On the Correctness of Bubbling

Sergio Antoy
Portland State University

RTA'06, Seattle, WA, August 11, 2006

Joint work with Daniel Brown and Su-Hui Chiang

Partially supported by the NSF grant CCR-0218224

Introduction

- Non-determinism simplifies modeling and solving problems in many domains, e.g., defining a language and/or parsing a string:

$$\begin{array}{lll} Expr & ::= & Num \mid Num\ BinOp\ Expr \\ BinOp & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\ Num & ::= & Digit \mid Digit\ Num \end{array}$$

- Non-determinism is a major feature of Functional Logic Programming.

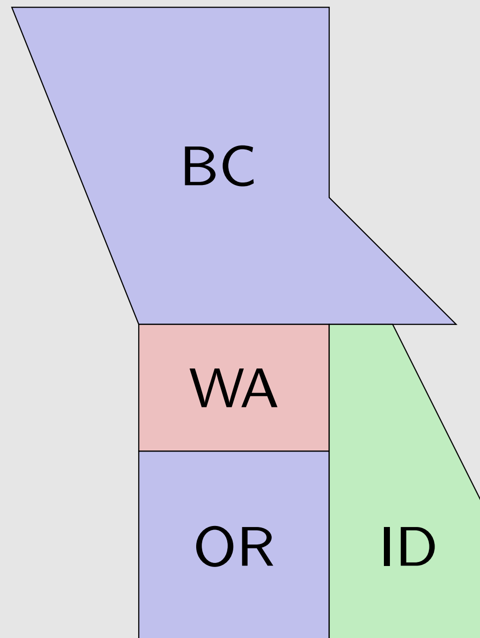
- A functional logic program is non-deterministic when some expression evaluates to *distinct* values, e.g., in Curry:

```
coin = 0 ? 1
```

- The predefined operator `?` yields either of its arguments.

An example - 1

Consider a program to color a map, e.g., the Pacific NW.



The map topology is defined as follows:

```
data State = WA | OR | ID | BC
states = [WA,OR,ID,BC]
adjacent = [(WA,OR),(WA,ID),(WA,BC),(OR,ID),(ID,BC)]
```

An example - 2

The states of the map are colored non-deterministically. The "function" `paint` paints its argument by associating a color to it.

```
data Color = Red | Green | Blue
paint x = (x, Red ? Green ? Blue)
```

Non-determinism makes coloring the map very easy.

```
theMap = map paint states
```

Since colors are assigned non-deterministically, adjacent states may have the same color. Therefore, the program constrains `theMap` to ensure the problem's condition (next slide).

The higher order function `map` is predefined. It applies `paint` to all the `states`.

An example - 3

With the above definitions, the complete program is a single function that ensures that adjacent states in the map have different colors:

```
solve | all diffColor adjacent = theMap
      where theMap = map paint states
            diffColor (x,y) = colorOf x /= colorOf y
            lookup ((s,c):t) x = if s==x then c
                                 else lookup t x
            colorOf = lookup theMap
```

Non-determinism greatly reduces the effort to design and code both data structures and algorithms for solving this problem.

Semantics

A *program* is a ***graph*** rewriting system.

```
sort x | sorted y = y
  where y = permute x
```

This is *somewhat* equivalent to

```
sort x | sorted (permute x) = (permute x)
```

The two occurrences of `y` in `sort` must be bound to the same value.

Sharing is essential in the presence of non-determinism.

