Prototyp eines S-Expression-basierten Rahmenwerks für sprachübergreifende Metaprogrammierung

Diplomarbeit

Benjamin Teuber
Erstbetreuer: Daniel Moldt
Zweitbetreuer: Leonie Dreschler-Fischer

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

18. Januar 2011

Themengebiet

- Thema: Metaprogrammierung
  - Hier: Code-Generierung
  - Generative Metaprogrammierung
  - Statische Metaprogrammierung
- Starke Verbreitung
  - Model-Driven Architecture
  - Domain-Specific Languages
  - Auch im AOSE-Projekt

Motivation

- Problem: Generator bauen ist aufwändig
- Viele Komponenten nötig
  - Parser für Quellsprache
  - Compiler in normaler Programmiersprache
  - Templates für Zielsprache
- Wie können wir dies vereinfachen?

Zielsetzung

- Entwurf eines eigenen Frameworks
- Davor:
  - Vergleich bestehender Systeme
  - Festlegen der Anforderungen
  - Planung der Architektur

Inhalt

- Vorarbeit
  - Vorhandene Technologien
  - Anforderungen
- MagicL
  - Architektur
  - Demo

Anwendungsbeispiel

- Slide, eine DSL für Vortragsfolien
- Wird nach Latex kompiliert

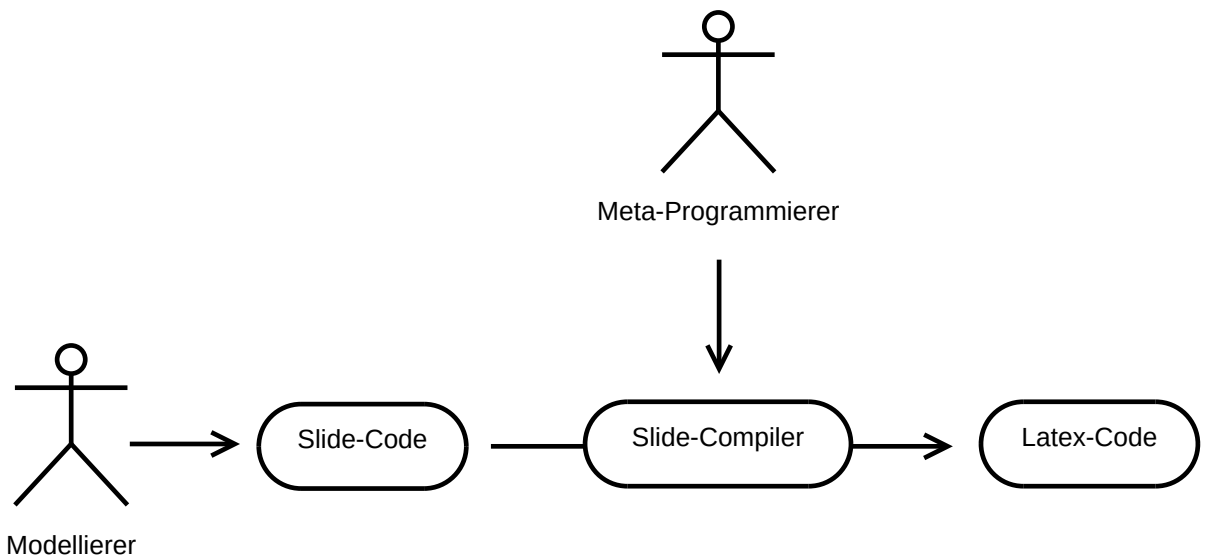


Abbildung: Der Slide-Compiler

Vorhandene Technologien

Unterschiede bestehender Werkzeuge

- Modellrepräsentation
  - Zeichenketten
  - Objekte
  - Bäume
- Art der Code-Erzeugung
  - Templates
- Meta-Programmiersprache
  - Mächtigkeit
  - Ausdrucksstärke

Modellrepräsentation

- Zeichenketten
- Pro:
  - Nötig für Serialisierung/Ausgabeformat
  - Einfache Manipulation
  - Universell
- Contra:
  - Low-Level
  - Anfällig für Syntaxfehler
  - Unstrukturiert

Modellrepräsentation (2)

- Typisierte Objekte
- Pro:
  - Typsicherheit
  - Effizienz
- Contra:
  - Typdeklarationen nötig
  - Manipulation komplizierter

Modellrepräsentation (3)

