Polymorphic Algebraic Data Type Reconstruction

Tom Schrijvers, Maurice Bruynooghe

K.U.Leuven, Belgium
{toms,maurice}@cs.kuleuven.be

PPDP 2006 - Venice, Italy - June 10, 2006
Overview

Motivation
Type System
Approach
Properties
Extensions
Prototype Implementation
Conclusion

Tom Schrijvers, Maurice Bruynooghe
Polymorphic ADT Reconstruction
Motivation

Types are a boon!
- programmer documentation
- program analysis
- optimized compilation
- verification (type checking)

Types are a burden!
- types have to be typed
- encumbers rapid prototyping
Motivation

Type Inference

- Hindley-Milner algorithm
- relieves typing burden: type declarations inferred
- type definitions still required

Example

data List a = Nil | Cons a (List a)

append :: List a -> List a -> List a

append Nil l = l
append (Cons x xs) ys = Cons x (append xs ys)
Motivation

Type Definition Reconstruction

- type declarations inferred
- type definitions inferred

Example

```haskell
data List a = Nil | Cons a (List a)

append :: List a -> List a -> List a
append Nil l = l
append (Cons x xs) ys = Cons x (append xs ys)
```

Tom Schrijvers, Maurice Bruynooghe
Type Definitions

- polymorphic Algebraic Data Types (Haskell-style) e.g.
  \[
  \text{data } T \ a \ b = K \ x \mid L \ y
  \]

- constructor overloading (like Mercury) e.g.
  \[
  \text{data } \text{List } a = \text{Nil} \mid \text{Cons } a \ (\text{List } a) \\
  \text{data } \text{Stream } a = \text{Cons } a \ (\text{Stream } a)
  \]

- built-in type for functions: \textit{arrow} type \(a \to b\)
Type Judgements

Expression $e$ has type $\tau$ in environment $\Gamma$

$$\Gamma \vdash e : \tau$$

where $\Gamma$ captures

- ADT definitions
- function typings $f : \tau$
- variable typings $x : \tau$

respecting judgement rules, e.g.:

$$\frac{(\text{data } \tau = \ldots \mid K[\tau_i]_i \mid \ldots) \in \Gamma \quad \Gamma \vdash e_i : \tau'_i \quad \tau'_i = \tau_i\theta}{\Gamma \vdash K[e_i]_i : \tau\theta}$$
$\Gamma \vdash e : \Diamond$

asserts that $e$ is well-typed in $\Gamma$, according to judgement rules, e.g.

\[
\text{(Def)} \quad \frac{\Gamma \vdash e : \tau \quad f : \tau \in \Gamma}{\Gamma \vdash f = e : \Diamond}
\]

\[
\text{(Prog)} \quad \frac{\Gamma \vdash f_i = e_i : \Diamond \quad \Gamma \vdash e : \tau}{\Gamma \vdash [f_i = e_i]_i e : \Diamond}
\]
Overview

Motivation

Type System

Approach

Properties

Extensions

Prototype Implementation

Conclusion
Constraint Programming Approach

Judgement rules not *executable*, but imply constraints:

- **type inference**: constraints on types of expressions
- **type reconstruction**: constraints on type definitions

⇒ Apply Constraint Programming

1. determine variables: unknown typings and definitions in $\Gamma$
2. impose constraints: infer from program based on judgement rules
3. “solve” constraints: normalize
4. interpret solution: extract type expressions and definitions
Kinds of constraints:

- Type equality: \( \tau_1 = \tau_2 \)
  
e.g. \( \text{Bool} = \text{Bool} \)

- Polymorphic type instance: \( \tau_1 <: \tau_2 \)
  
e.g. \( \text{List Bool} <: \text{List } a \)

- Arrow type: \( \text{arrow}(\tau, \tau_1, \tau_2) \)
  
e.g. \( \text{arrow}(\text{List } a \rightarrow \text{Bool}, \text{List } a, \text{Bool}) \)

- ADT constructor: \( \tau \supseteq K \tau_1 \ldots \tau_n \)
  
e.g. \( \text{List } a \supseteq \text{Cons } a \)
Deriving constraints from the program, e.g.:

\[
\begin{align*}
\text{(CONS)} & \quad e_i : \tau_i \quad K[e_i] : \tau \\
& \quad \tau \supseteq K[\tau_i] \\
\text{(ABS)} & \quad x : \tau_1 \quad e : \tau_2 \quad \lambda x. e : \tau_3 \\
& \quad \text{arrow}(\tau_3, \tau_1, \tau_2) \\
\text{(CASE)} & \quad p_i : \tau_{1,i} \quad e_i : \tau_{2,i} \\
& \quad e : \tau_1 \quad (\text{case } e \text{ of } [p_i \rightarrow e_i]) : \tau_2 \\
& \quad \tau_{1,i} = \tau_1 \quad \tau_{2,i} = \tau_2 \\
& \quad \ldots
\end{align*}
\]
Logic-based Approach

