Template Metaprogramming for Haskell

Nicole Stender

Goethe Universität Frankfurt am Main

21. Januar 2014

"Compile-time program optimizations are similar to poetry: more are written than are actually published in commercial compilers. [. . .] An alternative is to allow programmers to define their own compile-time optimizations."

Quelle: Sheard and Jones [7]

Inhaltsverzeichnis I

Grundbegriffe

Was ist Metaprogrammierung?

> *Metaprogrammierung ist die Disziplin, Programme zu schreiben, die andere Programme oder sich selbst repräsentieren oder manipulieren.*

(Quelle: Meixner [3])

Template Haskell:

in Haskell geschriebene Erweiterung

Grundbegriffe

Was ist Metaprogrammierung?

> *Metaprogrammierung ist die Disziplin, Programme zu schreiben, die andere Programme oder sich selbst repräsentieren oder manipulieren.*

(Quelle: Meixner [3])

Template Haskell:

> in Haskell geschriebene Erweiterung

Template Haskell Operatoren

Splice `$(...)`
Ausdrücke werden zur Compile-Zeit ausgeführt

Quasi-Quotes `[| ... |]`
Ausdrücke werden zur Compile-Zeit transformiert

Template Haskell Operatoren

Splice `$(...)`
Ausdrücke werden zur Compile-Zeit ausgeführt

Quasi-Quotes `[| ...|]`
Ausdrücke werden zur Compile-Zeit transformiert

Aufbau der Template Haskell Erweiterung

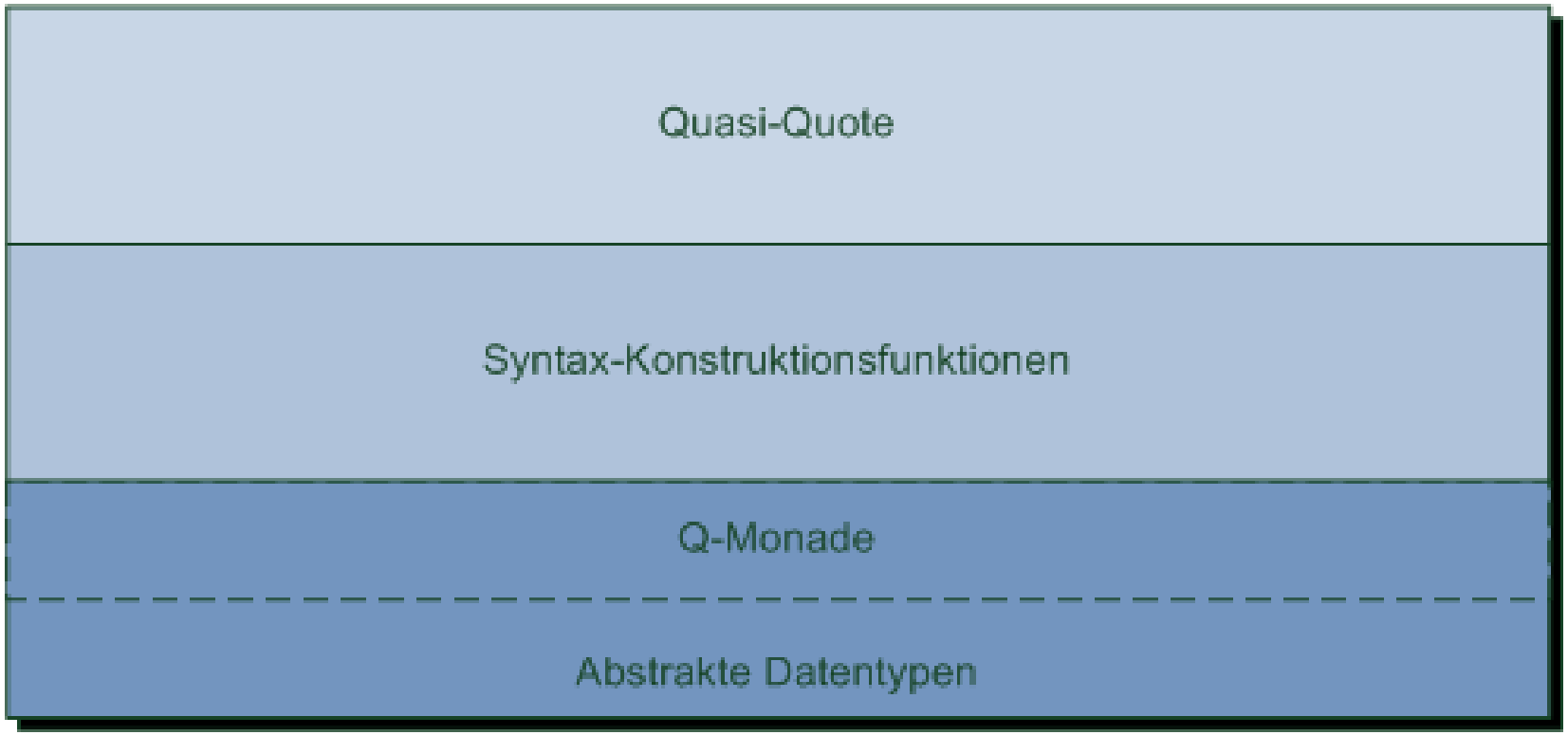

Abbildung: 1 Aufbau der Template Haskell Bibliothek (Quelle: Pribbernow [5])

Algebraische Datentypen

- Fundament der Haskell Bibliothek
- Darstellung von Haskell Programmen
  - Ausdrücke - `Exp`
  - Deklarationen - `Dec`
  - Typen - `Typ`
- **Vorteil:** Analyse
- **Nachteil:** hoher Aufwand, wenig Unterstützung von semantischen Funktionen (Scoping, Typüberprüfung)

Algebraische Datentypen

- Fundament der Haskell Bibliothek
- Darstellung von Haskell Programmen
  - Ausdrücke - `Exp`
  - Deklarationen - `Dec`
  - Typen - `Typ`
- **Vorteil:** Analyse
- **Nachteil:** hoher Aufwand, wenig Unterstützung von semantischen Funktionen (Scoping, Typüberprüfung)

Algebraische Datentypen

- Fundament der Haskell Bibliothek
- Darstellung von Haskell Programmen
  - Ausdrücke - `Exp`
  - Deklarationen - `Dec`
  - Typen - `Typ`
- **Vorteil:** Analyse
- **Nachteil:** hoher Aufwand, wenig Unterstützung von semantischen Funktionen (Scoping, Typüberprüfung)