- Ungetypte Bäume (XML, S-Expressions, JSON)
- Pro:
  - Einfach
  - Strukturiert
  - Universell
  - Sicherer als Strings
- Contra:
  - Nicht so sicher/effizient wie Objekte

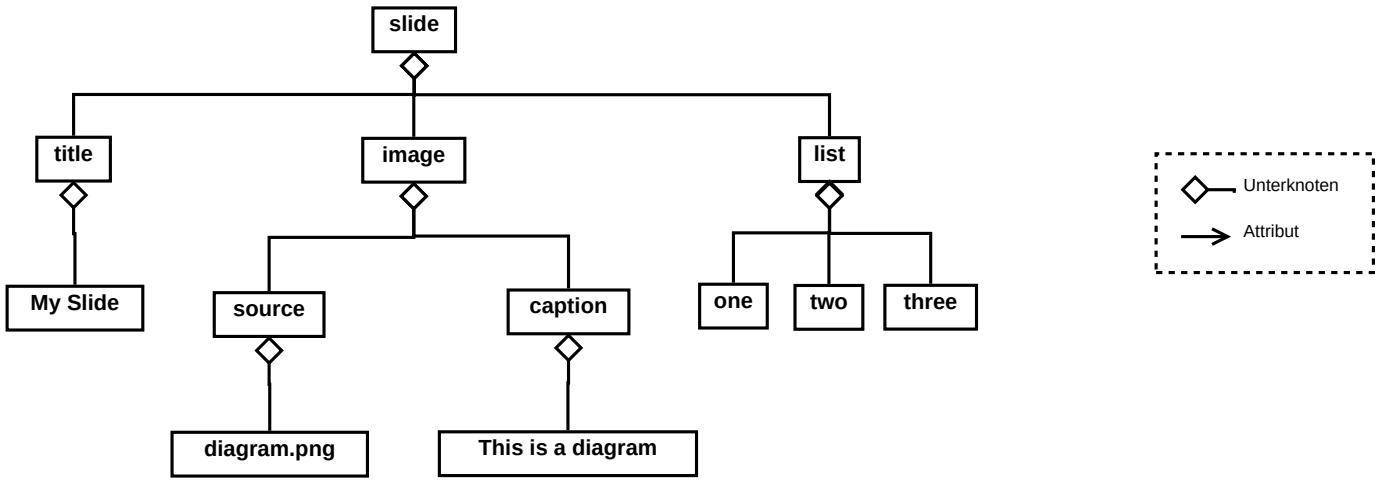
S-Expressions

- 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
- Minimale Komplexität (vgl. XML)

S-Expressions (2)

Beispiel: Folie in S-Expressions

```
(slide
  (title My Slide)
  (image
    (source diagram.png)
    (caption This is a diagram))
  (list one two three))
```



Templates

- Für Dokumenterzeugung genutzt
- “Schablone mit Platzhaltern”
- Funktion: $\mathrm{input} \rightarrow \mathrm{doc}$
- Typen input und doc variieren

PHP

- Object → String
- Zeichenketten problematisch da Fehleranfällig
- Komplette Programmiersprache

```
<ul>
  <? foreach ($items as $item) {
       echo("<li>" . $item . "</li>");
     }
  ?>
</ul>
```

XSL Transformation

- $\mathrm{XML} \rightarrow \mathrm{XML}$
- Mächtig, aber umständliche Syntax
  - $\Rightarrow$ Praxis: Komplexe Verarbeitung in externer Programmiersprache
- Ermöglicht direkten Aufruf sowie Matching über XPath
- XPath: `/book[@price>35]/title`

XSLT (2)

Eingabe

```
<list>
  <item>foo</item>
  <item>bar</item>
  <item>baz</item>
</list>
```

Erwünschte Ausgabe

```
<ul>
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ul>
```

XSLT (3)

```
<xsl:template match="/list">
  <ul>
    <xsl:for-each select="item">
      <li><xsl:value-of select="." /></li>
    </xsl:for-each>
  </ul>
</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
- Template-Syntax:
  - Quasiquote mit ‘
  - Unquote mit ,

Lisp-Makros (2)

Eingabe

```
(list
  (item foo)
  (item bar)
  (item baz))
```

Erwünschte Ausgabe

```
(ul
  (li foo)
  (li bar)
  (li baz))
```

Lisp-Makros (3)

Makro-Umsetzung in Common Lisp

```
(defmacro item (text)
  `(li ,text))

