

Higher-Order Abstract Syntax and Higher-Order Term Rewriting for Program Transformation

Jan Heering

Department of Software Engineering

CWI

Amsterdam

`Jan.Heering@cwi.nl`

`www.cwi.nl/~jan`

March 1, 2004

Table of contents

- Applications of program transformation
- Program transformation by first-order term rewriting
- Higher-order term rewriting
- Higher-order abstract syntax
- Systems: HOPS, MAG, and Phobos
- Literature

Applications of program transformation

- Optimization of time/space use
 - well-defined metric
 - behavior preserving
 - result not for human consumption
 - classic and important subject
- Optimization of program structure (restructuring/refactoring)
 - metric often implicit and not well-defined
 - behavior preserving
 - layout preservation/prettyprinting
 - very fashionable!

Applications of program transformation (cont.)

- Program evolution
 - **not** behavior preserving
 - semantics aware editing rather than unsafe edit scripts
 - layout preservation/prettyprinting
 - new subject driven by SE concerns
- Transformational program synthesis
 - program development methodology
 - wide-spectrum languages (SETL, CIP-L, COLD, ...)
 - fashionable in the 1980s
 - optimistic view: Bob Paige in *HOSC* **16** (2003) 7–13

Program transformation by f-o term rewriting

Program represented as abstract-syntax tree/prefix expression/term

Abstract syntax straightforward or “deep”

(Conditional) rewrite rule

$\text{conditions} \Rightarrow \text{pattern} \rightarrow \text{replacement}$

- **pattern** and **replacement** are terms with free f-o variables
(no function variables)
- f-o pattern matching is unitary
- **conditions** optional
- no “new” free variables allowed in replacement
if rule is unconditional $FV(\text{pattern}) \supseteq FV(\text{replacement})$

Some problems with f-o term rewriting (I)

f-o term rewriting similar to f-o functional programming

Advantage: simple mechanism

Disadvantage: limited expressiveness for program transformation

No rules like

$$\text{map}(F, \text{map}(G, L)) \rightarrow \text{map}(\text{compose}(F, G), L)$$

$$\text{if}(B, H(X), H(Y)) \rightarrow H(\text{if}(B, X, Y))$$

with h-o variables F, G, H

Some problems with f-o term rewriting (II)

No (implicit) context traversal like in

$$\begin{aligned} & \text{program}(\text{decls}(D_1, \text{decl}(Id, Type), D_2), \text{Stats}(Id)) \rightarrow \\ & \text{program}(\text{decls}(D_1, \text{decl}(Id, Type), D_2), \text{Stats}(\text{type2id}(Type))) \end{aligned}$$

with h-o variable *Stats*

Extensions to f-o

- traversal strategies (Stratego)
- traversal functions (ASF+SDF)

Some problems with f-o term rewriting (III)

Conditions

- **logical** (inherent in underlying semantics)
- **strategic** (control of rewriting process)

Specify logic first, add control later, **but** (*pace* Bob Kowalski)

- logic and control often **hard to distinguish**
- control **hard to express/factor out**

Strategic f-o rewriting approach (Stratego)

Not specific to f-o — equally useful in h-o rewriting case!

Summary

feature	(pure) f-o	h-o
function variables	—	+
context traversals	—	+
strategies	—	—
h-o abstract syntax (to be discussed)	—	+

but h-o rewriting is **much more complicated**

General issue: transfer of logic framework technology to program transformation systems (tactics/strategies, h-o abstract syntax, h-o term rewriting, ...)

Higher-order term rewriting

Many different definitions — from limited to general

Some limited forms of h-o

- **F-matching** (fold/unfold, John Darlington, 1970s)
- **applicative rewrite systems** (straightforward extension of f-o)
- **second-order** (Gérard Huet and Bernard Lang, 1978)
- **pattern rewrite systems** (Isabelle logic framework)

Liberal position in this talk in the vein of David Wolfram's book

Take weird phenomena for granted

H-O rewrite rule

conditions \Rightarrow pattern \rightarrow replacement

- **pattern** and **replacement** are **typed λ -terms** with free variables
- h-o pattern matching in general **not unitary**
- **conditions** optional
- no “new” free variables allowed in replacement
if rule is unconditional $FV(\text{pattern}) \supseteq FV(\text{replacement})$ **is not enough (variable containment problem)**
- abstract syntax representation of program as before
(for the time being)

Example (I)

h-o pattern matching **not unitary**

h-o signature

sorts $s, bool$

functions

$a : s$

$f, g : s \rightarrow s$

$if : bool \times s \times s \rightarrow s$

variables

$X, Y : s$

$H : s \rightarrow s$ (second-order variable)

$B, B' : bool$

Example (I) (cont.)

