Ensuring Pointer Safety Through Graph Transformation

Adam Bakewell, Mike Dodds, Detlef Plump,
Colin Runciman
The University of York (UK)

Project *Safe Pointers by Graph Transformation*

Aim: more reliable pointer programming through

- a powerful type system for pointer-data structures (shapes)
- a static type-checker for operations upon shapes

Approach:

- *Graph reduction specifications* model shapes
- Graph transformation rules model operations upon shapes
- Automatic verification that operations are *shape safe*, that is, always preserve shapes

Project webpage: `http://cs-people.bu.edu/bake/spgt/`

Pointer structures as graphs

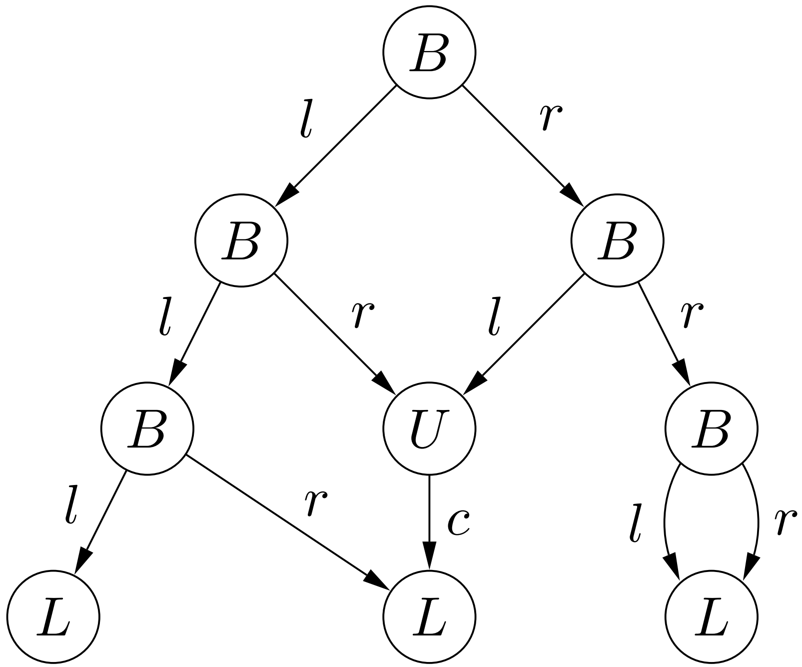
Graphs model tagged records connected by pointers

- **Tags have fixed sets of record fields**
- **Data is ignored**

Example: Pointer structure in C

```
struct B { data d;
           node *l;
           node *r; };
struct U { data d;
           node *c; };
struct L { data d; };
```

where `node` is the union of `B`, `U`, `L`



Signatures and Σ-graphs

- ***Signature*** $\Sigma = \langle \mathcal{C}_V, \mathcal{C}_N, \mathcal{C}_E, \mathrm{type}\colon \mathcal{C}_V \to 2^{\mathcal{C}_E} \rangle$
  - $\mathcal{C}_V$**: finite set of vertex labels (tags)**
  - $\mathcal{C}_N \subseteq \mathcal{C}_V$**: set of non-terminals**
  - $\mathcal{C}_E$**: finite set of edge labels (record fields)**
  - $\mathrm{type}(l)$**: set of record fields of a tag** l
- ***Σ-graphs***
  - **nodes may be unlabelled (in rules)**
  - **edges outgoing from a node labelled** l **have labels in** $\mathrm{type}(l)$
  - **different outgoing edges have different labels**
- ***Σ-total graphs*** **model pointer structures**
  - **all nodes are labelled**
  - **for a node labelled** l**, each label in** $\mathrm{type}(l)$ **is the label of an outgoing edge**

Σ-rules

Σ-rule $\langle L \supseteq K \subseteq R \rangle$

- L, K **and** R **are** Σ**-graphs**
- **unlabelled nodes in** L **are preserved, remain unlabelled and have the same outlabels in** L **and** R
- **preserved nodes that are not relabelled have the same outlabels in** L **and** R
- **relabelled nodes have a complete set of outlabels in** L **and** R**; labelled nodes in** L **must not be unlabelled in** R
- **deleted nodes have a complete set of outlabels**
- **allocated nodes are labelled and have a complete set of outlabels**

Σ-rules and direct derivations

Σ-rule $r = \langle L \supseteq K \subseteq R \rangle$**:** L, K, R **are Σ-graphs satisfying certain conditions on unlabelled nodes and "outlabels"**

Direct derivation $G \Rightarrow_r H$ **according to DPO approach with injective matching and relabelling:**

1. **Find injective morphism $L \to G$ satisfying the dangling condition,**
2. **remove image of $L - K$,**
3. **add $R - K$,**
4. **label the images of K-nodes with their labels in R.**

Theorem

Let $G \Rightarrow_r H$ be an application of a Σ-rule. Then
(1) ***G is a Σ-graph iff H is a Σ-graph, and***
(2) ***G is a Σ-total graph iff H is a Σ-total graph.***

