WWU
MÜNSTER

ERCIS
European
Research
Center for
Information
Systems

Structured Traversal of Search Trees in Constraint-logic Object-oriented Programming

C A U
Kiel University
Christian-Albrechts-Universität zu Kiel

Jan C. Dageförde (WWU Münster), Finn Teegen (CAU Kiel)

September 11, 2019

Constraint-logic Object-oriented Programming

- Imperative/OO programming, combined with
- features from (constraint) logic programming

  $\Rightarrow$ Non-deterministic execution of imperative programs

Example, in “Münster Logic-Imperative Language” (Muli)

```
boolean flipCoin() {
  int coin free;
  if (coin == 0) return false;
  else return true; }
```

Motivation

- Branching $\rightsquigarrow$ Choice points $\rightsquigarrow$ Implicit search tree
- Depth unknown prior to execution
- Explicit structure facilitates: Search strategies, debugging, . . .

Structured Traversal of Search Trees

in Constraint-logic Object-oriented Programming

Constraint-logic Object-oriented Programming

Concepts

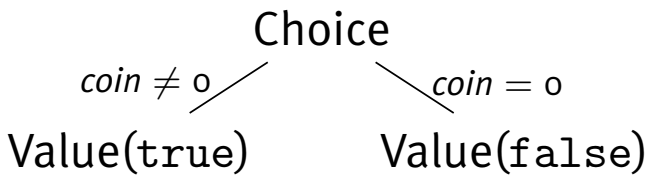
- Free variables (and fields) [DK18]

```
int y free;
```

- Constraints imposed by evaluating control structures

```
while (y > 5) { [...] }
```

- Non-deterministic execution in encapsulated search
- Search region written as lambda or Method Reference



- Implicit search tree, leaves are solutions (incl. exceptions)

Constraint-logic Object-oriented Programming

Muli Logic Virtual Machine

- Based on Muggl SJVM (test case generator), modified for Muli [MK11; DK18, DK19]
- Executes (Muli/Java) bytecode
- Execution state:

Frame stack	Operand stack	PC	Constraint stack	Choice point stack	Trails

- Non-deterministic branching $\Rightarrow$ choice point
- Choice point implementation
  - Previously: tracks choices and trails, controls execution
  - Now (Spoilers!): only declarative; MLVM in full control

Search Tree Representation

For comparison: Curry

[AHT19, BHH04]

```
x ? _ = x
_ ? y = y
```

e. g. `coin = 0 ? 1`

```
data ST a = Val a | Fail
          | Choice (ST a) (ST a)

Choice (Val 0) (Val 1)
```

In Muli

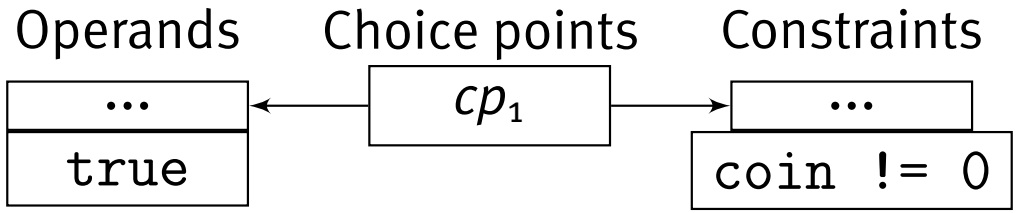
```
Muli.getAllSolutions( () -> {
  int coin free;
  if (coin == 0) {
    return false;
  } else {
    return true; } }
```

```
Solution[] {
  Solution(false), Solution(true) }
```

- additionally: single path on stack
- unstructured, transient

Search Tree Representation

Option: Implicit via choice point stack




```
Muli.getAllSolutions( () -> {
  int coin free;
  if (coin == 0) {
    return false;
  } else {
    return true; } }
```

Option: Explicit search tree

```
data ST a = Value a | Exception
          | Fail | Choice (ST a) (ST a)

Choice (Value false) (Value true)
```

Consider: Side effects ⇒ execution state!

