Compiling Collapsing Rules in Certain Constructor Systems

Sergio Antoy
Portland State University
Joint work with Andy Jost

LOPSTR 2015 – Siena, Italy
Thanks NSF CCF-1317249

Outline

- The virtues of FLP
- Graph Rewriting
- The Problem
- The Solution
- Discussion

The virtues of FLP

Coloring a map of the PNW

```
data State = WA | OR | ID | BC
states = [WA,OR,ID,BC]
adjacent = [(WA,OR),(WA,ID),...]
data Color = Red | Green | Blue
color x = (x, Red ? Green ? Blue)

main = solve (map color states) adjacent
solve (_++[(s1,c)]++_++[(s2,c)]++_)
      (_++[(s1,s2)]++_) = failed
default solve x _ = x
```

Not current Curry, but according to proposed extensions.

Implementation

A functional logic program is transformed into a graph rewriting system which is executed by rewriting.

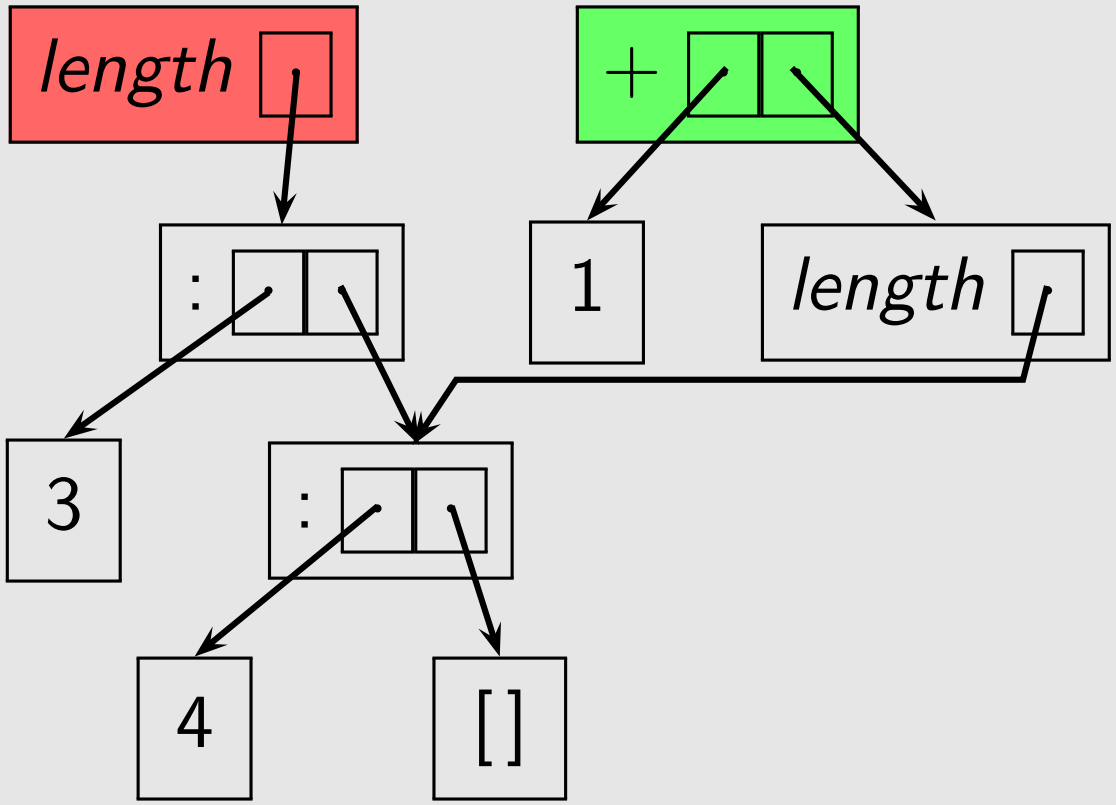
Graphs are represented by nodes (with attribues such as a label) and edges (pointers from nodes to nodes).

