Lazy Context Cloning for Non-Deterministic Graph Rewriting

Sergio Antoy
Portland State University

TERMGRAPH'06, Vienna, Austria, April 1, 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:

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

- 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 operator `?`, defined in the *Prelude*, selects either of its arguments.

An example

Consider a program to find a donor for a blood transfusion. The type `BloodTypes` defines the 8 blood types:

```
data BloodTypes = Ap | An | ABp | ...
```

The non-deterministic function `receive` defines which blood types can receive the argument of the function:

```
receive Ap = Ap ? ABp
receive Op = Op ? Ap ? Bp ? ABp
...
```

The function `hasType` returns the blood type of its argument, a person:

```
hasType "John" = ABp
hasType "Doug" = ABn
hasType "Lisa" = An
```

An example, cont'd

The whole program is a single non-deterministic function, `donorFor`, that takes a person x and return a donor, if it exists, for a blood transfusion to x:

```
donorFor x | receive (hasType y) =:= hasType x
           & x =/= y
           = y
           where y free
```

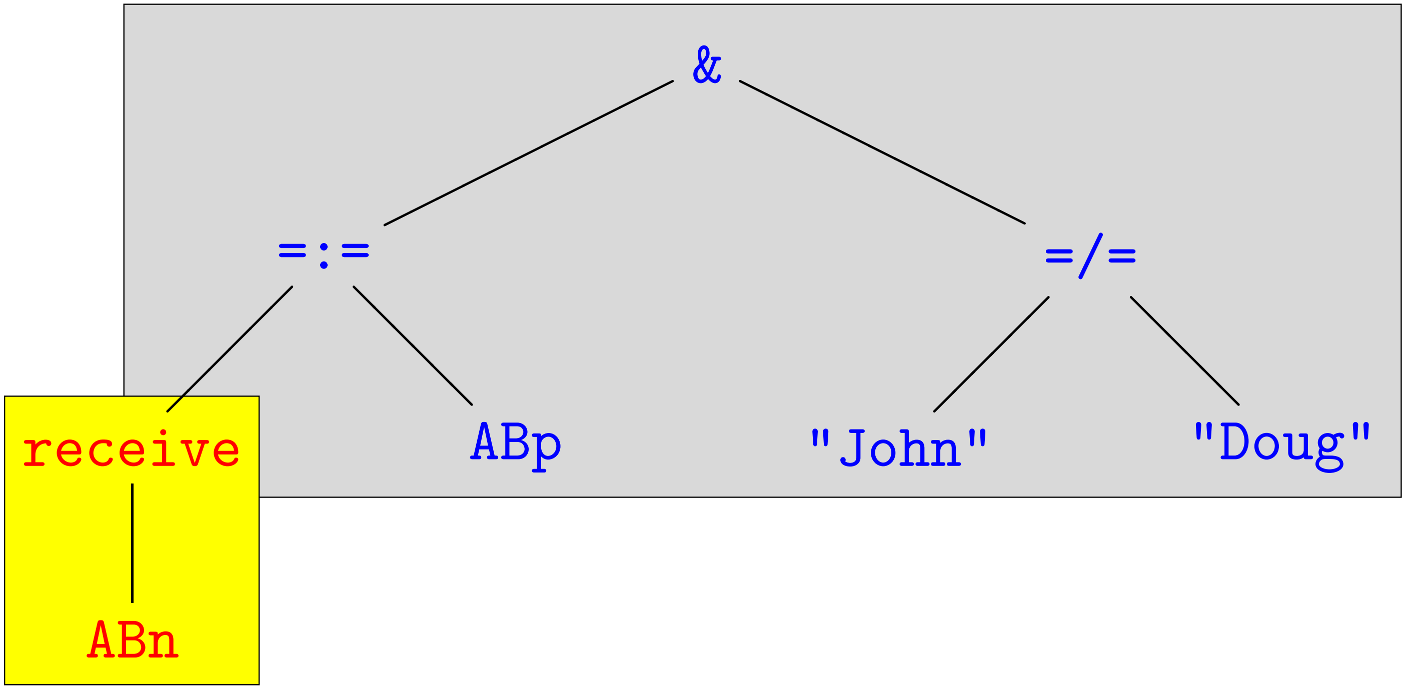
E.g.:

`donorFor "John"` yields `"Doug"` or `"Lisa"`
`donorFor "Lisa"` fails

Non-determinism greatly reduces the effort to design and code both data structures and algorithms for handling a many-to-many relation.

