Prototyp eines universellen Frameworks für S-Expression-basierte Code-Generierung

Diplomarbeit

Benjamin Teuber
Erstbetreuer: Daniel Moldt

TGI-Oberseminar
Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Department Informatik

19. Januar 2010

Themengebiet: Code-Generierung

- Code-Generierung immer stärker genutzt
  - Model-Driven Architecture
  - Domain-Specific Languages
- Problem: Generator bauen ist aufwändig
- Viele Komponenten nötig
  - Parser für Quellsprache
  - Compiler in normaler Programmiersprache
  - Templates für Zielsprache
- Ziel: Framework aus einem Guss

Inhalt

- Vergleich bestehender Technologien
- Ableiten von Anforderungen
- Entwurf eines prototypischen Frameworks
  - Architektur
  - Komponenten

Vorhandene Technologien

Übersicht: Vergleich bestehender Technologien

- Verschiedene Typisierungsgrade von Modellen
  - Zeichenketten - low level
  - Objekte - typisiert, aber unflexibel
  - Bäumrepräsentationen wie XML oder S-Expressions
- Code-Erzeugung
  - Templates - verschiedene Mächtigkeiten
  - Programmatisch - z.B. Stringmanipulationen
- Kontrollfluss
  - Direkter Aufruf von Templates
  - Impliziter Aufruf über Struktur

Exkursion: Lisp

- Eine der ältesten Programmiersprachfamilien (LISP: 1958)
- Minimale, uniforme Syntax (S-Expressions $\simeq$ XML-Subset)
- Sprache selbst kann durch Makros erweitert werden

  "The programmable programming language"

Zitat

"Any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp."
- Philip Greenspun

S-Expressions

- Ein S-Expression ist entweder:
  - Ein Atom, z.B. eine Zahl, ein String, eine Variable
  - Eine Liste von S-Expressions in Notation
    $(sexp_1 \ sexp_2 \ .. \ sexp_n)$
- Konvention: Knotenname an erster Stelle
- *Strukturelle* Verarbeitung

Beispiel: Funktionsdefinition in Common Lisp

```
(defun my-add (a b)
  (+ a b))
```

S-Expressions (2)

- Modell: Bäume mit benannten Knoten und Zeichenketten als Blättern
- Kompromiss zwischen Flexibilität und Strukturierung

Auto-Modell in S-Expressions

```
(car
  (color red)
  (specials AC Navigation))
```

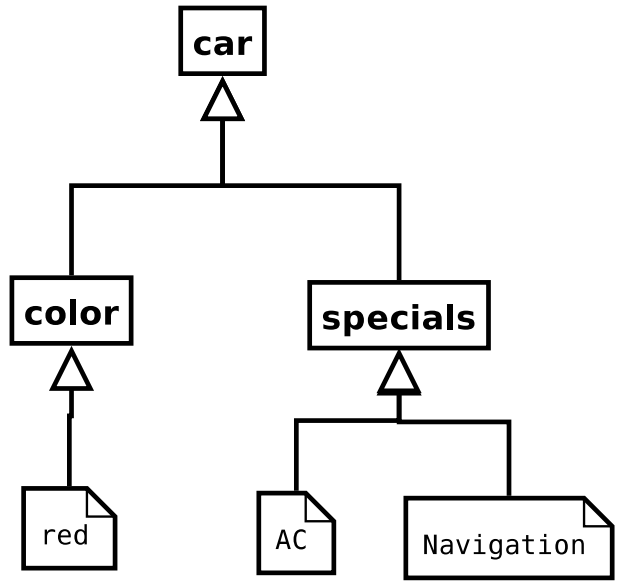


Abbildung: Sexp-Auto-Modell als Baum

Kritik an XML

- These: Attribute sind unnötig
- Problem: Keine Listen von Blättern

Auto-Modell in XML

```
<car color="red">
  <specials>
    <special>AC</special>
    <special>Navigation</special>
  </specials>
</car>
```

