Structural Recursion for Querying Ordered Graphs

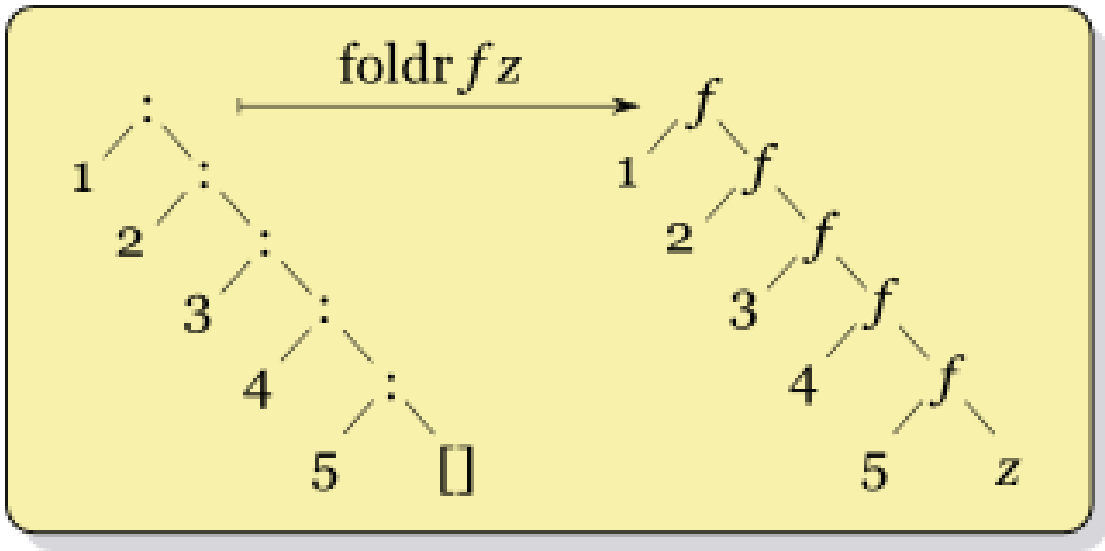
Soichiro Hidaka, Zhenjiang Hu, Kazuyuki Asada, Hiroyuki Kato, Keisuke Nakano

April 1, 2014

Motivation

Structural recursion

Structural recursion plays an important role in functional programming by providing a systematic way for constructing and manipulating functional programs

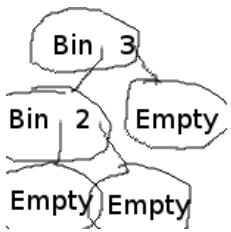


With trees

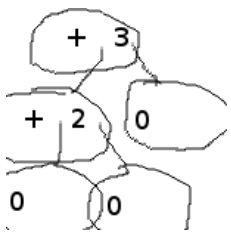
```
data Tree a = Empty | Bin (Tree a) a (Tree a)

foldTree _ z Empty = z
foldTree f z (Bin l a r) = f (foldTree f z l) a (foldTree f

t = Bin (Bin Empty 2 Empty) 3 Empty
```




```
foldTree (\a b c -> a + b + c) 0 t
```



With graphs

Graphs are essentially not inductive

How to resolve?

Use special trees - could overlay a tree on a graph and represent leftover edges as pointers or embedded functions

Not too successful because of the gap between trees and graphs, requiring programmers to bridge the gap by explicitely programming these specific pointers and embedded functions

Treat graphs as trees

Would it be possible to define a structural recursion on graphs as if the graphs were trees (special kinds of graphs) while using it to manipulate general graphs?

