Bachelor Thesis
Final Presentation
From Jinja Bytecode to Computation Graphs

Name: Mario Pirker (0919614)

Computationial Logic
Institute of Computer Science
University of Innsbruck

7. Oktober 2012

Supervisors: Assoz.-Prof. Dr. Georg Moser
BSc. Michael Schaper

Outline

Aim Of My Thesis

Vision:

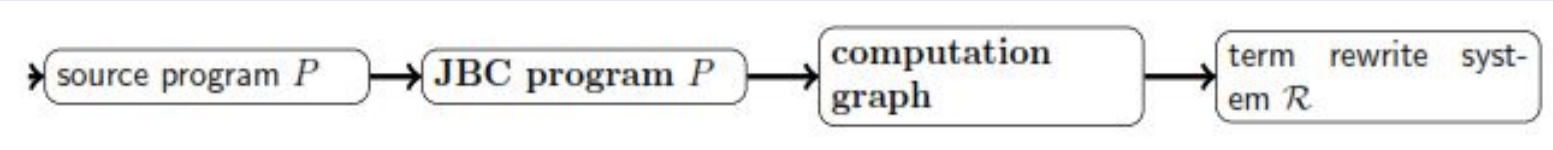

Abbildung: Vision of the Big Picture.

My part:

- small transformation tool for Jinja bytecode programs.
- input = Jinja bytecode program
- output = finite computation graph

Abstract States I

Why abstraction?

- all reachable states inside a program can not be computed because those are infinite.
- using suitable abstractions, the computation graph becomes finite.

Difference between concrete and abstract states:

- abstract class variable, abstract integer value, abstract boolean value.
- sharing information.
- concrete state: no abstract variables, all addresses on the heap can't be shared further.

Abstract States II

Instance concept:

- important when analysing loops.
- when reaching a node again, check whether the new state is an instance of the state visited before.

Example:

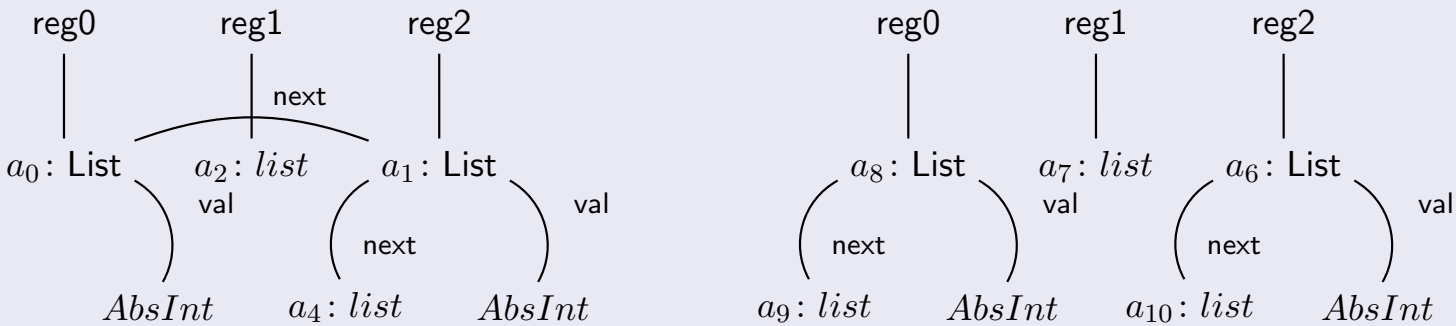

Abbildung: States *s27* and *s49* from List example.

Computation Graph I

A state of the Jinja Virtual Machine is a pair consisting of heap and frame.

Heap:

- global memory.
- mapping from addresses to objects.
- object has a class name and a fieldtable(= mapping from (class name, fieldid) to a Jinja value) or is an abstract class variable.

Frame:

- represents execution environment of a method.
- (opstack , registers, class name, method name, pc)

Computation Graph II

Brief description:

- $G = (V, E)$. V are abstract states and E are the edges.
- abstract states: set of concrete states of the JVM.
- edges: symbolic evaluation together with refinement and abstraction steps.
- directed and acyclic graph.
- representation of all runs of a JBC program.

Computation Graph III

Restrictions:

- non-recursive methods.
- tree-shaped objects.

Why computation graphs?

The motivation behind computation graphs is that they can handle all aspects of Jinja that cannot be expressed easily in *term rewriting systems*.

