Polymorphic Algebraic Data Type Reconstruction

Tom Schrijvers, Maurice Bruynooghe

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

Namur, Belgium - December 11-15, 2006
Overview

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

Motivation

Type System

Approach

Properties

Extensions

Prototype Implementation

Conclusion
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

```
:- type list(T) ---> [] ; [T | list(T)].

:- pred append(list(E),list(E),list(E)).

append([],L,L).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```
Motivation

Type Definition Reconstruction

- type declarations inferred
- type definitions inferred

Example

```prolog
:- type list(T) ---> []; [T | list(T)].

:- pred append(list(E), list(E), list(E)).

append([], L, L).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```
Previous Work

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

- monomorphic reconstruction
- for Prolog
Previous Work

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

- monomorphic reconstruction $\Rightarrow$ polymorphic
- for Prolog
Previous Work

_Inference of well-typing for logic programming with application to termination analysis_, SAS 2005, Bruynooghe, Gallagher & Van Humbeeck

- monomorphic reconstruction ⇒ polymorphic
- for Prolog ⇒ also for functional programming
Previous Work

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

- monomorphic reconstruction $\Rightarrow$ polymorphic
- for Prolog $\Rightarrow$ also for functional programming
- $\Rightarrow$ overall better understanding (implementation/variation)
Type Definitions

- polymorphic Algebraic Data Types e.g.
  
  ```prolog
  :- type maybe(X) --> yes(X) ; no.
  
  :- type bool --> true ; false.
  ```

- polymorphic instances e.g. maybe(bool)

- constructor overloading (like Mercury) e.g.
  
  ```prolog
  :- type list(E) --> nil ; cons(E,list(E)).
  
  :- type stream(E) --> cons(E,stream(E)).
  ```
Predicate Signatures
declares types of predicate arguments e.g.

:- pred append(list(E),list(E),list(E)).

Predicate Calls
polymorphic instance of signature e.g.

call: append(cons(true,nil),nil,L)

L : list(bool)
Expression $e$ has type $\tau$ in environment $\Gamma$

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

where $\Gamma$ captures

- ADT definitions
- predicate signatures $p(\overline{\tau})$
- variable typings $x : \tau$

respecting judgement rules, e.g.:

$$\frac{\vdash \text{type } \tau = \ldots ; k(\overline{\tau_i}) ; \ldots \in \Gamma}{\vdash \overline{e_i} : \tau_i'} \quad \tau_i' = \tau_i \theta}

$$

$$(\text{CONS}) \quad \overline{\vdash e_i : \tau_i'} \quad \tau_i' = \tau_i \theta}

\overline{\vdash k(\overline{e_i}) : \tau \theta}$$
\[ \Gamma \vdash e : \diamond \]

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

\[
\begin{align*}
\text{(TRUE) } & \Gamma \vdash \text{true} : \diamond \\
\text{(CALL) } & \frac{p(\tau_1, \ldots, \tau_n) \in \Gamma \quad \Gamma \vdash t_i : \tau'_i \quad \tau'_i = \tau_i \theta}{\Gamma \vdash p(t_1, \ldots, t_n) : \diamond}
\end{align*}
\]
Overview

1. Motivation
2. Type System
3. Approach
4. Properties
5. Extensions
6. Prototype Implementation
7. Conclusion
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 entities**: unknown types 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. \( \text{list(bool)} <: \text{list(E)} \)

- **ADT constructor**: \( \tau \supseteq k(\tau_1, \ldots, \tau_n) \)
  
  e.g. \( \text{list(E)} \supseteq \text{cons(E,list(E))} \)
Deriving constraints from the program, e.g.:

\[(\text{UNIF})\]
\[
\frac{t_1 : \tau_1 \quad t_2 : \tau_2 \quad t_1 = t_2 \in P}{\tau_1 = \tau_2}
\]

\[(\text{CONS})\]
\[
\frac{t_i : \tau_i \quad k(\bar{t}_i) : \tau}{\tau \supseteq k(\bar{\tau}_i)}
\]

...
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 (of 5):

$$\forall \tau, k, \tau_i, \tau'_i : \tau \supseteq k(\bar{\tau}_i) \land \tau \supseteq k(\bar{\tau}'_i) \Rightarrow \bigwedge_i \tau_i = \tau'_i$$
Example

1. Axiom (of 5):

\[ \forall \tau, k, \tau_i, \tau'_i : \tau \supseteq k(\bar{\tau}_i) \land \tau \supseteq k(\bar{\tau}'_i) \Rightarrow \bigwedge_i \tau_i = \tau'_i \]

2. Rewrite Rule:

if \( \tau \supseteq k(\bar{\tau}_i) \land \tau \supseteq k(\bar{\tau}'_i) \) then add \( \bigwedge_i \tau_i = \tau'_i \)
Constraint Handling Rules (CHR) implementation

Example

\[ \text{contains}(T,\text{Cons}1), \text{contains}(T,\text{Cons}2) \Rightarrow \]
\[ \text{functor}(\text{Cons}1,F,A), \]
\[ \text{functor}(\text{Cons}2,F,A) \]
\[ \mid \]
\[ \text{Cons}1 = \text{Cons}2. \]
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

\[- \text{type } t42(A) \longrightarrow \text{nil} ; \text{cons}(A, t42(A)). \]
Overview 21/39

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

Polymorphic ADT Reconstruction
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:

$$\forall C_1, C_2 : (C_1 \leftrightarrow C_2) \Rightarrow (\models C_1 \leftrightarrow C_2)$$
No principal typing!

when data constructor overloading

⇒ type inference ambiguity wrt. inferred type definitions

Example

- Code: p(a). q(a).

