Stepwise construction of simple Agda programs

Steps towards an interactive Agda programming tutor

Carlos Tomé Cortiñas Fabian Thorand
Ferdinand van Walree Renate Eilers

November 2nd, 2016

Department of Information and Computing Sciences, Utrecht University

Table of contents

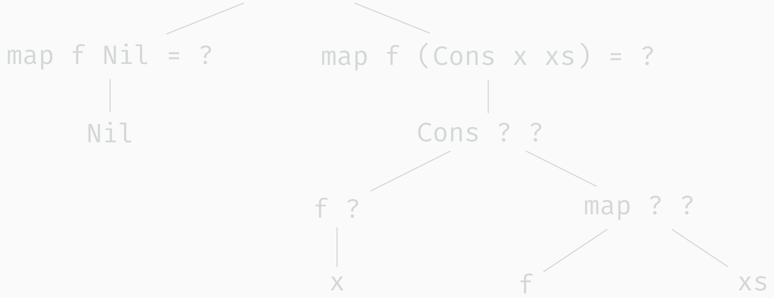
Introduction

Research question

What (sub-)class of semi-decidable programs (or, equivalently, proofs) can be automatically constructed in the programming language Agda through a series of mechanical steps that would normally be performed by a user writing a program, and how can we automatically extract these steps for programs within these classes?

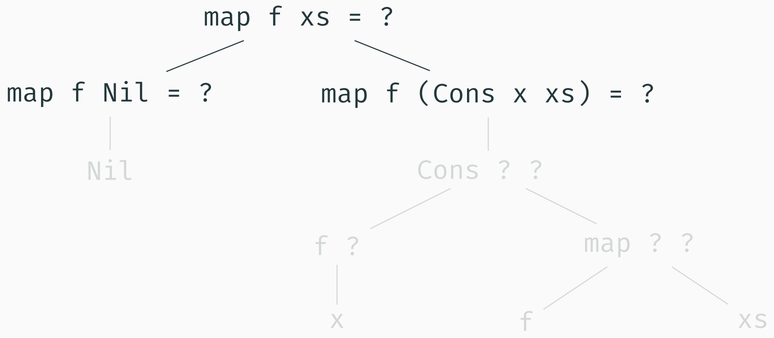
Example

```
map : ∀ {A B n} → (A → B)  → (Vec A n) → (Vec B n)
            map f xs = ?
```



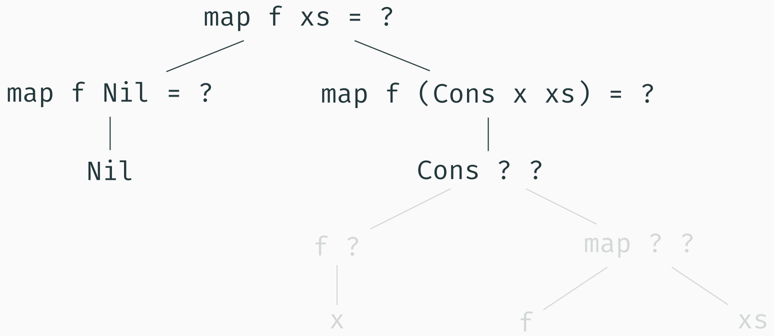
Example

```
map : ∀ {A B n} → (A → B)  → (Vec A n) → (Vec B n)
```



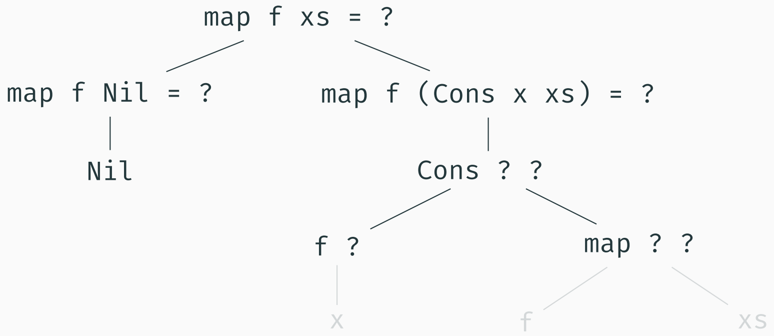
Example

```
map : ∀ {A B n} → (A → B)  → (Vec A n) → (Vec B n)
```



Example

```
map : ∀ {A B n} → (A → B)  → (Vec A n) → (Vec B n)
```



Example

```
map : ∀ {A B n} → (A → B) → (Vec A n) → (Vec B n)
```

$$\begin{tikzcd}
 & \texttt{map f xs = ?} \arrow[dl, no head] \arrow[dr, no head] & & & \\
\texttt{map f Nil = ?} \arrow[d, no head] & & \texttt{map f (Cons x xs) = ?} \arrow[d, no head] & & \\
\texttt{Nil} & & \texttt{Cons ? ?} \arrow[dl, no head] \arrow[dr, no head] & & \\
 & \texttt{f ?} \arrow[d, no head] & & \texttt{map ? ?} \arrow[dl, no head] \arrow[dr, no head] & \\
 & \texttt{x} & \texttt{f} & & \texttt{xs}
\end{tikzcd}$$

Approach

Strategy

Fill Hole

- First try to solve a hole using the proof search algorithm. [1]
- If that fails, invoke the case split strategy.

[1]Based on the paper *Auto in Agda* by Kokke and Swierstra
[2]We bound the depth so we can stop and backtrack

Strategy

Fill Hole

- First try to solve a hole using the proof search algorithm. [1]
- If that fails, invoke the case split strategy.

Case-Split

- Generates candidate variables for splitting based on their depth in patterns.
- Candidates are checked one at a time (split may fail).
- When the split succeeds, invokes the fill-hole strategy to recursively solve the newly generated clauses.[2]

[1] Based on the paper *Auto in Agda* by Kokke and Swierstra
[2] We bound the depth so we can stop and backtrack

Proof Search

- the goal type and the types of the definitions in scope are translated into Prolog-like terms
- a term with a given type is found by solving the corresponding Prolog query

Proof Search

- the goal type and the types of the definitions in scope are translated into Prolog-like terms
- a term with a given type is found by solving the corresponding Prolog query

```
nil  : Vec A zero
cons : A → Vec A n → Vec A (suc n)

foo : Vec Nat (suc zero)
foo = ?
```

Proof Search

- the goal type and the types of the definitions in scope are translated into Prolog-like terms
- a term with a given type is found by solving the corresponding Prolog query

```
nil  : Vec A zero
cons : A → Vec A n → Vec A (suc n)

foo : Vec Nat (suc zero)
foo = ?
```

nil rule: `Vec(`x_A`, zero).`

cons rule: `Vec(`x_A`, suc(`x_n`)) :- `x_A`, Vec(`x_A`, `x_n`).`

goal query: `?- Vec(Nat, suc(zero))`

Demonstration

Future Work

Future Work

- Improve case splitting by finding an effective and computationally cheap heuristic to select the variable.
  - Example: Analyze dependencies between variables and select the one with highest impact.
- Improve proof search to handle some restricted version of higher order unification
- Specify properties that help to prune the proof search state

Conclusion

- Using Agda as a library is hard (and unsupported).
- Expressiveness of dependent type system allows to automatically generate solutions for exercises, no model solutions required.