Evaluation

The evaluation of `donorFor "John"` goes through the following term:



The redex `receive ABn` has two values. The *context* of each value is the same. Therefore the context of this redex must be "used twice."

Approaches

To rewrite in a non-confluent systems, the context of some redex must be used multiple times. There are two common approaches to this 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**

  Make a copy of the context for each replacement. *Can evaluate non-deterministic choices concurrently*.

Problems

Both backtracking and copying have significant problems:

- **Backtracking**

  If the computation of "the first" replacement does not terminate, the value for the other replacements, if such exists, is never found (*incompleteness*).

- **Copying**

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

We propose an approach, called *bubbling*, that ensures completeness and minimizes copying.

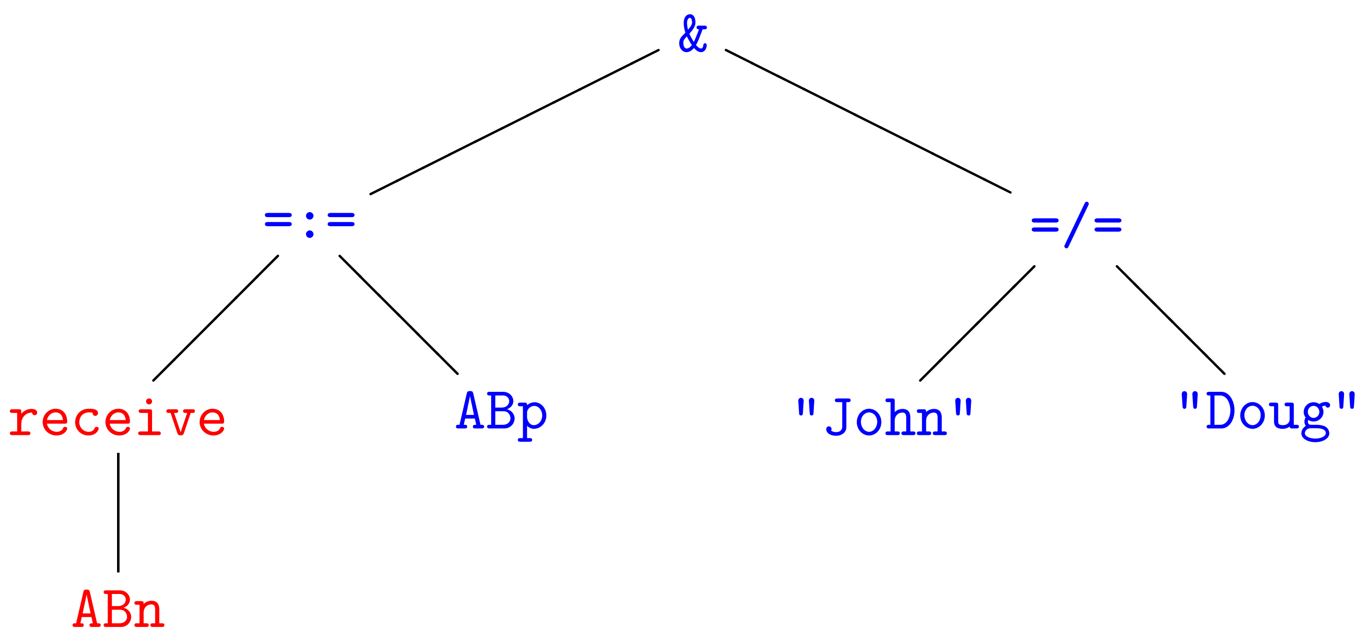
Bubbling

An expression to evaluate is a *term graph*. We are concerned with the evaluation of an expression to a constructor *head normal form*.

- The symbol `?` becomes a data *constructor* (the application of the rules of `?` is delayed).
- The arguments of `?` are *evaluated concurrently*.
- When an argument of `?` becomes constructor-rooted, `?` moves up its context.
- Only the portion between the origin and the destination of the move of `?` is copied.
- The move is *sound* only if the destination of `?` *dominates* it.

