A Caml Light Compiler for the JVM

Lars Hupel, Marek Kubica, Stefan Schulze Frielinghaus,
Edgar Müller, Dmitriy Traytel

TU München

20. August 2010

Übersicht

1. Eingabe: CamlLight (mit leichter Modifikation)
2. Scanning mit JFlex
3. Parsing mit CUP2
4. Magic
5. Code-Generierung nach MaMA (ebenfalls modifiziert)
6. Übersetzung nach JVM

Ablauf

- für den Compiler steht ein komplettes Ant-Skript zur Verfügung:
  `ant jar`
- Aufruf über Shell-Skript:
  `bin/compiler prog.cl -o foo`

Lexer

Technisches

- JFlex
- Kommentare: `(* foo (* bar *) baz *)`
- Identifier: `foo_bar_42`
- Integer-Literale: `-0x42`; `0b101010`; `0o52`
- Floating-point-Literale: nicht existent
- Character-Literale: `'a'`; `'\042'`
- String-Literale: `"foo"`; `"bar \023"`
- Keywords: einige...

Parser

Technisches

- CUP2 + Scala-Layer
- Caml Light-Syntax
- pro Reduce-Schritt werden die Zeilen- und Spaltennummern hinterlegt
- Post-processing: Normalisierung

Typinferenz

Constraint based typing nach Hindley-Milner

- ca. 700 LOC, $\approx$ 50 Tests
- Lambda-Kalkül + `let`-Ausdrücke: easy ☺
- Unterstützung von `match`, `let rec`, Records, Record-Zugriff, Tupel, Tupel-Zugriff, binäre/unäre Operatoren, if-then-else
- Pattern Matching für
  - Literale
  - Underscore
  - Listen (Cons, Nil)
  - Tupel
  - Records

Typinferenz

Beispiel: Record Pattern

```
let { foo = (_,(x,y)); bar = z::zs } =
    { foo = ('a',(1,2::[])); bar = [true;false;true] }
 in (x + (hd y) == 3) == z
```

Beispiel: sum

```
let rec foldr = fun
   f acc []     -> acc
 | f acc x::xs -> f x (foldr f acc xs);;
let rec add = fun x y -> x + y;;
let xs = [1;2;3] in foldr add 0 xs
```

Typinferenz

Weiteres Beispiel

```
let x = 3 in let y = 4 in (x,y) + 5
```

Fehlermeldungen

```
Type checking failed
ERROR: Couldn't unify types:
   TypeTuple(List(TypeInt(), TypeInt()))
   TypeInt() in expression:
(((x),(y)))+(5)
```

Codegenerierung: MaMa

Übersetzung AST $\Longrightarrow$ MaMa

- Vorgehen wie in

  [Wilhelm, Seidl – Übersetzerbau: Virtuelle Maschinen]

Erweiterungen

- Records
  - Implementierung als namenlose Tupel
  - Feldnamensauflösung mit Hilfe der Typinferenz
- Pattern matching mit beliebigen Patterns

Codegenerierung: MaMa

Übersetzung AST $\Longrightarrow$ MaMa

- Vorgehen wie in

  [Wilhelm, Seidl – Übersetzerbau: Virtuelle Maschinen]

Erweiterungen

- Records
  - Implementierung als namenlose Tupel
  - Feldnamensauflösung mit Hilfe der Typinferenz
- Pattern matching mit beliebigen Patterns
  - kompliziert...

Codegenerierung: MaMa – Pattern Matching

Pattern Matching im AST

`match` e_0 `with` p_1 `->` e_1 `|` ... `|` p_n `->` e_n

Idee

- Generiere rekursiv sogenannten *matching code* für jedes einzelne Pattern p_i.
- Invariante: nach Ausführung des *matching code* liegt auf dem Stack eine 1, falls e_0 p_i matcht, sonst 0.
- Zusammengesetzte Patterns (wie z.B. Tupel) generieren den *matching code* rekursiv und führen `AND`-Instruktionen aus.
- Dabei wird der Code für e_0 nur einmal generiert.