1. Formulate *constraint theory*:
   - formulate axioms that define the constraints
   - of the form: $\forall \bar{x} : C_1 \Rightarrow \exists \bar{y} C_2$

2. Extract rewrite algorithm from axioms
   - set rewriting
   - set = constraint store = conjunction of constraints
   - rules: if $C_1$ then add $C_2$
   - implement type equality (=) as substitution
   - apply rewriting rules exhaustively
Example

1. Axiom:

\[ \forall \tau, K, \tau_i, \tau_i': \tau \supseteq K[\tau_i] \land \tau \supseteq K[\tau_i'] \Rightarrow [\tau_i = \tau_i'] \]

2. Rewrite Rule:

if \( \tau \supseteq K[\tau_i] \land \tau \supseteq K[\tau_i'] \) then add \([\tau_i = \tau_i']\)

See paper for

- all 16 axioms
- extended occurs check based on [Henglein 1993]
Interpret Solution

Turn final constraints (solved form) into:

- type definitions
- type declarations

Example

\[ \tau \supseteq \text{Nil} \land \tau \supseteq \text{Cons } \alpha \tau \]

becomes

```
data T a = Nil | Cons a (T a)
```
Overview

Motivation
Type System
Approach

Properties
Extensions
Prototype Implementation
Conclusion
Soundness

The algorithm’s result is a well-typing of $P$.

Completeness

In case of failure $P$ has no well-typing.

Intuition: rewriting preserves logical equivalence:

$$\forall C_1, C_2 : (C_1 \leftrightarrow C_2) \Rightarrow (|= C_1 \leftrightarrow C_2)$$
No principal typing!
when data constructor overloading
⇒ type inference ambiguity wrt. inferred type definitions

Example

- **Code**: \( f = A \quad g = A \)

- **Result**: 
  - `data T1 = A`
  - `data T2 = A`
  - `f :: T1`
  - `g :: T2`

- **Alternative maximal typings wrt ADT definitions**:
  - `f :: T1` \( \quad g :: T1 \) or `f :: T2` \( \quad g :: T1 \) or
  - `f :: T2` \( \quad g :: T2 \)
Other measures of maximal generality:

**Maximally Used ADTs**
The inferred typing is maximal and uses all inferred ADTs.

**Maximally Distinctly Typed Expressions**
No more expressions in $P$ can be given distinct types.

**Maximally Distinct ADTs**
No more ADTs can be used in a maximal well-typing.
Surprisingly General

```haskell
data List a = Nil | Cons a (List a)
data Stream a = Cons a (Stream a)

append :: List a -> Stream a -> Stream a

append Nil l = l
append (Cons x xs) ys = Cons x (append xs ys)
```
Theoretically no termination:

Polymorphic recursion is undecidable! [Henglein 1993]

In practice:
- Priority of rewriting rules required!
- No non-terminating program known
- Type declarations are not a solution: undecidable because type expression can be disproportionally bigger
Overview

1 Motivation
2 Type System
3 Approach
4 Properties
5 Extensions
6 Prototype Implementation
7 Conclusion
### Type System Variations

- predefined ADTs (like function type \( a \to b \))
- type declarations
- no constructor overloading
- monomorphic recursion

### Additional Language Features

- polymorphic let expressions
- don’t care patterns
- Logic Programming
Different syntax, same type constraints.

- port (pure) Prolog to Mercury
- support functional logic programming

Example: unification rule

\[(\text{UNIF}) \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau \]
\[\quad \Gamma \vdash t_1 = t_2 : \Diamond\]
Overview

Motivation
Type System
Approach
Properties
Extensions
Prototype Implementation
Conclusion

Tom Schrijvers, Maurice Bruynooghe
Polymorphic ADT Reconstruction
Front-Ends
- *Prolog*: infers types for port to Mercury
- *Haskell*

CHR Constraint Solver
- multi-set rewrite language
- embedded in Prolog (unification for free)
- trivial implementation: $C_1 \Rightarrow C_2$
- but lacks rule priorities
- remedied with simple hack... but causes $\sim O(n^3)$ complexity
Overview

1. Motivation
2. Type System
3. Approach
4. Properties
5. Extensions
6. Prototype Implementation
7. Conclusion

Tom Schrijvers, Maurice Bruynooghe

Polymorphic ADT Reconstruction
Conclusion

Contributions

- polymorphic ADT reconstruction algorithm
- many extensions
- constraint-based approach
- soundness, completeness, maximality
Some possibilities

- additional features
  - pre-defined *extensible* ADT definitions
  - existential types, GADTs
  - partial application
- program analysis (alias analysis, termination, ...)
- complexity and efficient implementation
  - add *user-defined* control to CHR (with L. De Koninck)

\[ CP = Logic + Control \]
Want to know more?

- *Towards Constraint-based Type Inference with Polymorphic Recursion for Functional and Logic Programs*, IFL 2005, Schrijvers & Bruynooghe

- *Inference of well-typing for logic programming with application to termination analysis*, SAS 2005, Bruynooghe, Gallagher & Van Humbeeck

- AMTypRe prototype available