Optimisation of Control Flow Graphs using Graph Rewriting

Maik Marschner

First examiner: Prof. Dr. F.-E. Wolter
Second examiner: Prof. Dr. M. Rohs
Advisor: M. Sc. M. Klein

Leibniz Universität Hannover
Institut für Mensch-Maschine-Kommunikation
Lehrstuhl Graphische Datenverarbeitung

www.welfenlab.de

28th September 2016

Leibniz
Universität
Hannover

Outline

1. Motivation
2. Introduction to Buggy
3. Graph rewrite systems
4. Rewrite rules
5. Recursion
6. Results
7. Outlook

Motivation

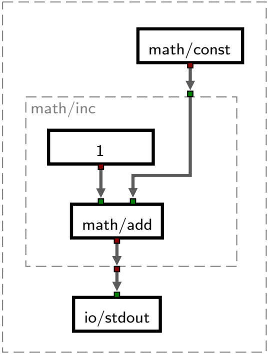
- Buggy is a language-agnostic programming platform currently in development at the *Welfenlab*

Functional code (e.g. *Lisgy*)		Another language
Abstract problem descriptions	$\rightarrow$ Buggy graph $\rightarrow$	$\rightsquigarrow$ Machine code
Formulas		Graph images
...		...

- Optimisation of programs should not depend on the used programming languages
- Idea: Optimise programs by transforming Buggy graphs

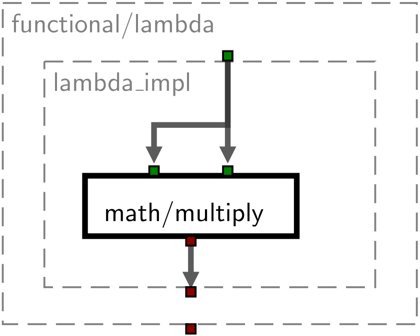
Buggy graphs

- Acyclic, directed port graphs
- Contain the entire semantic of the program (data flow and control flow)
- Composed of *atomic nodes* and *compound nodes*



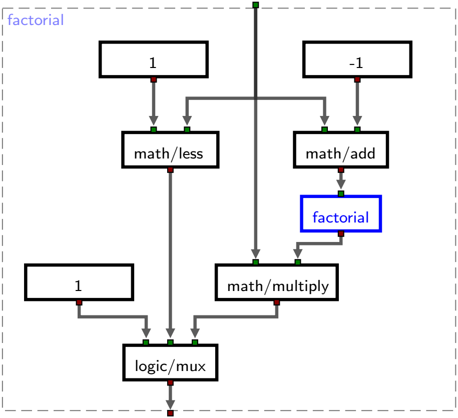
Lambda functions

- Use functions as values
- Bind arguments with `functional/partial` node
- Evaluate with `functional/call` node



Recursive functions

- Possible by using *compound nodes*
- Recursive call by a node with ID of the recursive compound



Automatic optimisation using Graph Rewriting

- Graph rewrite system: a set of graph rewrite rules
- Graph rewrite rule:
  - Graph g_l that can be replaced by the rule
  - Graph g_r that the matched graph is replaced with
  - Embedding description M that controls how g_r is embedded into the existing graph
- **`while`** any rule r matches **`do`** apply r
- g_l may be given implicitly, g_r may depend on g_l
  $\rightarrow$ use two functions `match(g)` and `rewrite(g, match)`
- Problems: termination, confluence, performance (subgraph matching is NP-complete)

Problems of graph rewrite systems

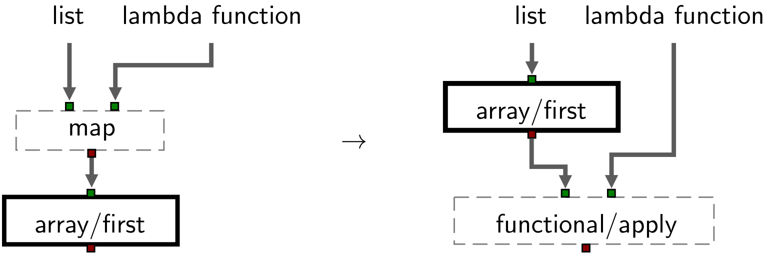
- Termination and Confluence
  - Rules may produce matches of other rules
    $\rightarrow$ cyclic rule application
  - More than one rule may match the same subgraph
    $\rightarrow$ resulting graph is non-deterministic
  - Solved by *stratification*: Put conflicting rules into different graph rewrite systems, successively apply them on the graph
- Performance
  - Buggy graphs are *labeled*, efficient subgraph matching possible
  - Fast replacement possible with efficient data structures