Graph reduction specifications

Graph languages model pointer-data structures

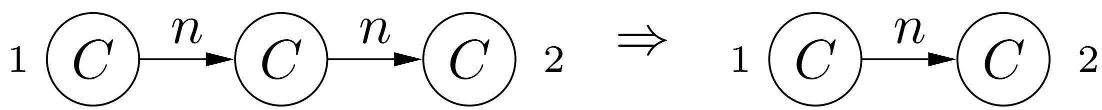
- ***Graph reduction specification*** **(GRS)** $S = \langle \Sigma, \mathcal{R}, Acc \rangle$
  - Σ**: signature**
  - $\mathcal{R}$**: finite set of** Σ**-rules**
  - Acc**, the** ***accepting graph*****:** Σ**-total graph irreducible by** $\mathcal{R}$
- **Specified** ***graph language***

$$\mathcal{L}(S) = \{G \mid G \Rightarrow^*_{\mathcal{R}} Acc \text{ and } G \text{ has no labels in } \mathcal{C}_N\}$$

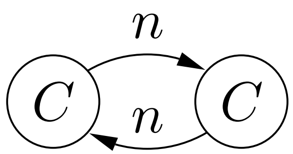
Note: all graphs in $\mathcal{L}(S)$ **are** Σ**-total**

Example: Cyclic lists

`Unlink`:

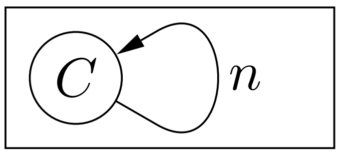


`TwoLoop`:

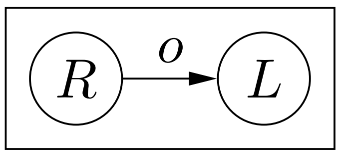


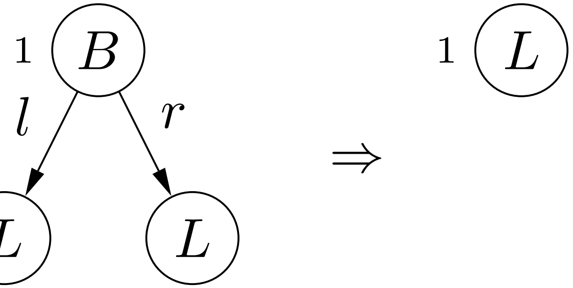
$\Rightarrow \; Acc$

$Acc =$



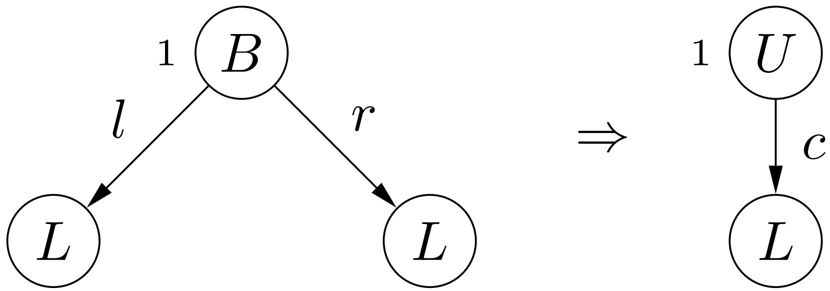
Example: Rooted binary trees

$Acc_{BT} =$ 

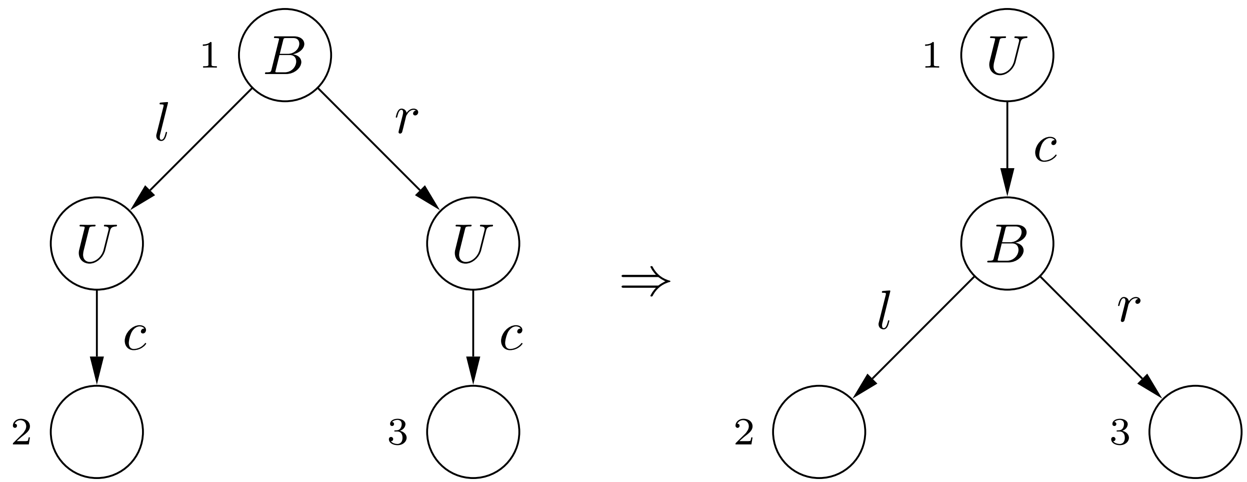
$\texttt{BtoL}:$ 

Example: Balanced binary trees

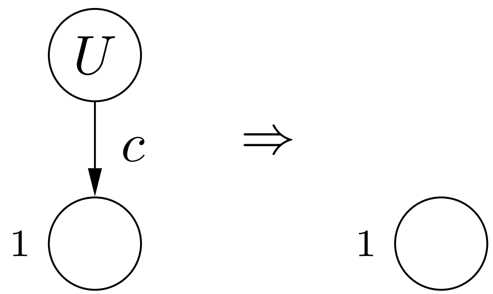
PickLeaf:



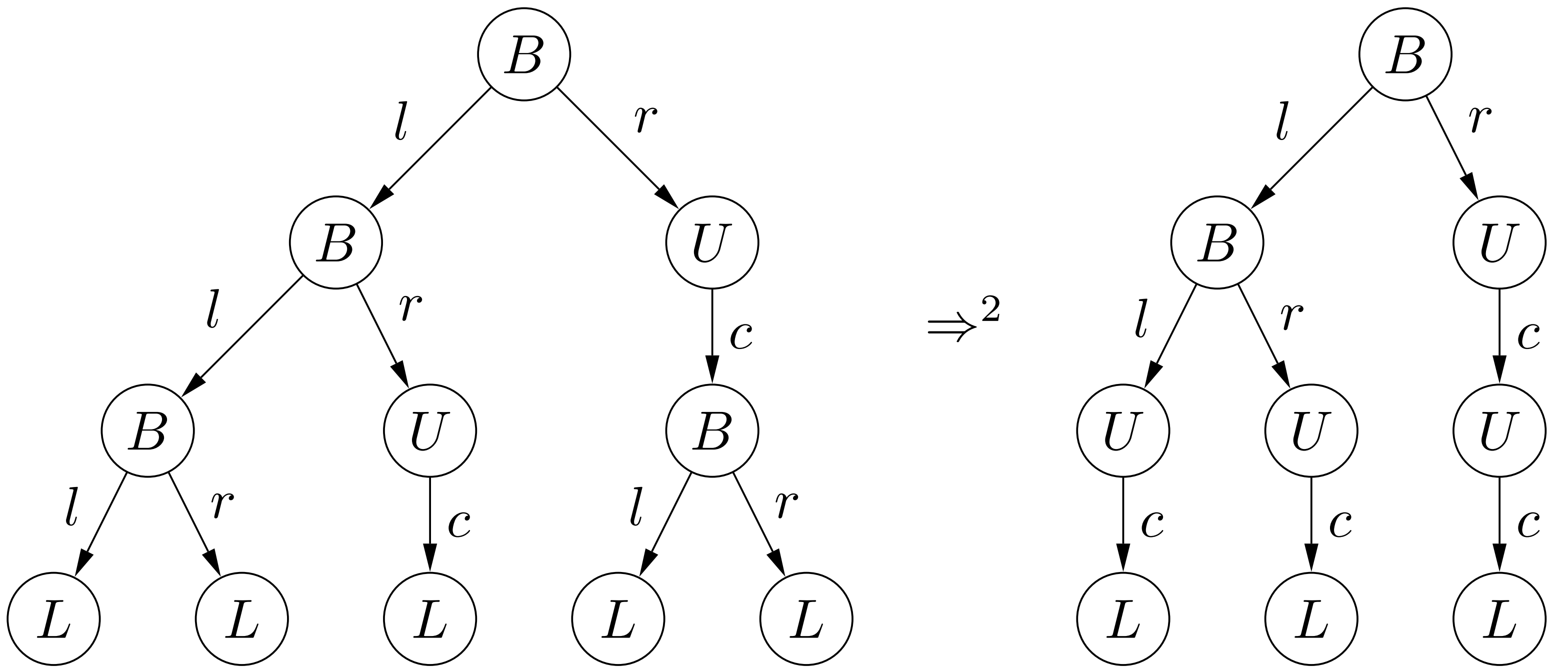
PushBranch:



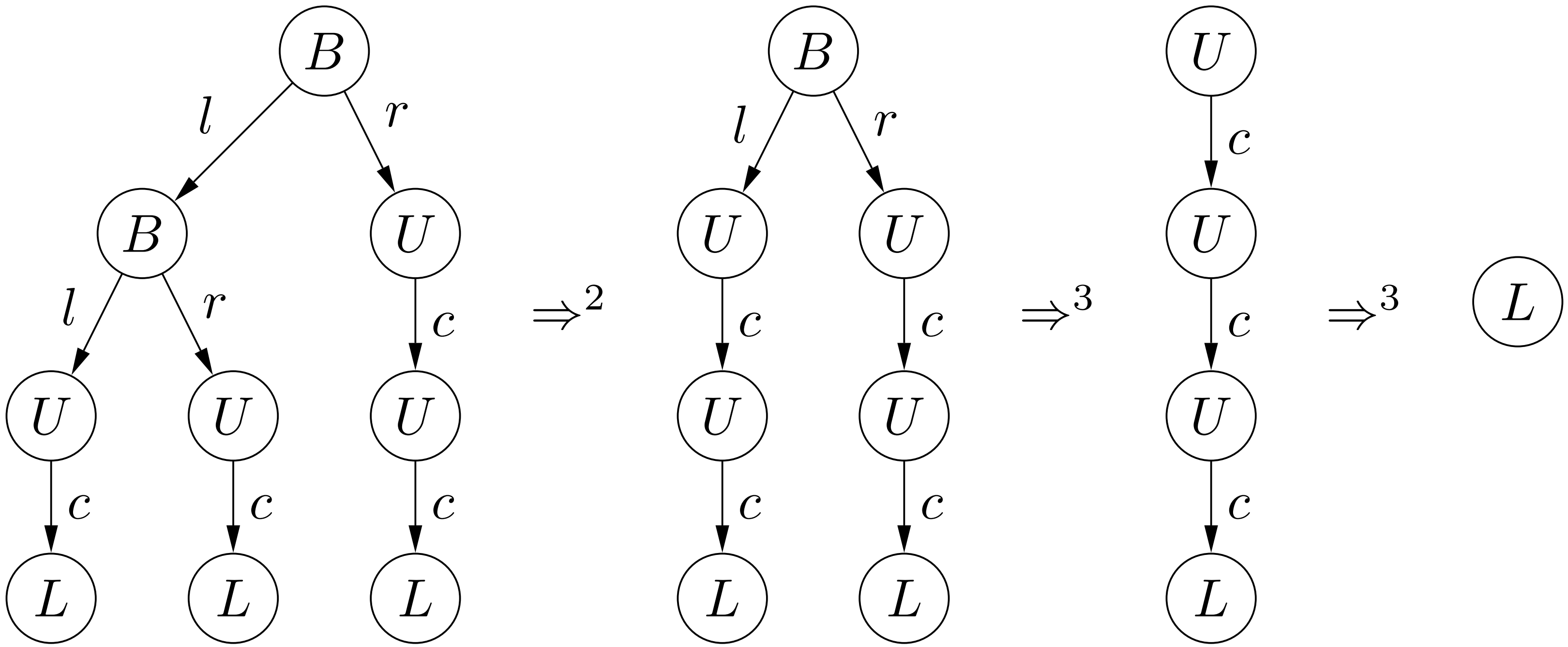
FellTrunk:



Example: Reduction of a balanced binary tree



Example: Reduction of a balanced binary tree (cont'd)



Membership checking (1)

Checking individual structures for language membership

- **to test and debug specifications**
- **to dynamically type-check structures generated by unsafe methods**

A GRS $\langle \Sigma, \mathcal{R}, Acc \rangle$ **is**

- ***terminating*** **if there is no infinite derivation** $G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \ldots$
- ***polynomially terminating*** **if there is a polynomial** p **such that for every derivation** $G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_n$**,** $n \leq p(\text{size}(G_0))$
- ***size-reducing*** **if for each rule** $\langle L \supseteq K \subseteq R \rangle$ **in** $\mathcal{R}$**,** $\text{size}(L) > \text{size}(R)$

Note:

- **size-reducing** $\Rightarrow$ **polynomially terminating** $\Rightarrow$ **terminating**
- **GRSs for (balanced) binary trees and cyclic lists are size-reducing**

Membership checking (2)

A GRS $\langle \Sigma, \mathcal{R}, Acc \rangle$ **is**

- ***closed*** **if for every step** $G \Rightarrow_{\mathcal{R}} H$, $G \Rightarrow^*_{\mathcal{R}} Acc$ **implies** $H \Rightarrow^*_{\mathcal{R}} Acc$
- ***confluent*** **if whenever** $H_1 \Leftarrow^*_{\mathcal{R}} G \Rightarrow^*_{\mathcal{R}} H_2$**, there are derivations** $H_1 \Rightarrow^*_{\mathcal{R}} H \Leftarrow^*_{\mathcal{R}} H_2$

Note:

- **confluent** $\Rightarrow$ **closed (converse does not hold)**
- **confluence of terminating GRSs can be checked by analyzing "critical pairs" of rules**
- **non-overlapping GRSs (no critical pairs) are always confluent**
- **GRSs for (balanced) binary trees and cyclic lists are confluent**

Membership checking (3)

A polynomially terminating and closed GRS is a *polynomial* GRS, a PGRS for short

Theorem

Membership in PGRS languages is decidable in polynomial time.