Algebraische Datentypen

- Fundament der Haskell Bibliothek
- Darstellung von Haskell Programmen
  - Ausdrücke - `Exp`
  - Deklarationen - `Dec`
  - Typen - `Typ`
- **Vorteil:** Analyse
- **Nachteil:** hoher Aufwand, wenig Unterstützung von semantischen Funktionen (Scoping, Typüberprüfung)

Q-Monade

- erweitert Haskells Monadensystem
- **Hauptaufgabe :** Generierung 'frischer' Namen
- Namenskonflikte sollen vermieden werden

```
gensym :: String → Q String
```

- **Problem:** ist nur innerhalb der Q-Monade nutzbar

Q-Monade

- erweitert Haskells Monadensystem
- **Hauptaufgabe :** Generierung 'frischer' Namen
- Namenskonflikte sollen vermieden werden

```
gensym :: String → Q String
```

- **Problem:** ist nur innerhalb der Q-Monade nutzbar

Q-Monade

- erweitert Haskells Monadensystem
- **Hauptaufgabe :** Generierung 'frischer' Namen
- Namenskonflikte sollen vermieden werden

```
gensym :: String → Q String
```

- **Problem:** ist nur innerhalb der Q-Monade nutzbar

Q-Monade

- erweitert Haskells Monadensystem
- **Hauptaufgabe :** Generierung 'frischer' Namen
- Namenskonflikte sollen vermieden werden

```
gensym :: String → Q String
```

- **Problem:** ist nur innerhalb der Q-Monade nutzbar

Syntax-Konstruktionsfunktionen

Transformation von algebraischen Datentypen in monadische Datentypen

Transformationsregeln:

- dreistellige Typnamen sind die Bezeichnung eines *algebraischen* Datentyps - `Exp`
- vierstellige Typnamen sind die Bezeichnung eines *monadischen* Datentyps - `Expr`
- großgeschriebene Funktionen sind die Bezeichnung eines *algebraischen* Datenkonstruktors - `App`
- kleingeschriebene Funktionen sind die Bezeichnungen eines *monadischen* Datenkonstruktors - `app`

Syntax-Konstruktionsfunktionen

Transformation von algebraischen Datentypen in monadische Datentypen

Transformationsregeln:

- dreistellige Typnamen sind die Bezeichnung eines *algebraischen* Datentyps - `Exp`
- vierstellige Typnamen sind die Bezeichnung eines *monadischen* Datentyps - `Expr`
- großgeschriebene Funktionen sind die Bezeichnung eines *algebraischen* Datenkonstruktors - `App`
- kleingeschriebene Funktionen sind die Bezeichnungen eines *monadischen* Datenkonstruktors - `app`