(defmacro list (&rest items)
  `(ul ,@items))
```

Anforderungen

Lisp-inspiriertes Framework

- S-Expressions
- Verallgemeinerte Makros
- Quasiquote
- Funktionale Metasprache

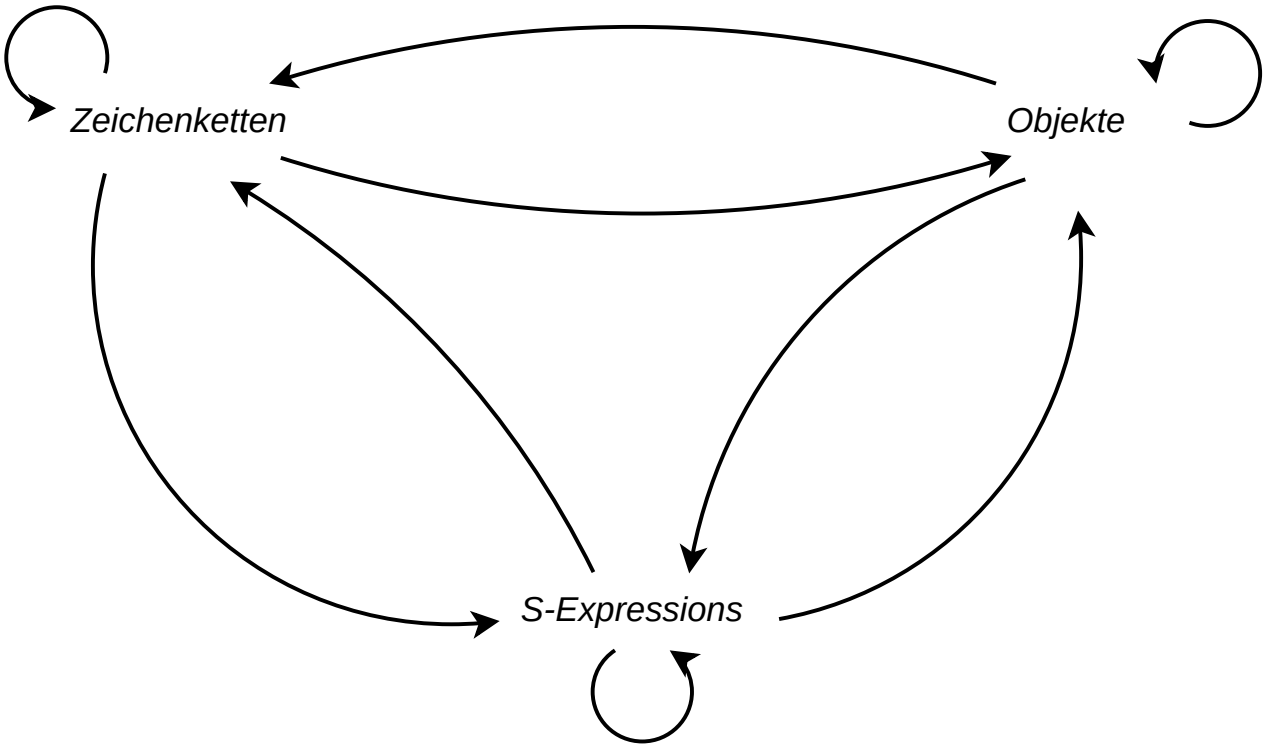
Unterstützte Modelltypen

- S-Expressions
  - Hauptfokus
  - Standard für User-DSL
  - $\Rightarrow$ Parser entfällt
- Strings
  - Parser
  - Generatoren
  - möglichst vom Benutzer versteckt
- Objekte
  - Für komplexere Berechnungen
  - Typsicherheit
  - Effizienz

Unterstützte Modelltypen (2)

- Alle drei Modelltypen verwendbar
  - Als Eingabe
  - Als Ausgabe

  $\Rightarrow$ Neun Mögliche Compiler-Arten



S-Expressions

- Einfachste Baumstruktur
- Hier: **Keine** weitere Syntax

$\Rightarrow$ Strings etc. müssen eingebettet werden

```
(Str Hallo Welt)
```

Quasiquote

- Syntax angepasst
- beliebig viele Rückgaben möglich

Quasiquote-Beispiele