Rule

$$if(B, H(X), H(Y)) \rightarrow H(if(B, X, Y))$$

matches

$$if(B', g(f(a)), g(f(f(a))))$$

in three different ways

$$H = \lambda V.g(f(V)) \quad X = a \quad Y = f(a) \quad B = B'$$

$$H = \lambda V.g(V) \quad X = f(a) \quad Y = f(f(a)) \quad B = B'$$

$$H = \lambda V.V \quad X = g(f(a)) \quad Y = g(f(f(a))) \quad B = B'$$

First match is probably the preferred one

Last match may give rise to infinite loop

Matching strategies for single redex needed in “h-o Stratego”

Example (I) (cont.)

$$if(B', a, a)$$

yields matches

$$H = \lambda V.a \quad X = X \quad Y = Y \quad B = B'$$

$$H = \lambda V.V \quad X = a \quad Y = a \quad B = B'.$$

First match leaves X and Y uninstantiated

Both variables are eliminated by β -reduction after substitution in the replacement part

$$if(B', a, a) \rightarrow (\lambda V.a)(if(B', X, Y)) \xrightarrow{(\beta)} a$$

Happy coincidence

Example (II)

$$\text{compose}(F, G) \rightarrow \lambda V. F(G(V))$$

$$\text{map}(F, \text{nil}) \rightarrow \text{nil}$$

$$\text{map}(F, \text{cons}(X, L)) \rightarrow \text{cons}(F(X), \text{map}(F, L))$$

$$\text{map}(\lambda V. V, L) \rightarrow L$$

$$\text{map}(F, \text{map}(G, L)) \rightarrow \text{map}(\text{compose}(F, G), L)$$

First three are usual definitions in functional language

Last two are **not constructor cases** — problem in functional language

h-o term rewriting as **extension of functional language**

Example (III)

Variable containment problem

Condition $FV(\text{pattern}) \supseteq FV(\text{replacement})$ is not enough

Earlier *if*-example harmless, but ...

$$f(g(F(X), F(a))) \rightarrow f(X)$$

matches

$$f(g(a, a))$$

in two ways

$$F = \lambda V.V \quad X = a$$

$$F = \lambda V.a \quad X = X$$

Last match yields result $f(X)$ with free variable X

Stefan Kahrs, *HOA '95*, LNCS, Vol. 1074, 109–123

Example (IV)

Implicit context traversal

Higher-order matching can take care of subterm lookup

Extend rule $p \rightarrow r$ to

$$H(p) \rightarrow H(r)$$

with polymorphic “context variable” H not free in p and r

Reject useless instantiation of H to $\lambda V.s$ where V not free in s by suitable loop check

Can be used to experiment with h-o rewriting in [lambda Prolog](#)

Experiments in lambda Prolog

h-o matching is special case of h-o unification

h-o unification basic mechanism of lambda Prolog

Extension of Prolog to typed λ -terms

Prolog	lambda Prolog
Herbrand base	closed polymorphically typed λ -terms
free	α -, β -, and η -rule
f-o unification	polymorphic h-o unification
Horn clauses	higher-order hereditary Harrop formulas (whatever they may be ...)

Implementations: (old) LP2.7 interpreter, Terzo interpreter, Prolog/Mali compiler, Teyjus compiler, ...

