# Diplomarbeit Abschlussvortrag


**Christian Müller**

## Run-Time Byte Code Compilation, Interpretation and Optimization for *Alice*

**Betreuer:** Guido Tack

**Verantwortlicher Prof.:** Gert Smolka

# Die nächsten 30 Minuten ...

- Was sind *Alice* und *Seam*?
- Bytecode Framework
  - Maschinenmodell und Bytecode
  - einfaches Übersetzungsschema
  - Optimierungen
- Vergleich: neues/altes System vs. Moscow ML

# Alice & Seam

# Was ist Alice?

- am Lehrstuhl entwickelte Erweiterung von Standard ML
- Nebenläufigkeit („lightweight threads“)
- Laziness, Lazy Linking von Komponenten
- Verteilte und offene Programmierung
  - Grundlage:

    plattformunabhängige Repräsentation von Code und Daten
  - → Alice Abstract Code (repräsentiert als DAG)

# Was ist SEAM?

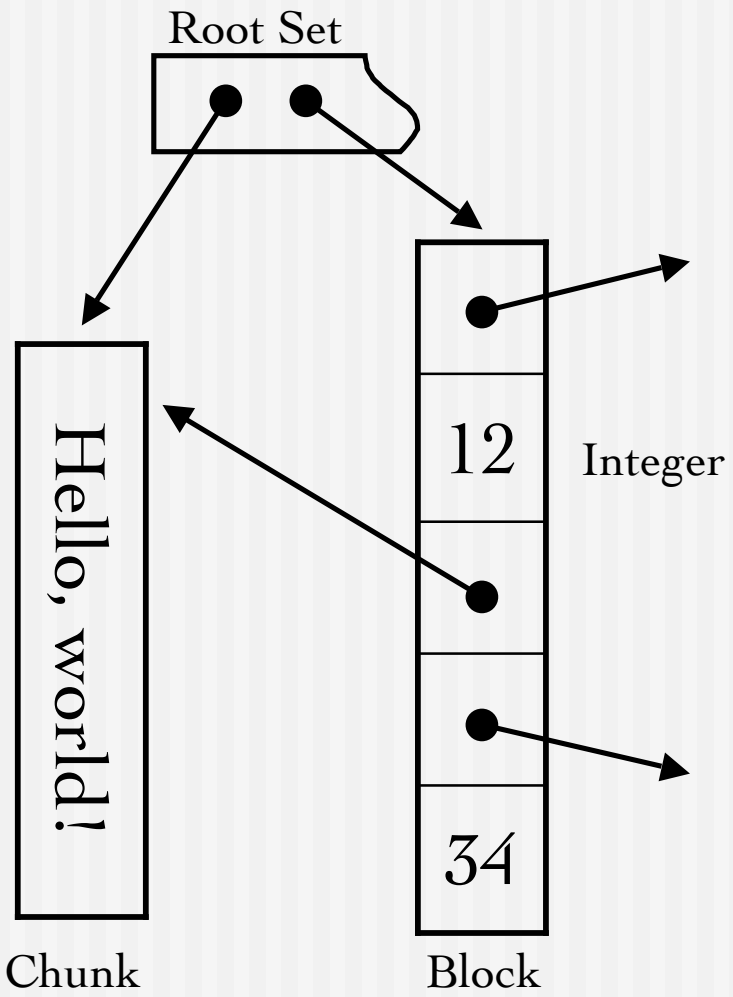
- **S**imple **E**xtensible **A**bstract **M**achine
- am Lehrstuhl entwickeltes Framework zur einfachen Implementierung von **Virtuellen Maschinen** (VMs)
  - abstraktes Datenmodell

