Under preparation for CPP 2015

Auto in Agda

Programming proof search

Pepijn Kokke Wouter Swierstra
Universiteit Utrecht
pepijn.kokke@gmail.com w.s.swierstra@uu.nl

Abstract

As proofs in type theory become increasingly complex, there is a growing need to provide better proof automation. This paper shows how to implement a Prolog-style resolution procedure in the dependently typed programming language Agda. Connecting this resolution procedure to Agda's reflection mechanism provides a first-class proof search tactic for first-order Agda terms. Furthermore, the same mechanism may be used in tandem with Agda's instance arguments to implement type classes in the style of Haskell. As a result, writing proof automation tactics need not be different from writing any other program.

1. Proof search in Agda

The following section describes our implementation of proof search à la Prolog in Agda. This implementation abstracts over two data types for names—one for inference rules and one for term constructors. These data types will be referred to as RuleName and TermName, and will be instantiated with types (with the same names) in section **??**.

Terms and unification

The heart of our proof search implementation is the structurally recursive unification algorithm described by McBride [1]. Here the type of terms is indexed by the number of variables a given term may contain. Doing so enables the formulation of the unification algorithm by structural induction on the number of free variables. For this to work, we will use the following definition of terms[1]:

```
data PsTerm (n : ℕ) : Set where
  var : Fin n → PsTerm n
  con : TermName → List (PsTerm n) → PsTerm n
```

In addition to a restricted set of variables, we will allow first-order constants encoded as a name with a list of arguments.

For instance, if we choose to instantiate PsName with the following Arith data type, we can encode numbers and simple arithmetic expressions:

```
data Arith : Set where
  Suc  : Arith
  Zero : Arith
  Add  : Arith
```

The closed term corresponding to the number one could be written as follows:

```
One : PsTerm 0
One = con Suc (con Zero [] :: [])
```

Similarly, we can use the var constructor to represent open terms, such as $x + 1$. We use the prefix operator # to convert from natural numbers to finite types:

```
AddOne : PsTerm 1
AddOne = con Add (var (# 0) :: con One [] :: [])
```

Note that this representation of terms is untyped. There is no check that enforces addition is provided precisely two arguments. Although we could add further type information to this effect, this introduces additional overhead without adding safety to the proof automation presented in this paper. For the sake of simplicity, we have therefore chosen to work with this untyped definition.

We shall refrain from further discussion of the unification algorithm itself. Instead, we restrict ourselves to presenting the interface that we will use:

```
unify : (t₁ t₂ : PsTerm m) → Maybe (∃ [n] Subst m n)
```

The unify function takes two terms t_1 and t_2 and tries to compute a sustitution—the most general unifier. Substitutions are indexed by two natural numbers m and n. A substitution of type Subst m n can be applied to a PsTerm m to produce a value of type PsTerm n. As unification may fail, the result is wrapped in the Maybe type. In addition, since the number of variables in the terms resulting from the unifying substitution is not known *a priori*, this number is existentially quantified over. For the remainder of the paper, we will write ∃ [x] B to mean a type B with occurrences of an existentially quantified variable x, or ∃ (λ x → B) in full.

Occasionally we will use a more general function unifyAcc, which takes a substitution as an additional parameter. It applies this substitution to t_1 and t_2 before attempting to unify.

```
unifyAcc : (t₁ t₂ : PsTerm m)
         → ∃ [n] Subst m n → Maybe (∃ [n] Subst m n)
```

Inference rules

We encode inference rules as records containing a rule name, a list of terms for its premises, and a term for its conclusion:

```
record Rule (n : ℕ) : Set where
  field
    name       : RuleName
```

[1] We will use the name PsTerm to stand for *proof search term* to differentiate them from Agda's *reflection terms*, or AgTerm.

```
    premises    : List (PsTerm n)
    conclusion  : PsTerm n
```

Once again the data-type is quantified over the number of variables used in the rule. Note that the variables are shared between the premises and the conclusion.

Using our newly defined Rule we can give a simple definition of addition. In Prolog, this would be written as follows.

```
add(0, X, X).
add(suc(X), Y, suc(Z)) :- add(X, Y, Z).
```

Unfortunately, the named equivalents in our Agda implementation are a bit more verbose. Note that we have, for the sake of this example, instantiated the RuleName and TermName to String and Arith respectively.

```
AddBase : Rule 1
AddBase = record {
  name       = "AddBase"
  conclusion = con Add ( con Zero []
                       :: var (# 0)
                       :: var (# 0)
                       :: [])
  premises   = []
  }

AddStep : Rule 3
AddStep = record {
  name       = "AddStep"
  conclusion = con Add ( con Suc (var (# 0) :: [])
                       :: var (# 1)
                       :: con Suc (var (# 2) :: [])
                       :: [])
  premises   = con Add ( var (# 0)
                       :: var (# 1)
                       :: var (# 2)
                       :: [])
               :: []
  }
```