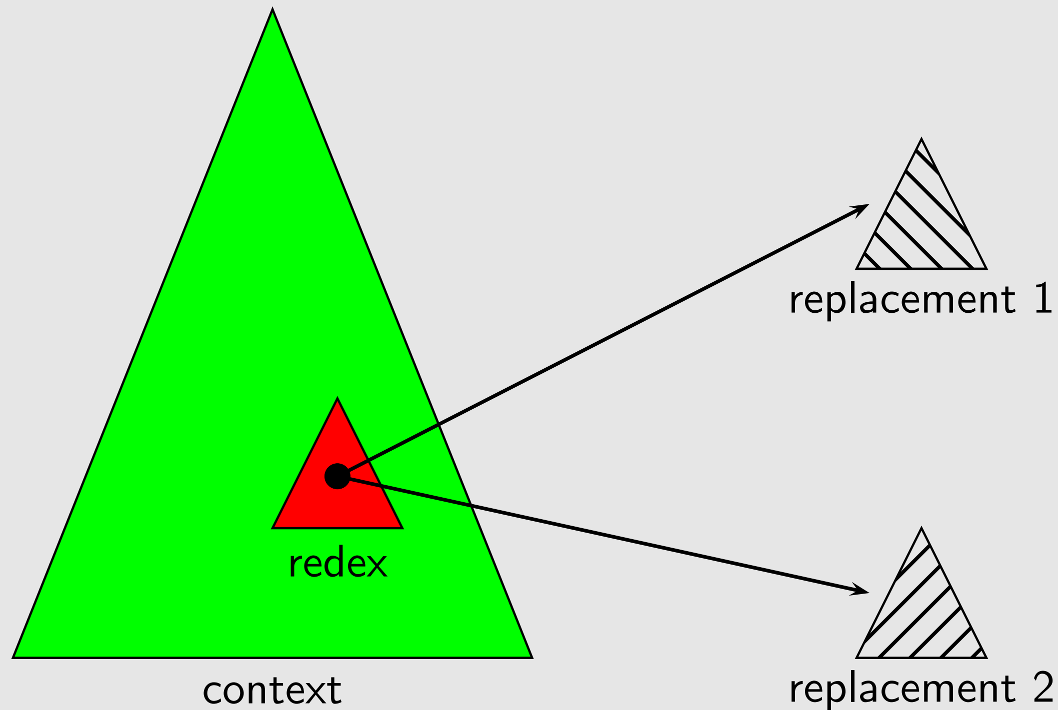
This semantics is called *call-time choice*.

Computations are admissible graph rewriting sequences.

Other properties of programs: conditional, constructor-based, overlapping inductively sequential, GRSs.

The Problem

In a non-confluent systems, the context of some redex must be used *multiple* times.



Approaches

There are various solutions to the problem.

- **Backtracking**

  Use the context for "the first" replacement. If and when the computation completes, recover the context and use it for other replacements.

- **Copying**

  Clone the context for each replacement. *Can evaluate non-deterministic choices concurrently*.

- **Bubbling** *new*

  Keep all the replacements in a single term. Clone the context incrementally sharing it as long and as much as possible.

Drawbacks of backtracking and copying

Backtracking and copying have significant drawbacks:

- **Backtracking**

  If the computation of “the first” replacement does not terminate, a value for another replacement, if such exists, is never found (*incompleteness*).

- **Copying**

  The computation of some replacement may fail before (a large portion of) its context is ever used. Therefore, copying the whole context is wasteful.

With some caution, bubbling avoids these drawbacks.

Bubbling

Distribute the parent of ? to the alternatives.

$$c[f(x\ ?\ y)] \to c[f(x)\ ?\ f(y)]$$

- Under appropriate conditions, ? moves up its context. Only the portion between the origin and the destination of the move of ? is cloned.
- The arguments of ? should be *evaluated concurrently*. If one fails, the choice *disappears.*
- The symbol ? becomes almost like a data *constructor*. The application of the rules of ? is delayed ... forever.

Nice and dandy... however, a small problem might arise. Before addressing the problem, let's see the advantages.

