

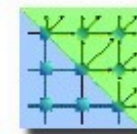
# Moderne Programmiermethoden



# Template Haskell

Johannes Pirkel

Sommersemester 2006

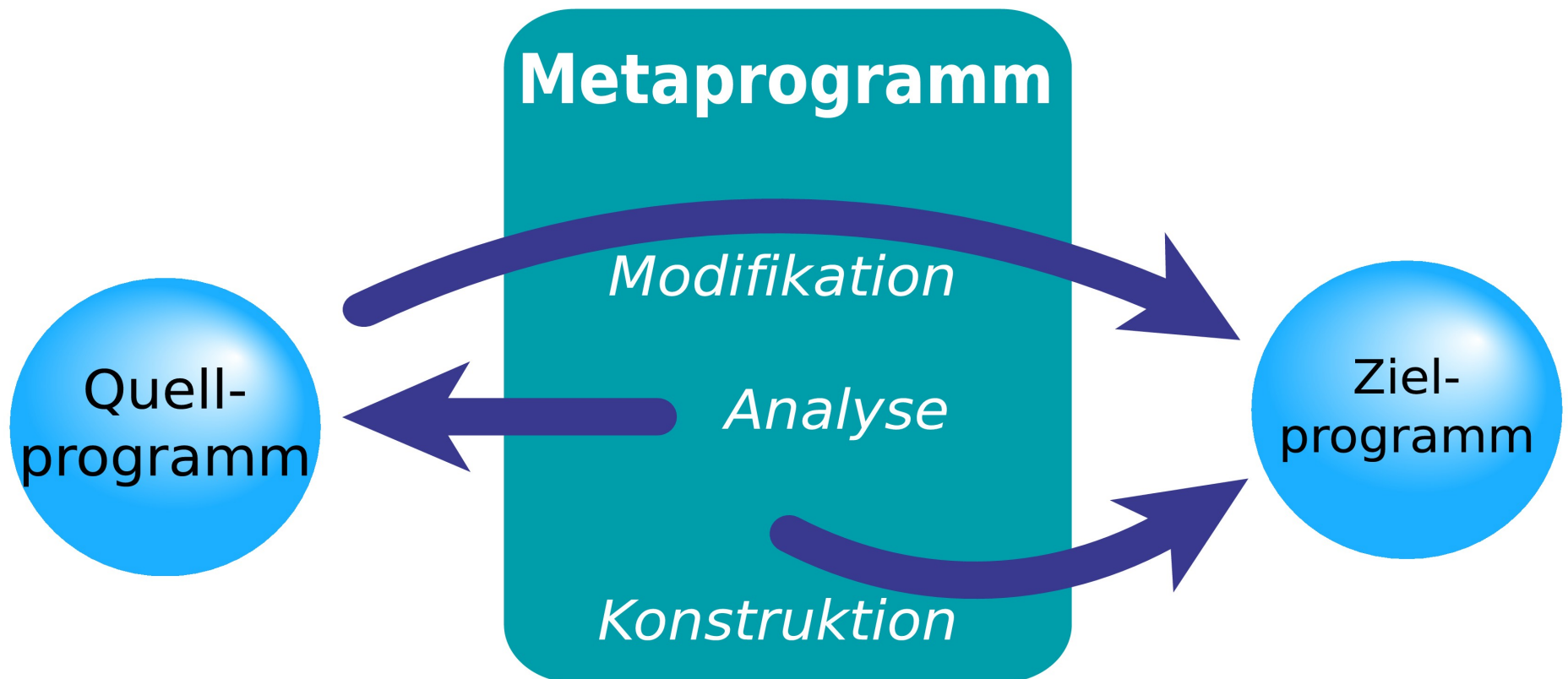


Lehrstuhl für  
Programmierung

# I. Template Haskell als Lösung

# Metaprogrammierung

- Programme als Objekte
- Metaprogramme und Objektprogramme



# Einordnung von Template Haskell

	<b>MetaOCaml</b>	<b>C++-Templates</b>	<b>Template Haskell</b>
Codegenerierung	Laufzeit	Compilierung	Compilierung
Garantie	Typsicherheit	gültige Syntax	gültige Syntax
Codeanalyse	nein	kaum	ja
Kapselung	ja	ja	nein
Multistaging	ja	nein	nein
Homogenität	ja	nein	fast