Optimisation in three phases

1. High-level optimisation
   Compound nodes still exist, can be re-ordered and removed
2. Optimisation
   Compound nodes are removed, allows rewrite rules that cross the borders of compound nodes
3. Cleanup
   Undo some rewrite rules that produced overhead to make other rewrite rules possible

High-level rewrite rules

- Some compound nodes contain semantic information (e.g. `map`, `filter`, `sort`)
- Use this information and create more efficient programs that produce the same result



`(first (map list fn))` → `(apply fn (first list))`

More high-level rewrite rules

- Filter lists before sorting them
  `(filter (sort list) filterFunction)`

More high-level rewrite rules

- Filter lists before sorting them

```
(filter (sort list) filterFunction)
                 ↓
(sort (filter list filterFunction))
```

More high-level rewrite rules

- Filter lists before sorting them
- Concatenate lambda functions
  `(map (map list fn1) fn2)`

More high-level rewrite rules

- Filter lists before sorting them
- Concatenate lambda functions

```
(map (map list fn1) fn2)
            ↓
(map list (lambda [x] (fn2 (fn1 x))))
```

More high-level rewrite rules

- Filter lists before sorting them
- Concatenate lambda functions
- Eliminate sorting
  `(array/first (sort list))`

More high-level rewrite rules

- Filter lists before sorting them
- Concatenate lambda functions
- Eliminate sorting

```
(array/first (sort list))
            ↓
(minimum list)
```

More high-level rewrite rules

- Filter lists before sorting them
- Concatenate lambda functions
- Eliminate sorting
- More rules will be possible, not only for lists
    - Matrices
    - String manipulation

Low-level optimisation

- Optimisation across compound nodes
- Semantical information from atomic node types and graph structures
- All unnecessary compound nodes are removed
  $\rightarrow$ simplifies rewrite rules
- Compound nodes may become unnecessary by applying rules
  $\rightarrow$ re-check after every rewrite

Optimisation rules

- Constant folding
- Algebraic optimisation rules
- Dead code removal
- Optimise functional programming elements
- Optimise recursive functions

Constant folding & algebraic rules

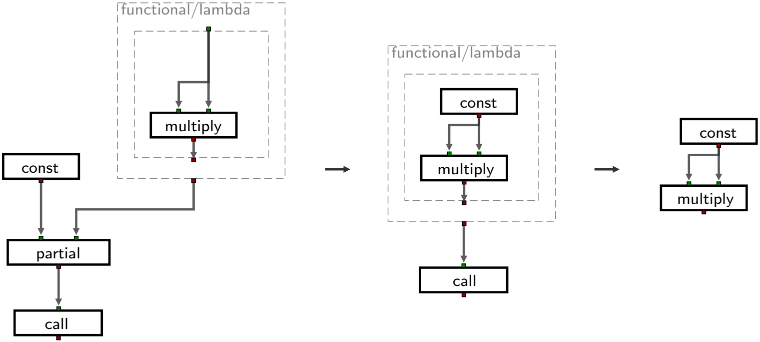
- Optimiser evaluates common operations with constants and inserts the result
  e.g. addition, multiplication, integer to string conversion
- Boolean algebra (negation, De Morgan's law)
  $\rightarrow$ also works with non-constants
- Remove multiplication by 1, additon with 0, disjunction with `true` etc.