Steps

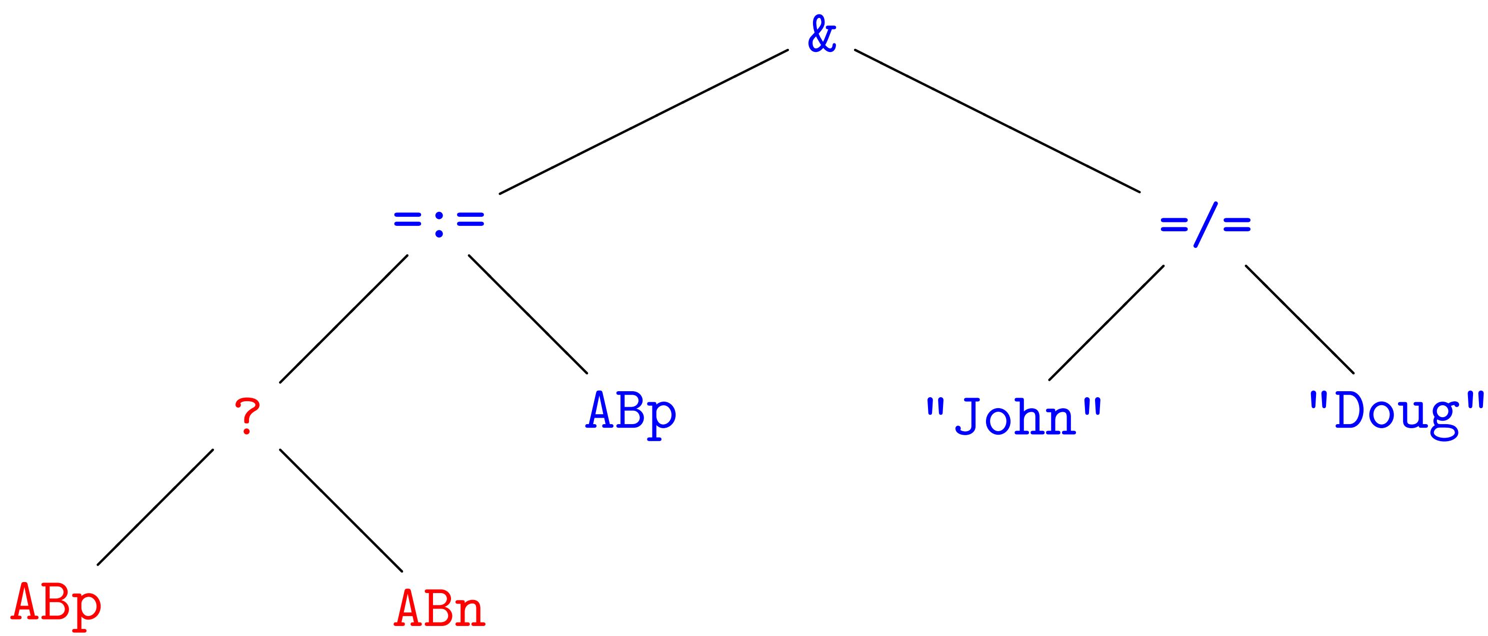
Steps of an evaluation



Reduce the redex `receive ABn` to `ABP ? ABn`.