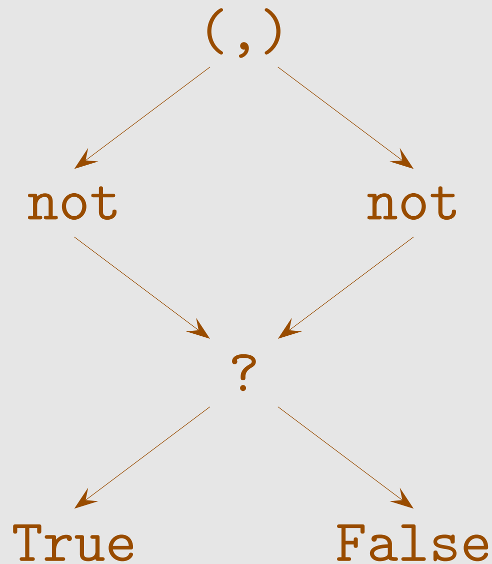
Advantages

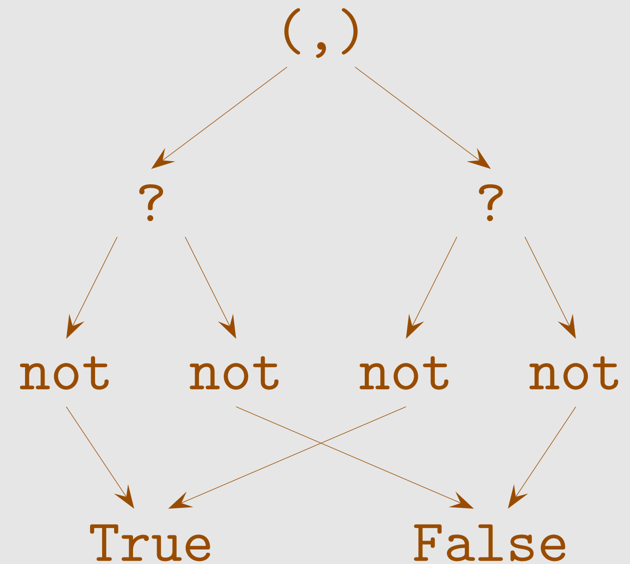
A (contrived) computation with bubbling

```
1+(2+(3 / (0 ? 1)))
  →  1+(2+((3 / 0) ? (3 / 1)))
  →  1+(2+(fail ? 3))
  →  1+(2+3)
```

- Bubbling enjoys some advantages of both backtracking and copying without their drawbacks.
- Only a small portion of the context of the choice has been copied.
- Typically and frequently, alternatives of choices fail.

Unsoundness





The term on the left has 2 values, `(True,True)` and `(False,False)`.

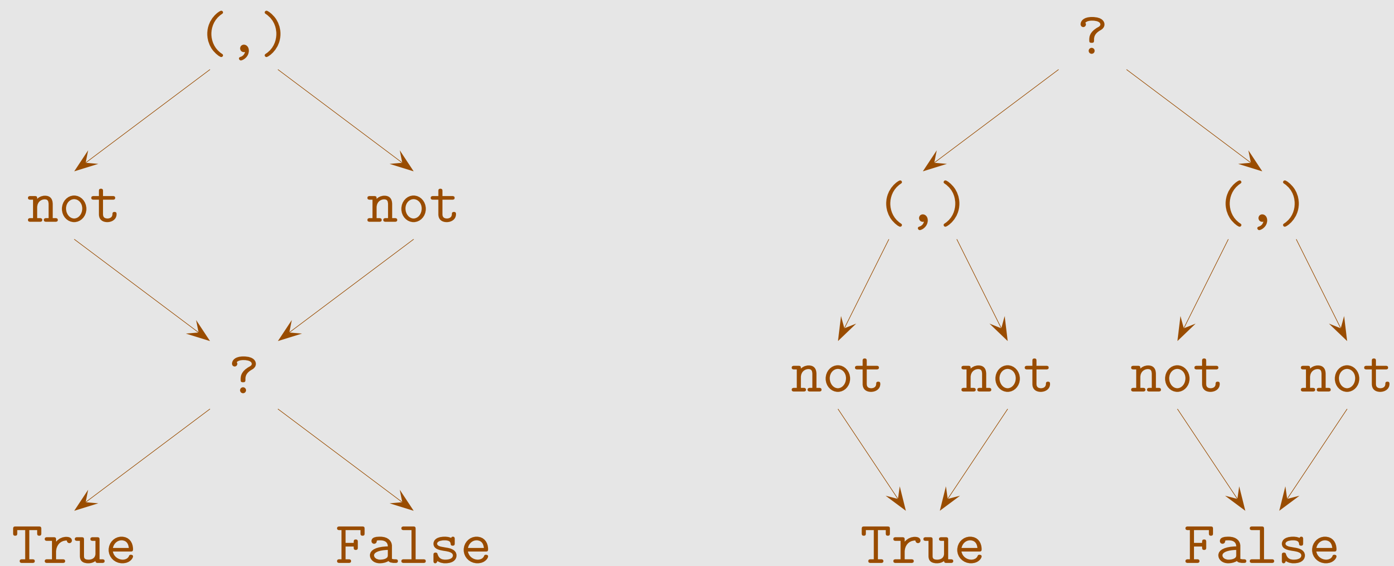
The term on the right is obtained by bubbling the term on the left.