# Syntax von Template Haskell

- Der Splice-Operator „\$(...)“ markiert Code, der zur Compile-Zeit ausgewertet werden soll, um Objektcode an dieser Stelle einzufügen.
- In Quasiquotierung „[|...|]“ eingeschlossener Code wird in ein Objekt umgewandelt, das im Metaprogramm weiterverarbeitet werden kann.
- Typische Verwendung:  

```
foo = $(transform [| ... |] )
```

# Automatische Codegenerierung

- Algorithmische Programmkonstruktion
  - Manchmal ist das Metaprogramm attraktiver als das Zielprogramm
- Domain Specific Languages
  - Darstellung als Metaprogramm
  - Erzeugung von (ggf. komplexem) problemspezifischem Objektcode
- Vermeidung von „Copy & Paste“-Programmierung

# Beispiel: printf

## Ziel:

Formatierungsfunktion à la „sprintf“ in C mit im String eingebetteten Platzhaltern

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

## Lösung:

```
-- Datentyp fuer Platzhalter und Literale  
data Format = D | S | L String
```

```
parse :: String -> [Format]  
[...]
```

# Beispiel: printf (Fortsetzung)

```
gen :: [Format] -> ExpQ -> ExpQ
gen []          x = x
gen (S : xs)    x = [| \s-> $(gen xs [| $x++s      |])|]
gen (D : xs)    x = [| \n-> $(gen xs [| $x++show n |])|]
gen (L s : xs)  x = gen xs [| $x ++ s |]
```

```
printf :: String -> ExpQ
printf s = gen (parse s) [| "" |]
```

# Beispiel: printf (Fortsetzung)

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

Liefert nun:

```
(\s0 -> \n1 ->  
  ""++"Error: "++ s0 ++" on line "++ show n1) msg  
                                             line
```

# Effizienz Aspekte

- Auswertung zur Compilezeit
  - Inlining
  - partielle Auswertung
- Ausnutzung von domänenspezifischem Spezialwissen

# Beispiel: pow

## Ziel:

Potenzfunktion mit Auswertung von Konstanten im Exponenten bereits zur Compile-Zeit

`$(pow 3) x`

## Lösung:

```
pow :: Int -> ExpQ
```

```
pow 0 = [| \x -> 1 |]
```

```
pow n | n>0 = [| \x -> x * $(pow (n-1)) x |]
```

# Beispiel: pow (Fortsetzung)

`$(pow 3) x`

Liefert also durch Auswertung der Konstante „3“:

```
(\x1 ->(x1 * (\x2 ->(x2 * (\x3 ->(x3 * (\x4 ->1)
                                     x3)))
          x2 ))
      x1))
```

x

Man beachte die hierbei automatisch vorgenommene Erzeugung frischer Namen.

# Beispiel: pow'

## Ziel:

Intelligenteres pow ohne überflüssige Lambdaausdrücke

`$(pow' 3) x`

## Lösung:

```
prod :: ExpQ -> Int -> ExpQ
```

```
prod x 0 = [| 1 |]
```

```
prod x n | n>0 = [| $x * $(prod x (n-1)) |]
```

```
pow' :: Int -> ExpQ
```

```
pow' n = [| \x -> $(prod [|x|] n) |]
```

# Beispiel: pow' (Fortsetzung)

`$(pow' 3) x`

Liefert nun:

`(\x1 -> (x1 * (x1 * (x1 * 1)))) x`

# Polytypische Programmierung

- Normales Typsystem evtl. zu starr
  - Länge eines Tupels legt Typ mit fest
- Lösung im Metaprogramm
  - Erzeugung eines passenden Ausdrucks
  - Parameter bestimmen Signatur im Zielcode

# Beispiel: sel

## Ziel:

Auswahlfunktion für Tupel beliebiger Länge;  
Dazu Erzeugung eines passenden Pattern-Matches im  
Metaprogramm

```
$(sel (0,3)) x
```

## Lösung:

```
sel :: (Int,Int) -> ExpQ
sel (i,n) =
  do
    xs <- sequence [ newName ("x"++show j)
                      | j<-[0..n-1] ]
    lamE [tupP (map varP xs)] (varE (xs!!i))
```