    ➔ Abstract Store
  - abstraktes Ausführungsmodell

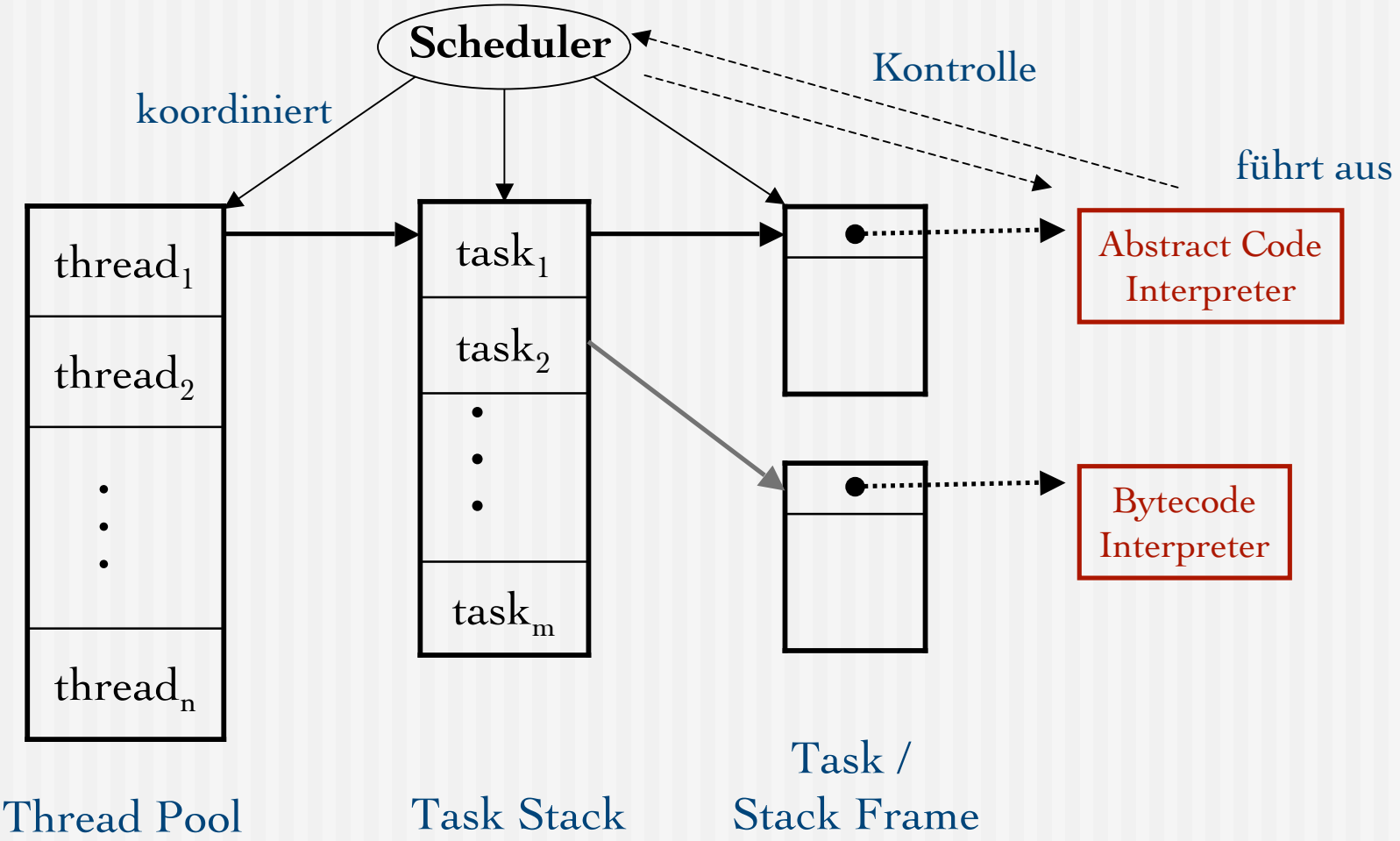
    ➔ Scheduler

# SEAM Abstract Store

- Graph
- 3 Knotentypen:
  - Block
  - Chunk
  - Integer
- Werte:
  - Zeiger auf Block/Chunk
  - Integer



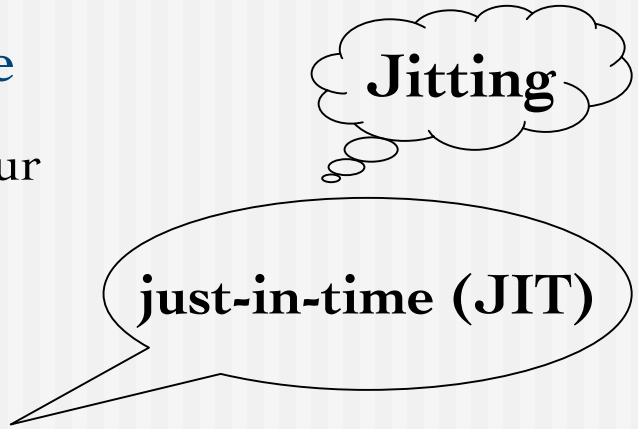
# SEAM Ausführungs-Modell



# Alice SEAM VM

Zwei Ansätze existieren:

1. Interpreter für den Abstract Code
   - einfache, gut zu wartende Struktur
   - plattformunabhängig
   - *Aber:* naiv und langsam
2. Laufzeit-Übersetzung in Maschinencode
   - deutlich schneller (genaueres später)
   - *Aber:* komplexe, kaum zu wartende Struktur
   - *Aber:* nicht plattformunabhängig