```
(' a b c d e)
(' (+ 3 (, x)))
(' (a b c (,@ list) h i))
```

Verallgemeinerte Makros

- Parser auf S-Expressions
- Aufruf genauer steuerbar
  - Nicht nur Matching vom Knotennamen
  - Lokale Makros möglich
  - XPath
  - EBNF

List-Makro in MagicL

```
(macro "item"
  (>>^ take
       (fun text
            (' (li (, text))))))
```

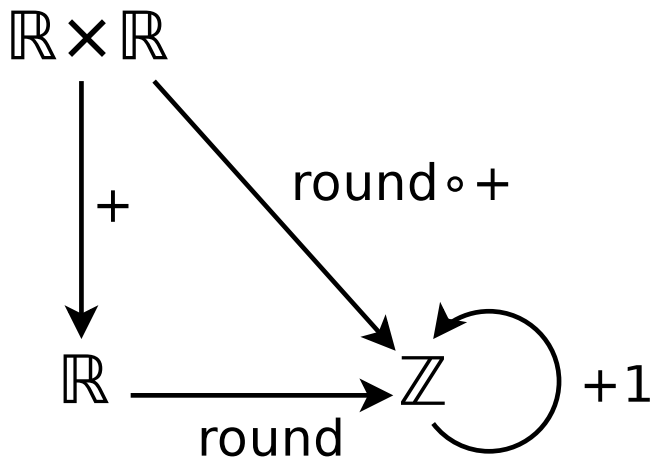
Architektur

Programmiersprache

- Haskell
- Funktional
- statische Typinferenz
- Kategorientheorie-Bibliotheken

Kategorientheorie

- Sehr abstrakter Bereich der Mathematik
- Grundidee
  - "Rechnen mit Funktionstypen"
  - Von Funktionen abstrahieren
  - $\Rightarrow$ flexible Verarbeitungsprozesse
