A proof system embedded in Haskell

Gergely Dévai

Eötvös Loránd University, Budapest, Hungary

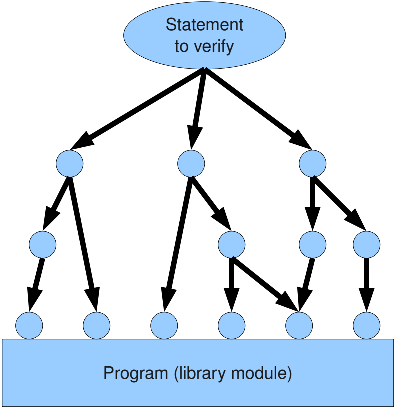
CEFP 2009, Budapest-Komárom

May 27, 2009

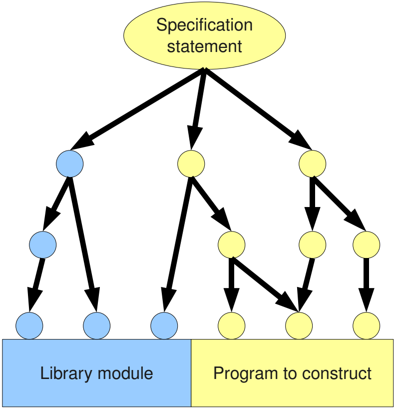
Goals

- to have programs that are *proved correct*
  - verification
  - *correctness by construction*
- both for *imperative and functional* programs
- to be able to *construct proofs in a flexible way*
- similar (but different) systems: B-method, Agda, ...

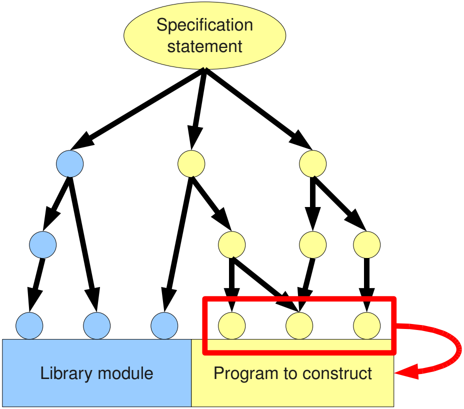
Verification



Correctness by construction



Correctness by construction



Refinement rules

- Sequence
- Selection (case distinction)

Refinement rules

- Sequence
- Selection (case distinction)
- Introduction and elimination of parameters
- Induction

Example: Peano numbers

```
data Nat = Z | S Nat

add :: Nat -> Nat -> Nat
add Z n = n
add (S n) m = S (add n m)
```

Example: Peano numbers

```
data Nat = Z | S Nat

add :: Nat -> Nat -> Nat
add Z n = n
add (S n) m = S (add n m)
```

- From the datatype declaration:

$$true \;\Rightarrow\; n = Z \;\vee\; \exists n' \,.\, n = S\; n'$$

Example: Peano numbers

```
data Nat = Z | S Nat

add :: Nat -> Nat -> Nat
add Z n = n
add (S n) m = S (add n m)
```

- From the datatype declaration:

$$true \;\Rightarrow\; n = Z \vee \exists n' \,.\, n = S\,n'$$

- From the definition of addition:

$$true \;\Rightarrow\; add\,Z\,n \;=\; n$$

$$true \;\Rightarrow\; add\,(S\,n)\,m \;=\; S\,(add\,n\,m)$$

Equality axioms

- Reflexivity:

$$\mathit{true} \;\Rightarrow\; n \;=\; n$$

- Replacement:

$$n = m \wedge f\ n \;\Rightarrow\; f\ m$$

Verification example

- property to verify: $true \Rightarrow add\ a\ Z = a$

Verification example

- property to verify: $\mathit{true} \Rightarrow \mathit{add}\ a\ Z = a$
- using the datatype axiom: $a = Z \lor \exists a' \,.\, a = S\ a'$

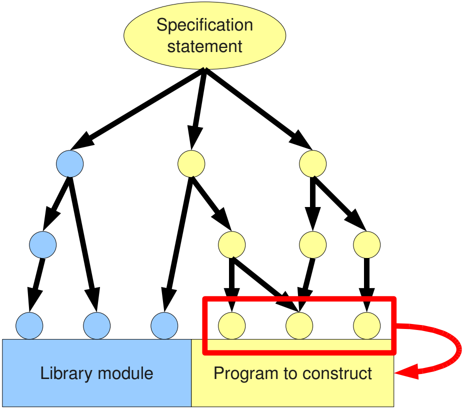
Verification example

- property to verify: $true \Rightarrow add\ a\ Z = a$
- using the datatype axiom: $a = Z \lor \exists a' . a = S\ a'$
- splitting the two cases:
  - $a = Z$
    - using the first axiom of *add*: $add\, Z\, Z = Z$
    - using a replacement: $add\, a\, Z = a$