# Neuer Ansatz: Bytecode Jitting

- Erzeuge Bytecode anstelle von Maschinencode
- Führe Bytecode auf einem Interpreter aus

➔ Ziel der Arbeit:

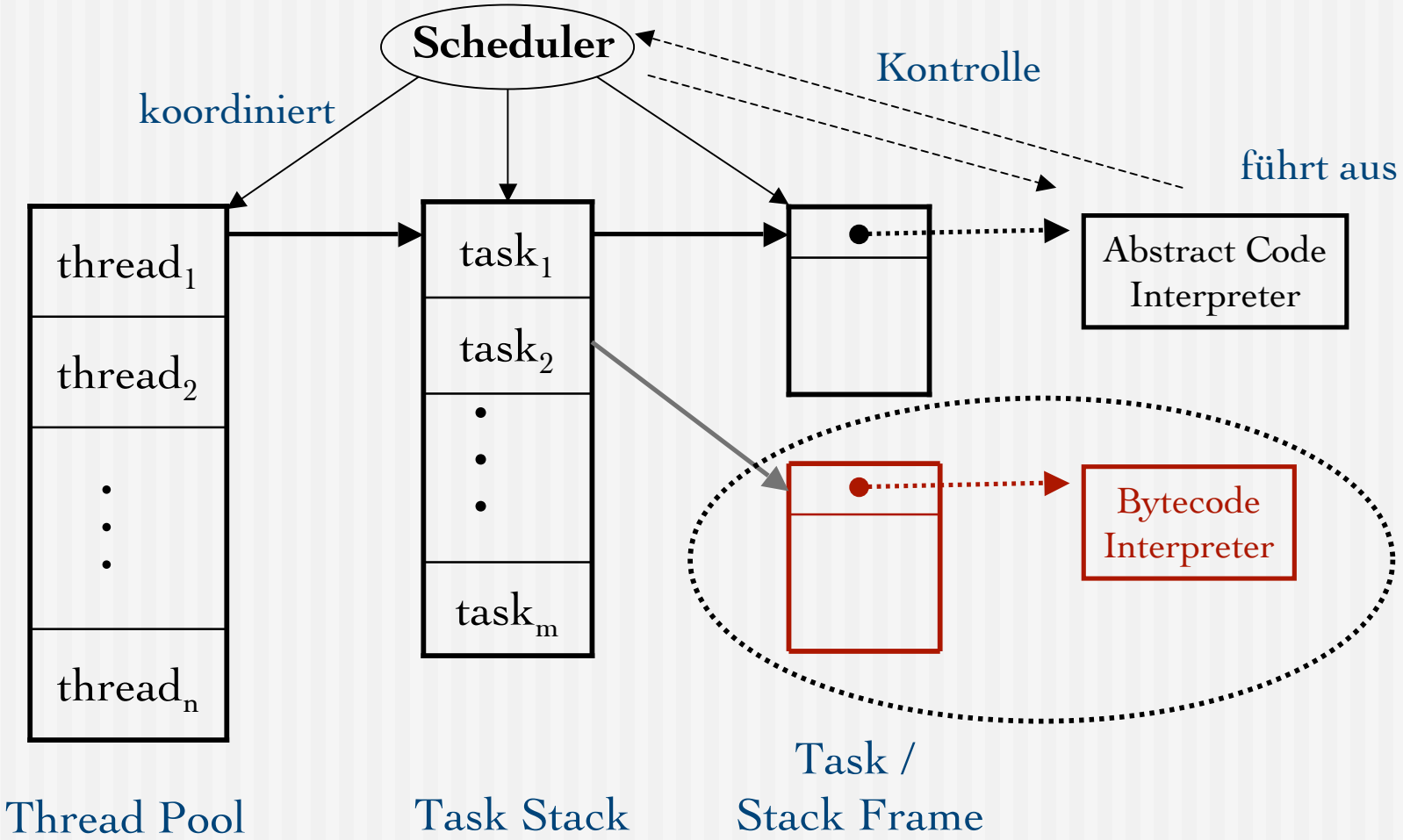
wartbares, plattformunabhängiges System mit guter Performanz

➔ Struktur der Arbeit:

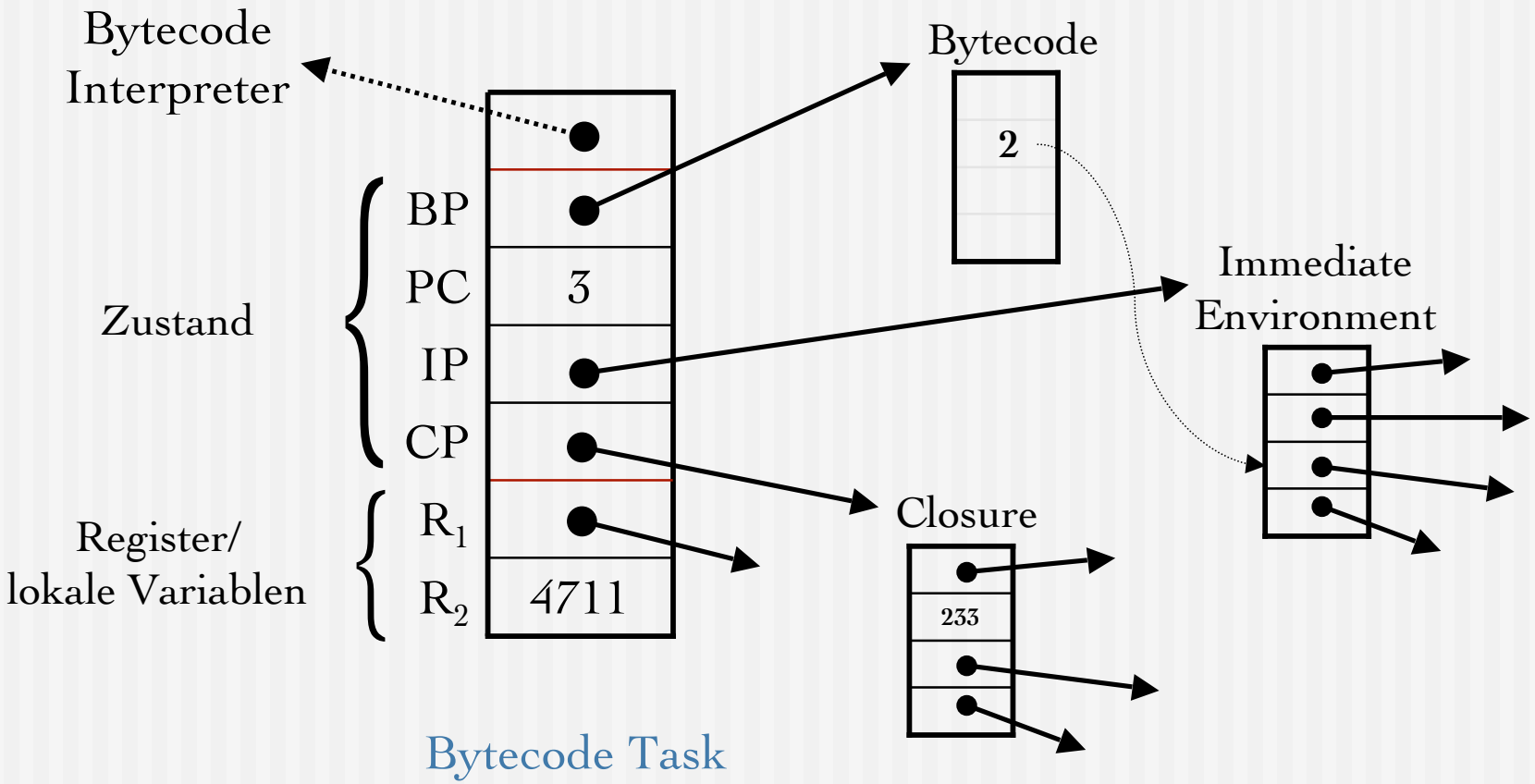
1. Spezifikation eines Bytecodes für Alice
2. Entwicklung eines Interpreters auf Alice VM
3. Entwicklung eines Bytecode Jitters
4. Optimierung des Jitters und Interpreters

# Bytecode Framework

# SEAM Ausführungs-Modell



# Maschinenmodell: Registermaschine



# 1. Bytecode Spezifikation

- eigener Alice Bytecode
- Bytecode ≈ Linearisierung und Spezialisierung des Abstract Codes
- Registercode
- **instr ::= opcode reg* integer***
- **Beispiel:** Tupel

```
(* val p = (x,y) *)
new_tup R1 2
init_tup R1 R5 0
init_tup R1 R6 1
```