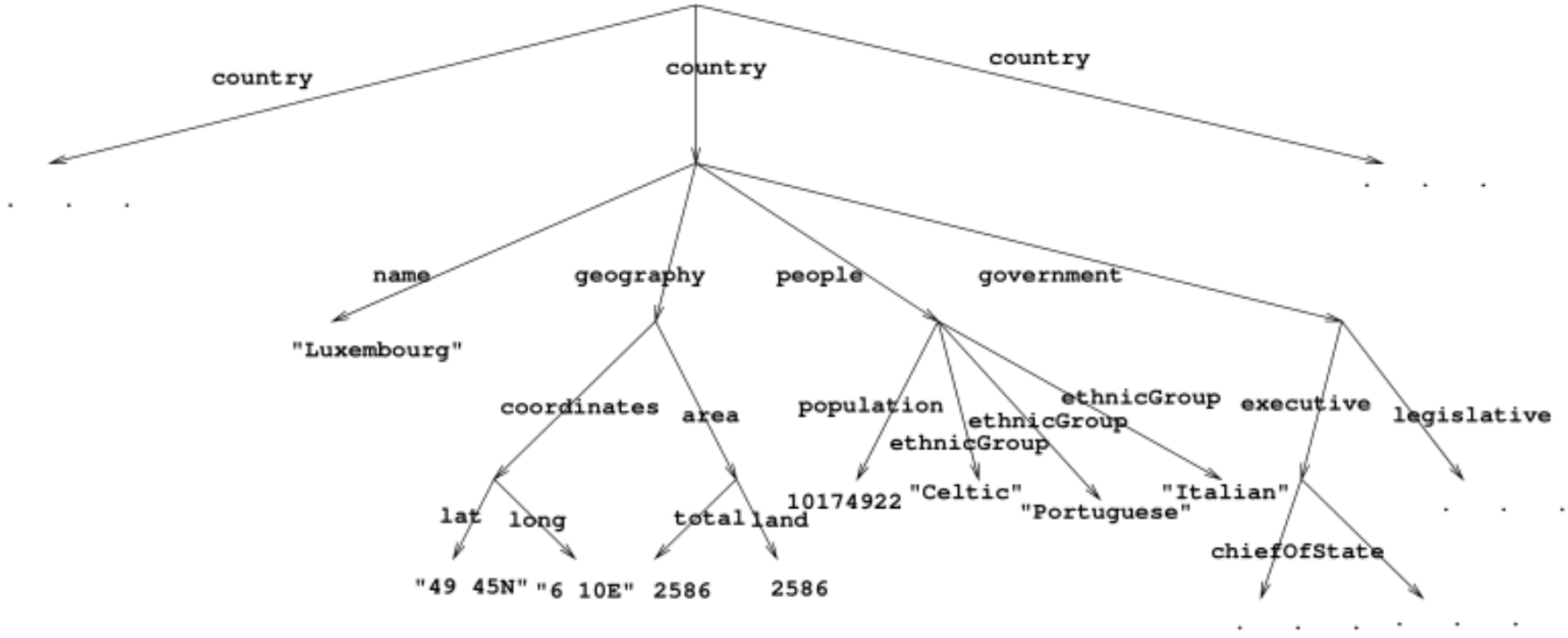
Yes, proven in databaes community with UnCAL and UnQl.

Querying using Structured Recursion

Definition

Definition of a tree

$t ::= t \cup t \mid \{l : t\} \mid \{\} \mid a$


country
country
country
name
geography
people
government
"Luxembourg"
coordinates
area
population
ethnicGroup
ethnicGroup
ethnicGroup
executive
legislative
lat
long
total
land
10174922
"Celtic"
"Portuguese"
"Italian"
chiefOfState
"49 45N"
"6 10E"
2586
2586

Structural recursion on trees

```
fun f1(T1 U T2) = f1(T1) U f1(T2)
  | f1({L:T})   = if L = ethnicGroup then
                    {result: T} else f1(T)
  | f1({})      = {}
  | f1(V)       = {} .

{result: "Celtic", result: "English",
 result: "Portuguese", result: "Italian",
 result: "Fleming", result: "Walloon", ...}.
```

Can guarantee termination using three laws:

1. The right-hand side of the clause with the union pattern should always have the form

   `f(T1 U T2) = f(T1) U F(T2).`

2. The only recursive calls in the right-hand side of the singleton clause, `f({L:T})`, are for the argument `T`. Moreover, the results of the recursive calls can only be used in constructors, and not passed as arguments to other functions or predicates. In particular, recursive calls of the form `f({a:{b:T}})` or `g(f(T))` are not allowed.
3. The result of `f({})` should always be `{}`.

Tree equality

Definition (Value-equality for trees)

$t \equiv t'$ is defined as:

When t and t' are atomic values, $t \equiv t'$ if those values are the same.

If $t = \{l_1 : t_1, \ldots, l_m : t_m\}$ and $t' = \{l'_1 : t'_1, \ldots, l'_m : t'_n\}$, and if

- for each $i \in \{1, \ldots, m\}$, there exists $j \in \{1, \ldots, n\}$ such that $l_i = l'_j$ and $t_i \equiv t'_j$
- for each $j \in \{1, \ldots, n\}$, there exists $i \in \{1, \ldots, m\}$ such that $l_i = l'_j$ and $t_i \equiv t'_j$

then $t \equiv t'$.

Graph data model

Markers are labels for input and output, denoted as $\&x, \&y, \&z, \ldots$

There is a distinguished marker '&', typically used as the default input marker (the root).