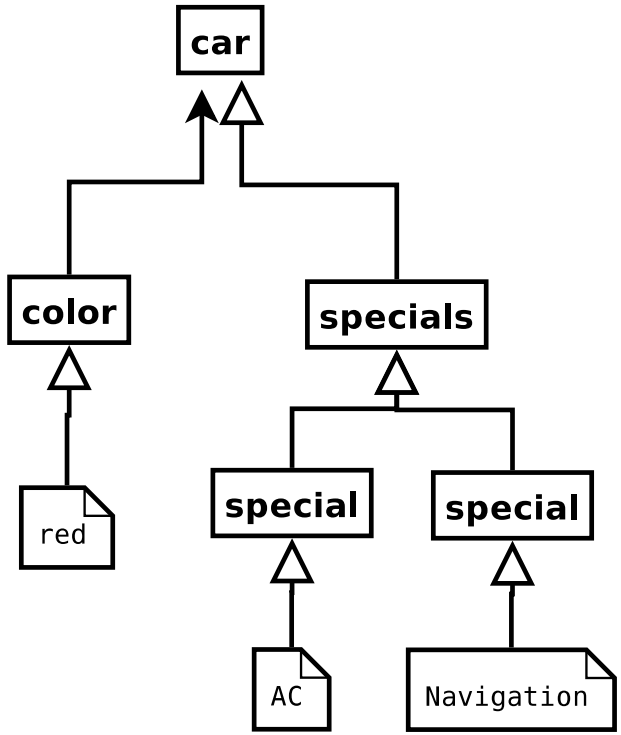


Abbildung:
XML-Auto-Modell als Baum

Vorteile von S-Expressions

- Simple Struktur - vereinfachen:
  - Parsing (vgl. XML)
  - Verarbeitung
  - Erzeugung
- Dennoch ausdrucksstark

$\Rightarrow$ Basis für Lisp-Makros

Templates

- Für Dokumenterzeugung genutzt
- “Schablone mit Platzhaltern”
- Funktion: $\mathrm{input} \rightarrow \mathrm{doc}$
- Unterschiede:
  - Typen input und doc
  - Syntax der Metasprache
  - Mächtigkeit

PHP

- Object → String
- Zeichenketten problematisch
  - Syntaxfehler
  - Einrückung für lesbaren Code?
- Komplette Programmiersprache

```
<html>
  <head><title>Current Date</title></head>
  <body>
    <? print($current_date); ?>
  </body>
</html>
```

XSL Transformation

- XML → XML
- Mächtig, aber umständliche Syntax
  - ⇒ Praxis: Komplexe Verarbeitung in externer Programmiersprache
- Ermöglicht direkten Aufruf sowie Matching über XPath
- XPath: `/book[@price>35]/title`

XSLT (2)

Eingabe

```
<addresses>
  <person firstname="Heinz" name="Schulz" />
  <person firstname="Walter" name="Meier" />
</addresses>
```

Erwünschte Ausgabe

```
<table>
  <tr><td>Walter</td><td>Meier</td></tr>
  <tr><td>Heinz</td><td>Schulz</td></tr>
</table>
```

XSLT (3)

```
<xsl:template match="/addresses">
  <table>
    <xsl:for-each select="person">
      <xsl:sort select="@name"
                order="ascending"
                data-type="text" />
        <tr>
          <td><xsl:value-of select="@firstname" /></td>
          <td><xsl:value-of select="@name" /></td>
        <tr>
      </xsl:sort>
    </xsl:for-each>
  </table>
</xsl:template>
```

Lisp-Makros

- $\text{Sexp} \rightarrow \text{Sexp}$
- Metasprache = Zielsprache = Quellsprache = Lisp
- “Compiler-Plugins” in kurzer, eleganter Notation
  - Ermöglichen inkrementelle Erweiterung des Lisp-Compilers
  - “Embedded DSLs” - in die ursprüngliche Sprache integriert
- Einschränkungen:
  - Typen
  - Kontrollfluss
  - Kein “SPath”

Lisp-Makros (2)

Makro-Aufruf

```
(addresses
  (person "Heinz" "Schulz")
  (person "Walter" "Meier"))
```

Zu erzeugender Code

```
(table
  (tr (td "Walter") (td "Meier"))
  (tr (td "Heinz")  (td "Schulz")))
```

Lisp-Makros (3)

Makro-Umsetzung in Common Lisp

```
(defun sort-by-name (persons) ...)

(defmacro addresses (&rest persons)
  `(table ,@(sort-by-name persons)))

(defmacro person (first-name name)
  `(tr (td ,first-name) (td ,name)))
```

Anforderungen

MagicL

- M: Models
- A: Architecture
- G: Generation
- I: Interface
- C: Control Flow
- L: Lisp

Anforderungen

- Lisp
  - Makro-Ansatz
- Modelle
  - S-Expressions
  - Objekte
  - Strings
- Generierung von Quelltext
  - Nicht auf Zeichenketten-Ebene
  - Modell für Code
  - automatische Formatierung

Anforderungen (2)

- Architektur
  - theoretische Fundierung
  - visualisierbar (vgl. Petri-Netze)
  - flexible Verarbeitungsprozesse
- Compiler-Schnittstelle
  - universell - Parser, Makros
  - automatische Auswahl über Typ
- Kontrollfluss
  - direkter Aufruf
  - impliziter Aufruf durch Matching
    - XPath?
    - EBNF?