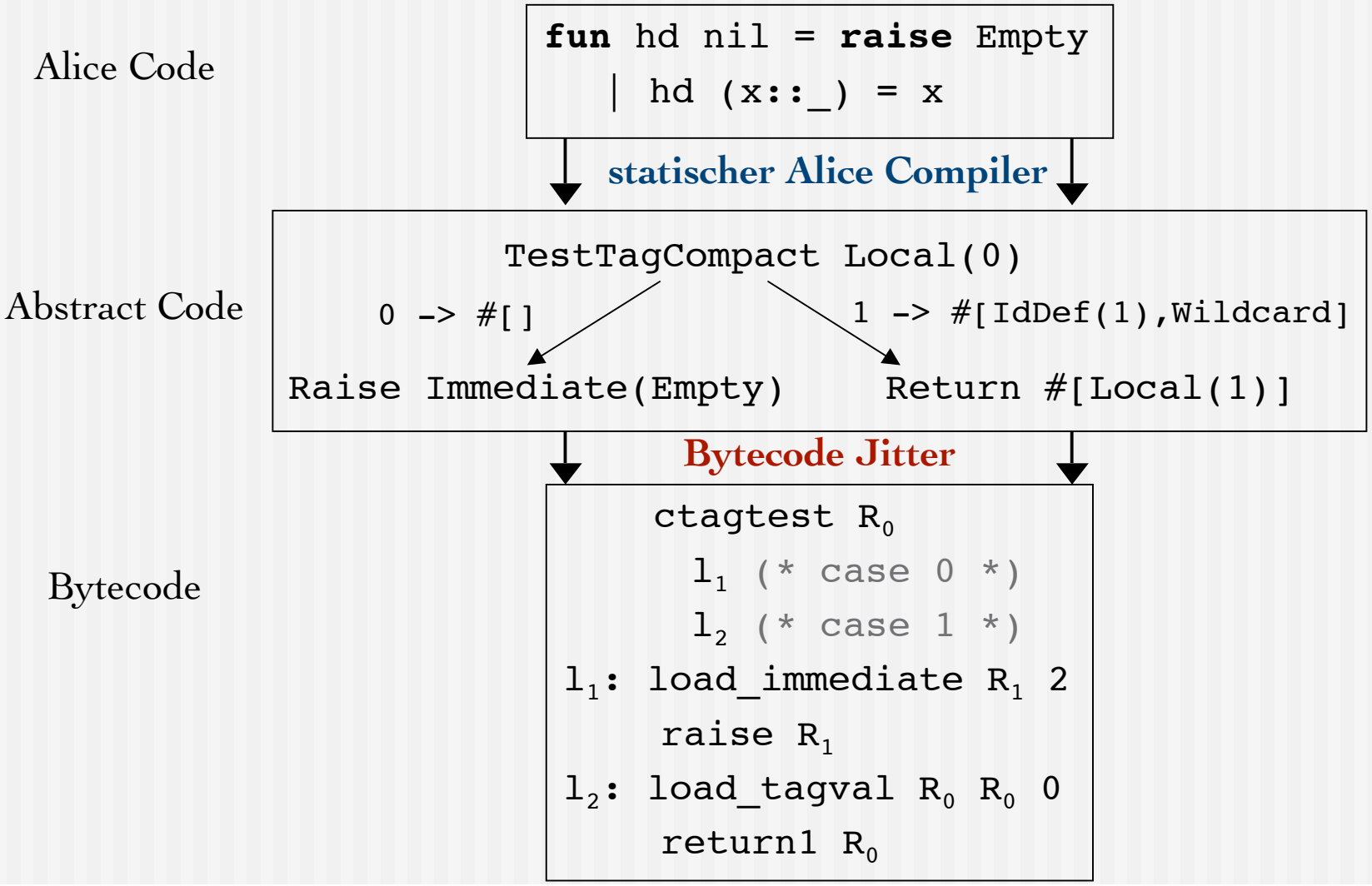
```
(* val z = #1 p *)
select_tup R7 R1 0
```

# 2. Bytecode Interpreter

- implementiert das Maschinenmodell
  - ➔ Abarbeitung eines Bytecode Tasks
- implementiert die Semantik der Instruktionen

  (ca. 150 Stück)

# 3. Bytecode JIT-Compiler



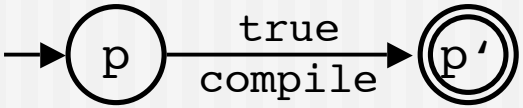
# 3. Bytecode JIT-Compiler

- Infrastruktur:
  - Offset-Tabelle zum Linearisieren des Codes
  - Verwalten von Registern
    - Registerallokation für lokale Variablen
    - temporäre Register für Zwischenergebnisse
  - Aufbau der Immediate Environment
  - Code-Alignment

