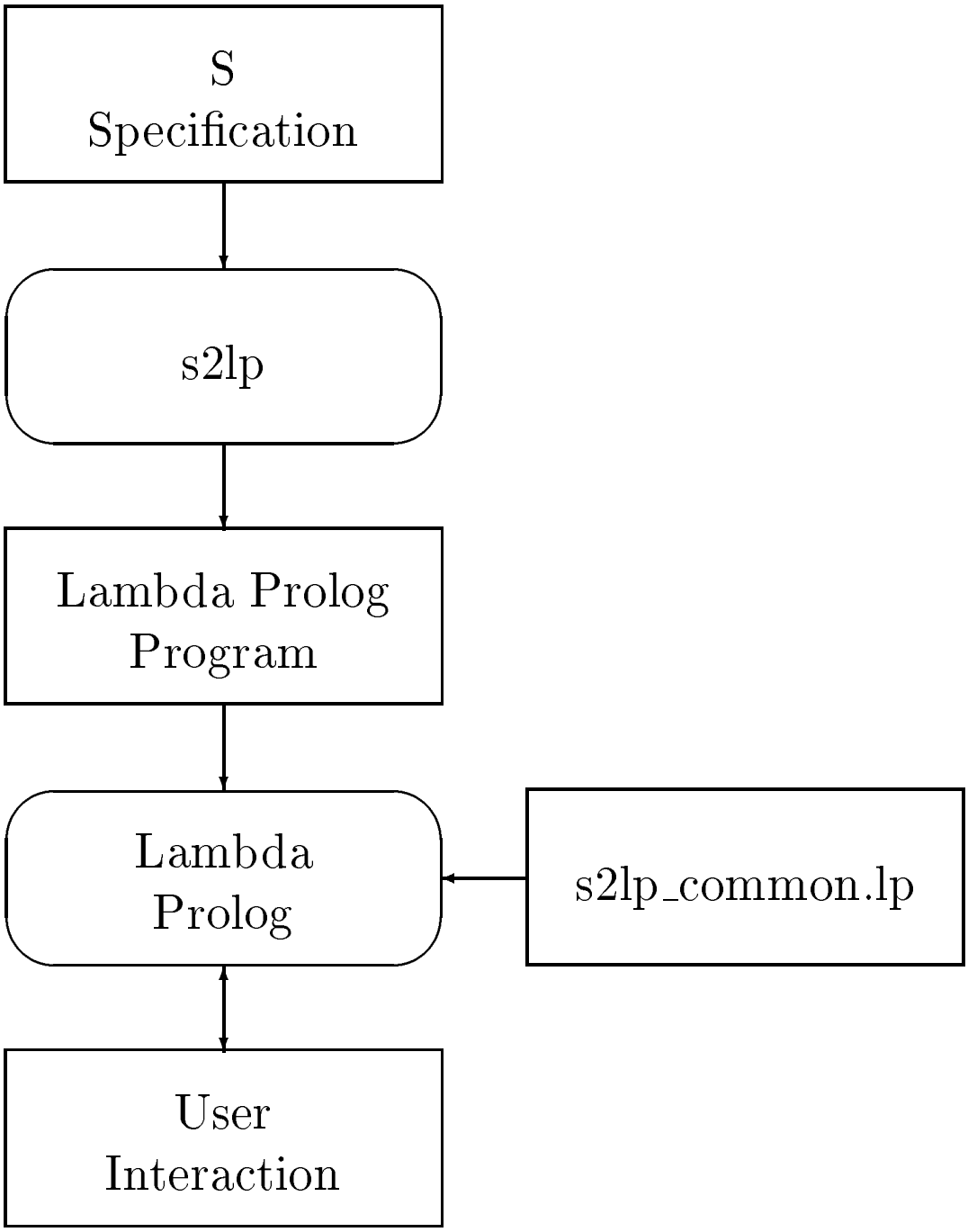
- High quality specs $\Rightarrow$ lower development costs
- Formal notations $\Rightarrow$ high quality specs
- Higher order typed logic $\Rightarrow$ safer, more expressive formal notations

Why execute formal specs?

- Incorrect/ambiguous specs $\Rightarrow$ lower quality
- Theorem proving very labour-intensive
- (Partial) execution $\Rightarrow$ greater confidence in correctness

How to execute higher order formal specs?