Architektur

Die Sprache Haskell

- Funktional (pur)
- statische Typinferenz
- Typklassen entsprechen Java-Interfaces
- Seiteneffekte dank Monaden/Arrows integrierbar

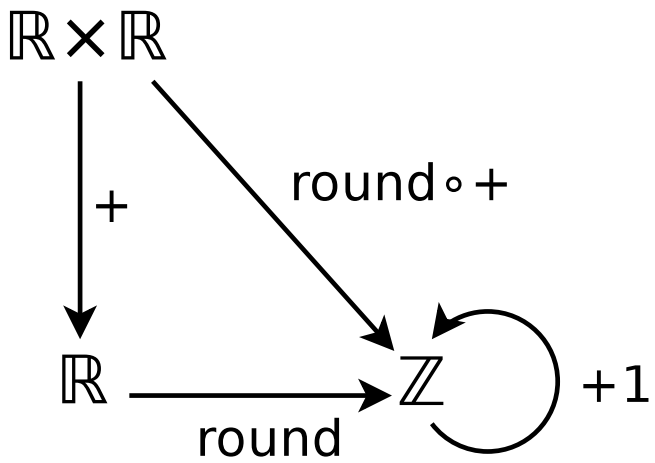
imperatives Beispielprogramm

```
getName :: IO String
getName = do putStrLn  "Getting Name"
             return "Harald"

main :: IO ()
main = do name <- getName
          putStrLn name
```

Kategorientheorie

- Sehr abstrakter Bereich der Mathematik
  - Praktisch alle mathematischen Strukturen sind Kategorien
  - Motivation: “Rechnen mit Funktionstypen”



Kategorientheorie (2)

- Morphismen zwischen Objekten: $f : \mathrm{A} \to \mathrm{B}$
  - Interpretation frei
  - Komposition: $g \circ f$ analog Funktionen
  - Identität: id_A
  - Produkte $(f \times g)$ und Coprodukte $(f + g)$
- Funktoren: Abbildungen zwischen Kategorien
- Weiteres: Natürliche Transformationen, Monaden etc.

Kategorien in Haskell

- Objekte: Haskell-Datentypen
- Arrows: Verarbeitungsprozesse
  - pure Funktionen (Kategorie **Hask**)
  - Funktionen mit Side-Effects (**IOArrow**)
- eigene Arrow-Formalismen möglich
  - müssen Komposition etc. implementieren
  - Funktor von **Hask** erforderlich

Funktoren in dieser Arbeit

- Gleicher Arrow in verschiedenen Kategorien
  - Unterschiedliche Typsignatur möglich

  $\Rightarrow$ "Verstecken" von Teilen der Signatur
- "Domain Specific Categories"
- Beispiel 1: Möglichkeit zum Scheitern ($\mathbf{C}_F$)
  - in $\mathbf{C}_F$: $\mathrm{A} \rightarrow \mathrm{B}$
  - in $\mathbf{C}$: $\mathrm{A} \rightarrow \mathrm{B} + \mathrm{String}$
- Beispiel 2: Zustände ($\mathbf{C}_S$)
  - in $\mathbf{C}_S$: $\mathrm{A} \rightarrow \mathrm{B}$
  - in $\mathbf{C}$: $\mathrm{A} \times \mathrm{state} \rightarrow \mathrm{B} \times \mathrm{state}$

Überblick Architektur

- Komplett Arrow-basiert
  - $\Rightarrow$ Generische Schnittstelle für alle Arten von Compilern
- Compiler: $\mathrm{a} \longrightarrow \mathrm{b}$
- Parser: $[\mathrm{token}] \longrightarrow \mathrm{b}$
- Makros: $[\mathrm{Sexp}] \longrightarrow \mathrm{b}$
- Lisp-Makros: $[\mathrm{Sexp}] \longrightarrow [\mathrm{Sexp}]$

Parser als Arrows

- Eigenschaften von Parsern:
  - Fehlschlagen / Alternativen ($\Rightarrow \mathbf{C}_F$)
  - Zustand: Position im Eingabestream ($\Rightarrow \mathbf{C}_S$)
- Kombination: $\mathbf{C}_P = \mathbf{C}_{FFS}$
  - “innerer Fail” möglich

  $\Rightarrow$ bessere Fehlermeldungen

Parser als Arrows (2)

- Signaturen eines Parsers
  - Parser-Kategorie: $\emptyset \rightarrow \mathrm{A}$
  - "Wirklichkeit": $[\mathrm{token}] \rightarrow (\mathrm{A} \times [\mathrm{token}] + \mathrm{String}) + \mathrm{String}$
  - nach Umwandlung in Compiler: $[\mathrm{token}] \rightarrow \mathrm{A}$
- Token-Typ Variabel: Char und Sexp verwendet