# Optimierungen

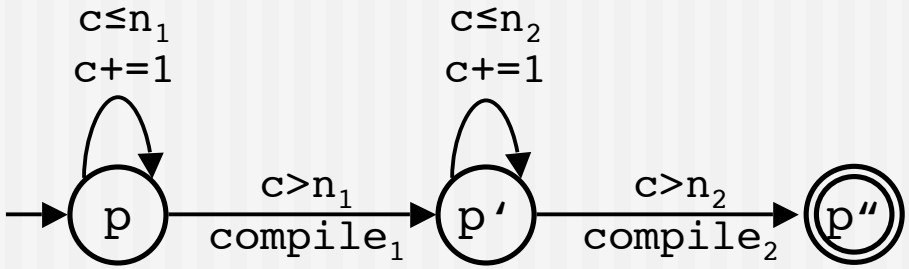
# 4. Selektive Übersetzung

- ursprüngliches Design: Lazy-Übersetzung

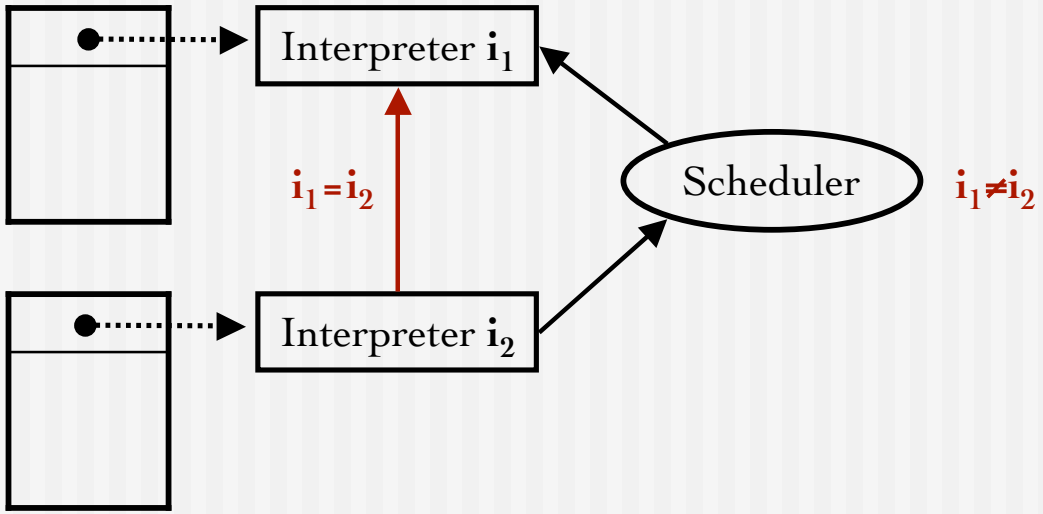



lazy future Bytecode

- Erweiterung: Selektive Übersetzung



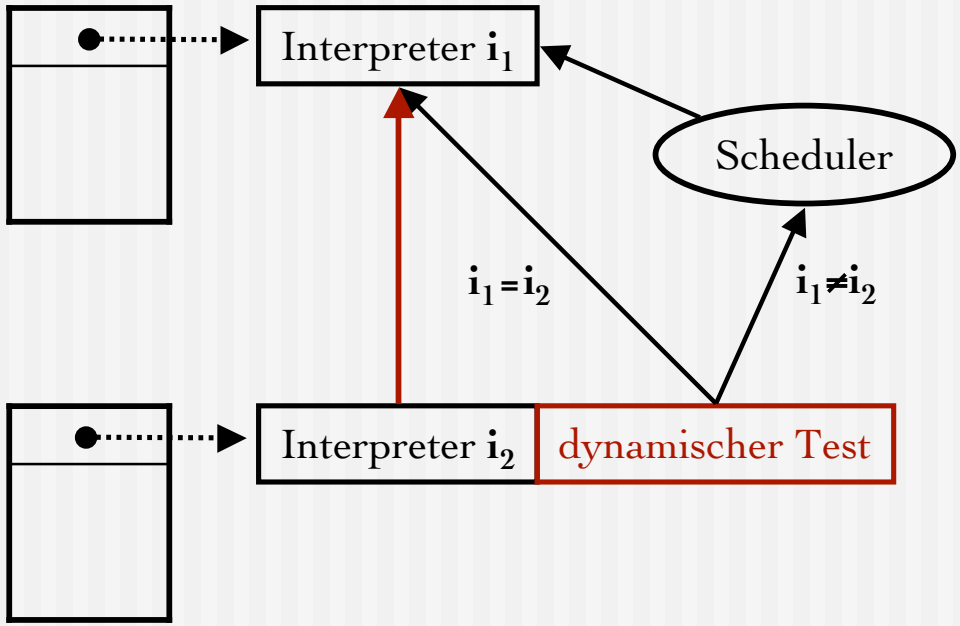
# 4. Umgehen des Schedulers




Maschinencode-Jitters: Test zur Übersetzungszeit

