Proof reusing

The case of Hindley Algorithm

D. Sotés, L. Lambán, J. Rubio

Departamento de Matemáticas y Computación
University of La Rioja (Spain)

November 16, 2007

Type inference

Hindley algorithm in functional programming

- Type inference: the ability to infer types of functions
- It is an important feature present in some statically typed functional programming languages (Haskell, ML,. . .)
- This leaves the programmer free to omit type annotations
- While maintaining type safety
- Hindley (a.k.a Hindley-Milner) algorithm is in the basis of type inference in most of modern functional programming languages

Goals

Verification in ACL2

- Goal (and ongoing work):
    - Formal verification of Hindley algorithm in ACL2
    - For the moment, the monomorphic case (polymorphic `let` excluded)
- By-product goal: analyze proof reusing of a previously done formalization of Robinson's unification algorithm

λ-calculus
Terms in λ-calculus

The terms of the simply typed λ-calculus

- **Variables** $x, y \ldots$
- **Applications** $[M\ N]$, where M and N are terms
- **Abstractions** $\lambda x.M$, where M is a term and x is a variable

λ-calculus
Type system

The type system consists of the following type expressions

- **Type variables** α, β, $\gamma \ldots$
- **"arrow"-types** $t_1 \rightarrow t_2$ (t_1 and t_2 type expressions)

Type expression examples:

α $\qquad\qquad$ $\alpha \rightarrow \beta$ $\qquad\qquad$ $(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$

λ-calculus

Type assignment

- Problem: to assign a type expression τ to a λ-term M (from a Γ basis of type assumtions)
  - Notation: $\Gamma \vdash M : \tau$
- The following **type assignement rules** describe the relation between terms and types:
  - $\Gamma \cup \{x : \tau\} \vdash x : \tau$
  - $\begin{array}{l} \Gamma \vdash M : \sigma \rightarrow \beta \\ \Gamma \vdash N : \sigma \end{array} \Rightarrow \Gamma \vdash [M\ N] : \beta$
  - $\Gamma \cup \{x : \sigma\} \vdash M : \tau \Rightarrow \Gamma \vdash \lambda x.M : \sigma \rightarrow \tau$
- For example:

$$\begin{array}{lll} & \text{Main type} & \text{Valid type} \\ \{x : \sigma\} \Longrightarrow & \vdash \lambda x.x : \sigma \rightarrow \sigma & (\beta \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \end{array}$$

Hindley Algorithm

Purpose:

- To find the most general valid type for a given λ-term

Input:

- M_0 (a λ-term)

Output:

- FAIL if M_0 is not typeable, else τ_0 (the main type of M_0)

Hindley Algorithm

$E = \emptyset$
$G = \{\emptyset \vdash M_0 : \alpha_0\}$
While $G \neq \emptyset$
  $g \leftarrow \Gamma \vdash M : \tau \in G$
  case g of:
    -$\Gamma \vdash x : \tau \Rightarrow E = E \cup \{\tau \simeq \Gamma(x)\}$
    -$\Gamma \vdash [M_1\ M_2] : \tau \Rightarrow G = G \cup \{\Gamma \vdash M_1 : \alpha \rightarrow \tau, \Gamma \vdash M_2 : \alpha\}$
    -$\Gamma \vdash \lambda x \bar{M} : \tau \Rightarrow E = E \cup \{\tau \simeq \alpha_1 \rightarrow \alpha_2\}$
      $G = G \cup \{\Gamma \vee \{x : \alpha_1\} \vdash \bar{M} : \alpha_2\}$
end while
$\Phi \leftarrow$ unify(E)
if $\Phi \equiv$FAILURE, return FAILURE
else, return $\alpha_0\Phi$