Decision procedure

Given a fixed PGRS $\langle \Sigma, \mathcal{R}, Acc \rangle$ and an input graph G,

1. **check that G only has terminal labels,**
2. **apply the rules from $\mathcal{R}$ (nondeterministically) as long as possible,**
3. **check that the resulting graph is isomorphic to Acc.**

PGRS Power

PGRSs are a powerful formalism for specifying pointer-data structures

- **They can specify important context-sensitive shapes, such as various forms of balanced trees.**
- **More PGRS examples: red-black trees, 2-3(-4) trees, AVL trees, binary DAGs, doubly-linked lists, rectangular grids, singly threaded trees.**

Shape Safety

A *Shape* is a class of graphs with common properties. E.g. binary trees, red-black trees, binary DAGs.

***Shape safety* means that a program ensures membership of the required shape. Program $P\colon S \times T$ is *shape-safe*:**

> **if applying P to structure G of shape S results in structure H, then H belongs to shape T.**

Note:

- **Partial correctness property**
- **P can temporarily violate the shape**

Insert into a binary search tree

Is the result of applying `insert()` to a binary tree also a binary tree?

```
BT *insert(datum d, BT *t) = {
  a := t;
  while branch(a) && a->data != d do
    if a->data > d
    then a := a->left
    else a := a->right;
  if leaf(a)
  then *a := branch{data=d,
                    left=leaf,
                    right=leaf};
  return(t)
}
```

`insert()` should not introduce:

- **sharing**
- **cycles**
- **pointers out of the tree**

Solution using graph transformation

Approach:

- **Pointer structures (without data) are graphs**
- **Shapes are graph languages defined by PGRS**
- **Pointer manipulations are modelled as graph transformations**
- **Check graph transformations w.r.t PGRS shapes**

Given program $P\colon S \times T$ **abstracted as graph transformation program** g_P**,** P **is shape safe if:**

$$G \in \mathcal{L}(S) \wedge G \rightarrow_{g_P} H \Rightarrow H \in \mathcal{L}(T)$$

Abstracting to graph transformations

Abstract program to a corresponding graph program.

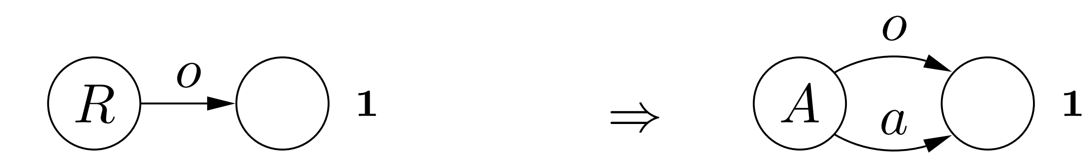
```
BT *insert(datum d, BT *t) = {
  a := t;
  while branch(a) && a->data != d do
    if a->data > d
    then a := a->left
    else a := a->right;
  if leaf(a)
  then *a := branch{data=d,
                    left=leaf,
                    right=leaf};
  return(t)
}
```

Insert : $BT \times BT$

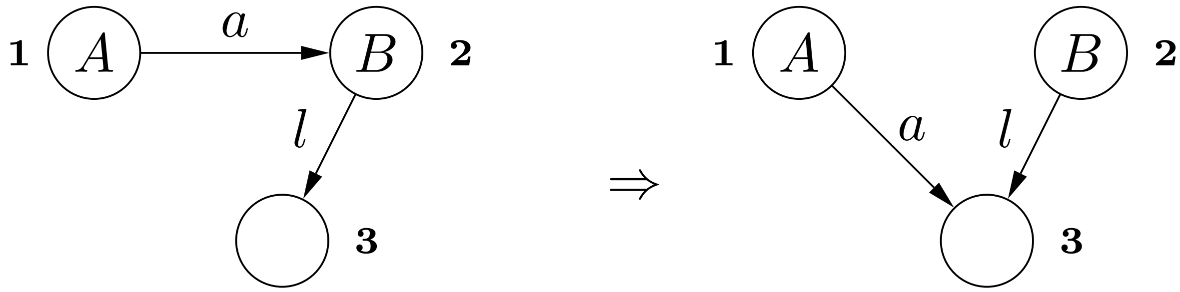
```
Insert =
  Begin;
  (GoLeft,
   GoRight)*;
  (Found,
    Ins)
```