Steps

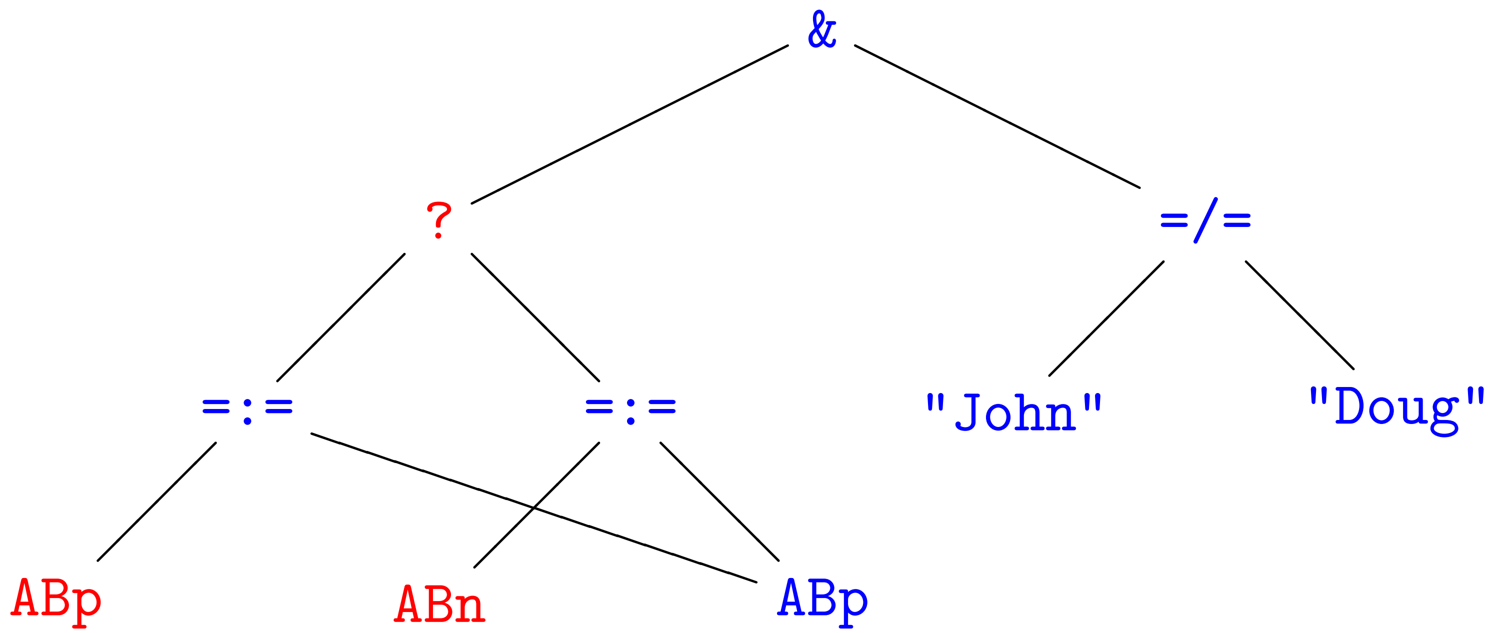
Steps of an evaluation



Bubble the non-deterministic choice.

Steps

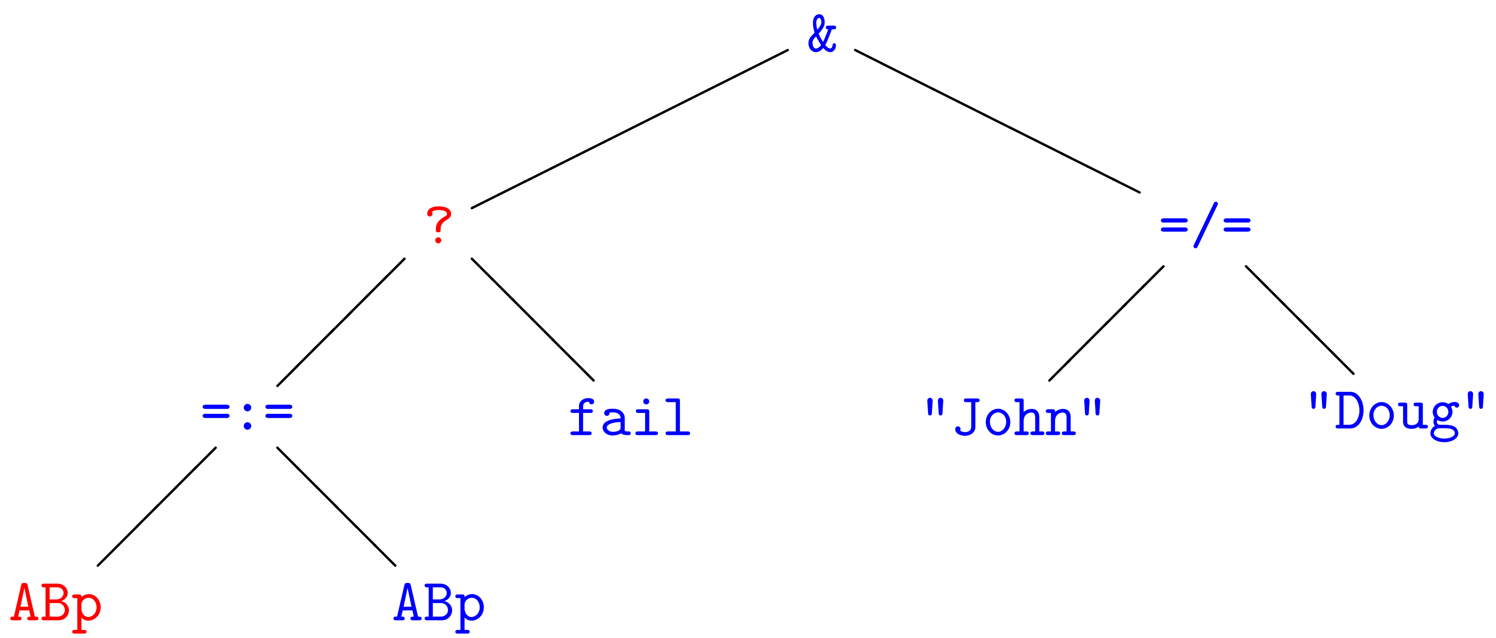
Steps of an evaluation



Evaluate `ABn =:= ABp`.