- visualisierbar
- Vergleich mit Petri-Netzen
  - "Punktfrei"
  - Komposition überladbar

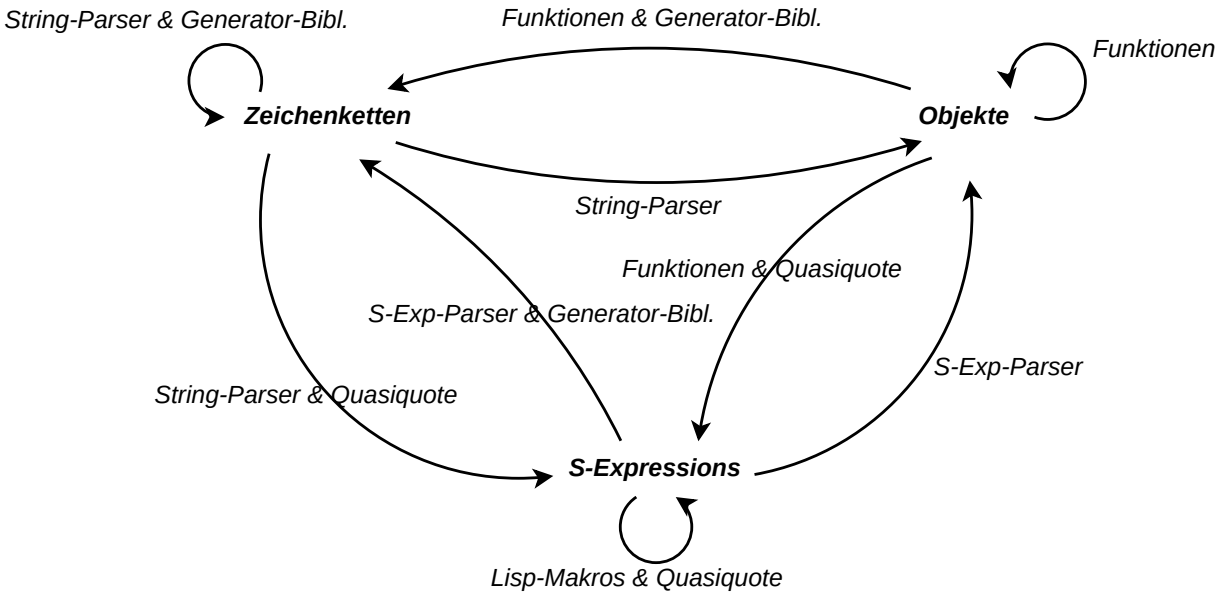


Kategorien im MagicL

- Überall genutzt
  - Parser
  - Compiler
  - Diagramme in Doku
- Umsetzung kategorientheoretischer Konzepte
  - teils bereits in Haskell
  - teils in Arrow-Bibliothek

Unterstützung der Modelltypen

- Strings
  - Parser
  - Generatoren
- S-Expressions
  - Parser
  - Quasiquote
- Objekte
  - Funktionen



Bereitgestellte DSLs

- Haskell-DSL
  - Funktionale S-Expression-Sprache
  - “geklammerte Haskell-Version”
  - Als Metasprache verwendet
  - Nach Haskell übersetzt