A labeled graph with input markers $\mathcal{X}$ and output markers $\mathcal{Y}$ is $g = (V, E, I, O)$, where V is a (possibly infinite) set of nodes, $E \subseteq V \times Label_\epsilon \times V$ are the edges, $I \subseteq \mathcal{X} \times V$, and $O \subseteq V \times \mathcal{Y}$.

$DB^{\mathcal{X}}_{\mathcal{Y}}$ is the set of data graphs with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$

Trees have input & and no outputs.

$t ::= a \mid \&y \mid \{l : t, \ldots, l : t\}$

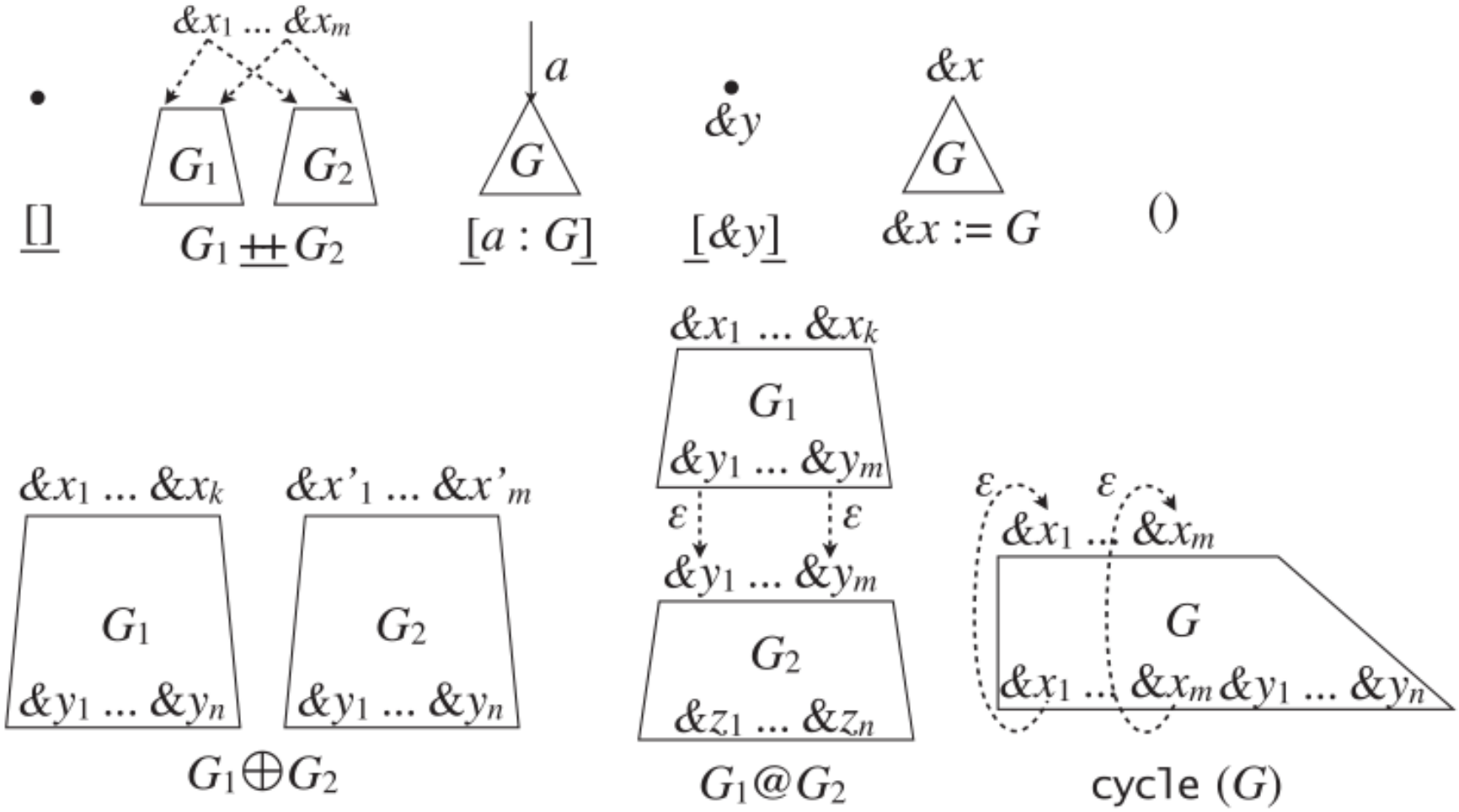


Figure: graph constructors

Unfolding

given $d \in DB^{\mathcal{X}}_{\mathcal{Y}}, d = (V, E, I, O), \mathit{unfold}(d) = (V', E', I', O')$ where