# Beispiel: sel (Fortsetzung)

$\$(\text{sel } (0,3)) \text{ } x$

Liefert nun:

$( \ \backslash(x1, x2, x3) \rightarrow x1 \ ) \ \ x$

# Vorteile bei der Entwicklung

- Conditional Compilation
  - optionale Codeteile
  - Debugging
  - Anpassung an Plattform, Konfiguration, ...
- verbesserte Fehleranalyse
  - Reification
  - Insbesondere Verweis auf fehlerhafte Stelle

## II. Template Haskell im Überblick

# Splicing

- Syntax: „\$( . . . )“
- Fügt Code in das Objektprogramm ein und wertet dazu sein Argument aus
  - Expression Splicing
  - Declaration Splicing
  - Type Splicing (geplant)

# Quasiquotierung

- Erzeugung von Objektcode im Metaprogramm
  - In der Quotierung: normale Syntax
  - Splices sind erlaubt
  - Automatische Vermeidung von Namenskonflikten mit eingesplicetem Code

Ausdrücke            [ | . . . | ]

Deklarationen        [ d | . . . | ]

Typen                 [ t | . . . | ]

# Cross-Stage Persistence

- Verwendung von Werten zur Compile-Zeit gebundener Variablen im generierten Code
- Unabhängig vom Scope der Applikationsstelle
  - Keine Verdeckung
  - ursprüngliche Variable muss nicht sichtbar sein
- Analog zur herkömmlichen Semantik also Lexikalisches Scoping

# Beispiel: Cross-Stage Persistence

## Definition

```
module T (genswap) where
  swap (a,b) = (b,a)
  genswap x = [| swap x |]
```

} Lexikalischer Scope,  
also „swap“

## Applikation

```
module Foo where
  import T (genswap)
  swap = True
  foo = $(genswap (4,5))
```

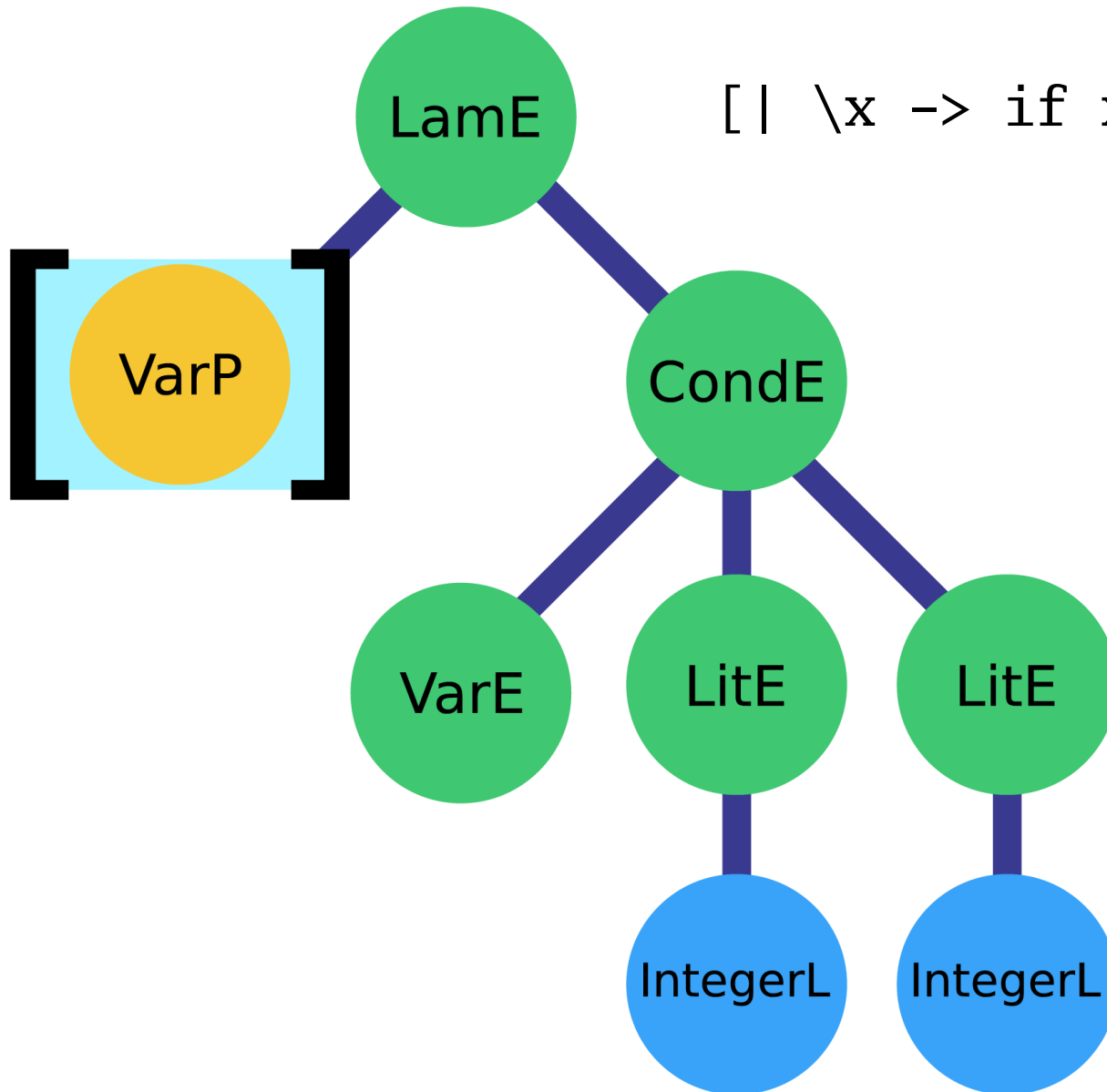
} Keine Verdeckung durch  
„swap“, sondern  
Verwendung von „swap“