Generalised injection and raising

Before we can implement some form of proof search, we define a pair of auxiliary functions. During proof resolution, we will need to work with terms and rules containing a different number of variables. We will use the following pair of functions, inject and raise, to weaken bound variables, that is, map values of type Fin n to some larger finite type.

```
inject : ∀ {m} n → Fin m → Fin (m + n)
inject n zero    = zero
inject n (suc i) = suc (inject n i)

raise : ∀ m {n} → Fin n → Fin (m + n)
raise zero    i = i
raise (suc m) i = suc (raise m i)
```

We have tried to visualize the behaviour of inject and raise, embedding Fin 3 into Fin (3 + 1) in Figure 1. On the surface, the inject function appears to be the identity. When you make all the implicit arguments explicit, however, you will see that it sends the zero constructor in Fin m to the zero constructor of type Fin (m + n). Hence, the inject function maps Fin m into the *first* m elements of the type Fin (m + n). Dually, the raise function maps Fin n into the *last* n elements of the type Fin (m + n) by repeatedly applying the suc constructor.

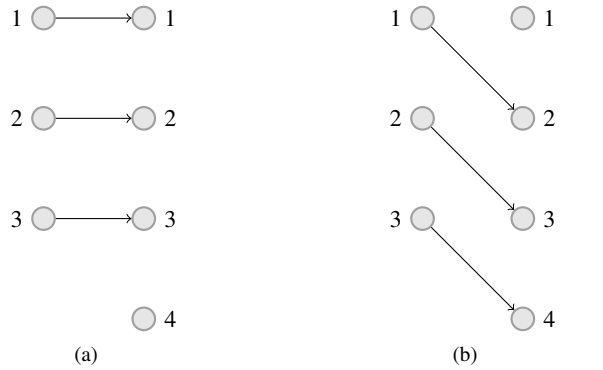

Figure 1. The graph of the inject function (a) and the raise function (b) embedding Fin 3 in Fin (3 + 1)

We can use these inject and raise to define similar functions that work on our Rule and PsTerm data types, by mapping them over all the variables that they contain.

Constructing the search tree

Our search tree is a potentially infinite rose tree.

```
data SearchTree (A : Set) : Set where
  leaf : A → SearchTree A
  node : List (∞ (SearchTree A)) → SearchTree A


data Proof : Set where
  con : (name : RuleName) (args : List Proof) → Proof

Proof′ : ℕ → Set
Proof′ m = ∃ [k] Vec (Goal m) k × (Vec Proof k → Proof)

con′ : (r : Rule n) → Vec Proof (arity r + k) → Vec Proof (suc k)
con′ r xs = new :: rest
  where
    new  = con (name r) (toList $ take (arity r) xs)
    rest = drop (arity r) xs

solve : Goal m → HintDB → SearchTree Proof
solve g rules = solveAcc (just (m, nil)) (1, g :: [], head)

solveAcc : Maybe (∃ [n] Subst (δ + m) n)
  → Proof′ (δ + m) → SearchTree Proof
solveAcc nothing _ = node []   -- fail
solveAcc (just (n, s)) (0, [], p) = leaf (p [])
solveAcc (just (n, s)) (suc k, g :: gs, p) = node (map step rules)

step : ∃ [δ′] Rule δ′ → ∞ (SearchTree Proof)
step (δ′, r) = ♯ solveAcc mgu prf
  where
    prf : Proof′ (δ′ + δ + m)
    prf = arity r + k, prm′ ++ gs′, p′
      where
        gs′ : Vec (Goal (δ′ + δ + m)) k
        gs′ = inject δ′ gs
        prm′ : Vec (Goal (δ′ + δ + m)) (arity r)
        prm′ = map (raise (δ + m)) (fromList (premises r))
        p′  : Vec Proof (arity r + k) → Proof
```

```
    p′  =  p ∘ con′ r
mgu : Maybe (∃ [n] (Subst (δ′ + δ + m) n))
mgu = unifyAcc g′ cnc′ s′
  where
    g′  : PsTerm (δ′ + δ + m)
    g′  = inject δ′ g
    cnc′ : PsTerm (δ′ + δ + m)
    cnc′ = raise (δ + m) (conclusion r)
    s′  : ∃ [n] Subst (δ′ + δ + m) n
    s′  = n + δ′, injectSubst δ′ s
```

Searching for proofs

Acknowledgements We would like to thank the Software Technology Reading Club at the Universiteit Utrecht for their helpful feedback.

References

[1] Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13:1061–1075, 11 2003. ISSN 1469-7653. doi: 10.1017/S0956796803004957. URL `http://journals.cambridge.org/article_S0956796803004957`.