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

```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 $\Sigma$-graphs

- **Signature** $\Sigma = \langle C_V, C_N, C_E, \text{type}: C_V \rightarrow 2^{C_E} \rangle$
  - $C_V$: finite set of vertex labels (tags)
  - $C_N \subseteq C_V$: set of non-terminals
  - $C_E$: finite set of edge labels (record fields)
  - $\text{type}(l)$: set of record fields of a tag $l$

- **$\Sigma$-graphs**
  - nodes may be unlabelled (in rules)
  - edges outgoing from a node labelled $l$ have labels in $\text{type}(l)$
  - different outgoing edges have different labels

- **$\Sigma$-total graphs** model pointer structures
  - all nodes are labelled
  - for a node labelled $l$, each label in $\text{type}(l)$ is the label of an outgoing edge
\(\Sigma\)-rules

**\(\Sigma\)-rule** \(\langle L \supseteq K \subseteq R \rangle\)

- \(L, K\) and \(R\) are \(\Sigma\)-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 $L \rightarrow G$ 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)** $S = \langle \Sigma, \mathcal{R}, \text{Acc} \rangle$
  - $\Sigma$: signature
  - $\mathcal{R}$: finite set of $\Sigma$-rules
  - Acc, the **accepting graph**: $\Sigma$-total graph irreducible by $\mathcal{R}$

- **Specified graph language**

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

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


**Example: Cyclic lists**

Unlink: \[ 1 \xrightarrow{n} 1 \xrightarrow{n} 2 \Rightarrow 1 \xrightarrow{n} 2 \]

TwoLoop: \[ \xrightarrow{n} \quad \xrightarrow{n} \Rightarrow \text{Acc} \]

\[ \text{Acc} = \]

\[ \xrightarrow{n} \]

Example: Rooted binary trees

\[ Acc_{BT} = \begin{array}{c}
\begin{array}{c}
\circlearrowright R
\end{array}
\rightarrow
\begin{array}{c}
L
\end{array}
\end{array} \]

\[ B_{toL} : \]

\[ \begin{array}{c}
\begin{array}{c}
1
\end{array}
\end{array} B \]

\[ \begin{array}{c}
\begin{array}{c}
l
L
\end{array}
\end{array} \]

\[ \begin{array}{c}
\begin{array}{c}
r
L
\end{array}
\end{array} \]

\[ \Rightarrow \]

\[ 1 L \]
Example: Balanced binary trees

PickLeaf:

```
    B
   / \ 1
  l   r
 L   L
```

⇒

```
    U
   / \ 1
  c   c
 L   L
```

PushBranch:

```
    B
   / \ 1
  l   r
 U   U
```

⇒

```
    B
   / \ 1
  l   r
 U   U
```

FellTrunk:

```
    U
   / \ 1
  c   c
```

⇒

```
   U
  / \ 1
 c   c
```

```
```

```
```

```
```

Example: Reduction of a balanced binary tree

\[
\begin{array}{c}
\text{B} \quad \text{L} \\
\text{B} \quad \text{U} \\
\text{L} \\
\end{array}
\quad \Rightarrow^2
\begin{array}{c}
\text{B} \\
\text{U} \\
\text{L} \\
\end{array}
\]
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
A GRS $\langle \Sigma, \mathcal{R}, \text{Acc} \rangle$ is

- **closed** if for every step $G \Rightarrow_{\mathcal{R}} H$, $G \Rightarrow^{*}_{\mathcal{R}} \text{Acc}$ implies $H \Rightarrow^{*}_{\mathcal{R}} \text{Acc}$

- **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 $\langle \Sigma, \mathcal{R}, \text{Acc} \rangle$ 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 $\text{Acc}$. 
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.
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 $P: S \times T$ 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 \texttt{insert()} to a binary tree also a binary tree?

\begin{verbatim}
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)
}
\end{verbatim}

\texttt{insert()} should not introduce:

\begin{itemize}
    \item sharing
    \item cycles
    \item pointers out of the tree
\end{itemize}
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 $P: S \times T$ abstracted as graph transformation program $g_P$, $P$ is shape safe if:

$$G \in \mathcal{L}(S) \land G \xrightarrow{g_P} H \Rightarrow H \in \mathcal{L}(T)$$
Abstracting to graph transformations

Abstract program to a corresponding graph program.

\[
 BT \ast \text{insert}(\text{datum } d, BT \ast t) = \{
 \hspace{1cm}
a := t; \\
 \hspace{1cm}\text{while branch}(a) \&\& a->data != d \text{ do} \\
 \hspace{1cm}\hspace{1cm}\text{if } a->data > d \\
 \hspace{1cm}\hspace{1cm}\hspace{1cm}\text{then } a := a->left \\
 \hspace{1cm}\hspace{1cm}\hspace{1cm}\text{else } a := a->right; \\
 \hspace{1cm}\text{if } \text{leaf}(a) \\
 \hspace{1cm}\hspace{1cm}\text{then } *a := \text{branch}\{\text{data}=d, \\
 \hspace{1cm}\hspace{1cm}\hspace{1cm}\text{left}=\text{leaf}, \\
 \hspace{1cm}\hspace{1cm}\hspace{1cm}\text{right}=\text{leaf}\}; \\
 \hspace{1cm}\text{return}(t)
 \}
\]

\[\text{Insert} : BT \times BT\]
\[\text{Insert} = \] 
\[\text{Begin};\] 
\[\text{GoLeft},\] 
\[\text{GoRight})*;\] 
\[\text{Found},\] 
\[\text{Ins}\]
Rules

\[
\text{Begin:} \quad BT \times AT \quad = \quad \begin{array}{c}
R \\
A \\
B
\end{array} \xrightarrow{o} \begin{array}{c}1 \\
2 \\
3 \end{array} \quad \Rightarrow \quad \begin{array}{c}
A \\
B
\end{array} \xrightarrow{a} \begin{array}{c}1 \\
2 \end{array}
\]

\[
\text{GoLeft:} \quad AT \times AT \quad = \quad \begin{array}{c}
1 \quad A \\
2 \quad B
\end{array} \xleftarrow{l} \begin{array}{c}3 \end{array} \quad \Rightarrow \quad \begin{array}{c}
1 \quad A \\
2 \quad B
\end{array} \xleftarrow{l} \begin{array}{c}3 \end{array}
\]

\[
\text{Found:} \quad AT \times BT \quad = \quad \begin{array}{c}
1 \quad o \quad A \\
2 \quad a \quad B
\end{array} \xrightarrow{l} \begin{array}{c}2 \end{array} \quad \Rightarrow \quad \begin{array}{c}
1 \quad o \quad R \\
2 \quad B
\end{array}
\]

\[
\{1=2, 1\neq 2\}
\]

\[
\text{Insert:} \quad AT \times BT \quad = \quad \begin{array}{c}
1 \quad o \quad A \\
2 \quad a \quad L
\end{array} \xrightarrow{l} \begin{array}{c}2 \end{array} \quad \Rightarrow \quad \begin{array}{c}
1 \quad o \quad R \\
2 \quad L
\end{array}
\]

\[
\{1=2, 1\neq 2\}
\]
Binary tree with auxiliary pointer PGRS

\[ \text{Acc}_{AT} = \begin{array}{c}
A \quad o \\
a \quad L
\end{array} \]

BtoLl:

\[ \begin{array}{c}
1 \quad A \\
\quad a \\
\quad l \\
\quad \quad L \\
\quad \quad r \\
\quad \quad L \\
\quad 2 \quad B
\end{array} \quad \Rightarrow \quad \begin{array}{c}
1 \quad A \\
\quad a \\
\quad L \\
\quad 2 \quad L
\end{array} \]

BtoLr:

\[ \begin{array}{c}
1 \quad B \\
\quad l \\
\quad \quad L \\
\quad \quad \quad L \\
\quad \quad \quad r \\
\quad \quad \quad a \\
\quad \quad \quad A \\
\quad \quad \quad 2 \\
\quad 1 \quad L
\end{array} \quad \Rightarrow \quad \begin{array}{c}
1 \quad L \\
\quad a \\
\quad A \\
\quad 2 \quad L
\end{array} \]
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**:

  $$C_{ij} \cup L \Rightarrow^*_S B_i \cup L \Rightarrow^*_S Acc_S$$

  - *non-basic reductions* $C_{ij} \cup L \Rightarrow^*_S B_i \cup L$ 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 (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:

\[
\begin{align*}
(x_i, y_i) &\in \{(l, r), (r, l)\} \\
n &\geq 0
\end{align*}
\]
Language inclusion: all basic reductions for the LHS must also reduce to \textit{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 \textit{includes} left ARG.
Shape congeniality

All non-basic contexts $C_{ij} \cup R$ 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

\[
\begin{align*}
C & \quad \xleftarrow{\text{translate}} \quad \text{C-GRS} \quad \xrightarrow{\text{abstract}} \quad \text{graph transformation}
\end{align*}
\]
Example: C-GRS shape declaration

```c
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

```c
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

```c
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

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

  into

  ```c
  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

\[ C \xleftarrow{\text{translate}} \text{C-GRS} \xrightarrow{\text{abstract}} \text{graph transformation} \]

\[ G \xrightarrow{\mathcal{G}[F]} G' \quad \Sigma\text{-total graphs} \]

\[ S \xrightarrow{\mathcal{C}[F]} S' \quad \text{C pointer-structures} \]

- \( F \) is a transformer over signature \( \Sigma \)
- \( S \) is a pointer structure consistent with \( \Sigma \)
- \( \alpha_\Sigma \) abstracts pointer structures consistent with \( \Sigma \) to \( \Sigma\)-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/