Steps

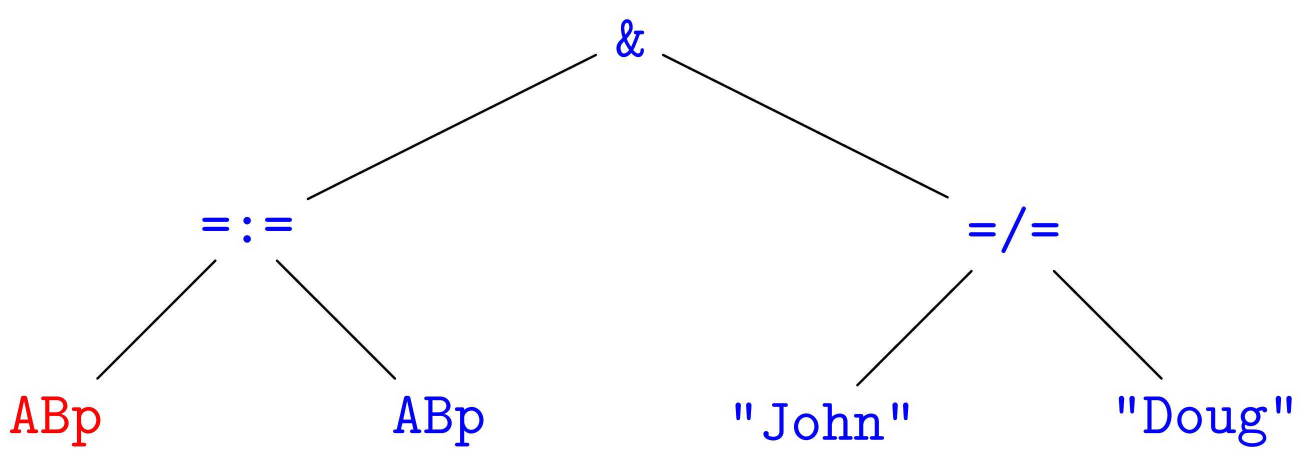
Steps of an evaluation



Eliminate the irrelevant choice.

Steps

Steps of an evaluation



Continue the evaluation.
No significant context has been copied.
Backtracking is not used.

Distributing

A computation is a sequence of rewriting and/or bubbling steps.

A bubbling step is similar to the application of a distributive law.

In the example, we distributed the parent of the occurrence of `?`:

`(x ? y) =:= z` $\rightarrow$ `(x =:= z) ? (y =:= z)`

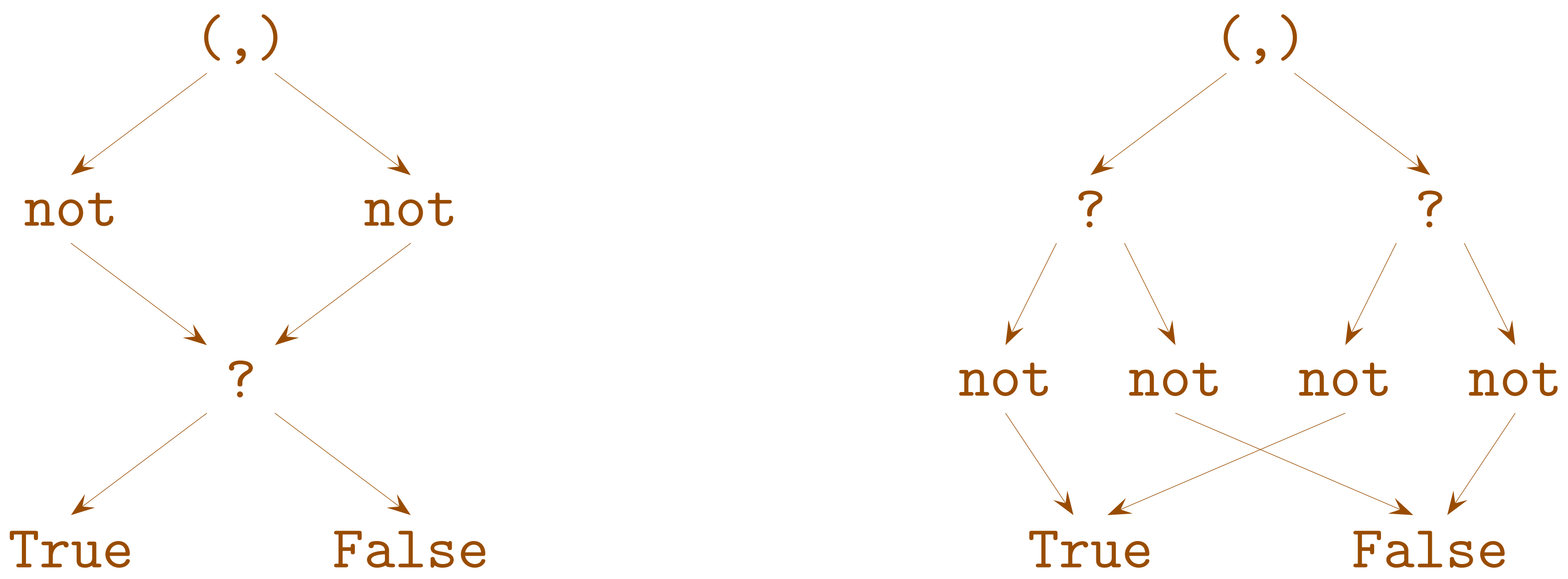
Unfortunately, distributing is unsound in some cases. Consider:

```
f x = (not x, not x)
```

and evaluate:

```
f (True ? False)
```

Unsoundness



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

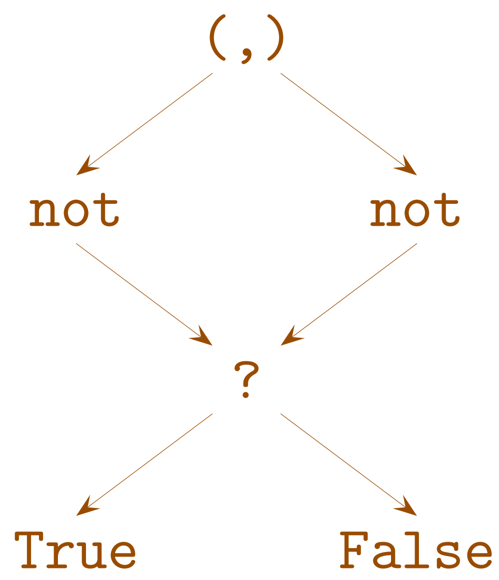
The term on the right is obtained by bubbling the term on the left.

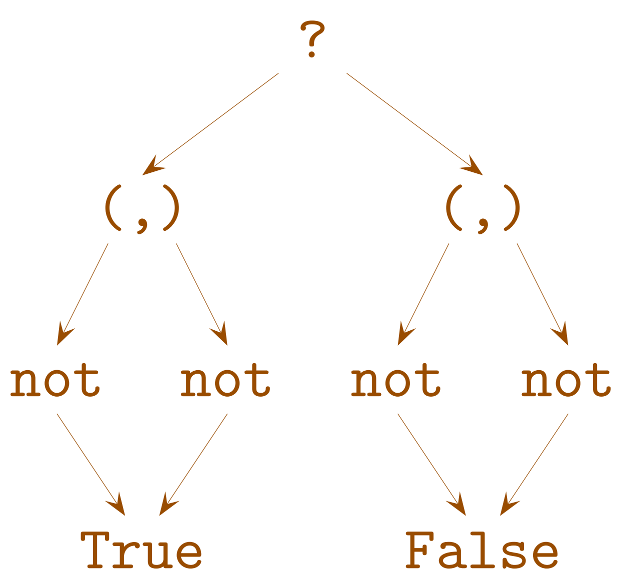
This term has 4 values, including `(True,False)`, which cannot be derived from the term on the left.

Soundness

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.

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 "leave 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)
  ```

  Evaluate concurrently the arguments of `?`

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

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

  Bubble and continue with:

  ```
  (1 + 4) ? (1 + 3*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
- Very recently bubbling has been proved sound and complete
- It is not known if steps are needed (modulo non-det. choices)
- There exists a prototypical implementation for rewriting
- The extension to narrowing is under way

The End