➔ Optimierungen funktionieren nicht bei selektiver Übersetzung

# 4. Umgehen des Schedulers: Bytecode-Interpreter




`rewrite_call f` $R_3$ ⋯▶ `bci_call f` $R_3$

Umschreiben der Instruktion, sobald `f` in Bytecode vorliegt

# 4. „Procedure Integration“

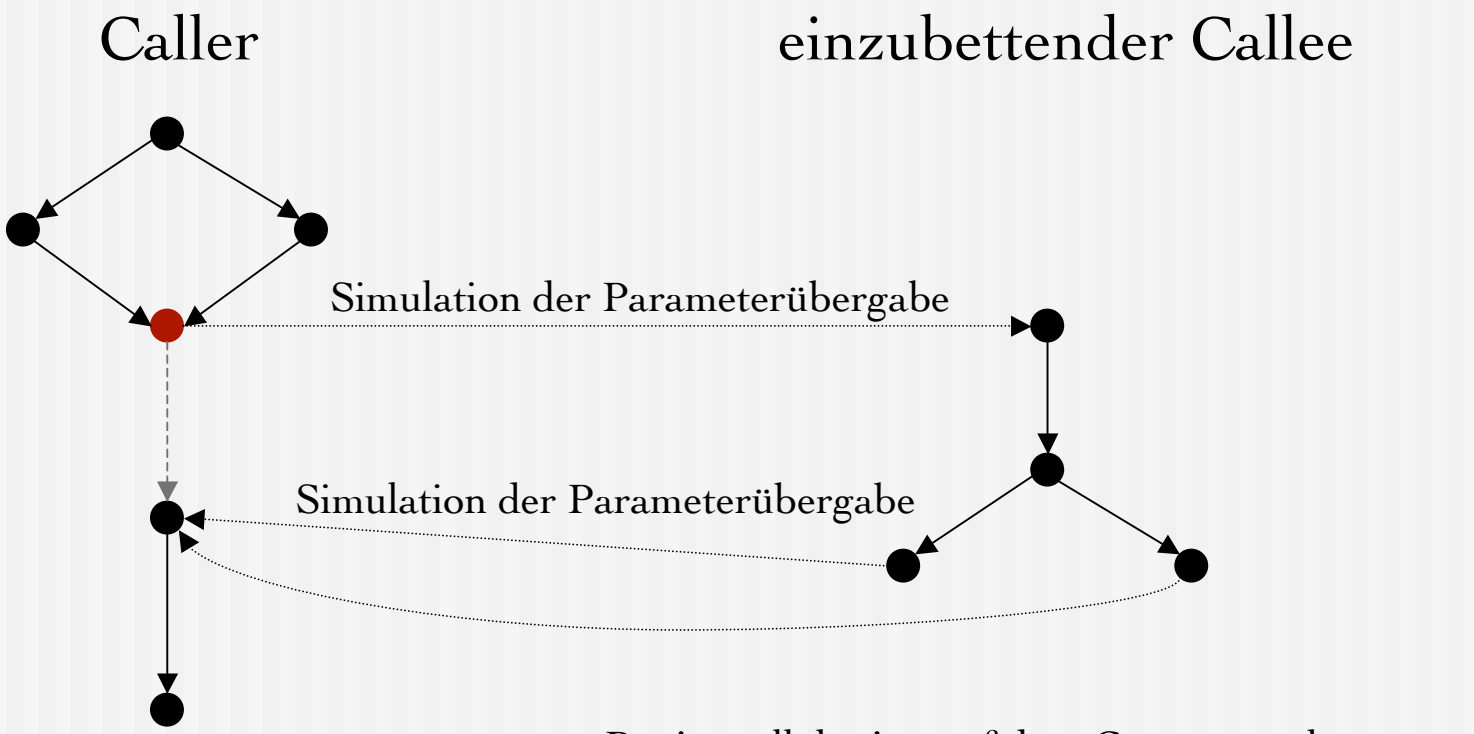
```
fun do_magic x = x + 1

fun f nil = nil
  | f (x::xr) = do_magic x :: f xr
```

