# Diplomarbeit Antrittsvortrag


**Christian Müller**

Run-time byte code compilation, interpretation and optimization for *Alice*

**Betreuer:** Guido Tack

**Verantwortlicher Prof.:** Gert Smolka

# Die nächsten 15 Minuten ...

- Was ist *Alice*?
- Was ist Seam?
- Bestehende Ansätze zur Ausführung von Alice Code
- Neuer Ansatz und Optimierungen

# Was ist Alice?

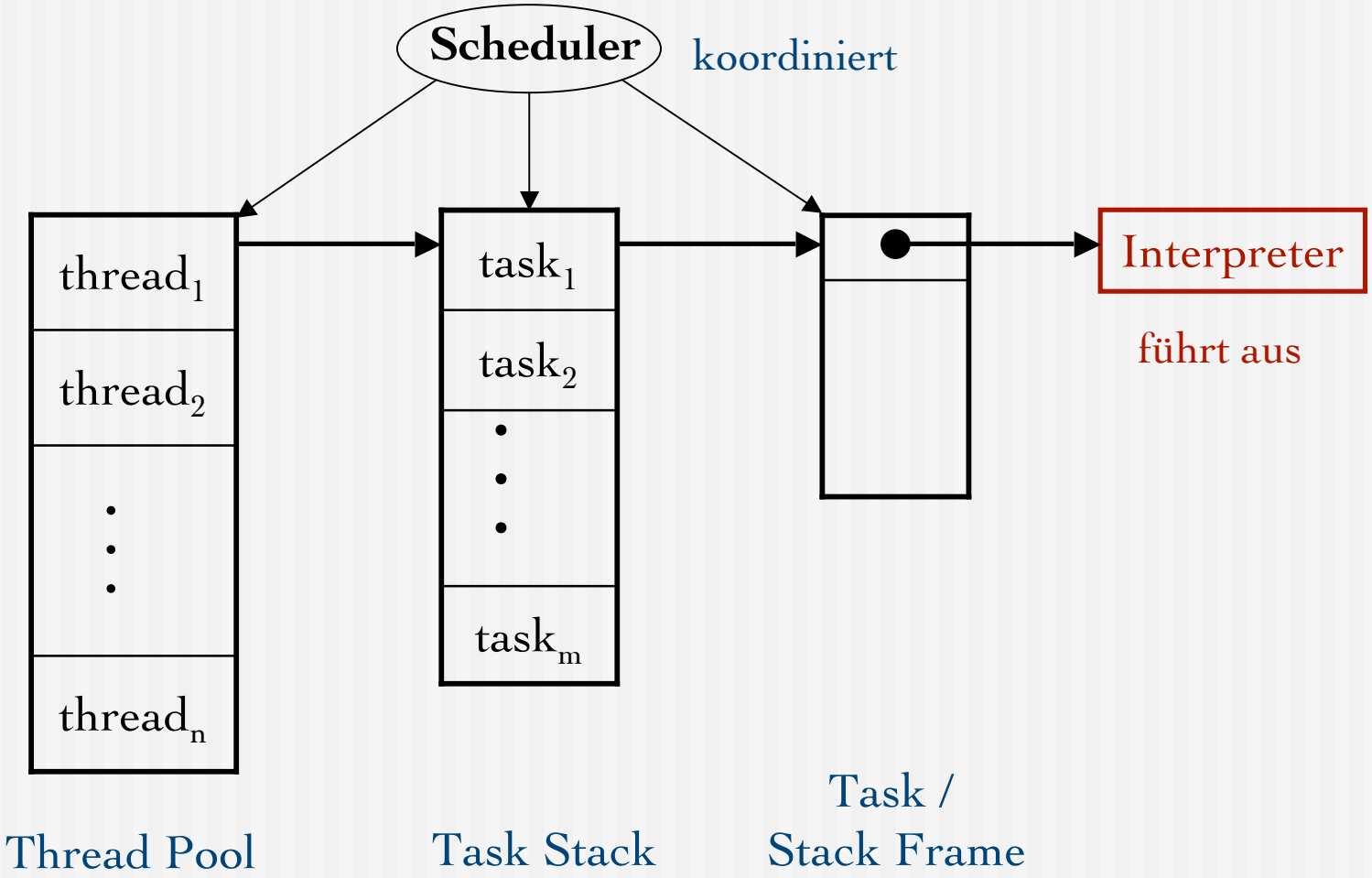
- Am Lehrstuhl entwickelte Erweiterung von Standart ML
- Nebenläufigkeit
- Lazy Linking von Komponenten
- Verteilte und offene Programmierung
  - Grundlage: Abstrakte Repräsentation von Code und Daten
  - Abstrakt ≈ Uniformität von Code und Daten, Plattform unabhängig
  - → Alice Abstract Code