A rewrite step replaces a subgraph rooted by r with another subgraph rooted by c.

For that, every pointer to r must be redirected to c.
This is the focus of our work.

Example

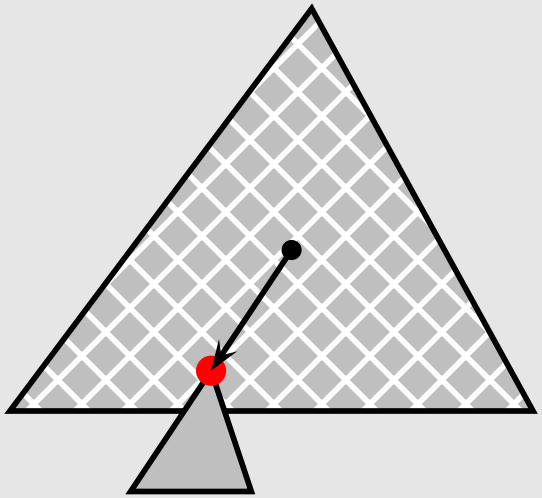
```
length (_:xs) = 1 + length xs
```



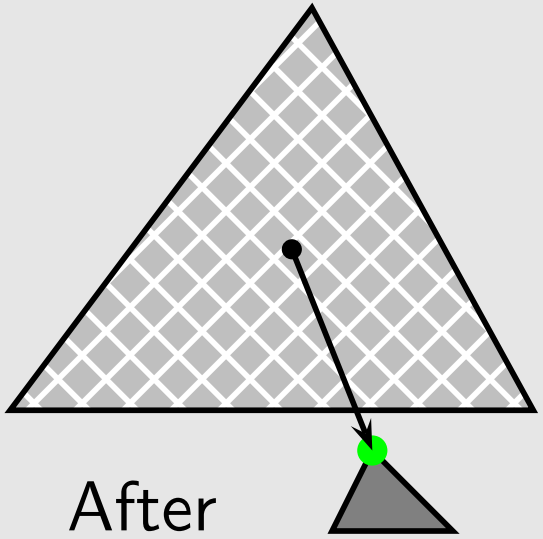
The Problem

- The redex is a subexpression of a larger expressions called the *context.*
- Within the context, there can be several pointers pointing to the redex.
- The contractum replaces the redex, within the context. The context is almost entirely preserved.
- Hence, all the pointers pointing to the redex must be redirected to the contractum.

The Problem (pictorially)



Before



After

Solution (typical)

Every node of an expression is accessed through an indirection pointer.

PROS: in a replacement, only the indirection pointer is redirected.

CONS: extra memory allocated when a node is constructed, extra cycles executed when a node is accessed.

The effort of rewriting is almost entirely allocating (and garbage collecting) nodes for the rhs and accessing nodes for pattern matching.

Solution (proposed)

Replace the content of the redex root node,
with the content of the contractum root node.
We call this *ripping* (rewriting in place).

No indirection pointer used.