Verification example

- property to verify: $\mathit{true} \Rightarrow \mathit{add}\ a\ Z = a$
- using the datatype axiom: $a = Z \vee \exists a' \,.\, a = S\ a'$
- splitting the two cases:
  - $a = Z$
    - using the first axiom of *add*: $\mathit{add}\, Z\, Z = Z$
    - using a replacement: $\mathit{add}\, a\, Z = a$
  - $\exists a' \,.\, a = S\ a'$
    - introducing the parameter a': $a = S\ a'$
    - using the second axiom of *add*: $\mathit{add}\,(S\ a')\, Z = S\,(\mathit{add}\ a'\ Z)$
    - using the inductive hypothesis: $\mathit{add}\ a'\ Z = a'$
    - using two replacements: $\mathit{add}\, a\, Z = a$

Recall: Correctness by construction



Function definition axiom

- Definition of function with two arguments:

$$true \Rightarrow f\ arg_1\ arg_2 = expr$$

- Whenever it is used, it generates a new equation into the program
- The system has to check:
  - Are we allowed to define f in the program?
  - Are the arguments valid patterns?
  - ...

Constructing the subtraction function

- Specification: $a = add\, b\, c \Rightarrow sub\, a\, b = c$

Constructing the subtraction function

- Specification: $a = add\,b\,c \Rightarrow sub\,a\,b = c$
- The proof structure is similar to the previous one.
- Two instantiations of the function definition axiom were used to complete the proof:
  - $true \Rightarrow sub\,a\,Z = a$
  - $true \Rightarrow sub\,(S\,a')\,(S\,b') = sub\,a'\,b'$

Constructing the subtraction function

- Specification: $a = add\, b\, c \Rightarrow sub\, a\, b = c$
- The proof structure is similar to the previous one.
- Two instantiations of the function definition axiom were used to complete the proof:
  - $true \Rightarrow sub\, a\, Z = a$
  - $true \Rightarrow sub\,(S\, a')\,(S\, b') = sub\, a'\, b'$
- These yield the following definition:

```
sub a Z = a
sub (S a') (S b') = sub a' b'
```

Embedding in Haskell

- Haskell datatypes for: expressions, formulas, proofs, ...
- The compiler (proof checker) works on these Haskell datatypes.
- A set of Haskell functions are defined to have handy syntax.

Embedding in Haskell

- Haskell datatypes for: expressions, formulas, proofs, ...
- The compiler (proof checker) works on these Haskell datatypes.
- A set of Haskell functions are defined to have handy syntax.
- Advantages:
  - no scanner and parser needed
  - easier to modify language definition while experimenting
  - all the power of Haskell is there to create tricky functions (tactics, proof strategies) that generate proofs

Embedding in Haskell

- Haskell datatypes for: expressions, formulas, proofs, ...
- The compiler (proof checker) works on these Haskell datatypes.
- A set of Haskell functions are defined to have handy syntax.
- Advantages:
  - no scanner and parser needed
  - easier to modify language definition while experimenting
  - all the power of Haskell is there to create tricky functions (tactics, proof strategies) that generate proofs
- Disadvantages:
  - syntax is slightly limited
  - error reporting is problematic (no line number info, etc.)

Example: Proof strategy

- Let's have the following axiom about the functions f and g:

$$\neg f \;\Rightarrow\; g$$

Example: Proof strategy

- Let's have the following axiom about the functions f and g:

$$\neg f \Rightarrow g$$

- How to prove $\neg g \Rightarrow f$?

Example: Proof strategy

- Let's have the following axiom about the functions f and g:

$$\neg f \Rightarrow g$$

- How to prove $\neg g \Rightarrow f$?
  - Case distinction on f:
    - if f holds then we are ready.
    - if $\neg f$ holds then we use the axiom above, and get $g \wedge \neg g$
    - from *false* we can prove everything, including f
- Every indirect proof can be done this way in this system:
  - We can create a function capturing this scheme!

Status & future work

- Previously a (standalone) language was implemented for imperative programs.
  - simple programs using *pointers* and *C++ STL* were proved
- Currently: a proof of concept implementation embedded in Haskell.
- Next tasks:
  - merge the features of the two implementations in the embedded version
  - clearly define the semantics for the construction of functional code
  - see how to use Haskell features in proof construction

Thank you for your attention!