# Syntaxkonstruktoren

- Repräsentation von Objektcode durch algebraische Datentypen
- Sprachelemente (Deklarationen, Ausdrücke usw.) als Datentypen
  - Verschiedene Unterarten der Sprachelemente (Lambdaausdruck, „if“) als Konstruktoren
  - Hierarchische Struktur
  - Repräsentation des Syntaxbaums
  - Konstruktionsfunktionen
- Typkorrekte Metaprogramme erzeugen syntaktisch korrekte Objektprogramme
- Keine automatische Vermeidung von Namenskonflikten

# Beispiel: Lambdaausdruck

```
[ | \x -> if x then 12 else 42 | ]
```



*Exp* Ausdrücke

*Pat* Patterns

*Lit* Literale

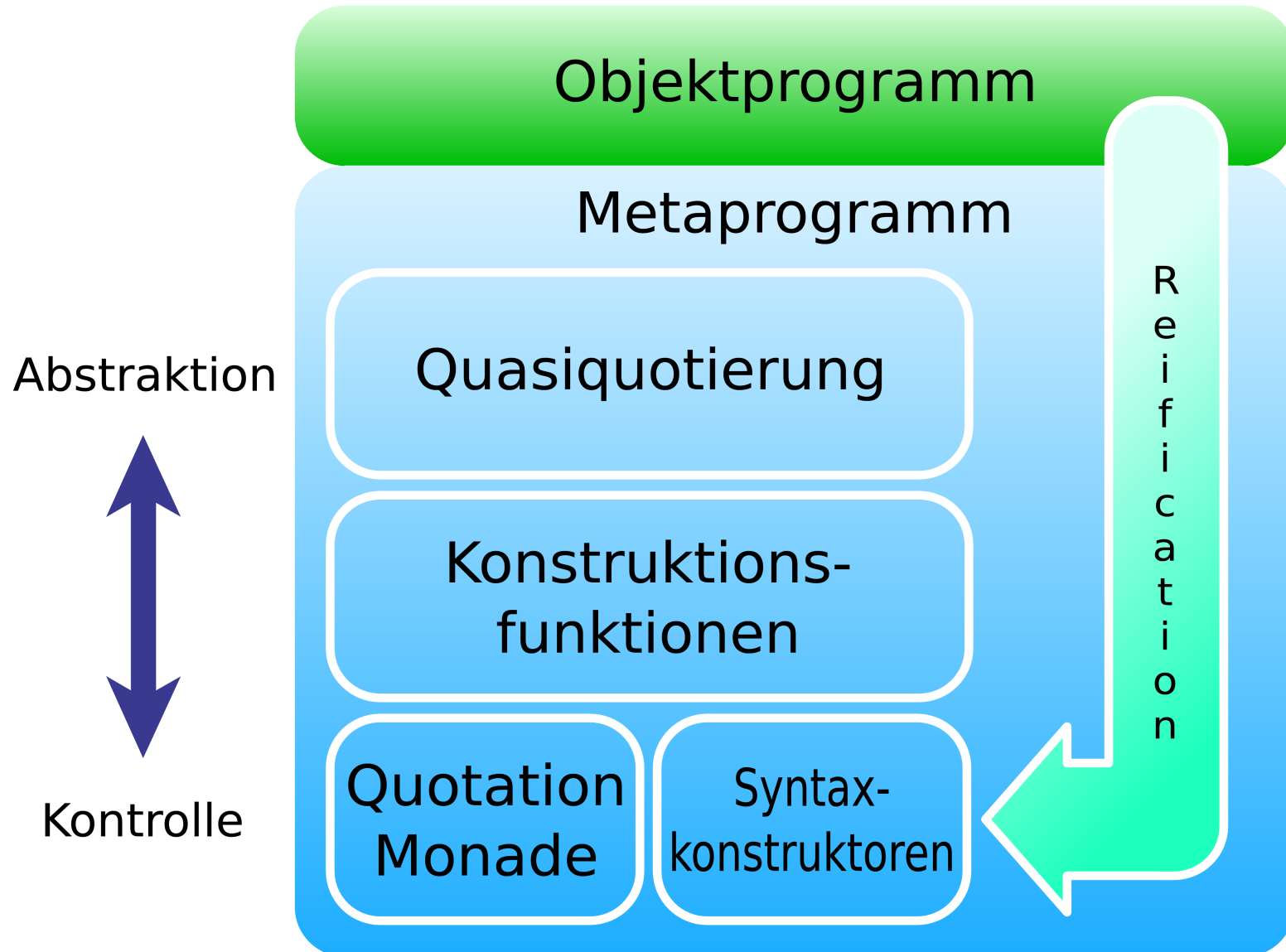
# Quotation Monade

- Problem: Namenskonflikte bei Verwendung von Syntaxkonstruktoren
- Monadische Lösung („Q“-Monade)
  - vergebene Namen intern gespeichert
  - „newName“ erzeugt frische Namen
- Quasiquotierung und Konstruktionsfunktionen benutzen „Q“
- Die Codeerzeugung findet dementsprechend immer in der Monade statt
  - „ExpQ“ ist nur ein Typsynonym für „Q Exp“

# Reification

- „reify“: Zugriff auf die Symboltabelle über Namen von Klassen, Typen, Konstruktoren und Variablen
- Ergebnisse vom Typ „Info“, die ggf. die jeweiligen „Dec“-Objekte enthalten
- Analyse bestehenden Objektcodes
  - Struktur eines Datentyps
  - Zugriff auf Definition einer Funktion als AST