Rules

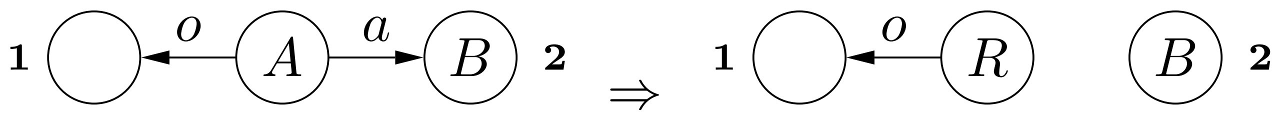
`Begin` :
$BT \times AT$ =



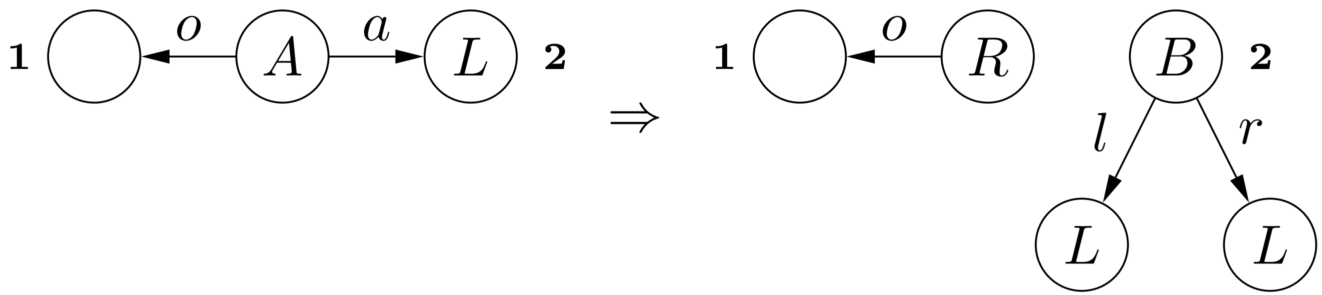
`GoLeft` :
$AT \times AT$ =



`Found` :
$AT \times BT$ =
$\{\mathbf{1{=}2},\mathbf{1{\neq}2}\}$

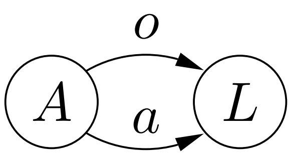


`Insert` :
$AT \times BT$ =
$\{\mathbf{1{=}2},\mathbf{1{\neq}2}\}$

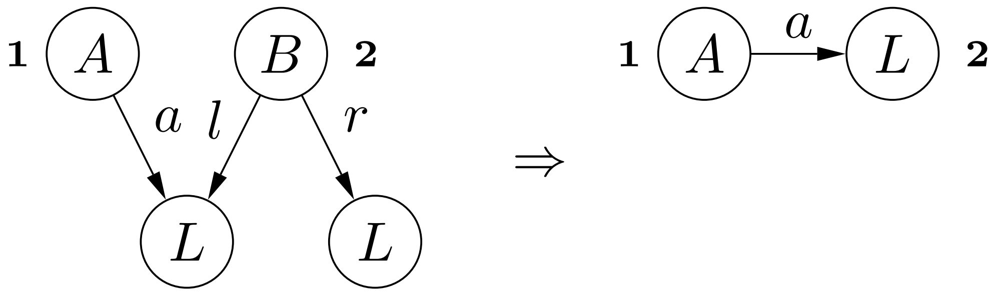


Binary tree with auxiliary pointer PGRS

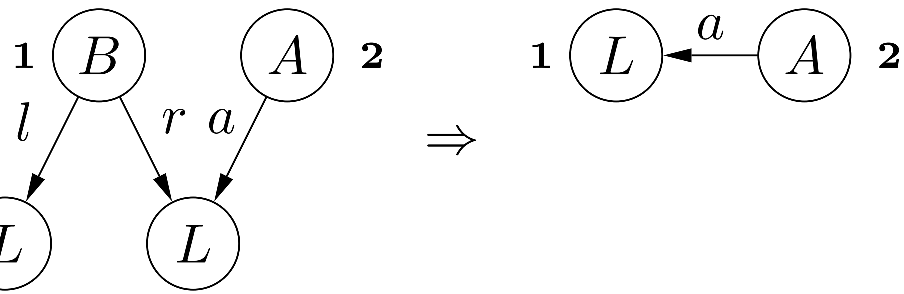
$Acc_{AT} =$



`BtoLl` :



`BtoLr` :



Check shape annotations

To check rule $r : S \times T$:

- **Consider every** ***graph context*** C**:** $C \cup L \Rightarrow_S^* Acc_S$
- **Split the reduction:**