Syntax-Konstruktionsfunktionen

Transformation von algebraischen Datentypen in monadische Datentypen

Transformationsregeln:

- dreistellige Typnamen sind die Bezeichnung eines *algebraischen* Datentyps - `Exp`
- vierstellige Typnamen sind die Bezeichnung eines *monadischen* Datentyps - `Expr`
- großgeschriebene Funktionen sind die Bezeichnung eines *algebraischen* Datenkonstruktors - `App`
- kleingeschriebene Funktionen sind die Bezeichnungen eines *monadischen* Datenkonstruktors - `app`

Syntax-Konstruktionsfunktionen

Transformation von algebraischen Datentypen in monadische Datentypen

Transformationsregeln:

- dreistellige Typnamen sind die Bezeichnung eines *algebraischen* Datentyps - `Exp`
- vierstellige Typnamen sind die Bezeichnung eines *monadischen* Datentyps - `Expr`
- großgeschriebene Funktionen sind die Bezeichnung eines *algebraischen* Datenkonstruktors - `App`
- kleingeschriebene Funktionen sind die Bezeichnungen eines *monadischen* Datenkonstruktors - app

Quasi-Quote

```
[| Expr |]
```

- Repräsentation von Haskell Programmen
- Erstellung der Templates
- Arbeitsweise ist vergleichbar mit einem Parser → transformiert algebraische Konstrukte in monadische Ausdrücke
- anstelle eines Ausdrucks `[e|...|]`, Pattern `[p|...|]`, Typ `[t|...|]`, Deklaration `[d|...|]`

Quasi-Quote

```
[| Expr |]
```

- Repräsentation von Haskell Programmen
- Erstellung der Templates
- Arbeitsweise ist vergleichbar mit einem Parser → transformiert algebraische Konstrukte in monadische Ausdrücke
- anstelle eines Ausdrucks `[e|...|]`, Pattern `[p|...|]`, Typ `[t|...|]`, Deklaration `[d|...|]`

Quasi-Quote

```
[| Expr |]
```

- Repräsentation von Haskell Programmen
- Erstellung der Templates
- Arbeitsweise ist vergleichbar mit einem Parser → transformiert algebraische Konstrukte in monadische Ausdrücke
- anstelle eines Ausdrucks `[e|...|]`, Pattern `[p|...|]`, Typ `[t|...|]`, Deklaration `[d|...|]`

Reifizierung

- Quellcode des Programmes zu analysieren
- Zustand der internen Symboltabelle abfragen

```
reify :: Name → Q Info
```

- Information kann analysiert und weiterverarbeitet werden

Reifizierung

- Quellcode des Programmes zu analysieren
- Zustand der internen Symboltabelle abfragen

$$\texttt{reify :: Name} \rightarrow \texttt{Q Info}$$

- Information kann analysiert und weiterverarbeitet werden

Reifizierung

- Quellcode des Programmes zu analysieren
- Zustand der internen Symboltabelle abfragen

$$\texttt{reify :: Name} \rightarrow \texttt{Q Info}$$

- Information kann analysiert und weiterverarbeitet werden

Typisierung

Haskell = stark getypt = Typüberprüfung zur Compile-Zeit

Ablauf einer Typüberprüfung:

1. Typüberprüfung
2. Kompilierung
3. Ausführung

Keine Typfehler zur Laufzeit!

Stufenweise Typisierung

```
$(printf "Error: %s on line %d") msg line
```

1. Typüberprüfung innerhalb der Splice

```
(printf "Error: %s on line %d") :: Expr
```

2. Kompiliere
3. Führe `splice` aus

```
(\s0 → \n1 →
"Error:" ++ s0 ++ "on line" ++ show n1)
```

4. Typüberprüfe dieses Ergebnis

Erkenntnis: Ausführung und Typüberprüfung wechseln sich ab

Problem: `f x = $(zipN x)` - ist nicht zulässig

Stufenweise Typisierung

```
$(printf "Error: %s on line %d") msg line
```

1. Typüberprüfung innerhalb der Splice

   ```
   (printf "Error: %s on line %d") :: Expr
   ```

2. Kompiliere
3. Führe `splice` aus

   ```
   (\s0 → \n1 →
   "Error:" ++ s0 ++ "on line" ++ show n1)
   ```

4. Typüberprüfe dieses Ergebnis

Erkenntnis: Ausführung und Typüberprüfung wechseln sich ab

Problem: `f x = $(zipN x)` - ist nicht zulässig

Stufenweise Typisierung

```
$(printf "Error: %s on line %d") msg line
```

1. Typüberprüfung innerhalb der Splice

   ```
   (printf "Error: %s on line %d") :: Expr
   ```

2. Kompiliere
3. Führe `splice` aus

   ```
   (\s0 → \n1 →
   "Error:" ++ s0 ++ "on line" ++ show n1)
   ```

4. Typüberprüfe dieses Ergebnis

Erkenntnis: Ausführung und Typüberprüfung wechseln sich ab
Problem: `f x = $(zipN x)` - ist nicht zulässig