Rules to be translated to lambda Prolog

$$\mathit{add}(X, \mathit{zero}) \rightarrow X$$

$$\mathit{add}(X, \mathit{succ}(Y)) \rightarrow \mathit{succ}(\mathit{add}(X, Y))$$

$$\mathit{if}(t, X, Y) \rightarrow X$$

$$\mathit{if}(f, X, Y) \rightarrow Y$$

$$\mathit{if}(B, F(X), F(Y)) \rightarrow F(\mathit{if}(B, X, Y))$$

$$\mathit{compose}(F, G) \rightarrow \lambda X.F(G(X))$$

$$\mathit{map}(F, \mathit{nil}) \rightarrow \mathit{nil}$$

$$\mathit{map}(F, \mathit{cons}(X, L)) \rightarrow \mathit{cons}(F(X), \mathit{map}(F, L))$$

$$\mathit{map}(\lambda V.V, L) \rightarrow L$$

$$\mathit{map}(F, \mathit{map}(G, L)) \rightarrow \mathit{map}(\mathit{compose}(F, G), L)$$

Translation to lambda Prolog — signature

```
kind nat      type.
kind bool     type.
kind lst      type -> type.

type zero     nat.
type succ     nat -> nat.
type add      nat -> nat -> nat.
type t        bool.
type f        bool.
type if       bool -> A -> A -> A.
type nil      (lst A).
type cons     A -> (lst A) -> (lst A).
type map      (A -> B) -> (lst A) -> (lst B).
type compose  (B -> C) -> (A -> B) -> A -> C.
```

Translation to lambda Prolog — rules

```
type reduce  A -> A -> o.
```

```
type extrule A -> A -> o.
```

```
extrule (H (add X zero))      (H X).
```

```
extrule (H (add X (succ Y))) (H (succ (add X Y))).
```

```
extrule (H (if t X Y))       (H X).
```

```
extrule (H (if f X Y))       (H Y).
```

```
extrule (H (if B (F X) (F Y))) (H (F (if B X Y))).
```

```
extrule (H (compose F G))    (H (X \ (F (G X)))).
```

```
extrule (H (map F nil))      (H nil).
```

```
extrule (H (map F (cons X L))) (H (cons (F X) (map F L))).
```

```
extrule (H (map X \ X L))    (H L).
```

```
extrule (H (map F (map G L))) (H (map (compose F G) L)).
```

Translation to lambda Prolog — fixed part

```
reduce X Y :- extrule X Z,  
              not(X = Z), %%% loop check - X,Z closed  
              reduce Z Y.
```

```
reduce X X.
```

Redex selection and matching strategies determined by

- order of translated rules
- β -reduction performed implicitly during h-o unification
- lambda Prolog's h-o unification strategy
projection vs. imitation — limited control

if transformation examples

?- reduce (if y (cons f nil) (cons t nil)) NF.

NF = cons (if y f t) nil .

with **generic constant** y acting as **universally quantified variable**

?- reduce (if y (add (succ zero) (succ zero))
 (succ (succ zero))) NF.

NF = succ (succ zero) .

?- reduce (if y (if y1 x0 x1) (if y1 x2 x1)) NF.

NF = if y1 (if y x0 x2) x1 .

with generic constants x_i, y_i

map transformation examples

```
?- reduce (map (X \ (compose succ X)) (cons succ nil)) NF.
```

```
NF = cons Var1612 \ (succ (succ Var1612)) nil .
```

```
?- reduce (map (X \ zero) (map succ lst)) NF.
```

```
NF = map (Var347 \ zero) 1 .
```

with generic constant lst

Rewriting under abstraction

?- reduce (Y \ (add Y zero)) NF.

NF = Y \ (add Y zero) .

No rewriting under abstraction

In accordance with usual functional programming practice, but
important for program transformation and h-o abstract syntax

Can be added to translation scheme using built-in `pi` predicate
(not shown)

h-o abstract syntax

h-o rewriting is **rewriting with bound variables**

Typed λ -calculus used **for substitution**,
not as general computation mechanism

Simply typed λ -calculus is **strongly terminating**

Similar to lambda Prolog (Dale Miller — FAQ page):

While it is the case that lambda Prolog contains λ -terms and such terms can be used to represent functional programs, the terms in lambda Prolog are simply typed and do not contain recursion or conditional. While it is possible to code up some functional computations using lambda Prolog's built-in beta-conversions, it would be hard to imagine doing general computations at that level.

h-o abstract syntax (cont.)