- Build customized engine: reinvents wheels
- Translate into functional language: misses some key features
- Translate into logic programming language: handle quantifiers, obtain added unification, backtracking features
- Need higher order logic programming language like Lambda Prolog


S
Specification
s2lp
Lambda Prolog
Program
Lambda
Prolog
s2lp_common.lp
User
Interaction

S and Lambda Prolog: Declarations

- S: developed 1994 by Joyce, Day, Donat (UBC)
- Lambda Prolog: developed 1986 by Nadathur, Miller (UPenn)

- Constant declaration

  S:
```
version_number: num;
```
  LP:
```
type version_number num.
```

- Type declaration

  S:
```
: process;
```
  LP:
```
kind process type.
```

- "Type definition"

  S:
```
: num_tree := leaf :num
            | branch :num_tree :num_tree;
```
  LP:
```
kind num_tree type.
type leaf num -> num_tree.
type branch
  num_tree -> num_tree -> num_tree.
```

Simplified CCS in S: Declarations

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

% Example labels
a, b, c: label;
```

Simplified CCS in S: Function Definitions

Functions without parameters:

```
process1 :=
  (andthen a (andthen b nullprocess));
process2 :=
  (andthen a (plus (andthen b nullprocess)
                   (andthen c nullprocess)));
```

Function with a parameter:

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

Translating Function Definitions

- Lambda Prolog has Prolog-style *clauses* for predicates
- Key concept: `eval` predicate
- Lambda Prolog query `(eval` *expr* `Result)` binds variable `Result` to "value" of *expr*
- `type eval A -> A -> o.`
- `s2lp`'s main task: produce `eval` clauses from S spec

- Function definition

S:
```
merge P Q :=
  if (P=nullprocess) then Q
  else (par P Q);
```

LP:
```
type merge process -> process -> process.
eval (merge P Q) R :-
  eval ('COND' (P=nullprocess)
               Q
               (par P Q))
       R.
```

Translating Constants

Recursion of `eval` bottoms out on declared constants

- Declared constant

  S: `a: label;`

  LP:
```
type a label.
eval a a.
```

Constructors evaluate their arguments

- Constructor

  S: `plus: process -> process -> process;`

  LP:
```
type plus process -> process -> process.
eval (plus X$1 X$2) (plus Y$1 Y$2) :-
  eval X$1 Y$1,
  eval X$2 Y$2.
```

`eval` aided by builtin declarations in `s2lp_common.lp`:

```
eval ('COND' Cond Then Else) Result :-
  eval Cond 'T',
  !,
  eval Then Result.
eval ('COND' Cond Then Else) Result :-
  eval Else Result.
```

Working with Translated Program

- Issue queries of form `eval` *expr* `Result`
- If *expr* fully instantiated and functional, returns value
  - e.g. `eval (merge process1 process2) Result`
- If quantification involved, does backtracking search
  - e.g. `eval (trace process2) Result`
- User must be aware of usual Prolog strategy
- Handles everything functional translations could handle
- Can do more if spec is constructed carefully

Efficiency:

- "Terzo" interpreter fairly slow on translated program
- Lambda Prolog compilers (e.g. Prolog/Mali) would improve

Design Decisions

Many decisions impinge on active research areas:

- How to integrate FP and LP?
- What is the boundary between LP and ATP?

Uninstantiated variables:

- Translated function cannot know whether var instantiated
- If `eval` given uninstantiated var to evaluate, diverges
- We want to give uninstantiated vars to equality operators
- Tip:
  - "`A == B`" evaluates `A` and `B`, unifies
  - "`A = B`" evaluates only `B`, unifies
  - Use "`A = B`" rather than "`A == B`" if you expect `A` might be uninstantiated

Design Decisions

Evaluation responsibility:

- "Caller evaluation":
  - Calling function pre-evaluates arguments
  - Called function assumes arguments evaluated
  - Bypasses problems of uninstantiated vars
  - Does not integrate well with lambda expressions: e.g.

```
apply X Y := (X Y);
foo A B :=
  apply (function A. bar (baz A)) B;
```

- "Callee evaluation":
  - Called function evaluates its own arguments
  - Actual scheme adopted by `s2lp`

Other decisions:

- Negation as failure
- No constraint processing

Epilogue

Conclusions:

- `s2lp` extends range of executability of specs
- Adding features to Lambda Prolog would extend further
- Similar scheme possible for other spec languages

Availability:

- `s2lp` adapted from `fuss` typechecker
- Source should be available within next year

Vision:

- Integrated functional/logic/spec language