Zustände

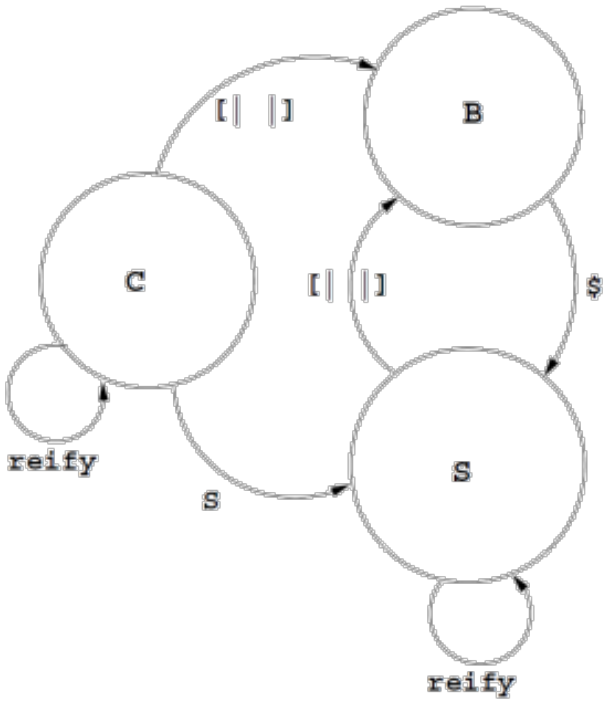

Abbildung: 2 Übersicht über die möglichen Zustände und Zustandsübergänge

- **C - Compiling:** Programmcode ohne Meta-Operatoren
- **B - Bracket:** Programmcode innerhalb von Quasi-Quotes
- **S - Splicing:** Ausdruck, der innerhalb einer Splice steht

Beispiel:

```
f :: Int → Expr
f x = [| foo $(zipN x) |]
```

Zustände

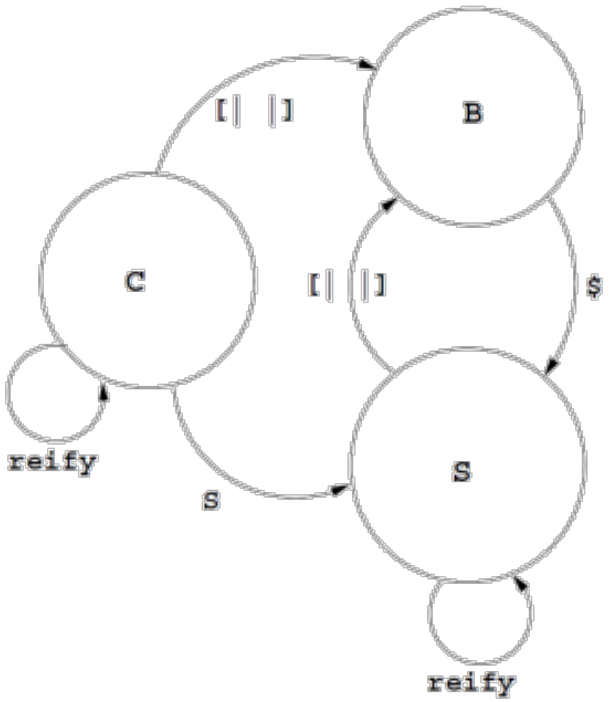

Abbildung: 2 Übersicht über die möglichen Zustände und Zustandsübergänge

- **C - Compiling:** Programmcode ohne Meta-Operatoren
- **B - Bracket:** Programmcode innerhalb von Quasi-Quotes
- **S - Splicing:** Ausdruck, der innerhalb einer Splice steht