Towards a Search Tree for CL-OO-P

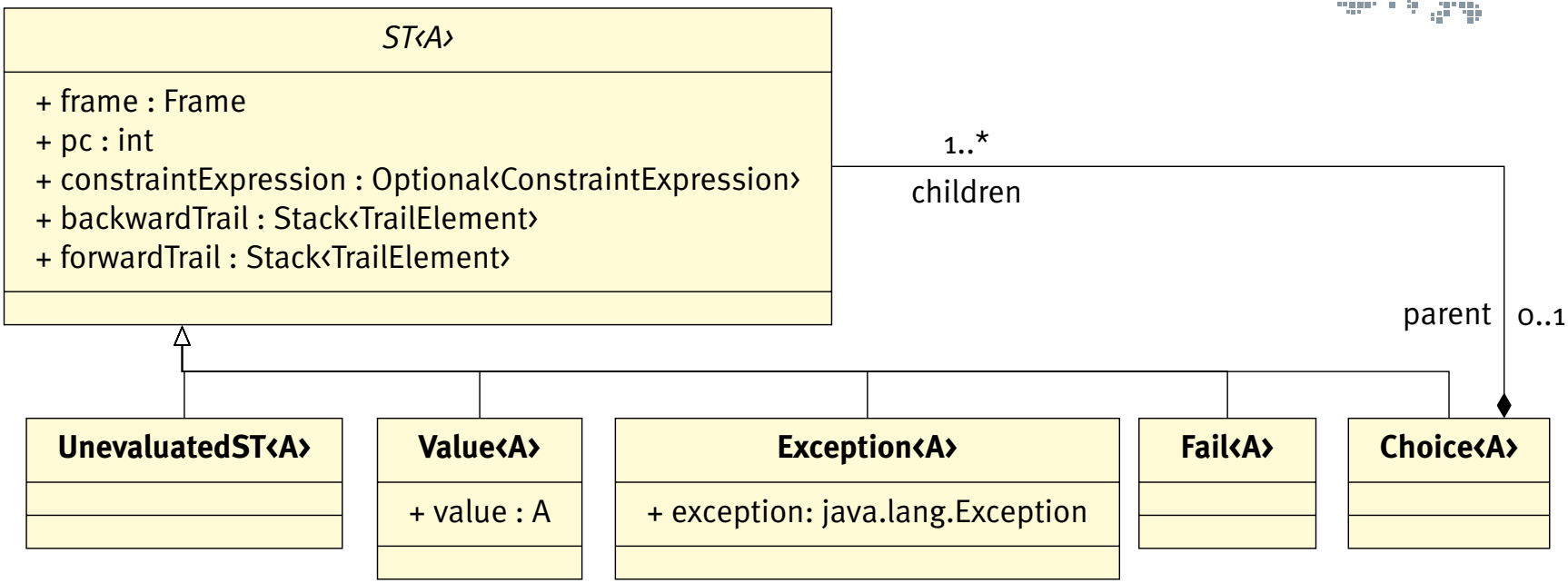
Types of choice

Triggering bytecode instruction	Type of choice	No. of decisions
`If<cond>`, `If_icmp<cond>`	`if` instruction, integer comp.	2
`FCmpg`, `FCmpl`, `DCmpg`, `DCmpl`	floating point comparison	2
`LCmp`	long comparison	3
`Lookupswitch`, `Tableswitch`	`switch` instruction	1 per case + 1

State

- Backtracking: return to former choice/state (e. g., operands and frame)
- Trails for navigation between choices
- During execution: Execute instruction with side-effect
  $\Rightarrow$ Push inverse of side-effect to backward trail
- During backtracking: Apply element from backward trail
  $\Rightarrow$ push inverse on forward trail

Search Tree Construction



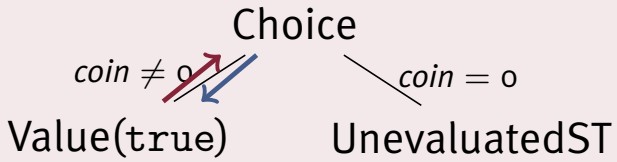
- New node ⇒ Return control to MLVM (every child is an `UnevaluatedST`)
- Non-strict evaluation
- Search strategies select next `UnevaluatedST`

```
Choice Uneval'dST Uneval'dST ⇝ Choice (Value false) Uneval'dST
```

Search Strategies

Primitives

- `navigateUpwards()`
- `navigateDownwards()`



- **BFS**

```
Choice
├─ Choice
│  ├─ UnevaluatedST
│  └─ UnevaluatedST
└─ Value(false)
```

- **DFS**

```
Choice
├─ Choice
│  ├─ Value(true)
│  └─ UnevaluatedST
└─ UnevaluatedST
```