Use of typed λ -terms to represent name binding in abstract syntax of logical formulas or programs

- Represent bound names by bound variables in typed λ -terms
- Built-in avoidance of name capture
- Uniform mechanism available to all client formalisms
but perhaps insufficiently flexible

Used in

- (some) logic frameworks
- Teyjus (and other?) implementation(s) of lambda Prolog
- Phobos language development and transformation system
uses MetaPRL logic framework

h-o abstract syntax — example

(After Pfenning & Elliott)

$$H(\text{if}(B, X, Y)) \rightarrow \text{if}(B, H(X), H(Y))$$

With first order-abstract syntax

$$\text{let } b = t \text{ in } \text{if}(b, 1, 2) \rightarrow \text{if}(b, \text{let } b = t \text{ in } 1, \text{let } b = t \text{ in } 2)$$

Use h-o abstract syntax for $\text{let } V_1 = E_1, \dots, V_n = E_n \text{ in } E$

$$\text{let}((\lambda V_1 \dots \lambda V_n. E), \langle E_1, \dots, E_n \rangle)$$

example (cont.)

let b = t in if (b, 1, 2) now represented as

let((λV .if (V, 1, 2)), < t >)

Try *if*-rule again — potential match

$H = \lambda U.$ *let((λV .U), < t >)*

$B = V$ (bound variable)

$X = 1$

$Y = 2$

rejected because of name capture

Systems

- **HOPS Higher Object Programming System**
second-order term graph rewriting
- **MAG program transformation system for Haskell subset**
suitably tamed form of higher-order matching
- **Phobos**
MetaPRL logic framework used as front-end for Mojave
extensible compiler

Outlook

Is h-o worthwhile for program transformation?

Remains to be seen

- It is complicated
- It has to be tamed
- Only very small scale experience so far
(in particular MAG — talk by Ganesh Sittampalam)
- Strong competition from existing f-o systems
(ASF+SDF with traversal functions, Stratego, TXL, ...)

But it is very interesting!

Literature

- Mark van den Brand, Paul Klint, and Jurgen Vinju, [Term rewriting with traversal functions](#), *ACM TOSEM* **12** (2003) 152–190
- Amy Felty, [A logic programming approach to implementing higher-order term rewriting](#), in: LNAI, Vol. 596, 1991, 135–161, <http://www.site.uottawa.ca/~afelty/bib.html>
- Adam Granicz and Jason Hickey, [Phobos: A front-end approach to extensible compilers](#), in: *Proceedings HICSS-36*, 2003, <http://mojave.caltech.edu/publications.php>
- Jan Heering, [Implementing higher-order algebraic specifications](#), in: *Proceedings of the 1992 Workshop on the lambda Prolog Programming Language*, <http://www.cwi.nl/~jan/HO.WLP.ps>

Literature (cont.)

- Wolfram Kahl, [HOPS — The Higher Object Programming System](http://www.cas.mcmaster.ca/~kahl/HOPS/), <http://www.cas.mcmaster.ca/~kahl/HOPS/>
- Dale Miller, [lambda Prolog home page](http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/index.html), <http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/index.html>
- Gopalan Nadathur, [The Teyjus implementation of lambda Prolog](http://teyjus.cs.umn.edu/index.html), <http://teyjus.cs.umn.edu/index.html>
- Frank Pfenning and Conal Elliott, [Higher-order abstract syntax](#), *ACM SIGPLAN Notices* **23** 7 (July 1988) 199–208
- Ganesh Sittampalam, [MAG — A little program transformation system for a subset of Haskell](http://web.comlab.ox.ac.uk/oucl/research/areas/progtools/mag/index.html), <http://web.comlab.ox.ac.uk/oucl/research/areas/progtools/mag/index.html>

Literature (cont.)

- Eelco Visser, [Stratego home page](http://www.stratego-language.org/),
`http://www.stratego-language.org/`
- David Wolfram, [The Clausal Theory of Types](#), Cambridge University Press, 1993