Beispiel:

```
f :: Int → Expr
f x = [| foo $(zipN x) |]
```

Zustände

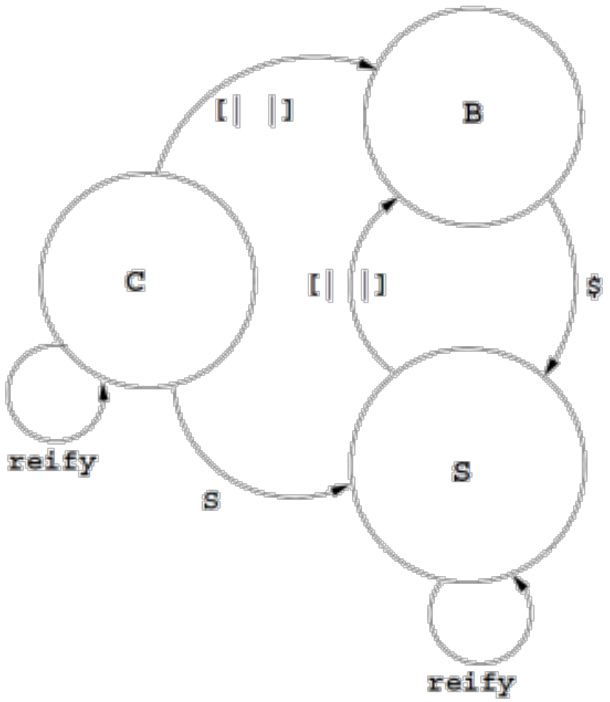

Abbildung: 2 Übersicht über die möglichen Zustände und Zustandsübergänge

- **C - Compiling:** Programmcode ohne Meta-Operatoren
- **B - Bracket:** Programmcode innerhalb von Quasi-Quotes
- **S - Splicing:** Ausdruck, der innerhalb einer Splice steht

Beispiel:

```
f :: Int → Expr
f x = [| foo $(zipN x) |]
```

Zustände

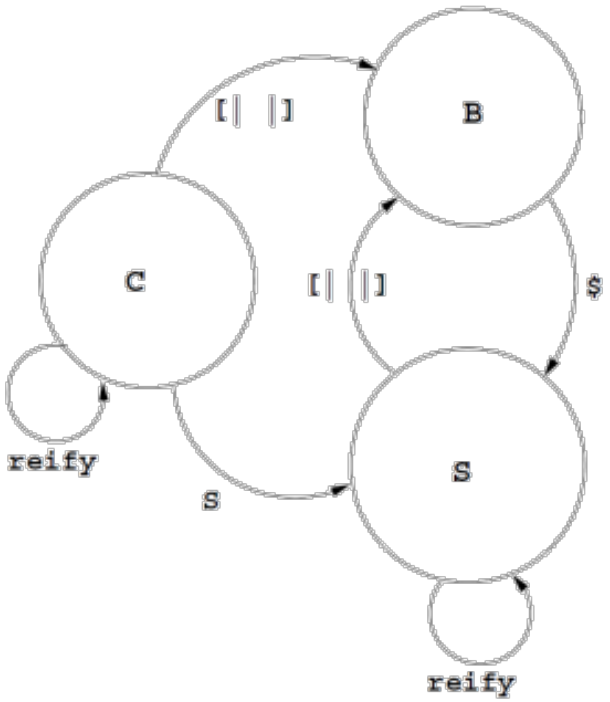

Abbildung: 2 Übersicht über die möglichen Zustände und Zustandsübergänge

- **C - Compiling:** Programmcode ohne Meta-Operatoren
- **B - Bracket:** Programmcode innerhalb von Quasi-Quotes
- **S - Splicing:** Ausdruck, der innerhalb einer Splice steht

