

From Monomorphic to Polymorphic Well-Typings and Beyond

Tom Schrijvers John Gallagher Maurice Bruynooghe

Catholic University of Leuven, Belgium
TOM.SCHRIJVERS@CS.KULEUVEN.BE

LOPSTR 2008 – July 17-18, 2008





- 1 Background & Motivation
- 2 SCC-based Type Analysis
- 3 Evaluation
- 4 Conclusion



- 1 Background & Motivation
- 2 SCC-based Type Analysis
- 3 Evaluation
- 4 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
- ▶ historically untyped: logic programming

Type definitions:

- ▶ Monomorphic type definitions
e.g. `:- type bool ---> true ; false.`
- ▶ Polymorphic type definitions
e.g. `:- type list(T) ---> [T|list(T)] ; [].`
- ▶ Instances of polymorphic types
e.g. `list(bool), list(list(bool)), list(list(T)), ...`

like in Mercury and FP (ML, Haskell)

Type signatures for predicates:

```
:- pred app(list(T),list(T),list(T)).
```

```
app([],L,L).
```

```
app([X|Xs],Ys,[X|Zs]) :-  
    app(Xs,Ys,Zs).
```

- ▶ 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 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).
```

MONO

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

- ▶ monomorphic reconstruction:
no instantiation of polymorphic types
- ▶ for Prolog

Call and Definition share same type:

```
p(R) :- app([a],[b],M), app([M],[M],R).
app([],L,L).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

:- type list ---> [] ; a ; b ; [list | list].

:- pred app(list,list,list).
:- pred p(list).
```

POLY

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

- ▶ polymorphic reconstruction:
instantiation of polymorphic types
- ▶ for Prolog (and FP languages)

Previously (2/2): Polymorphic Analysis

12/34

Call type is instance of Definition type:

```
p(R) :- app([a],[b],M), app([M],[M],R).
app([],L,L).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

:- type elem ---> a ; b.
:- type list1(T) ---> [] ; [T | list1(T)].
:- type list2(T) ---> [] ; [T | list2(T)].

:- pred app(list1(T),list2(T),list2(T)).
:- pred p(list2(list2(elem))).
:- call app1(list1(elem), list2(elem), list2(elem)).
:- call app2(list1(list2(elem)),
             list2(list2(elem)), list2(list2(elem))).
```

	MONO	POLY
constraints	$\tau_1 = \tau_2$ $\tau \supseteq f(\tau_1, \dots, \tau_n)$	$\tau_1 = \tau_2$ $\tau \supseteq f(\tau_1, \dots, \tau_n)$ $\tau_1 <: \tau_2$
RW algorithm	simple	complex
accuracy	weak	strong
cost	$\mathcal{O}(n)$	$\sim \mathcal{O}(n^3)$

We want a compromise!



- 1 Background & Motivation
- 2 SCC-based Type Analysis**
- 3 Evaluation
- 4 Conclusion

Idea: mimick POLY with MONO

- 1 duplicate definition for each call
- 2 MONO infer a signature for each copy

Step 1: Duplicate Definition

16/34

```
p(R) :-
    app1([a],[b],M),
    app2([M],[M],R).

app1([],L,L).
app1([X|Xs],Ys,[X|Zs]) :- app1(Xs,Ys,Zs).

app2([],L,L).
app2([X|Xs],Ys,[X|Zs]) :- app2(Xs,Ys,Zs).
```

Step 2: MONO on Copies

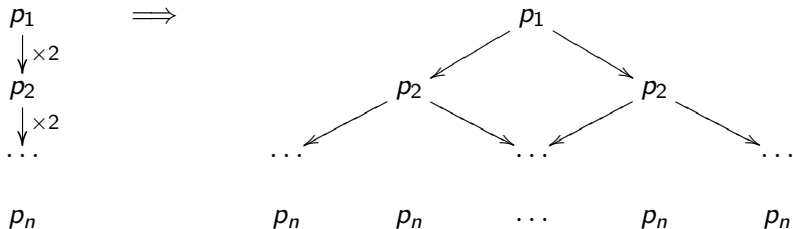
17/34

```

:- type elem ---> a ; b.
:- type elist1 ---> [elem|elist1] ; [].
:- type elist2 ---> [elem|elist2] ; [].
:- type elistlist1 ---> [elist2|elistlist1] ; [].
:- type elistlist2 ---> [elist2|elistlist2] ; [].