Parser-Bibliothek

- Kombinatoren:
  `empty, eq, member, many, optional etc.`
- Aufruf innerer Parser: `applyParser`

$\Rightarrow$ Makros:

```
macro "addresses" (many (macro "person"
                                compSymbol
                                compSymbol))
```

Komponenten

Komponenten von MagicL

- Modelle
  - Sexp
  - Code
- Eigene DSLs
  - S-Expression-Haskell
  - S-Expression-Compiler
- Außerdem
  - minimales Test-Framework
  - Build-Tool

Parser: String → Sexp

```
data Sexp = Symbol String
          | Node   [Sexp]

whitespace = skip (many (member " \t\n"))

parseSymbol = many1 (notMember " \t\n()") >>>
              symbol

parseNode = skip (eq '(') >>>
            (many parseSexp >>> node) >>>
            skip (eq')')

parseSexp = whitespace >>>
            (parseSymbol <+> parseNode) >>>
            whitespace
```

Quellcode-Modell

- Ansatz von Philip Wadler
- Beschreibung durch Haskell-Funktionen
  - Elementar: `text`, `newline`, `indent`, `append`, `group`
  - Abgeleitet: `lines`, `paragraphs`, `parens` etc.

Beispiel: Pretty-Printer für S-Expressions

```
layoutSexp :: Sexp -> Code
layoutSexp (Symbol sym) = text sym
layoutSexp (Node children) =
    format (map layoutSexp children)
  where format = parens . group . indent2 . lines
```

Sexp-Haskell

- Ziel: S-Expression-basierte Erzeugung von Haskell-Code
- Ansatz: Haskell-Variante in S-Expression-Syntax

Sexp-Haskell

```
(= (sumOfSquares x y)
   (+ x2 y2)
 (where
  (= x2 (* x x))
  (= y2 (* y y))))
```

Haskell

```
sumOfSquares x y = (x2 + y2)
  where
    x2 = (x * x)
    y2 = (y * y)
```

Implementation von Sexp-Haskell

- Zeigt Einbindung typisierter Modelle
- Drei Komponenten
  - Haskell-Modell
  - Parser $\mathrm{Sexp} \rightarrow \mathrm{Haskell}$
  - Funktionaler Compiler $\mathrm{Haskell} \rightarrow \mathrm{Code}$
- Nutzt automatisches Finden von Compilern

$\Rightarrow$ Formulierung etwas umständlich

Haskell-Modell

```
...

data Pattern = ListPattern [Pattern]
             | TuplePattern [Pattern]
             | ConsPattern [Pattern]
             | StringPattern String
             | CallPattern Call

...
```

Sexp → Haskell

```
...

instance Compilable (SexpParser Pattern) [Sexp] Pattern
 where
  comp =
    (liftA1 ListPattern (macro "List" comp)   <+>
     liftA1 TuplePattern (macro "Tuple" comp) <+>
     liftA1 ConsPattern (macro "Cons" comp)    <+>
     liftA1 StringPattern (macro "Str" (many comp >>^
                                        unwords)) <+>
     liftA1 CallPattern comp)

...
```

Haskell → Code

```
...

instance Compilable (Pattern -> Code) Pattern Code where
  comp (ListPattern pats)  = list $ map comp pats
  comp (TuplePattern pats) = tuple $ map comp pats
  comp (ConsPattern pats)  = parenFoldOp ":" (map comp
                                                pats)
  comp (StringPattern str) = string str
  comp (CallPattern call)  = comp call
...
```

Sexp-Compiler

- DSL für untypisierte S-Expression-Verarbeitung ähnlich Lisp
  - Quasiquote-Operator für S-Expression-Templates
  - Makro-Typ: $[\mathrm{Sexp}] \rightarrow [\mathrm{Sexp}]$
  - Lisp-Kontrollfluss: Auto-Makros
- Metarekursiv implementiert

Beispiel: Mehrere Rückgaben

```
(autoMac compDefArith defArithFuns
  (' (= (add x y) (+ x y))
     (= (sub x y) (- x y))))
```

Schluss

Zusammenfassung

- Prototyp eines universellen Compiler-Frameworks
- Sauber fundiert: Haskell, Kategorientheorie
- Lisp-inspiriert: S-Expressions, Makros
- Allgemeiner
  - Modelltypen
  - Verarbeitungsprozesse
  - Kontrollfluss

Ausblick

- Compiler-Definitionen sind etwas redundant

⇒ Plan: DSL für Modelldefinitionen

- Code-Modell in S-Expressions
- Unterstützung verschiedener Zielsprachen
- S-Expression-basierte Softwareentwicklung
  - Datenbank
  - Strukturelle IDE
  - Anpassbare Visualisierung

  ⇒ Model-View-Trennung von Code

Vielen Dank für ihre Aufmerksamkeit!