Beispiel:

```
f :: Int → Expr
f x = [| foo $(zipN x) |]
```

Zustände

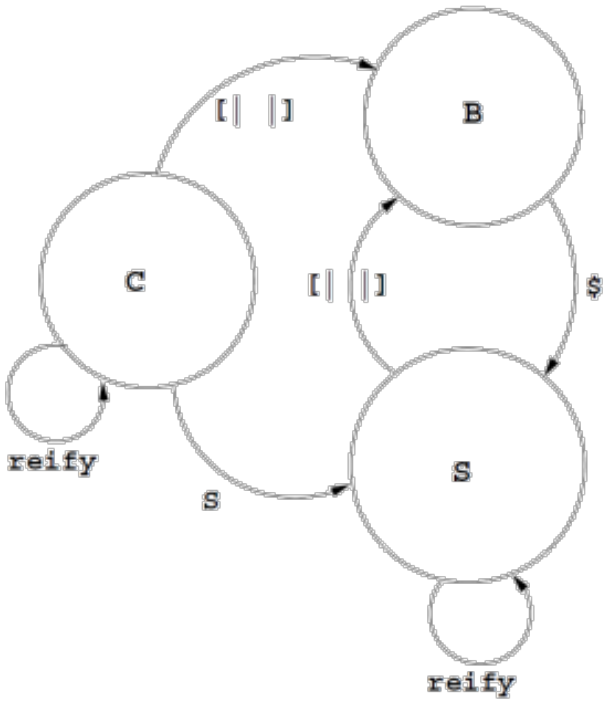

Abbildung: 2 Übersicht über die möglichen Zustände und Zustandsübergänge

- **C - Compiling:** Programmcode ohne Meta-Operatoren
- **B - Bracket:** Programmcode innerhalb von Quasi-Quotes
- **S - Splicing:** Ausdruck, der innerhalb einer Splice steht

Beispiel:

```
f :: Int → Expr
f x = [| foo $(zipN x) |]
```

Level

Unterscheidung von `Splice` innerhalb einer Quasi-Quote und einem Top-Level `Splice`

- Level startet mit 0
- Level wird erhöht: innerhalb einer Quasi-Quote
- Level wird erniedrigt: innerhalb einer Splice (Top-Level)

Beispiel:

```
g x = $(h [| x*2 |])
```

Level

Unterscheidung von `Splice` innerhalb einer Quasi-Quote und einem Top-Level `Splice`

- Level startet mit 0
- Level wird erhöht: innerhalb einer Quasi-Quote
- Level wird erniedrigt: innerhalb einer Splice (Top-Level)

Beispiel:

```
g x = $(h [| x*2 |])
```

Level

Unterscheidung von `Splice` innerhalb einer Quasi-Quote und einem Top-Level `Splice`

- Level startet mit 0
- Level wird erhöht: innerhalb einer Quasi-Quote
- Level wird erniedrigt: innerhalb einer Splice (Top-Level)

Beispiel:

```
g x = $(h [| x*2 |])
```

Level

Unterscheidung von `Splice` innerhalb einer Quasi-Quote und einem Top-Level `Splice`

- Level startet mit 0
- Level wird erhöht: innerhalb einer Quasi-Quote
- Level wird erniedrigt: innerhalb einer Splice (Top-Level)

Beispiel:

```
g x = $(h [| x*2 |])
```

Level

Unterscheidung von `Splice` innerhalb einer Quasi-Quote und einem Top-Level `Splice`

- Level startet mit 0
- Level wird erhöht: innerhalb einer Quasi-Quote
- Level wird erniedrigt: innerhalb einer Splice (Top-Level)

Beispiel:

```
g x = $(h [| x*2 |])
```

Weitere Typisierungsregeln

Es gibt weitere Typisierungsregeln für die Typüberprüfung von Ausdrücken und Deklarationen

$$\Gamma \vdash_s^n e : \tau$$

Γ: Umgebung, e: Ausdruck/Expression, τ: Typ,
s: Zustand der Typüberprüfung, n: Level

- Regeln für das korrekte Typisieren von Splice und Quasi-Quotes von Ausdrücken
- Typüberprüfung von Gruppen von Deklarationen

Weitere Typisierungsregeln

Es gibt weitere Typisierungsregeln für die Typüberprüfung von Ausdrücken und Deklarationen

$$\Gamma \vdash_s^n e : \tau$$

Γ: Umgebung, e: Ausdruck/Expression, τ: Typ,
s: Zustand der Typüberprüfung, n: Level

- Regeln für das korrekte Typisieren von Splice und Quasi-Quotes von Ausdrücken
- Typüberprüfung von Gruppen von Deklarationen

Anwendungsbeispiel

Embedded DSL - Bereitstellung von spezifischen Sprachen

- Reduzierung von Produktionskosten durch Programmkonstrukte
- Verringerung von redundanten Quellcode

Beispiel:

$$\text{n} \times \text{n Matrix M}$$
$$\text{M} * \textsc{inverse}\ \text{M} = \textsc{identity}$$

der Term wird erweitert zu

$$\text{M} * \textsc{inverse}\ \text{M} * \text{n}$$

Ziel: Compiler erkennt $\text{M} * \textsc{inverse}\ \text{M}$ als Identitätsmatrix und setzt diese gleich

$$\textsc{identity} * \text{n}$$

Anwendungsbeispiel

Embedded DSL - Bereitstellung von spezifischen Sprachen

- Reduzierung von Produktionskosten durch Programmkonstrukte
- Verringerung von redundanten Quellcode