# Im Zusammenhang



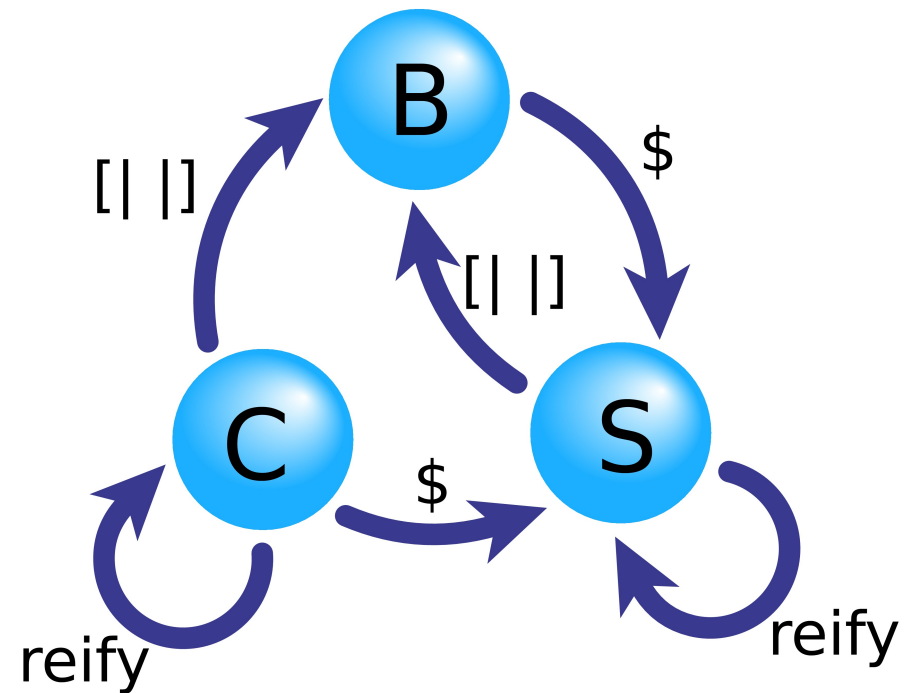
## III. Interna

# Typprüfung

- Typsicherheit soll gewährleistet sein
- Aber: Typen des resultierenden Objektcodes erst nach dem Splice bekannt
- Somit:
  - Typprüfung des Splice