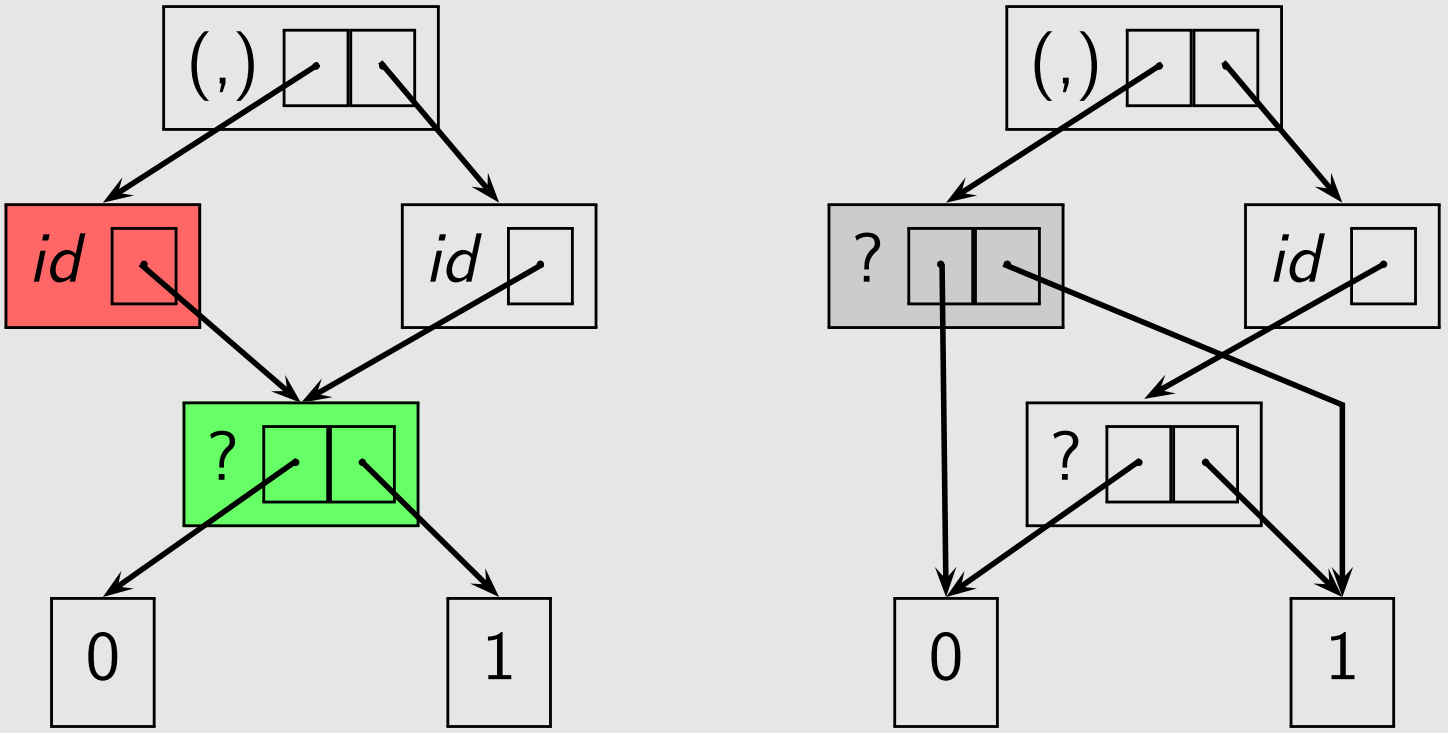
No extra memory allocated/collected,
no extra cycles executed.

Reuse a node (no allocation, no collection).

One drawback: unsound for **collapsing** rules.

Unsoundness example

```
(id x, id x) where x = 0 ? 1
```



Ripping produces some unintended values.

The Fix (typical)

Use a "forward node" (kind of indirection pointer) only for the application of collapsing rules.

Marginally inefficient, take some extra space and requires some extra cycle exactly like an indirection pointer, but only occasionally.

Complicates the run-time environment. Every time a node is accessed, the code must check whether it is a forward node, and if it is, act accordingly.

There can be chains of forward nodes. Forward nodes can be eliminated "just-in-time."

The Fix (proposed)

Evaluate the contractum of a collapsing rule to head-constructor form before applying the no-longer-collapsing rule.

Not a trivial idea:

- The contractum may not be a redex or its evaluation may not terminate or it may have multiple head-constructor forms.
- The graphs produced in this way are not the same as those produced by rewriting.