- $V' = \{p \mid p \text{ is a path in } d \text{ starting at some } I(\&x), \text{ for } \&x \in \mathcal{X}\}$.
- $E' = \{(p, a, p') \mid p' = p.a \text{ where } p, p' \in V' \text{ and } a \in \mathit{Label}_\epsilon\}$.
- $I' = \{(\&x, p) \mid p \text{ ends in } v \text{ and } (\&x, v) \in I\}$
- $O' = \{(p, \&x) \mid p \text{ ends in } v \text{ and } (v, \&x) \in O\}$

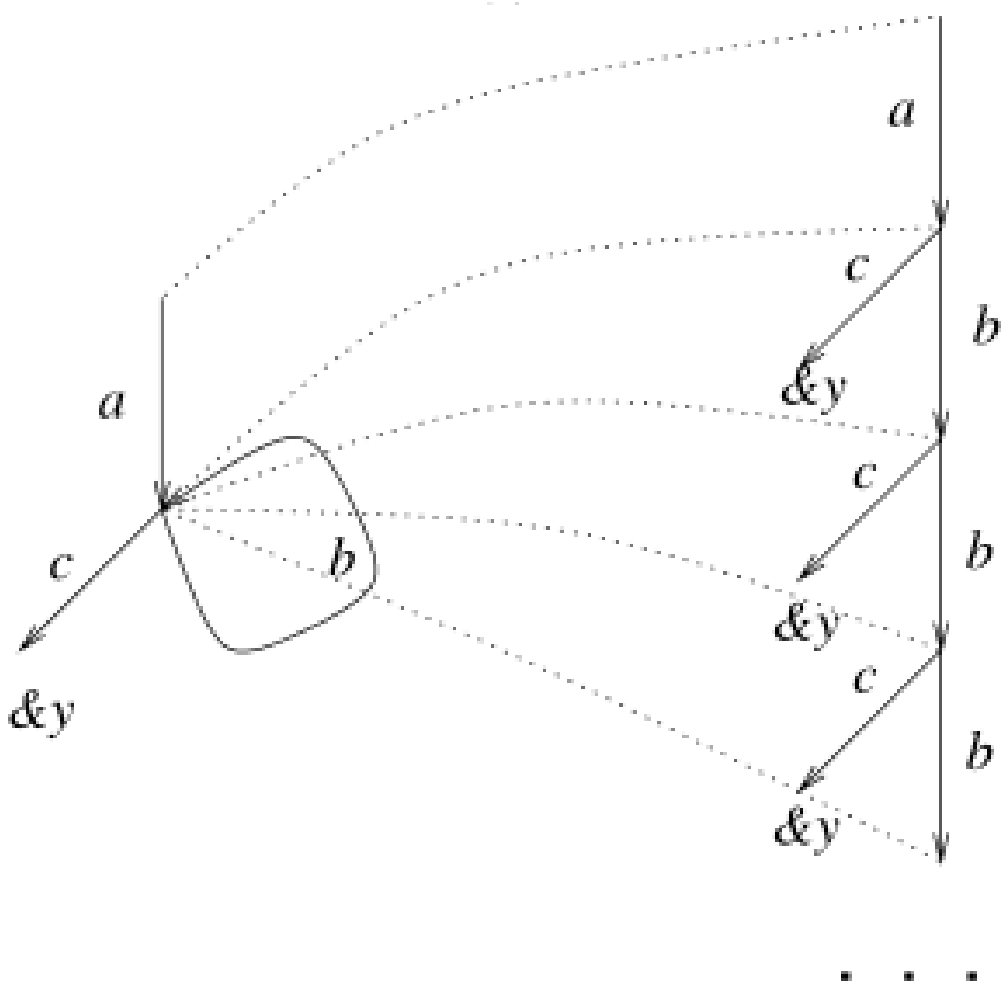


Figure: unfolding

Structural recursion on graphs

$$
\begin{aligned}
rec(e)(\{\}) &\equiv \{\} \\
rec(e)(\{l : d\}) &\equiv e(l, d)@rec(e)(d) \\
rec(e)(d_1 \cup d_2) &\equiv rec(e)(d_1) \cup rec(e)(d_2) \\
rec(e)(\&x := d) &\equiv \&x \cdot (rec(e)(d)) \\
rec(e)(\&y) &\equiv (\&z_1 := \&y \cdot \&z_1, \ldots, \&y \cdot \&z_p) \\
rec(e)() &\equiv () \\
rec(e)(d_1 \oplus d_2) &\equiv rec(e)(d_1) \oplus rec(e)(d_2) \\
rec(e)(d_1@d_2) &\equiv rec(e)(d_1)@rec(e)(d_2) \\
rec(e)(cycle(d)) &\equiv cycle(rec(e)(d)).
\end{aligned}
$$