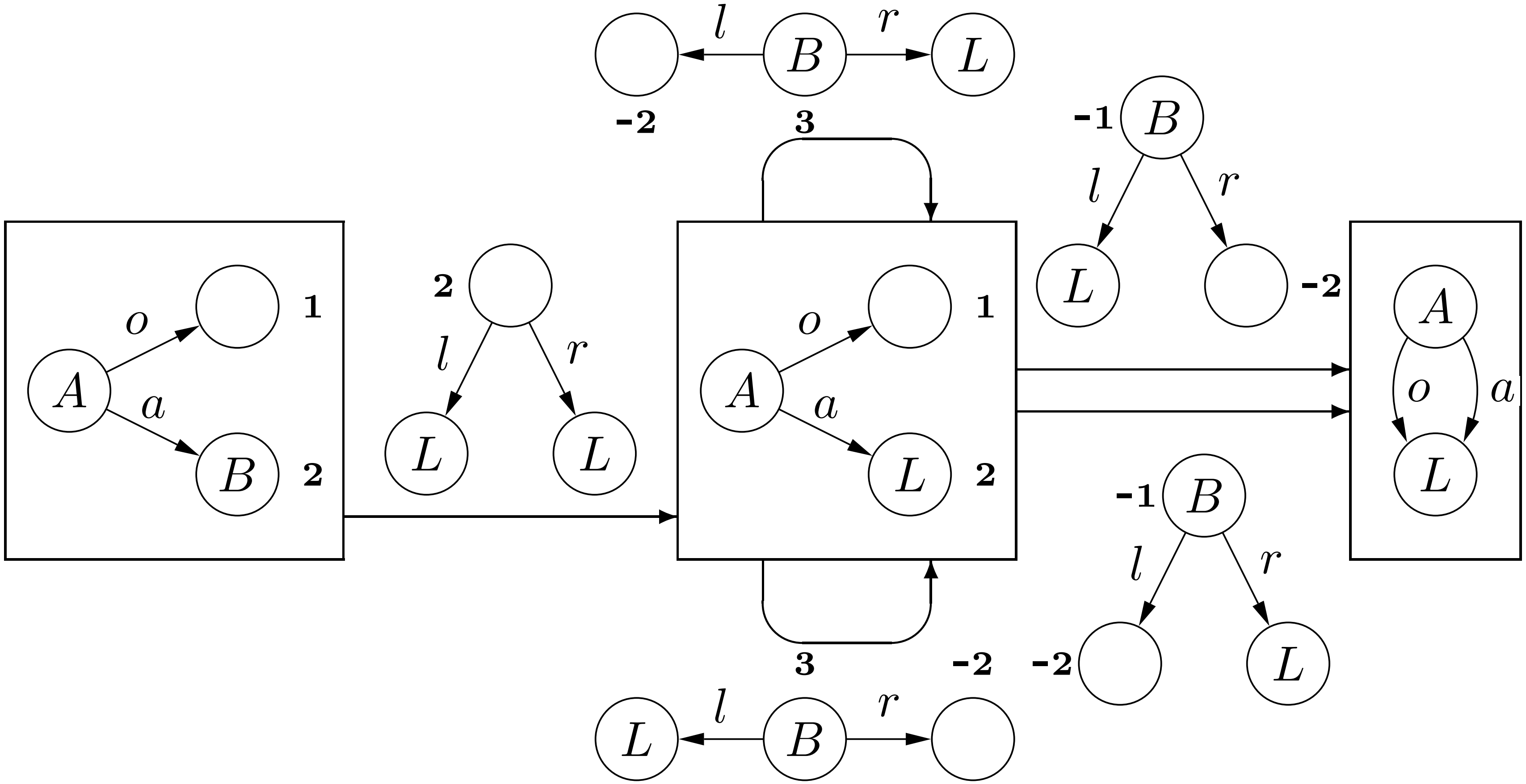
$$C_{i_j} \cup L \Rightarrow_S^* B_i \cup L \Rightarrow_S^* Acc_S$$

  - ***non-basic reductions*** $C_{i_j} \cup L \Rightarrow_S^* B_i \cup L$ **do not overlap with** L
  - ***basic reductions*** $B_i \cup L \Rightarrow_S^* Acc_S$ **overlap with** L

- **Check:**
  - $\bigwedge\{B_i \cup R \Rightarrow_T^* Acc_T\}$ **(language inclusion)**
  - $\bigwedge\{C_{ij} \cup R \Rightarrow_T^* B_i \cup R\}$ **(shape congeniality)**

Abstract Reduction Graph

Abstract Reduction Graph **(ARG) represents a set of basic contexts** $\{B_i\}$**.**

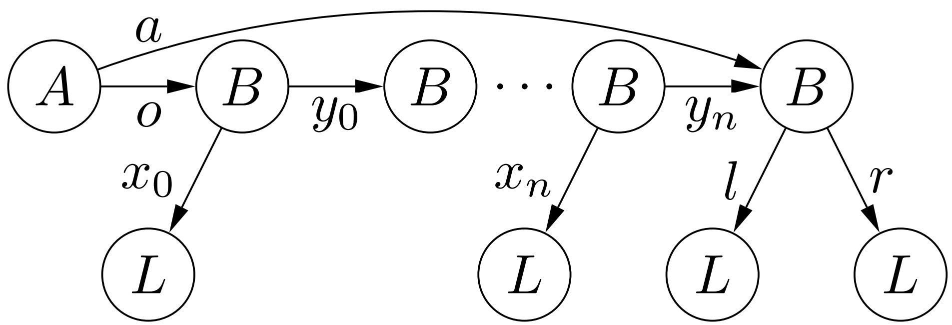


Meaning of an Abstract Reduction Graph

Meaning of an ARG:

- **edges are labelled with context graphs.**
- **nodes are labelled with the result of reductions.**
- **edge C exists between node G_1 and G_2 if $G_1 \cup C$ can be reduced to G_2 with some rule in $\mathcal{R}$**

Graphs represented by example ARG:



$$n \geq 0$$

$$(x_i, y_i) \in \{(l, r), (r, l)\}$$

Language inclusion

Language inclusion: all basic reductions for the LHS must also reduce to Acc when LHS is replaced with RHS.