Hindley Algorithm
Example

$$M_0 = \lambda f.\lambda x.[f\ [f\ x]]$$

G	**E**
$\emptyset \vdash \lambda f.\lambda x.[f\ [f\ x]] : \alpha_0$	
$\{f : \alpha_1\} \vdash \lambda x.[f\ [f\ x]] : \alpha_2$	$\alpha_0 \simeq (\alpha_1 \rightarrow \alpha_2)$
$\{f : \alpha_1, x : \alpha_3\} \vdash [f\ [f\ x]] : \alpha_4$	$\alpha_2 \simeq (\alpha_3 \rightarrow \alpha_4)$
$\{f : \alpha_1, x : \alpha_3\} \vdash f : \alpha_5 \rightarrow \alpha_4$	$(\alpha_5 \rightarrow \alpha_4) \simeq \alpha_1$
$\{f : \alpha_1, x : \alpha_3\} \vdash [f\ x] : \alpha_5$	$(\alpha_5 \rightarrow \alpha_6) \simeq \alpha_1$
$\{f : \alpha_1, x : \alpha_3\} \vdash f : \alpha_5 \rightarrow \alpha_6$	$\alpha_6 \simeq \alpha_3$
$\{f : \alpha_1, x : \alpha_3\} \vdash x : \alpha_5$	

- $Unify(E) = \Phi = \{\alpha_0 \rightsquigarrow ((\alpha_6 \rightarrow \alpha_6) \rightarrow (\alpha_6 \rightarrow \alpha_6)), \alpha_2 \rightsquigarrow (\alpha_6 \rightarrow \alpha_6), \alpha_1 \rightsquigarrow (\alpha_6 \rightarrow \alpha_6), \alpha_4 \rightsquigarrow \alpha_6, \alpha_5 \rightsquigarrow \alpha_6, \alpha_3 \rightsquigarrow \alpha_6\}$
- $\bullet\lambda f.\lambda x.[f\ [f\ x]] : (\alpha_6 \rightarrow \alpha_6) \rightarrow (\alpha_6 \rightarrow \alpha_6)$

Main properties of Hindley algorithm

- Algorithm terminates on every input
- If it does not fail, the returned type is valid for the input
- In that case, the returned type is the most general valid type for the input
- If it fails, the input is not typeable

Derivations

Type Checking *vs* type inference

- The relation $\Gamma \vdash M : \alpha$ has to be formalized in ACL2 by expliciting the derivation by means of the corresponding applications of type inference rules (represented as a tree)
- Example of a derivation witnessing that $\vdash \lambda f.\lambda x.[f\ x] : (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$:

$$\lambda : (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$$

$$f : \alpha \rightarrow \beta \qquad \lambda : \alpha \rightarrow \beta$$

$$x : \alpha \qquad [\,] : \beta$$

$$f : \alpha \rightarrow \beta \qquad x : \alpha$$

Derivations

Derivation checking and inference in ACL2

- Type derivations are represented in ACL2 as lists encoding its tree structure
- It is easy to define an ACL2 function for checking if a derivation is coherent with the type inference rules
  - Function `derivation-check`
- Hindley algorithm can be reprogrammed in ACL2, in such a way that it builds not only a type, but a type derivation
  - Function `derivation-inference`

Soundness

Valid type

If it does not fail, The type returned by the algorithm is a correct type for the input:

```
(defthm inference-soundness
  (implies
     (and (lambda-termp x)
          (not (equal (derivation-inference x) 'FAIL)))
    (derivation-check (derivation-inference x))))
```

Soundness

Main type

The type returned by the algorithm is more general than any other correct type:

```
(defthm inference-principal-derivation
    (implies (and (lambda-termp x)
                  (derivation-check d)
                  (eq-l-terms x (l-term-extract d)))
       (derivation-subs d  (derivation-inference x))))
```

Completeness

If the algorithm returns failure, then there is no valid type for the input term

```
(defthm inference-completeness
   (implies (and (lambda-termp x)
                 (equal (derivation-inference x) 'FAIL)
                 (eq-l-terms x (l-term-extract d)))
            (not (derivation-check d))))
```

- Status of the work

Unification algorithm and proof reusing

- The final part of Hindley algorithm relies on Robinson unification algorithm
- Their properties are essential for the verification of Hindley algorithm
- This algorithm has already been verified in ACL2
- It is a good example of *proof reusing*
  - How to test the degree of reusability of a book?
  - If used as a “black box”, how the structure of the book used (local and non-local events) influences the proof effort of an external user? And its generality?
  - Is this a good measure for “book reusability”?
  - Are there good general design principles for this?