Infinite recursion

Infinite loops caused by recursion. Solve by either

1. Memorization: remember all recursive calls and avoid entering infinite loop
2. Apply the recursive functions in parallel, on all the graph's edges – hence, each function will be applied only as many times as edges in the graph, and infinite loops are avoided: *bulk semantics*

Bisimulation

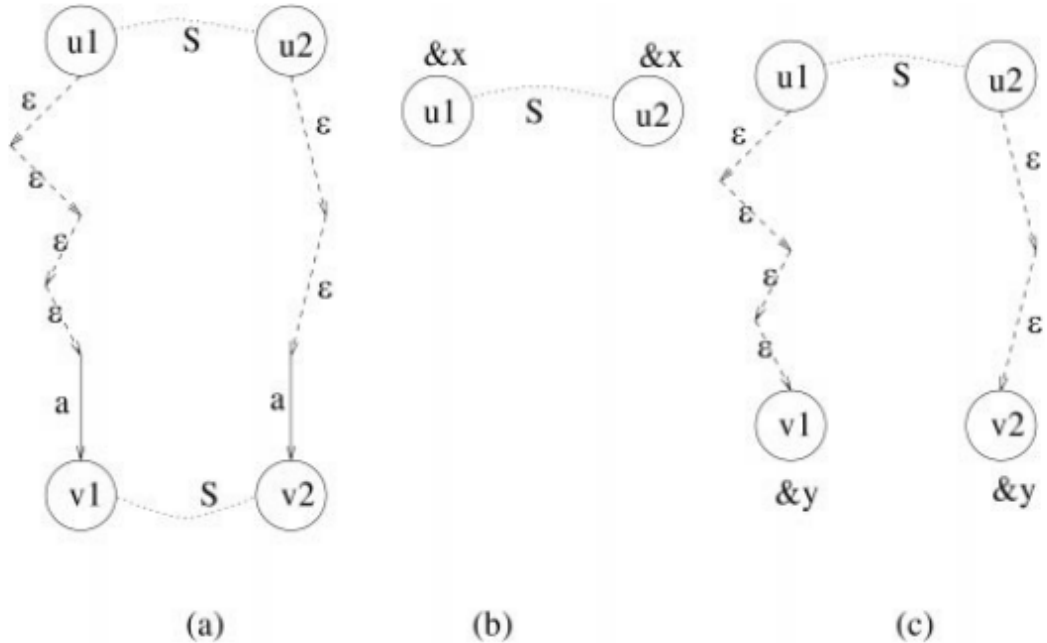
Let $g_1 = (V_1, E_1, I_1, O_1), g_2 = (V_2, E_2, I_2, O_2)$ be two labeled graphs, both with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$. Define $S \subseteq V_1 \times V_2$ as an extended simulation such that

1. if $(u_1, u_2) \in S \wedge (u_1, \epsilon^*.a, v_1) \in E_1$ with $a \neq \epsilon$, then there exists a node v_2 such that $(u_2, \epsilon^*.a, v_2) \in E_2$ and $(v_1, v_2) \in S$.
2. if $(u_1, u_2) \in S \wedge (\&x, u_1) \in I_1$ then $(\&x, u_2) \in I_2$.
3. if $(u_1, u_2) \in S \wedge (u_1, \epsilon^*, v_1) \in E_1 \wedge (v_1, \&y) \in O_1$ then there exists a node v_2 such that $(u_2, \epsilon^*, v_2) \in E_2 \wedge (v_2, \&y) \in O_2$.
4. $(I_1(\&x), I_2(\&x)) \in S$, for every $\&x \in \mathcal{X}$.

An extended bisimulation from g_1 to g_2 is an extended simulation S for which S^{-1} is also an extended simulation.

Two labeled graphs g_1 and g_2 are value equivalent ($g_1 \equiv g_2$) if there exists an extended bisimulation from g_1 to g_2.