Dead code removal

- Remove lambda function nodes without successors
- Remove nodes without successors, if they have no side effects
- Remove unused predecessor branches of `logic/mux` nodes if the condition is a constant

Optimise functional programing elements

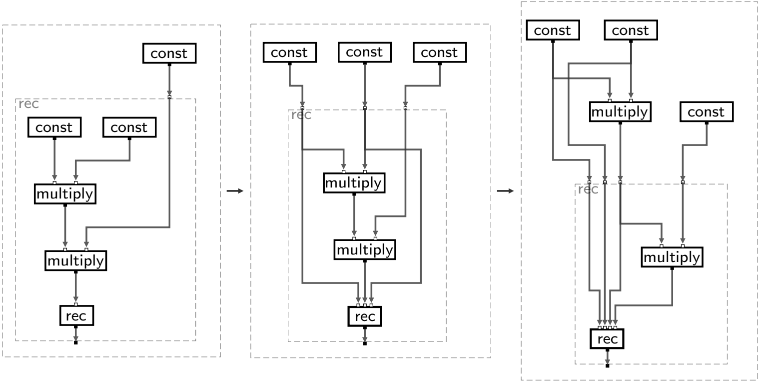
- Create lambda functions that contain bound arguments
- Inline lambda functions



Recursion

- Buggy doesn't have loops $\rightarrow$ recursion is very common
- Rewrite rules:
  - Move nodes out of the recursive compounds if they always have the same input values

Recursion


const
rec
const
const
multiply
multiply
rec
const
const
const
rec
multiply
multiply
rec
const
const
multiply
const
rec
multiply
rec

Recursion

- Buggy doesn't have loops $\rightarrow$ recursion is very common
- Rewrite rules:
  - Move nodes out of the recursive compounds if they always have the same input values

Recursion

- Buggy doesn't have loops $\rightarrow$ recursion is very common
- Rewrite rules:
  - Move nodes out of the recursive compounds if they always have the same input values
  - Transform tail recursion to loops
  - Transform recursion to tail recursion, if possible

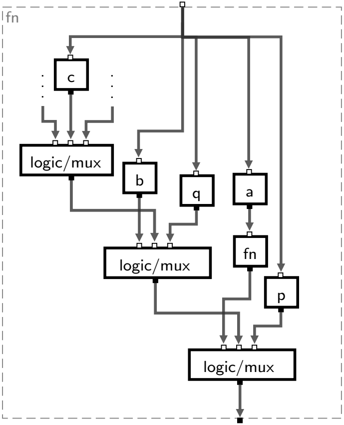
Tail recursion

Recursive function where the recursive call is always the last operation before returning

```
function fn(x) {
  start(x) // side effects
  if (p(x)) {
    return fn(a(x));
  } else if (q(x)) {
    return b(x);
  } else if (...) {
    ...
  } else {
    return c(x);
  }
}
```

Tail recursion in Buggy

Recursive compound node that ends with a `mux` chain, recursive calls only right before `mux` nodes



- `a`, `b`, `c` calculate the arguments for the recursive calls or the return value
- `p`, `q` are conditions
- Optimiser extracts these functions into lambda functions

What the code generator does

Extracted argument generators: a, b, c

Extracted conditions: p, q

```
function fn(x) {
  while (true) {
    if (p(x)) {
      x = a(x);
      continue; // was a recursive call
    }
    if (q(x)) {
      x = b(x);
      break;     // was a return
    }
    ...
  }
  return x;
}
```

Transform recursion to tail recursion

Make recursive functions tail-recursive by using an *accumulator*

- Last operation before return is associative, recursive calls only before this operation
- Neutral element must exist for the operation
- Factorial function:
  `(if (< 1 n) (* n (fac (- n 1))) 1)`

Transform recursion to tail recursion

Make recursive functions tail-recursive by using an *accumulator*

- Last operation before return is associative, recursive calls only before this operation
- Neutral element must exist for the operation
- Factorial function:

```
(if (< 1 n) (* n (fac (- n 1))) 1)
(if (< 1 n) (factr (- n 1) (* acc n)) acc)
```

Results

- Optimiser for Buggy that uses graph rewriting
- Common high-level operations can be optimised (only lists for now)
- Overhead of using functional programming (especially when using Lisgy) can often be reduced
- Tail recursion elimination → loops without stack overflow

Outlook

- Optimiser is already embedded in the Buggy toolchain
- Add more rewrite rules as Buggy gets new features and components
- Algebraic optimisation has more possibilities, allows to specify rewrite rules in a more general way
- Database or description format for rewrite rules and properties of components