

Polymorphic Algebraic Data Type Reconstruction

Tom Schrijvers, Maurice Bruynooghe

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

Namur, Belgium - December 11-15, 2006





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



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

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

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).
```

Type Definition Reconstruction

- ▶ type declarations **inferred**
- ▶ type definitions **inferred**

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).
```

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 \Rightarrow polymorphic
- ▶ for Prolog \Rightarrow 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)



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

Type Definitions

- ▶ polymorphic Algebraic Data Types e.g.

```
:- type maybe(X) ---> yes(X) ; no.
```

```
:- type bool ---> true ; false.
```

- ▶ polymorphic instances e.g. `maybe(bool)`
- ▶ constructor overloading (like Mercury) e.g.

```
:- 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)
```

Type Judgements

Expression e has type τ in environment Γ

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

where Γ captures

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

respecting judgement rules, e.g.:

$$\text{(CONS)} \frac{
 \begin{array}{l}
 (-: \text{type } \tau = \dots ; k(\bar{\tau}_i) ; \dots) \in \Gamma \\
 \Gamma \vdash e_i : \tau'_i \quad \tau'_i = \tau_i\theta
 \end{array}
 }{
 \Gamma \vdash k(\bar{e}_i) : \tau\theta
 }$$

Type Judgements - Well-Typing

$$\Gamma \vdash e : \diamond$$

asserts that e is *well-typed* in Γ , according to judgement rules, e.g.

$$\begin{array}{l}
 (\text{TRUE}) \quad \Gamma \vdash \text{true} : \diamond \\
 (\text{CALL}) \quad \frac{\rho(\tau_1, \dots, \tau_n) \in \Gamma \quad \Gamma \vdash t_i : \tau'_i \quad \tau'_i = \tau_i \theta}{\Gamma \vdash \rho(t_1, \dots, t_n) : \diamond}
 \end{array}$$



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

Constraint Programming Approach

14/39

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 Γ
- 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. `bool = bool`

- ▶ Polymorphic type instance: $\tau_1 <: \tau_2$

e.g. `list(bool) <: list(E)`

- ▶ ADT constructor: $\tau \supseteq k(\tau_1, \dots, \tau_n)$

e.g. `list(E) \supseteq cons(E, list(E))`

Deriving constraints from the program, e.g.:

$$\begin{array}{l}
 \text{(UNIF)} \quad \frac{t_1 : \tau_1 \quad t_2 : \tau_2 \quad t_1 = t_2 \in P}{\tau_1 = \tau_2} \\
 \text{(CONS)} \quad \frac{t_i : \tau_i \quad k(\bar{t}_i) : \tau}{\tau \supseteq k(\bar{t}_i)} \\
 \dots
 \end{array}$$

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):

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

Example

- 1 Axiom (of 5):

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

- 2 Rewrite Rule:

if $\tau \supseteq k(\bar{\tau}_i) \wedge \tau \supseteq k(\bar{\tau}'_i)$ then add $\bigwedge_i \tau_i = \tau'_i$

Constraint Handling Rules (CHR) implementation

Example

```
contains(T,Cons1), contains(T,Cons2) ==>
    functor(Cons1,F,A),
    functor(Cons2,F,A)
    |
    Cons1 = Cons2.
```

Turn final constraints (*solved form*) into:

- ▶ type definitions
- ▶ type declarations

Example

$$\tau \supseteq \text{nil} \wedge \tau \supseteq \text{cons}(\alpha, \tau)$$

becomes

```
:- type t42(A) ---> nil ; cons(A,t42(A)).
```



- 1 Motivation
- 2 Type System
- 3 Approach
- 4 Properties**
- 5 Extensions
- 6 Prototype Implementation
- 7 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:

$$\forall C_1, C_2 : (C_1 \rightsquigarrow C_2) \Rightarrow (\models C_1 \leftrightarrow C_2)$$

No principal typing!

when data constructor overloading

⇒ type inference ambiguity wrt. inferred type definitions

Example

- ▶ Code: `p(a). q(a).`
- ▶ Result:

<code>:- type t1 ---> a.</code>	<code>:- type t2 ---> a.</code>
<code>:- pred p(t1).</code>	<code>:- pred q(t2).</code>
- ▶ Alternative maximal typings wrt ADT definitions:

<code>:- pred p(t2).</code>	<code>:- pred q(t1).</code>	<code>or</code>
<code>:- pred p(t1).</code>	<code>:- pred q(t1).</code>	<code>or</code>
<code>:- pred p(t2).</code>	<code>:- pred q(t2).</code>	

Maximality Properties

24/39

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

```
:- 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

```
:- 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).
```

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).
```

- ▶ 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: λ -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)



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

Type System Variations

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

Functional Programming

- ▶ extra constraint $arrow(\tau, \tau_1, \tau_1)$, e.g.

$arrow(int \rightarrow bool, int, bool)$

- ▶ 5 axioms based on Henglein's arrow graph algorithm
- ▶ 1 interaction axiom:

$$\forall \tau : \tau \supseteq \dots \wedge arrow(\tau, \dots, \dots) \Rightarrow fail$$

Variations on a Theme

33/39

Functional Programming

- ▶ extra constraint $arrow(\tau, \tau_1, \tau_1)$, e.g.

$$arrow(int \rightarrow bool, int, bool)$$

- ▶ 5 axioms based on Henglein's arrow graph algorithm
- ▶ 1 interaction axiom:

$$\forall \tau : \tau \supseteq \dots \wedge arrow(\tau, \dots, \dots) \Rightarrow fail$$

\Rightarrow type reconstruction for Mercury, Haskell,...



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

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: $C_1 \implies C_2$
- ▶ but lacks rule priorities
- ▶ remedied with simple hack... but causes $\sim \mathcal{O}(n^3)$ complexity



- 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

Some possibilities

- ▶ additional features

- ▶ **non-uniform ADTs**

- ```
:- type seq(A) ---> nil ; cons(A, seq(pair(A,A))).
```

- ▶ 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