- Compiler-DSL
  - Für Compiler-Definitionen
  - Beinhaltet
    - Quasiquote
    - Lisp-Makros
    - Haskell-DSL
  - Nach Haskell-DSL übersetzt

Haskell-DSL-Beispiel

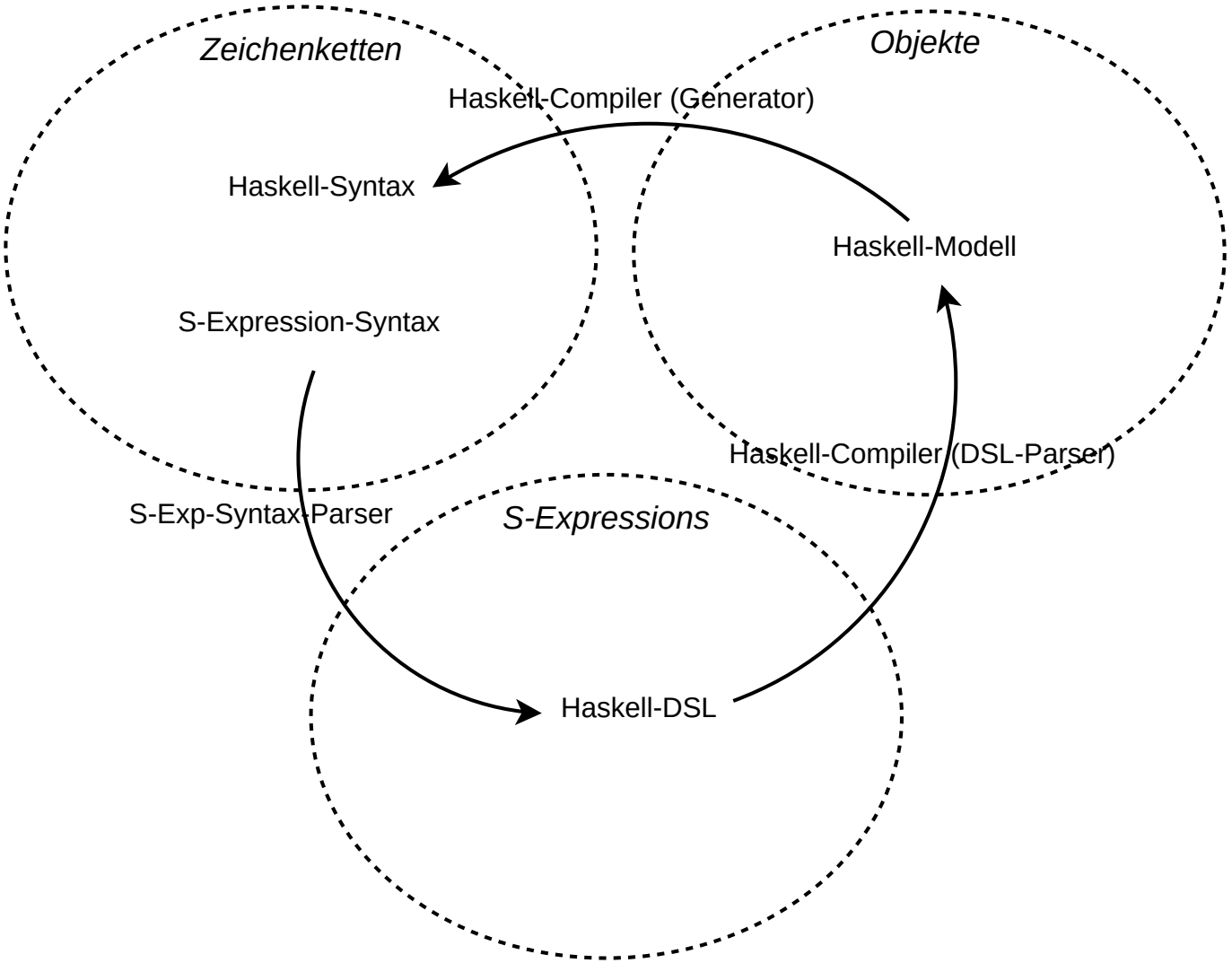
Haskell-DSL-Code

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

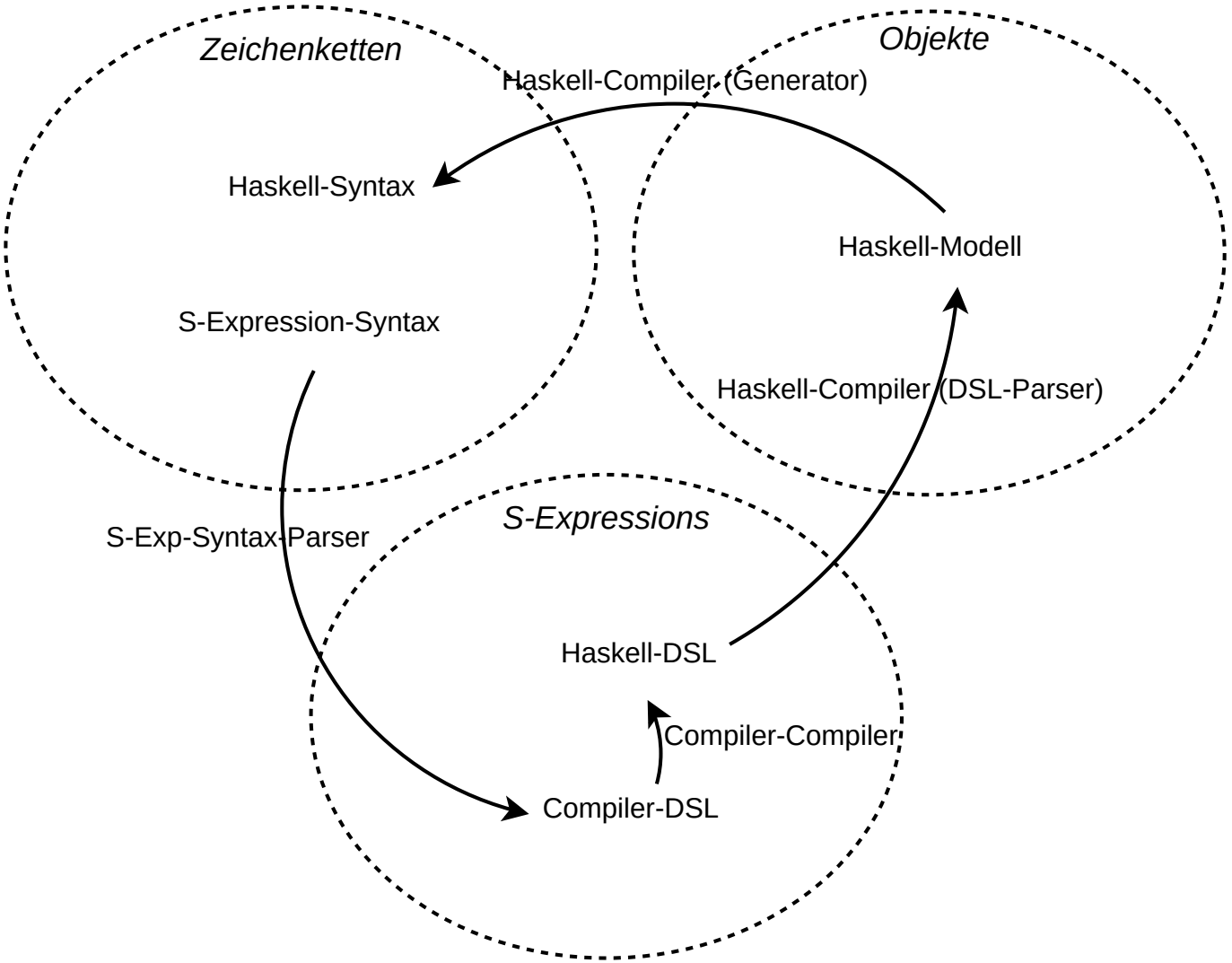
Erzeugter Haskell-Code

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

Übersetzung der Haskell-DSL



Übersetzung der Compiler-DSL



Parser-Bibliothek

- Konstruktoren
  - `empty`
  - `eq`
  - `member`
  - ...
- Kombinatoren
  - `many`
  - `optional`
  - ...

Generator-Bibliothek

- Funktionale Erzeugung von Code

Beispiel