→ Hochgradig dynamische Sprache

# Was ist Seam?

- **S**imple **E**xtensible **A**bstract **M**achine
- Am Lehrstuhl entwickeltes Framework zur einfachen Implementierung von **Virtuellen M**aschinen (VMs)
  - Unterstützung von Nebenläufigkeit
  - Speicherverwaltung (Garbage Collection und Memory Layout)
  - Abstraktes Ausführungsmodell

# Seam Ausführungs-Modell



# Alice Seam VM

Zwei Ansätze existieren:

1. Interpretiere den abstrakten Code
   - Einfache, gut zu wartende Struktur
   - Plattform unabhängig
   - *Aber:* naiv und langsam
2. Erzeuge zur Laufzeit Maschinencode
   - Etwa 3x so schnell
   - *Aber:* komplexe, kaum zu wartende Struktur
   - *Aber:* nicht Plattform unabhängig

# Neuer Ansatz: Bytecode Jitting

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

→ Struktur der Arbeit

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

# Stand der Arbeit

- Vertraut machen mit der Architektur und Literaturrecherche
- Erste Spezifikation eines Bytecodes
- Implementierung eines Assemblers in Alice
- Beginn der Implementierung eines Interpreter-Prototyps
  - Eingebettet in das Seam-Ausführungsmodell
  - Maschinenmodell:

    **Register Maschine** mit fester Anzahl von General Purpose Registers

# Ausblick und „Future Work“

Bytecode besser/schneller als Maschinencode???

# Bytecode besser/schneller als Maschinencode???

- Jitting findet nur *lokal* statt
  - Übersetzung je einer Funktionen
  - GC, Scheduler, ... weiterhin als C-Funktionsaufrufe
- Code-Größe:
  - Bei aktuellem Jitter bis zu 80MB im Extremfall
  - Bytecode ist wesentlich kompakter

# Optimierungen: Code

- Größe:
  - Möglichst wenige Instruktionen
  - Erledige viel Arbeit pro Instruktion
  - → Kleiner Dispatch Overhead
- Spezialisierung von Befehlen:
  - Z.B. tuple_select1, tuple_select2
  - → Kleinere, schnellere Befehle
- Direct Threaded Code:
  - Schneller Dispatch

# Optimierung: Direct Threaded Code

## Switch-basierend

```
typedef enum { hello, halt} Inst;

void interprete() {
        static Inst program[] = {hello, hello, halt };
        int pc = 0;
        while(true) {
                switch(program[pc]){
                case print:
                        cout << „hello“ << endl;
                        pc++;
                        break;
                case halt:
                        return;
                }
        }
}
```

## Direct Threaded

```
typedef void *Inst;

void interprete() {
        static Inst program[] =
                        {&&hello, &&hello, &&halt};
        int pc = 0;
        goto *program[pc];

hello:
        cout << „hello“ << endl;
        goto *program[pc++];
halt:
        return;
}
```

# Bytecode besser/schneller als Maschinencode???

- Dynamisches Optimieren (Jitter)
  - Umgehe Scheduler, bspw. bei Self-Call
  - Inlining von Primitiven
    AppPrim Int.+ (2,3) ➜ iadd 2 3

- Optimierung nicht nur zur Compile-Zeit,
  sondern auch zur Laufzeit
  - Inlining von Primitiven, die beim Jitting unbekannt waren
  - „Strictness Analysis“

# Zusammenfassung

- Erweitere Alice Seam VM um eine weitere Ausführungseinheit.
- JIT Compiler für dynamische Sprachen besonders gut geeignet.
- Lasse optimierten Bytecode Jitter gegen Maschinencode Jitter antreten!

→ „Es ist einen Versuch wert“

# 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
- Ertl, Threaded Code Variations and Optimizations , 2001