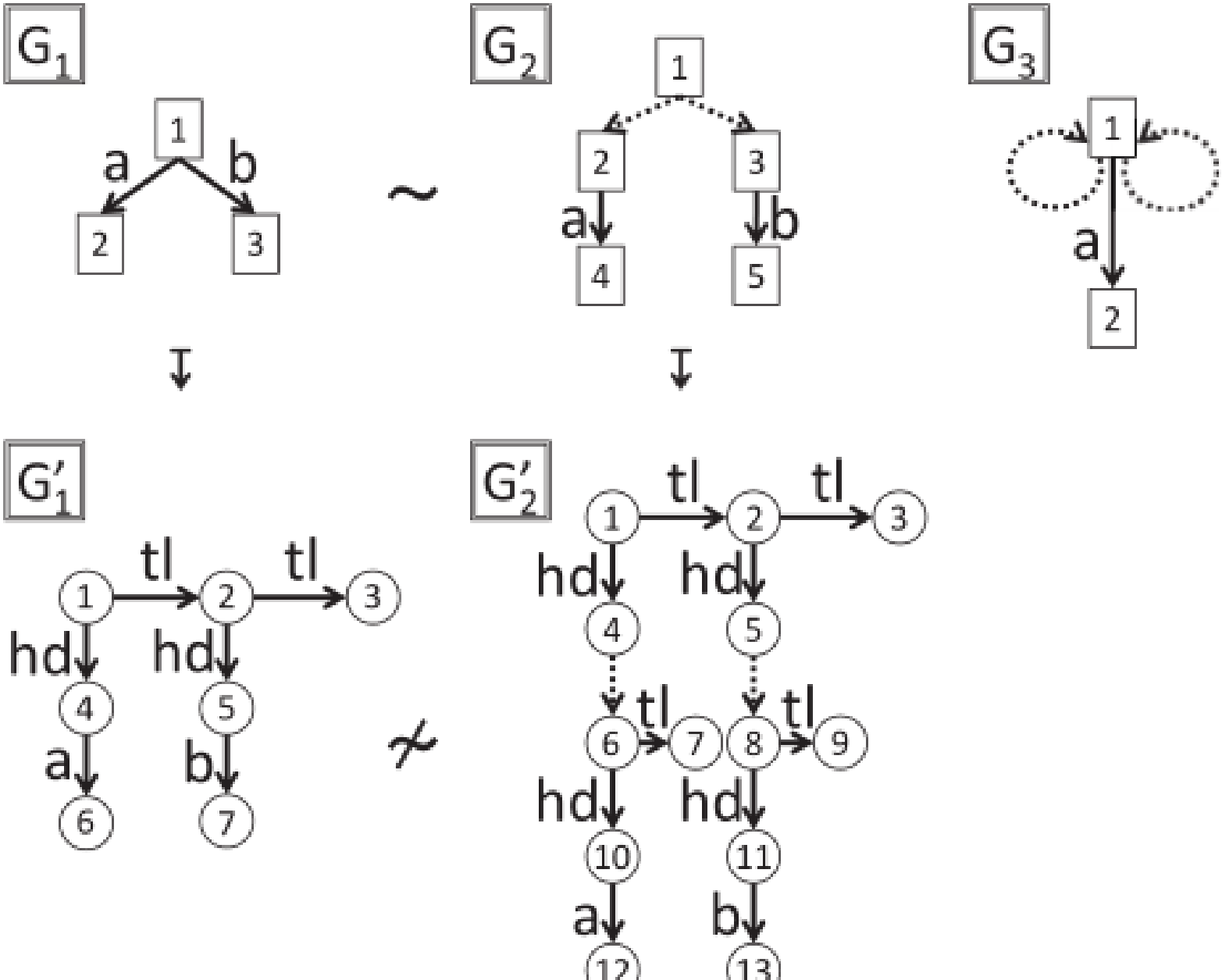
Bisimulation



Problem?

- “Sibling dimension” - cannot write a transformation in UnCAL that exracts all edges labeled with the average number on the labels of all siblings
- Cannot query ordered graphs (think xml)

List-like head/tail encoding?



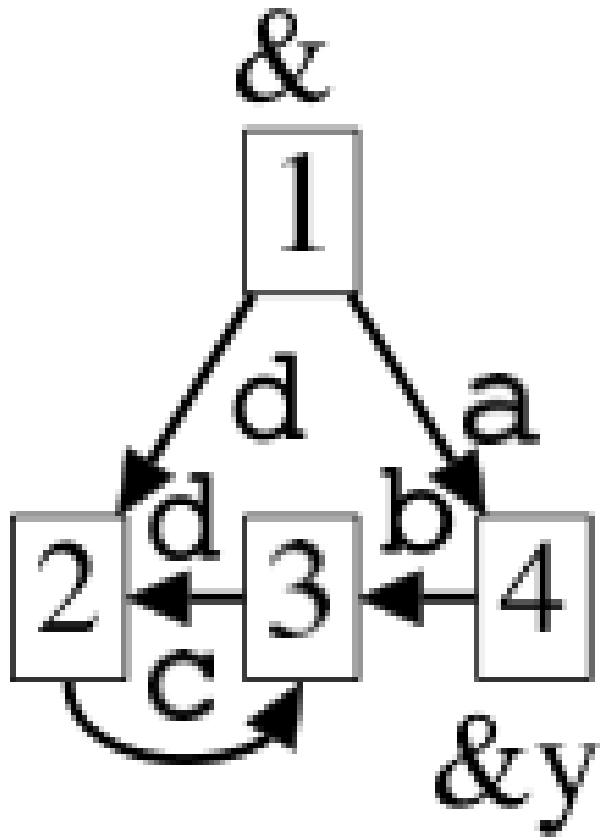
Solve?