:- pred p(elistlist2).
:- pred app1(elist1, elist2, elist2).
:- pred app2(elistlist1, elistlist2, elistlist2).
  
```

Problematic complexity: worst case $O(2^n)$



Incremental Approach:

- ▶ Avoid copying subtrees
- ▶ Analyse definition first
- ▶ Copy and *Extend* definition signature for each call

Proceed SCC by SCC: SCC Analysis

Step 1: Analyse Definition First

20/34

```
app([]).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

:- type list1(T) ---> [T|list1(T)] ; [].
:- type list2(T) ---> [T|list2(T)].

:- pred app(list1(T),list2(T),list2(T)).
```

Step 2: Copy and Extend Signature for Calls

21/34

```

p(R) :-
    app([a],[b],M),
    app([M],[M],R).
:- type elem ---> a ; b.

:- type list1_1 ---> [elem|list1_1] ; [].
:- type list2_1 ---> [elem|list2_1] ; [].
:- call app(list1_1,list2_1,list2_1).

:- type list1_2 ---> [list1_1|list1_2] ; [].
:- type list2_2 ---> [list2_1|list2_2] ; [].
:- call app(list1_2,list2_2,list2_2).

:- pred p(list2_2).
  
```

$$\mathcal{O} \left(m * \sum_{i=1}^k i \right)$$

k : number of SCCs

m : number of calls per SCC

$$\mathcal{O}(n^2)$$

n : program size = $m * k$



- 1 Background & Motivation
- 2 SCC-based Type Analysis
- 3 Evaluation**
- 4 Conclusion

- ▶ 45 small logic programs
- ▶ 4–44 lines of code
- ▶ $\text{SCC} = 1\text{--}3 \times \text{MONO}$
- ▶ $\text{POLY} = 10\text{--}100 \times \text{MONO}$

Benchmark Suite (2/2)

25/34

Program	MONO	SCC	POLY
ackermann	0.32	128.93 %	1066.04 %
append	0.12	258.33 %	2325.00 %
delete	0.78	155.13 %	14230.77 %
der	1.56	115.38 %	3846.15 %
factor	0.44	205.88 %	29411.76 %
flat	0.24	125.00 %	4583.33 %
flatlength	0.30	200.00 %	7000.00 %
frontier	0.40	178.39 %	15075.38 %
g	0.78	155.13 %	15384.62 %
in	0.42	166.67 %	16904.76 %
inorder	0.32	158.39 %	18633.54 %
insert	0.52	156.37 %	11583.01 %
length	0.12	166.67 %	8333.33 %
length1	0.22	181.82 %	5000.00 %

```

app([],L,L).
app([X|Xs],Ys,[X|Zs]) :-
    app(Xs,Ys,Zs).

r(R) :-
    app([a],[b],M1),
    app([M1],[M1],M2),
    ...,
    app([Mn],[Mn],R).

:- pred r(listn+1(elem)).
    
```

Scalable Append (2/2)

27/34

Program	MONO	SCC	POLY
app-1	0.38 ms	0.65 ms	12 ms
app-10	1.20 ms	2.14 ms	206 ms
app-100	9.60 ms	18.10 ms	88,043 ms
app-1000	115.80 ms	238.05 ms	T/O
app-10000	1,402.40 ms	2,955.95 ms	T/O

MONO linear

SCC linear

POLY cubic

Worst Case Benchmark (1/2)

28/34

```
list([]).
list([X|Xs]) :- list(Xs).

p0(L) :- L = [a], list(L).
p1(L) :- p0(A), L = [A], list(L).
...
pn(L) :- pn-1(A), L = [A], list(L).

:- pred pn(listn+1(elem)).
```




- 1 Background & Motivation
- 2 SCC-based Type Analysis
- 3 Evaluation
- 4 Conclusion**

- ▶ success typing (abstract interpretation)
 - ▶ depends on control flow
 - ▶ approximates success set
- ▶ polymorphic variants [Garrigue'98]
 - ▶ OCaml
 - ▶ extend type with more constructors

- ▶ new SCC-based type analysis
- ▶ more efficient than polymorphic analysis
- ▶ scales linearly in practice
- ▶ more informative than linear analysis
- ▶ easy to pinpoint bugs

Further evaluation and application:

- ▶ big Mercury programs
- ▶ type-based norms for termination analysis
- ▶ debugging problems

Thank you!

34/34

Questions?