Beispiel:

$$\mathrm{n} \times \mathrm{n}\ \text{Matrix}\ \mathrm{M}$$
$$\mathrm{M} * \textsc{inverse}\ \mathrm{M} = \textsc{identity}$$

der Term wird erweitert zu

$$\mathrm{M} * \textsc{inverse}\ \mathrm{M} * \mathrm{n}$$

Ziel: Compiler erkennt $\mathrm{M} * \textsc{inverse}\ \mathrm{M}$ als Identitätsmatrix und setzt diese gleich

$$\textsc{identity} * \mathrm{n}$$

Anwendungsbeispiel

Embedded DSL - Bereitstellung von spezifischen Sprachen

- Reduzierung von Produktionskosten durch Programmkonstrukte
- Verringerung von redundanten Quellcode

Beispiel:

$$\mathsf{n} \times \mathsf{n}\ \text{Matrix}\ \mathrm{M}$$
$$\mathrm{M} * \textsc{inverse}\ \mathrm{M} = \textsc{identity}$$

der Term wird erweitert zu

$$\mathrm{M} * \textsc{inverse}\ \mathrm{M} * \mathsf{n}$$

Ziel: Compiler erkennt $\mathrm{M} * \textsc{inverse}\ \mathrm{M}$ als Identitätsmatrix und setzt diese gleich

$$\textsc{identity} * \mathsf{n}$$

Anwendungsbeispiel

Embedded DSL - Bereitstellung von spezifischen Sprachen

- Reduzierung von Produktionskosten durch Programmkonstrukte
- Verringerung von redundanten Quellcode

Beispiel:

$$\mathrm{n} \times \mathrm{n}\ \text{Matrix}\ \mathrm{M}$$
$$\mathrm{M} * \textsc{inverse}\ \mathrm{M} = \textsc{identity}$$

der Term wird erweitert zu

$$\mathrm{M} * \textsc{inverse}\ \mathrm{M} * \mathrm{n}$$

Ziel: Compiler erkennt $\mathrm{M} * \textsc{inverse}\ \mathrm{M}$ als Identitätsmatrix und setzt diese gleich

$$\textsc{identity} * \mathrm{n}$$

Anwendungsbeispiel

Template Haskell

$$\mathrm{M} * \textsc{inverse}\ \mathrm{M} * \mathrm{n}$$

würde in Template Haskell lauten:

```
exp_mat = [| \M n → M * inverse M * n |]
```

- Quasi-Quote-Ausdruck wird transformiert in eine `Expr`
- Ausdruck `M * inverse M` wird reifziert → `identity`

Anwendungsbeispiel

Template Haskell

$$\mathrm{M} * \textsc{inverse}\ \mathrm{M} * \mathrm{n}$$

würde in Template Haskell lauten:

```
exp_mat = [| \M n → M * inverse M * n |]
```

- Quasi-Quote-Ausdruck wird transformiert in eine `Expr`
- Ausdruck `M * inverse M` wird reifziert → `identity`

Anwendungsbeispiel

Template Haskell

$$\mathrm{M} * \textsc{inverse}\ \mathrm{M} * \mathrm{n}$$

würde in Template Haskell lauten:

```
exp_mat = [| \M n → M * inverse M * n |]
```

- Quasi-Quote-Ausdruck wird transformiert in eine `Expr`
- Ausdruck `M * inverse M` wird reifziert → `identity`

Fazit

- knappe Dokumentation im ursprünglichen Paper von Sheard and Jones [7]
- übersichtlich aufgestellte Bibliothek
- **Nachteil:** Quasi-Quotes allein reichen nicht vollständig aus
- Quellcode wird auf den unteren Schichten unübersichtlich