This term has 4 values, including `(True,False)`, which cannot be derived from the term on the left.

The fix

The destination of bubbling must be a *dominator* of `?`

A node d *dominates* a node n in a rooted graph g,
if every path from the root of g to n goes through d.



These terms have the same set of values.

Correctness of Bubbling

The results hold for constructor-based GRSs with a very well-behaved form of overlapping.

- **Completeness**

  Any rewriting normal form of t remains reachable after a bubbling step of t by means of rewriting and possibly other bubbling steps.

- **Soundness**

  Any rewriting and/or bubbling normal form of t is reachable by pure rewriting of t.

These results are applied to the implementation of FLP languages, in particular the evaluation strategy [Termgraph'06].

Some interesting facts

Bubbling is transitive.

If t bubbles to u and u bubbles to v,
then t bubbles to v.

Bubbling and rewriting do *not* always commute.

$$\texttt{snd (1,2 ? 3)} \simeq \texttt{snd ((1,2) ? (1,3))}$$

`snd (1,2 ? 3)` → `2 ? 3` - - → ■ ← - - `snd ((1,2) ? (1,3))`

Strategy

The strategy is based on definitional trees.

It handles all the key aspects of the computation.

- **Redex computation**

  Extends INS, is aware of `?`
  Sometimes "leaves behind" occurences of `?`

- **Concurrency**

  Both arguments of `?` are evaluated in parallel.
  Other parallelism can be similarly accommodated.

- **Bubbling**

  Performed only to promote reductions
  (see next example).

Strategy behavior

Two major departures from considering `?` an operation.

- A needed argument is `?`-rooted, *but* no redex is available:

  ```
  1 + (2*2 ? 3*3*3)
  ```

  Evaluate concurrently the arguments of `?`

- A needed argument is `?`-rooted, *and* a redex is available:

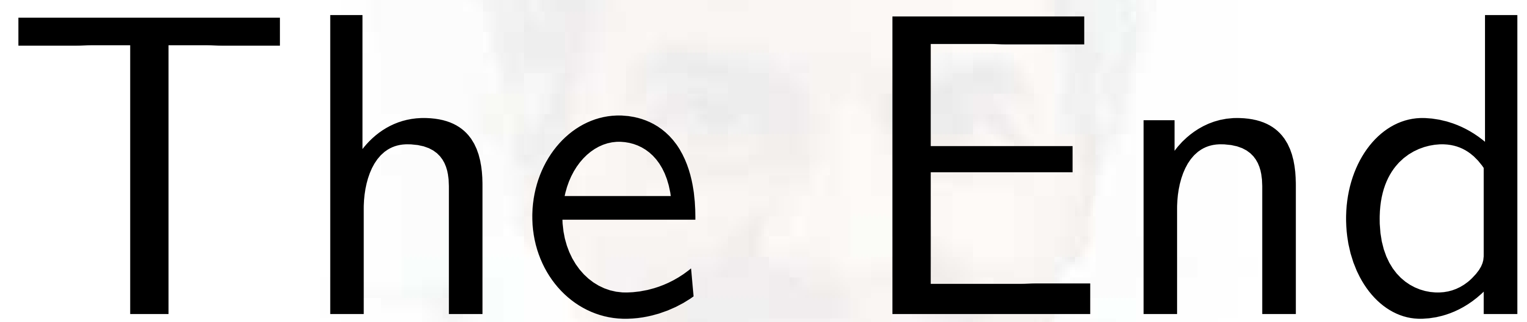
  ```
  1 + (4 ? 9*3)
  ```

  Bubble and continue with:

  ```
  (1 + 4) ? (1 + 9*3)
  ```

Conclusion

- New approach for non-confluent, constructor-based rewriting
- It finds application in functional logic language development
- It avoids the incompleteness of backtracking
- It avoids the inefficiency of context copying
- It is applied in a newly developed sound and complete strategy
- There exists a prototypical implementation for rewriting
- The extension to narrowing is under way


The End