Create new language that handles ordered graphs directly called λ_{FG}

New representation

- V is a set of *nodes*
- $B : V \to \mathit{List}(\mathcal{L}_\epsilon \times V + Y)$ is a branch function mapping a node to a list of branches: a branch is either a labeled edge $\mathrm{Edge}(l, v)$ or an output marker $\mathrm{Outm}(\&y)$
- $I : X \to V$ is a function which determines the input nodes of the graph

$$
\begin{aligned}
V &= \{1,2,3,4\} \\
B &= \{1 \mapsto [\mathrm{Edge}(d,2), \mathrm{Edge}(a,4)], 2 \mapsto [\mathrm{Edge}(c,3)], \\
&\quad\ 3 \mapsto [\mathrm{Edge}(d,2)], 4 \mapsto [\mathrm{Edge}(b,3), \mathrm{Outm}(\&y)]\} \\
I(\&) &= 1.
\end{aligned}
$$



Ordered graphs with countable widths

Since graphs may have a countable (including finite) width, we need to extend our definition of an ordered graph.

Ordered graph with a countable width $\mathcal{G}^{c\mathcal{X}}_{\mathcal{Y}}$.

Proper branches

A proper branch of a node is defined as a path from v going through zero or more ϵ-edges and reaching a non-ϵ-edge (or an out marker)

The set of all proper branches of v in G is denoted by $\mathrm{Pb}(G, v)$

Bisimilarity on ordered graphs

For two graphs $G = (V, B, I)$ and $G' = (V', B', I')$ in $\mathcal{G}^{c\mathcal{X}}_{\mathcal{Y}}$, $R \subseteq V \times V'$ is a *bisimulation relation*, if for any $(v, v') \in R$, there is an order isomorphism $f : (\mathrm{Pb}(G, v), \leq_{\mathrm{Pb}}) \to (\mathrm{Pb}(G', v'), \leq_{\mathrm{Pb}})$ satisfying the order-preserving propery: For any proper branch $p = (v \overset{\epsilon}{\to}_{i_0} \dots v_n \to_{i_n}) \in \mathrm{Pb}(G, v)$ with $f(p) = (v' \overset{\epsilon}{\to}_{i'_0} \dots v'_{n'} \to_{i'_{n'}}) \in \mathrm{Pb}(G', v')$ we have

- Edge Correspondence: if $B(v_n).i_n = \mathrm{Edge}(l, u)$ for some $l \in \mathcal{L}, u \in V$, then there exists $u' \in V'$ such that $B'(v'_{n'}).i'_{n'} = \mathrm{Edge}(l, u')$ and $(u, u') \in R$
- Marker Correspondence: if $B(v_n).i_n = \mathrm{Outm}(\&y)$ for some $\&y \in \mathcal{Y}$, then $B'(v'_{n'}).i'_{n'} = \mathrm{Outm}(\&y)$

Two graphs G and G' are bisimilar ($G \sim G'$) if there is a bisimulation relation R such that for every input marker $\&x \in \mathcal{X}, (I(\&x), I'(\&x)) \in R$.

Recursive semantics

$f = \mathbf{srec}(e, d)$

$f([l_1 : g_1] +\!\!+ \cdots +\!\!+ [l_n : g_n])$
$= d_\emptyset([e(l_1, g_1)@f(g_1), \ldots, e(l_n, g_n)@f(g_n)]).$