Novel for (ND) imperative programs

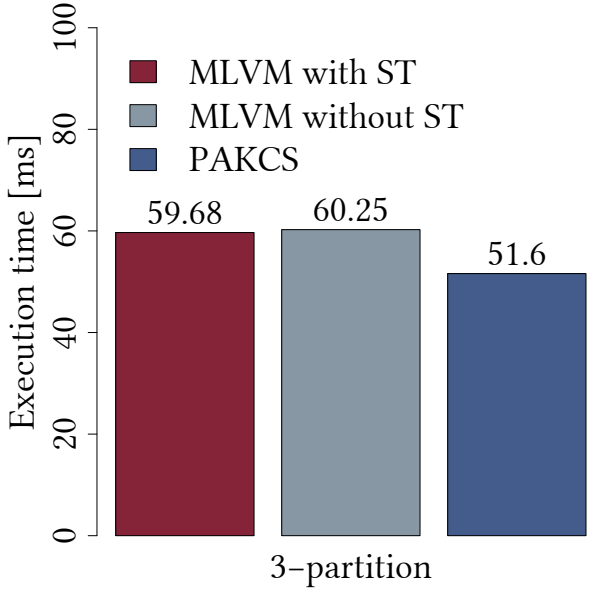
- **Iterative deepening DFS**

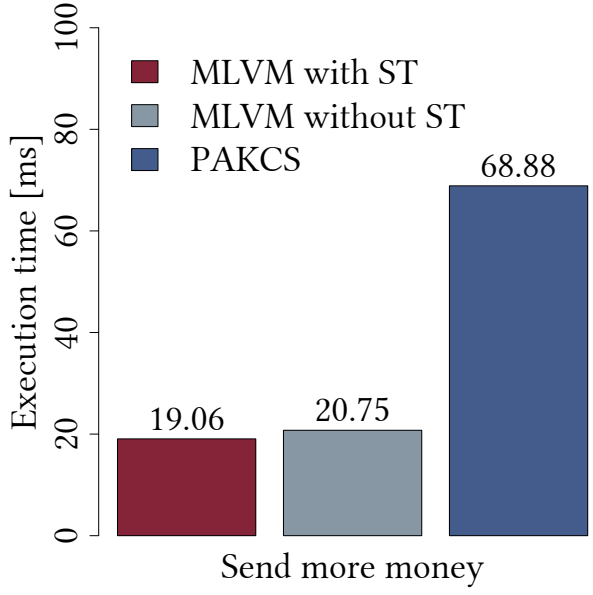
Key Results

- Useful for introspecting ND execution
- Foundation for complete search strategies
- Execution state changed:

Frame stack	Operand stack	PC	Constraint stack	Trails	Choice point stack
Frame stack	Operand stack	PC	Constraint stack	Trails	Search tree

- Increased memory requirements*, but performance unaffected





- *Optimisation: Purge trails of exhaustively evaluated subtrees

Concluding Remarks

Useful: Explicit structures (esp. in case of non-determinism)

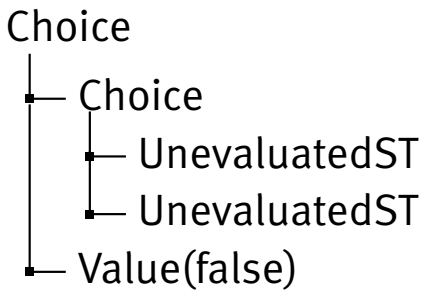
- Non-strict search tree representation
- Foundation for novel CL-OO search strategies
- Facilitates introspection

Implemented in MLVM, available on GitHub

- GPL, `https://github.com/wwu-pi/muli`

Future work

- Interactive search strategy
- Alternative representations



Jan C. Dageförde
University of Münster

j.d@wwu.de
http://erc.is/p/j.d
https://keybase.io/dagefoerde

THE IS RESEARCH NETWORK

www.ercis.org

References

[AHT19] Antoy, S., Hanus, M., Teegen, F. (2019). Synthesizing Set Functions. In Silva, J. (Ed.), Functional and Constraint Logic Programming.

[BHH04] Braßel, B., Hanus, M., Huch, F. (2004). Encapsulating Non-Determinism in Functional Logic Computations. In Journal of Functional and Logic Programming **2004**(6).

[DK18] Dageförde, J. C., Kuchen, H. (2018). A Constraint-logic Object-oriented Language. In Proceedings of the SAC 2018: Symposium on Applied Computing, Pau, Frankreich, pp. 1185–1194.

[DK19] Dageförde, J. C., Kuchen, H. (2019). Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space. In Proceedings of the SAC 2019: Symposium On Applied Computing, Limassol, Cyprus, pp. 1552–1561.

[MK11] Majchrzak, T. A., Kuchen, H. (2011). Muggl: The Muenster Generator of Glass-box Test Cases. In Working Papers, European Research Center for Information Systems (10).