```
layout 80 (braces
(indent2
(lines [text "foo",
  words [text "hello", text "world"],
  commaSep [text "1", text "2", text "3"]])))
```

Erzeugter Code

```
{foo
  hello world
  1, 2, 3}
```

Weitere Komponenten

- kleines Test-Framework
- Build-Tool

Demo

Demo

- Slide-DSL

⇒ Latex-DSL

⇒ Latex-Code

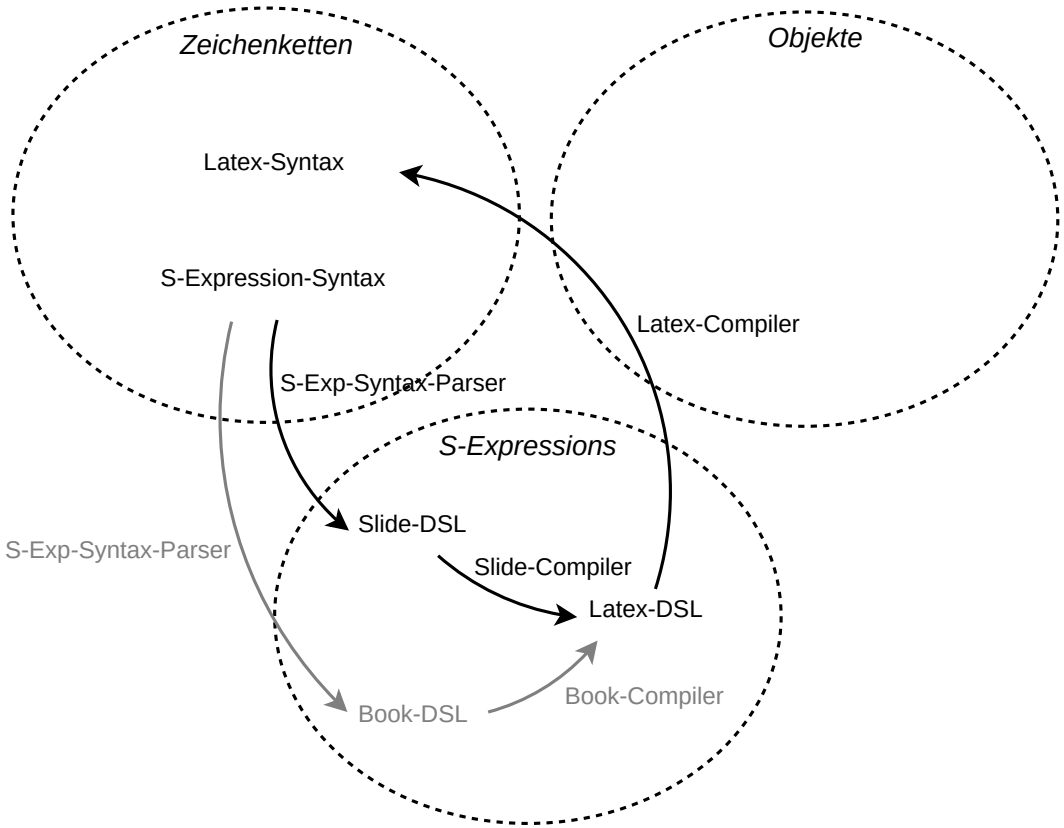


Abbildung: Übersetzung der Slide-DSL

Schluss

Zusammenfassung

- Prototyp eines universellen Compiler-Frameworks
- Lisp-inspiriert: S-Expressions, Makros
- Architektur
  - Haskell
  - Kategorientheorie
- Verallgemeinerungen gegenüber Lisp
  - Modelltypen
  - Verarbeitungsprozesse
  - Kontrollfluss

Fazit

- Themengebiet sehr interessant
- Framework funktioniert
- Manches noch etwas umständlich
  - Preis für Verallgemeinerung
  - Kann durch Code-Generierung vereinfacht werden

Ausblick

- Implementierung sehr komplex
  - $\Rightarrow$ nächstes Mal KISS-Prinzip!
- Einfacher: selbst ein Lisp verwenden
  - Bekannte Lisps wirkten veraltet und unsauber
  - Inzwischen gibt es Clojure!

Vielen Dank für ihre Aufmerksamkeit!