  - Ausführung des Splice
  - Typprüfung des resultierenden Programms

# Zustände bei der Typprüfung

Compiling (C)  
Bracket (B)  
Splicing (S)



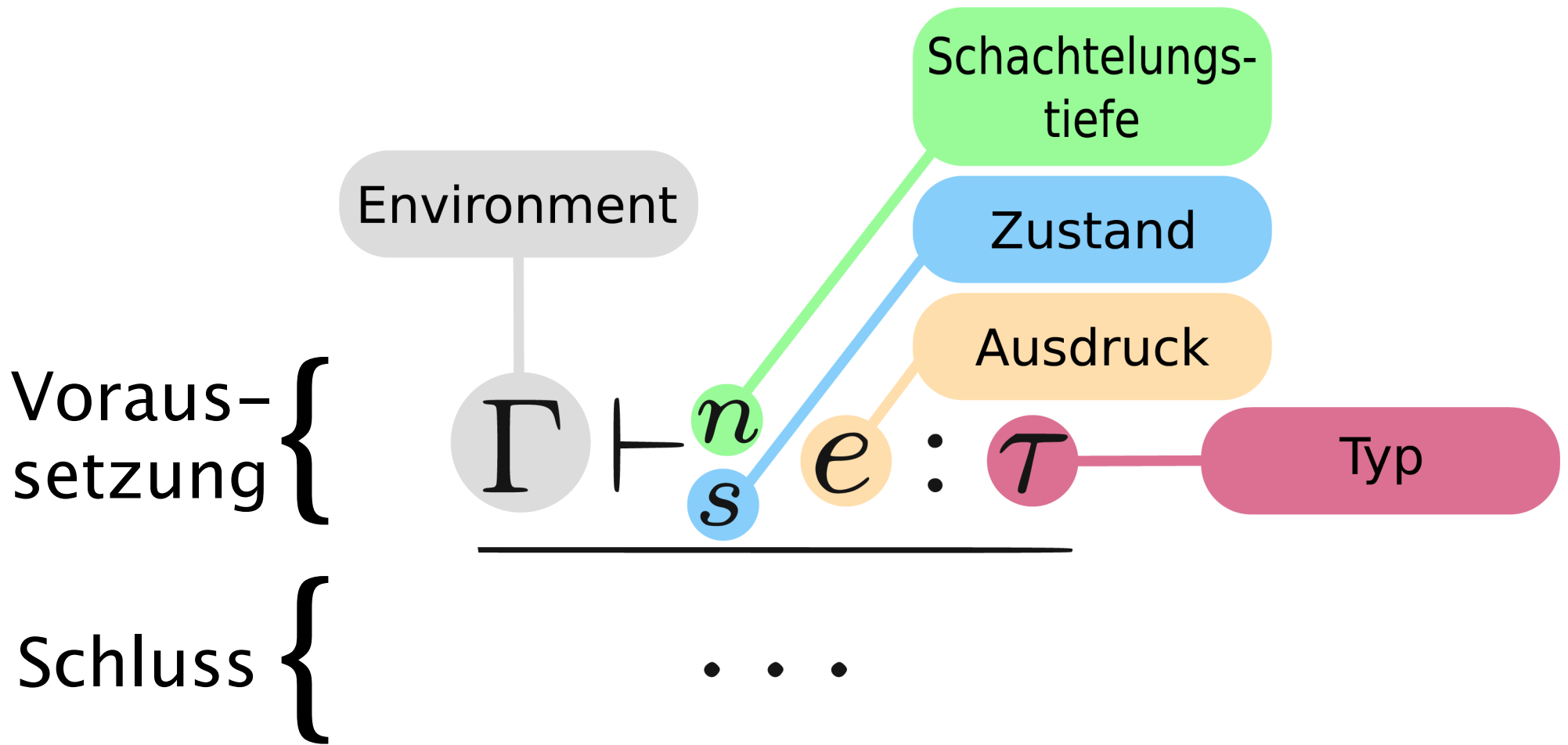
Zählen der Schachtelungstiefe:

- Inkrementierung in B
- Dekrementierung in S

# Inferenzregeln bei der Typprüfung

- Kein Zustandsübergang möglich → Fehler
- Formale Beschreibung der Typprüfung durch Inferenzregeln
- Inferenzregeln beziehen sich auf:
  - Zustand
  - Schachtelungstiefe

# Notation der Inferenzregeln



# Beispiel: „BRACKET“

$$\frac{\Gamma \vdash_B^{n+1} e : \tau}{\Gamma \vdash_{C,S}^n [|e|] : QExp}$$

- Quasiquotierungen sind vom Typ „Q Exp“
- Typ des inneren Ausdrucks verborgen
- Aber: Typing des inneren Ausdrucks nach dem Splice

# Beispiel: „ESCS“

$$\frac{(\Gamma \vdash_C^0 e' : \tau) \wedge (\text{run}Q(e) \mapsto e') \wedge (\Gamma \vdash_S^0 e : QExp)}{\Gamma \vdash_C^0 \$e : \tau}$$

- Argument von „\$“ ist vom Typ  $Q \text{ Exp}$
- Aber: Typ des erzeugten Ausdrucks hängt von der Berechnung („ $\text{run}Q$ “) ab
- Der resultierende Ausdruck  $e'$  wird in  $C$  der normalen Typprüfung unterzogen

# Implementierung von Cross-Stage Persistence

- Zu splicender Code kann sich auf Variablen und Funktionen beziehen, die an der Applikationsstelle nicht im Scope sind
  - nicht importierte top-level Variablen
  - lokale Variablen
- Für top-level Variablen werden intern „original names“ der Form „Modul:Variable“ eingeführt
- Werte lokaler Variablen werden intern mittels der Typklasse „Lift“ in eine Repräsentation über Syntaxkonstruktoren überführt

# Implementierung von „Q“

- Früher: Anreicherung der „IO“-Monade mit einem Environment zur Namensverwaltung
- Aktuell: „Q“ benutzt intern direkt die GHC-eigene Typcheckermonade „TcM“
  - Effizienz
  - Reification ist somit durch Bibliotheksfunktionen implementierbar, ohne wie früher eigene neue Sprachkonstrukte erforderlich zu machen

## IV. Fazit

# Beispielanwendungen

- „Hacanon“
  - Baut auf das Foreign Function Interface auf
  - Erzeugt den FFI-spezifischen low-level codes mit TH
- Web-Interface für „HERA“
  - „HERA“: Haskell Equational Reasoning Assistent
  - Template Haskell wird benutzt für:
    - Eingabe von geparstem und typgecheckten Code
    - Spezifikation der Gleichungen
  - Nur sehr wenige Informationen über das Projekt
- Forschungsprojekt am Lehrstuhl Programmierung zum Schaltkreisdesign mit Template Haskell
  - Größenparametrisierung (Bitzahl = Tupellänge)
  - Parallele und Sequentielle Komposition

# Bewertung

## Pro

- Softwaretechnisch saubere Umsetzung
- Laufzeiteffizienz des erzeugten Codes
- Vollständigkeit
- Für algorithmische Programmkonstruktion in Haskell das Mittel der Wahl

## Contra

- Übersetzung terminiert nicht notwendigerweise
- Verwirrende und/oder veraltete Dokumentation
- Nur für Experten