Effect on relation

The proposed solution changes the rewrite relation

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)
loop = loop
```

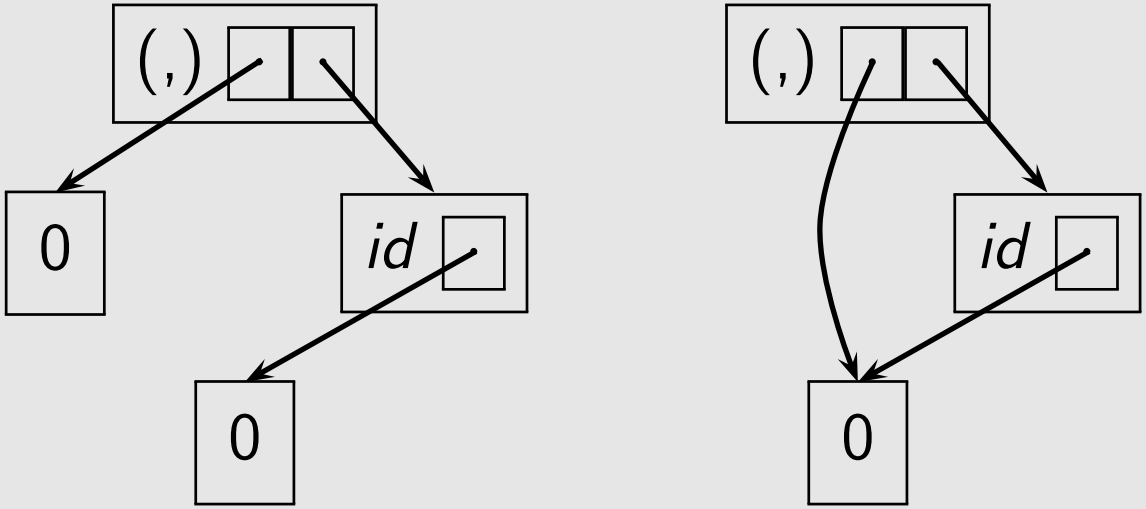
Transform the first rule into:

```
[] ++ [] = []
[] ++ (x:xs) = x:xs
```

Now, `[]++loop` is a redex in the orginal system, but it is not and never will be in the transformed system.

Effect on expressions

Evaluate `(id x, id x) where x = 0`



Value with ripping (left): `(0,0)`
Value with rewriting (right): `(x,x) where x=0`

Result #1

A program is an overlapping, inductively sequential graph rewriting system — ideal model for a functional logic program.

Transform (possibly implicitly) every collapsing rule

$$f \ldots x \ldots \to x$$

into

$$f \ldots (c\ x_1 \ldots x_n) \ldots \to c\ x_1 \ldots x_n$$

for every constructor c of arity n.

Transformed system is called the *uncollapsing variant*.

Computing in the uncollapsing variant produces the same values in same number of steps as in the original system.

Result #2

A program is an overlapping, inductively sequential graphs rewriting system without collapsing rules.

Adequate representation: a graph homomorphism such that distinct nodes with the same image are labeled by constructors.

Ripping provides adequate representations of the graphs obtained by rewriting.

If t is an adequate representation of u, then t and u are equal when seen as trees (bisimilar) and when common subexpressions are shared (fully collapsed).

Related work

Transformation originates from recent extended definition of need in constructor systems [Antoy, Jost 13].

Graph rewriting concerned with redex selection and/or G-machines [Kieburtz 85], [Burn, Peyton Jones, Robson 88], [Echahead, Janodet 95].

Forward nodes for collapsing rules are in the folklore [Kennaway, Klop, Sleep, de Vries 93].

Ripping is "mid-level."

Conclusion

1. FLP are executed by graph rewriting.
2. Collapsing rules are a problem.
3. Replace collapsing rules (implicitly or explicitly) by reducing the binding of the collapsing variable to head-constructor form.
4. Rewriting-in-place without collapsing rules computes the same values except for minor representation adjustments.
5. Our contribution is a marginal efficiency gain and a significant simplification of the run-time enviroment.


Thank you