+ bessere Performanz
– schlechtere Abstraktion
– statisch nicht immer feststellbar

```
fun f nil = nil
  | f (x::xr) = x + 1 :: f xr
```

Bytecode Jitter bettet `do_magic` automatisch ein

# 4. „Procedure Integration“




- Registerallokation auf dem Gesamtgraphen
- Konvertierung globaler Variablen zu Konstanten
  ➔Spezialisierung

# 4. Weitere Optimierungen

- Spezialisierung und Super-Instruktionen
- Inlining von Primitiven
- Switch-Intepreter und Direct Threaded Code
- Code/Instruktions-Layout: Platzverbrauch ⇔ Ausführungsgeschwindigkeit
- Selbstrekursion
- Loophole: `load_zero R0, await R0 [, kill R0]`
- „Alice Calling Convention“: Konvertierungstests vermeiden
- eingebettete Test-Tabellen
- Tag-Tests erschöpfend machen
- relative Sprünge besser als absolute Sprünge (relativ zum Beginn des Bytecode-Chunks)
- Sharing in der Immediate Environment
- Konstanten-Propagierung
- Register Allokation, Scratch Register (Unterschiede zum Maschinencode-Jitter)
- ...

- Assembler/Disassembler im Rahmen des Entwicklungsprozesses

# 4. Spezialisierung und Super-Instruktionen

- Beispiel: `val (x,y) = p`
  - Basisinstruktion:

```
select_tup R4 R3 0
select_tup R5 R3 1
```

  - Spezialisierung:

```
select_tup0 R4 R3
select_tup1 R5 R3
```

  - Super-Instruktion:

```
get_tup2 R4 R5 R3
```

- Ergebnis:
  - effizientere Implementierung
  - schnellerer Dispatch

# Vergleich der Systeme

## Antrittsvortrag:

- „Jitter 3x schneller als Abstract Code Interpreter“

➔ Ziel: Bytecode Jitter soll Maschinencode Jitter erreichen

# Vergleich der Systeme: Mikro-Benchmarks

| Zeit in ms | **fib** | **tak** | **ack** | **reverse** |
|---|---|---|---|---|
| Abstract Code | 1937 | 15305 | 31074 | 584 |
| Maschinencode | 108 | 1326 | 914 | 88 |
| Bytecode | 344 | 2552 | 3636 | 121 |
| **Moscow ML** | 109 | 1755 | 5990 | 243 |

Abstract Code/Maschinencode ≈ Faktor 17.5

Bytecode/Maschinencode ≈ Faktor 2.6

| Zeit in ms | **bubblesort** | **tree** |
|---|---|---|
| Abstract Code | 12313 | 15305 |
| Maschinencode | 2823 | 2471 |
| Bytecode | 2168 | 1186 |
| **Moscow ML** | 4081 | 1155 |

# Vergleich der Systeme: Alice-Applikationen

- Toplevel-Interpreter Startup:

| | ms |
|---|---|
| Abstract Code | 4241 |
| Maschinencode | 1728 |
| Bytecode | 1613 |

- Bootstrap:

| | min:sec |
|---|---|
| Abstract Code | 57:27 |
| Maschinencode | 20:55 |
| Bytecode | 20:35 |

# Zusammenfassung

- Bytecode-System für Alice
  - Compiler + Interpreter und Optimierungen
  - (frühe Version) in Alice 1.2 integriert
  - effizientes, plattformunabhängiges System für AMD64, PowerPC, ...
- Testumgebung für neue Optimierungen
  - wesentlich einfacher zu handhaben als Maschinencode-Jitter
- Performanz für reale Applikationen vergleichbar zum Maschinencode-Jitter

# Future Work Bytecode-System

- Strictness-Analyse
- Bytecode-Debugger (Erweiterung des Disassemblers)
- genauere Heuristik für Procedure Integration
- `inline` Pragma in der Source-Sprache

# Literatur


- Wilhelm, Maurer, Übersetzerbau, Springer 1997
- Rossberg et. al., Alice ML Through the Looking Glass, Techreport 2004
- Brunklaus, Kornstädt, A Virtual Machine for Multi-Language Execution, 2002
- Ralf Scheidhauer, Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz, PhD Thesis, 1998
- Poletta und Sakar, Linear Scan Register Allocation, ACM, 1999
- Davis, The Case For Virtual Register Machines, 2002
- Jones, Secrets of the Glasgow Haskell Compiler Inliner, 1999
- Ierusalimschy et. al., The Implementation of Lua 5.0, 2005