Computation Graph IV

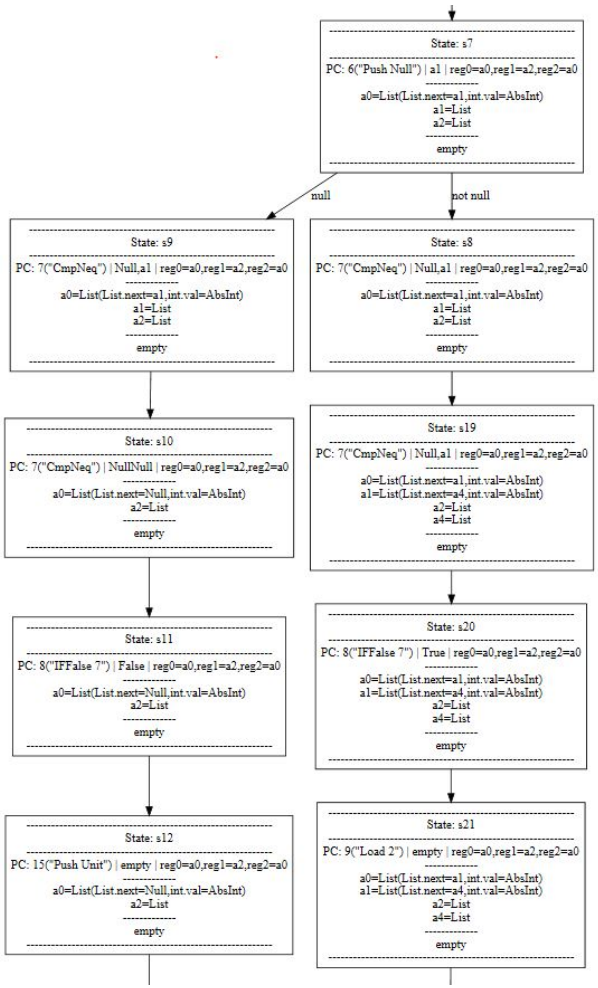

Abbildung: Parts of the Computation Graph for the List example.

Transformation Tool I

Introduction:

- written in Haskell.
- binary for Linux OS.
- converts JBC program into computation graph.

Approach:

1. execute instruction from the instruction list until program reachs a loop.
2. abstract states (summarizing) and perform loop again.
3. program termintes when the PC reaches end of the instruction list or an instance has been found during analysis of a loop.

Transformation Tool II

Additional features:

- provides interface between the computation graph data type and GraphViz API.
- cycle detection inside the heap.
- integrated parser that transforms the JBC program into a well-defined data type used for internal representation by the computation graph.

Example I

Source-code:

```
class List{
List next;
int val;

unit append(List ys){
        List cur = this;
        while(cur.next!=null){
                cur = cur.next;
        };
        cur.next = ys;
}
}
```

Abbildung: The Jinja source code for the method *append* declared in the class *List*.

Example II

Bytecode:

```
0:  Load 0                    12: Push unit
1:  Store 2                   13: Pop
2:  Push unit                 14: Goto -10
3:  Pop                       15: Push unit
4:  Load 2                    16: Pop
5:  Getfield next List        17: Load 2
6:  Push null                 18: Load 1
7:  CmpNeq                    19: Putfield next List
8:  IfFalse 7                 20: Push unit
9:  Load 2                    21: Return
10: Getfield next List
11: Store 2
```

Abbildung: The List JBC-program.

Example III

Live presentation

Difficulties

- erros in theory led to problems in implemenation.
- abstraction (implementation of summarizing).
- implementing unification between addresses on the heap.
- implementing morphism between states.
- termination of the program.

Schedule

- 13th March 2012: Initial presentation
  - March: Reading papers
  - April: JBC instruction set, refinements
  - May - June: Morphism, unification and instance
  - July - August: Summarizing
- 3th September 2012: End of programing
  - September: Writing thesis, testing, final presentation
- 16th October 2012: Final presenation

Conclusion

Achievements:

- transformation of JBC program into finite computation graph.
- implementation of the theoritical approach from [1].
- generate finite output with the help of abstraction (summarizing).

Future work:

- improve performance of the tool.
- output of Dot file representation.
- use of recursive methods in the program.
- implementation of transformation step from computation graph to TRS.

Bibliography

G. Moser and M. Schaper.
A complexity preserving transformation from jinja bytecode to rewrite systems.
CoRR, cs/PL/1204.1568, 2012.

End

Thank you for your attention!
Questions?