Aufbau der Template Haskell Erweiterung

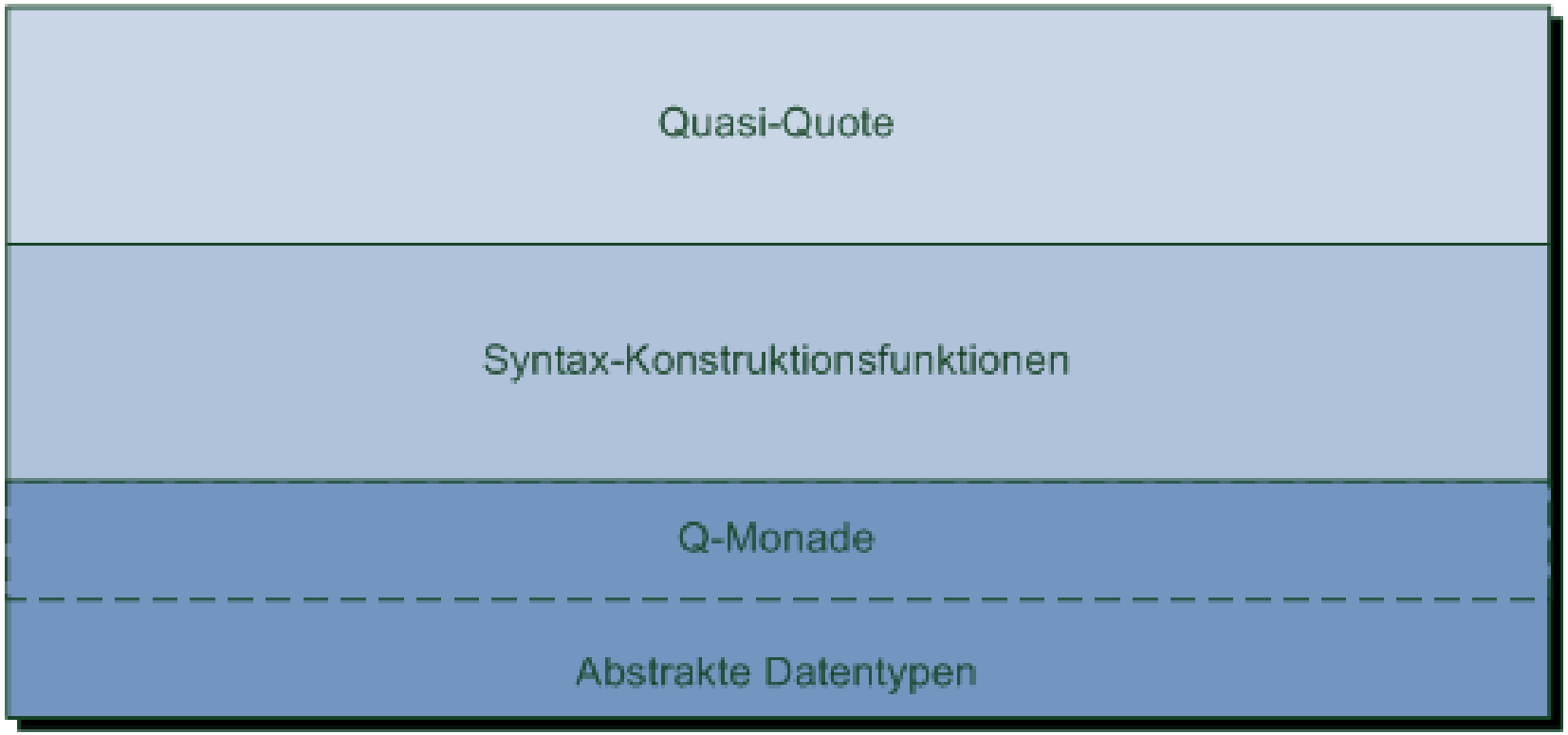

Abbildung: 1 Aufbau der Template Haskell Bibliothek (Quelle: Pribbernow [5])

Danke für Eure Aufmerksamkeit!

Fragen?

[1] Peter Findeisen. Template Haskell. http://www-ps.informatik.uni-kiel.de/~fhu/Seminar/WS05/Findeisen.pdf, 2005. [Online; accessed 14-January-2014].

[2] HaskellWiki2013. Template Haskell. http://www.haskell.org/haskellwiki/Template_Haskell#What_is_Template_Haskell.3F, 2013. [Online; accessed 14-January-2014].

[3] Prof. Meixner. Metaprogrammierung. http://www.hs-augsburg.de/~meixner/prog/skript/metaprogrammierung/Metaprogrammierung.html#GenericsUndReflection. [Online; accessed 14-January-2014].

[4] Ulf Norell and Patrik Jansson. Prototyping Generic Programming in Template Haskell. In Dexter Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 314–333. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22380-1. doi: 10.1007/978-3-540-27764-4_17. URL http://dx.doi.org/10.1007/978-3-540-27764-4_17.

[5] Philipp Pribbernow. Template Haskell. http://www.fh-wedel.de/~si/seminare/ss11/Ausarbeitung/03.TemplateHaskell/index5d2a.html?chapter=headline1#headline1, 2011. [Online; accessed 14-January-2014].

[6] Sean Seefried, Manuel Chakravarty, and Gabriele Keller. Optimising Embedded DSLs Using Template Haskell. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 186–205. Springer Berlin Heidelberg, 2004.

[7] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *ACM, Haskell Workshop*, 2002.