Check:

- **Construct normalised ARGs for LHS and RHS**
- **Check that every context represented by left ARG is represented by right ARG (undecidable in general)**
- **In practice, check whether right ARG *includes* left ARG.**

Shape congeniality

All non-basic contexts $C_{i_j} \cup R$ reduce to $B_i \cup R$, where B_i is a LHS basic context.

Sufficient condition:

- **Trivial for rules with the same domain and range shapes.**
- **If the domain shapes differ, unshared rules cannot be used in non-basic reductions.**

Limitations of shape-safety approach

Shape safety is undecidable:

- **ARG construction is non-terminating in general.**
- **Even if ARG construction terminates, language inclusion test may fail.**

The checking algorithm fails for more complex shapes, including most non-context-free shapes.

We have no characterisation of shapes that can be checked.

C-GRS: Applying shape safety to C

Plan:

- **Extend C with analogues of**
  - **PGRSs, for defining shapes of pointer structures**
  - **graph transformation rules, for operations upon shapes**
- **C-GRS programs should manipulate pointers only by rules**
- **Abstract C-GRS to graph transformation for checking shape safety**
- **Translate C-GRS to C for execution**

$$\mathbf{C} \quad \underset{\text{translate}}{\longleftarrow} \quad \mathbf{C\text{-}GRS} \quad \underset{\text{abstract}}{\longrightarrow} \quad \textbf{graph transformation}$$

Example: C-GRS shape declaration

```
shape bt {
  signature {
    nodetype btroot {
      edge top, aux;
    }
    nodetype branchnode {
      edge l, r;
      int val;
    }
    nodetype leafnode {}
  }
  accept {
    root btroot rt;
    leafnode leaf;
    rt.top => leaf;
    rt.aux => leaf;
  }
  rules {
    moveaux2root;
    branch2leaf;
  }
}
```

Example: C-GRS function for binary tree insertion

```
bt *insert(int i, bt *b) {
  int t;
  bt_auxreset(b);
  while ( bt_getval(b, &t) ) {
    if ( t == i ) return b;
    else if ( t > i ) bt_goleft(b);
    else bt_goright(b);
  }
  bt_insert(b, &i);
  return(b);
}
```

```
transformer
bt_insert( bt *tree,
           int *inval ) {
  left (rt, n1) {
    root btroot rt;
    leafnode n1;
    rt.aux => n1;
  }
  right (rt, n1, l1, l2) {
    branchnode n1;
    leafnode l1, l2;
    rt.aux => n1;
    n1.l => l1;
    n1.r => l2;
    n1.val = *inval;
  }
}
```

Rooted graph transformation

Two problems:

- **Graph transformation is non-deterministic whereas C is deterministic**
- **Matching of graph transformation rules is too slow: requires polynomial time for a given set of rules**

Solution: rooted shapes and rules

- **Shape members and left-hand sides of transformers contain at least one distinguished root node; distinct roots have distinct node types**
- **Every left-hand node of a transformer must be reachable from some root; transformers do not delete or add roots**
- **Matching is deterministic and requires only constant time: comparison starts at the roots and proceeds uniquely along edges**

Translating C-GRS to C

- **Node types (of non-roots) are translated to structure declarations**

```
nodetype branchnode {
  edge l, r;
  int val;
}
```

$\longmapsto$

```
struct branchnode {
  bt_node *l;
  bt_node *r;
  int val;
}
```

which are wrapped into a single union (`bt_node`)

- **Transformers are translated to C functions which first match the left-hand side and then transform it into the right-hand side**
- **Dangling condition is implemented by reference counting**
- **Transformer has no structural effect if matching fails**

Correctness of the translation

$$\textbf{C} \;\xleftarrow[\textbf{translate}]{\mathcal{C}}\; \textbf{C-GRS} \;\xrightarrow[\textbf{abstract}]{\mathcal{G}}\; \textbf{graph transformation}$$

$$\begin{array}{ccccl}
G & \xrightarrow{\mathcal{G}[\![F]\!]} & G' & & \textbf{Σ-total graphs} \\
\uparrow \alpha_\Sigma & = & \uparrow \alpha_\Sigma & & \\
S & \xrightarrow{\mathcal{C}[\![F]\!]} & S' & & \textbf{C pointer-structures}
\end{array}$$

- F **is a transformer over signature** Σ
- S **is a pointer structure consistent with** Σ
- α_Σ **abstracts pointer structures consistent with** Σ **to** Σ**-total graphs**
- **Failure of** $\mathcal{G}[\![F]\!]$ **implies** $G = G'$

Conclusions and Outlook

Prototype of the system has been implemented:

- **Implementation of the checking algorithm**
- **Compiler from C-GRS to C**

Further work:

- **Extending the power of the checking algorithm.**
- **More C-like syntax for application language.**

Project webpage: `http://cs-people.bu.edu/bake/spgt/`