Codegenerierung: MaMa – Fehlermeldungen

```
let hd = fun x::xs -> x in hd []
```

- Für jedes Pattern Matching lege auf den Stack:
  1. Den Match-Ausdruck
  2. Zeilen- und Spaltenangaben
- Virtuelle MaMa-Maschine produziert Ausgaben von der Form:

```
Exception in thread "main" java.lang.RuntimeException:
Pattern match failure in the expression
"match (427 @ 0) with [(4212);(_)] -> (4212)"
in line 1 and column 24
unlucky you!
```

Codegenerierung: JVM-Bytecode

Idee

- Verarbeitet erweiterten MaMa-Code: deineMaMa
  - Instruktionen die PC lesen, nehmen stattdessen Label als Parameter
  - Instruktionen die PC modifizieren, geben stattdessen Label aus
- Verwendung des JVM-Heaps
- Verwendung von `java.util.Stack`
- Instruktionen in Java geschrieben

ASM

- Library für Bytecodeinspektion und Modifikation
- Wird genutzt um unsere eigene `main()` einzuschleusen

Codegenerierung: Generierter Bytecode

main() nach Java übersetzt

```
Machine m = new Machine();
boolean terminate = false;
int _goto = 0;
while(!terminate) {
  switch(_goto) {
    case 0:
    case 1:
      _goto = m.eval(2);
      continue;
    case 2:
      m.loadc(0);
    case 3:
      terminate = true;
  }
}
```

Zustand

Was funktioniert

- Caml Light Syntax, mit (passenden) Fehlermeldungen
- Lambdas, Match, Tupel, Records, Strings, Listen
- Ausgabe von Java-Bytecode

Was nicht funktioniert

- Typkonstruktoren
- Seiteneffekte, Exceptions
- Threads

Testing

- eigene Testing-Umgebung, ähnlich zu ScalaTest
- $\approx$ 50 Testcases für Scanner/Parser
- $\approx$ 50 Testcases für Typinferenz
- $\approx$ 150 Zeilen Caml Light-Programme

Benchmarks

Ackermann(3, 5)

Zweifaches Pattern Matching

	Caml Light	Scala	TUM
Kompilieren	0.0 s	9.3 s	4.1 s
Ausführen	0.00 s	1.03 s	1.69 s

Factorial(10)

Rekursiv

	Caml Light	Scala	TUM
Kompilieren	0.0 s	9.1 s	3.9 s
Ausführen	0.00 s	0.98 s	0.13 s

System: Core2 Duo @ 1.5 GHz x86_64; OpenJDK 1.6.0_18; Scala 2.8.0; Caml Light 0.75

Spaß

Code

$\approx$ 3.300 LOC Scala, $\approx$ 650 LOC Java, $>$ 325 commits

Best of Commit-Messages

- „Whee, the machine can actually run more complicated code without failing“
- „Get out of the JAR, *.mama. You’re not even bytecode!“
- „Case classes should not inherit other case classes, you know? We have so much syntactic sugar, use that instead!“
- „Hey ., you should really consider matching newlines, too!“
- „Scala supports some mad syntactic sugar. Mad! I tell you.“
- „Marek goes insane and after that, realizes that bipush handles only +127 and then wraps around.“
- „I heard you want to know where exactly the typo in your souce code is. Here you are!“

Rückblick

Was war gut

- Scalas Pattern Matching
- Verwendung von Git (Branching!)
- Betreuung ☺

Was war weniger gut

- NetBeans
- Compilezeiten von `scalac`
- Rollo ging täglich um 14:30 rauf ☹

Zukunft

Mögliche Erweiterungen

- Stdlib bereitstellen
- Verwendung des JVM-Stacks
- Threads
- Optimierungen: TCO