- Result:
  :- type t1 ---> a. :- type t2 ---> a.
  :- pred p(t1). :- pred q(t2).

- Alternative maximal typings wrt ADT definitions:
  :- pred p(t2). :- pred q(t1). or
  :- pred p(t1). :- pred q(t1). or
  :- pred p(t2). :- pred q(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.
Maximality Properties

Surprisingly General

:- type list(A) ---> nil ; cons(A,list(A)).
:- type stream(A) ---> cons(A,stream(A)).

:- pred append(list(A),stream(A),stream(A)).

append(nil,L,L).
append(cons(X,XS),Ys,cons(X,Zs)) :- append(Xs,Ys,Zs).
### Surprisingly General

```prolog
:- type list(A) ---> nil ; cons(A,list(A)).
:- type list2(A) ---> nil ; cons(A,list2(A)).

:- pred append(list(A),list2(A),list2(A)).

append(nil,L,L).
append(cons(X,XS),Ys,cons(X,Zs)) :- append(Xs,Ys,Zs).

:- pred p(list(B),list2(B)).
p(X,Y) :- append(X,nil,Y).
```
Maximality Properties

Surprisingly General

:- type list(A) ---> nil ; cons(A,list(A)).

:- pred append(list(A),list(A),list(A)).

append(nil,L,L).
append(cons(X,XS),Ys,cons(X,Zs)) :- append(Xs,Ys,Zs).

:- pred p(list(B),list(B)).

p(X,Y) :- append(X,Y,X).
Termination

- We get non-termination for recursion!
We get **non-termination** for recursion!

Theoretically **no termination**: Polymorphic recursion is undecidable! [Henglein 1993]
We get **non-termination** for recursion!

Theoretically **no termination**:

Polymorphic recursion is undecidable! [Henglein 1993]

**ad hoc solutions:**
- limited solver iterations
- monomorphic recursion
## Similar problem: \(\lambda\)-calculus type inference

**Algorithm A** [Henglein]
- on-line cycle detection (extended occurs check)
- as rewriting
- of arrow graph

*terminates* in practice:
- well-typing
- cycle (failure)
Our algorithm (for Prolog):

- on-line cycle detection (extended occurs check)
- as rewriting
- of constraints (1 auxiliary constraint + 3 axioms)

**terminates** in practice:

- well-typing
- (short-circuit cycle)
Variations on a Theme

Type System Variations

- predefined ADTs (like function type $a \rightarrow b$)
- type declarations
- no constructor overloading
- monomorphic recursion
- **functional programming**
Variations on a Theme

Functional Programming

- extra constraint $\text{arrow}(\tau, \tau_1, \tau_1)$, e.g.

  $\text{arrow}(\text{int} \rightarrow \text{bool}, \text{int}, \text{bool})$

- 5 axioms based on Henglein’s arrow graph algorithm

- 1 interaction axiom:

  $\forall \tau : \tau \supseteq \ldots \land \text{arrow}(\tau, \ldots, \ldots) \Rightarrow \text{fail}$
Functional Programming

- extra constraint $\text{arrow}(\tau, \tau_1, \tau_1)$, e.g.

  $$\text{arrow}(\text{int} \rightarrow \text{bool}, \text{int}, \text{bool})$$

- 5 axioms based on Henglein’s arrow graph algorithm

- 1 interaction axiom:

  $$\forall \tau : \tau \supseteq \ldots \land \text{arrow}(\tau, \ldots, \ldots) \Rightarrow \text{fail}$$

$\Rightarrow$ type reconstruction for Mercury, Haskell, ...
### Front-Ends

- **Prolog**: infers types for port to Mercury
- **Haskell** (under construction)

### CHR Constraint Solver

- multi-set rewrite language
- embedded in Prolog (unification for free)
- trivial implementation: $C_1 \Longrightarrow C_2$
- 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
Contributions

- polymorphic ADT reconstruction algorithm
- many extensions
- constraint-based approach
- soundness, completeness, maximality
Future Work

Some possibilities

▪ additional features
  ▪ non-uniform ADTs
    
    <pre>:- type seq(A) ---&gt; nil ; cons(A,seq(pair(A,A))).</pre>
  ▪ existential types, GADTs
▪ program analysis (alias analysis, termination, ...)
▪ complexity and efficient implementation
  ▪ termination
  ▪ static priorities for CHR (with L. De Koninck)
  ▪ low-level implementation
Want to know more?

- *Polymorphic Algebraic Data Type Reconstruction*, PPDP 2006, Schrijvers & Bruynooghe
- *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