Problem: may not terminate if g_i loops back to root

Generalized recursive semantics

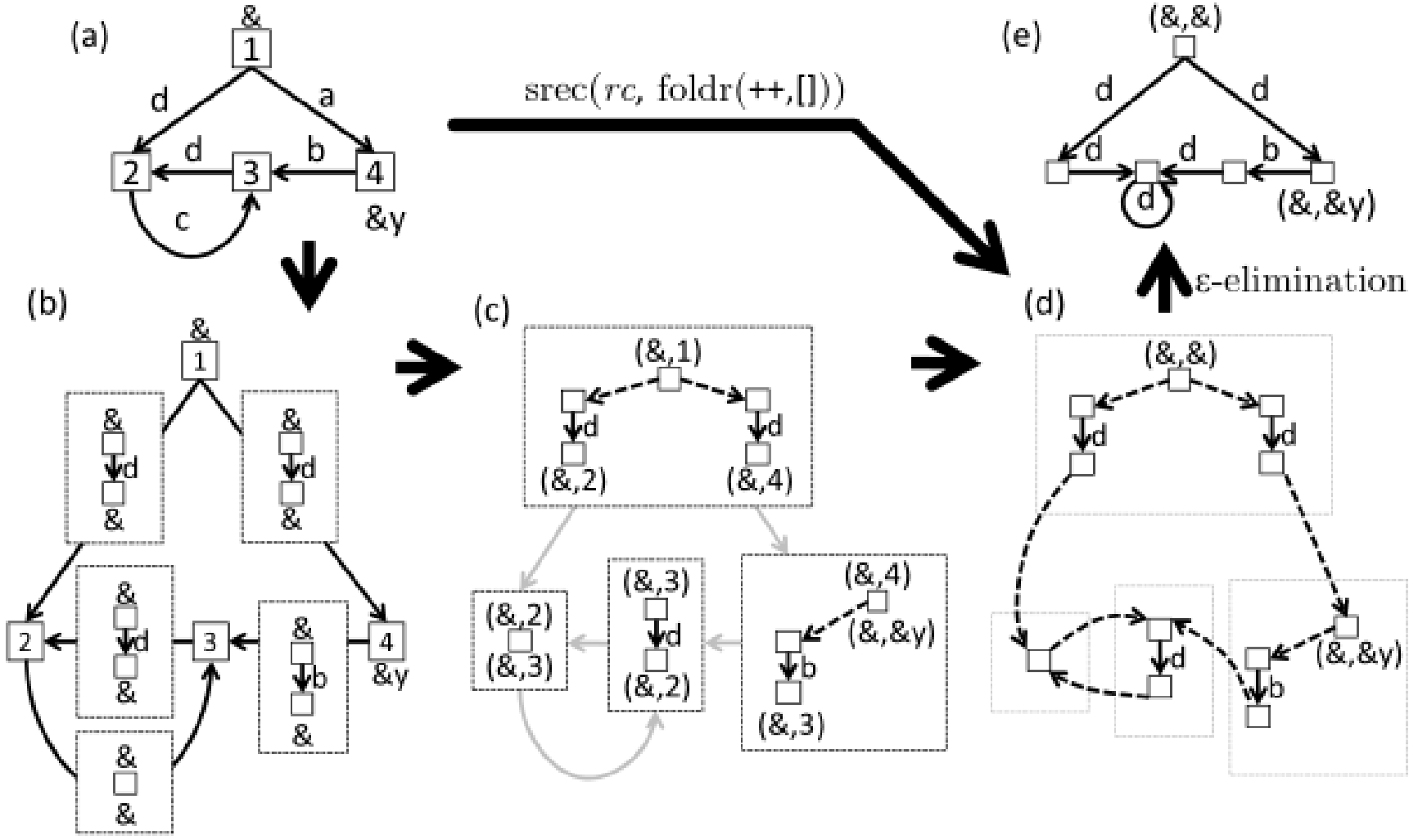
$$f([l_1 : g_1] +\!\!+ \cdots +\!\!+ [l_n : g_n])$$
$$= d_{\overline{n}}([e(l_1, g_1)@\lfloor \mathrm{in}_1 \rfloor, \ldots, e(l_n, g_n)@\lfloor \mathrm{in}_n \rfloor])@((\lfloor \mathrm{iso}_1 \rfloor @ f(g_1)) \oplus$$
$$\cdots \oplus (\lfloor \mathrm{iso}_n \rfloor @ f(g_n)))$$

$$\mathrm{in}_i(\&z) \stackrel{\mathrm{def}}{=} (\&z, i)$$
$$\mathrm{iso}_i((\&z, i)) \stackrel{\mathrm{def}}{=} \&z$$

$$f([l_1 : g_1] +\!\!+ \cdots +\!\!+ [l_n : g_n])$$
$$= d_{\overline{n}}([e(l_1, g_1), \ldots, e(l_n, g_n)])@(f(g_1) \oplus \cdots \oplus f(g_n)))$$

The difference is that d is applied before @

```
rc(l,g) = if l == a then [d : [&]]
          else if l == c then [&] else [l : [&]]
```


(a)
&
1
d
a
d
b
2
3
4
c
&y
srec(rc, foldr(++,[]))
(e)
(&,&)
d
d
d
d
b
d
(&,&y)
ε-elimination
(b)
(c)
(d)
(&,1)
(&,2)
(&,4)
(&,3)
(&,2)
(&,4)
(&,&y)
(&,3)
(&,3)
(&,2)