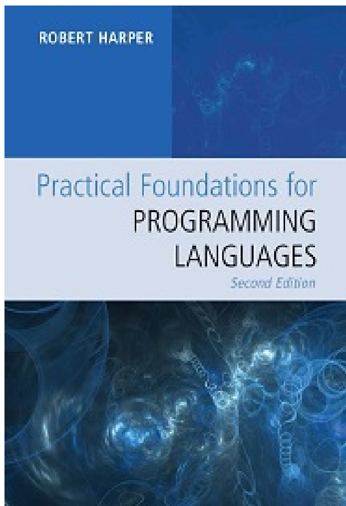
Fortgeschrittene Funktionale Programmierung in Haskell

Jonas Betzendahl
Stefan Dresselhaus

Vorlesung 13: *Meine eigene Programmiersprache*
Stand: 15. Juli 2016

Leseempfehlung (I): PFPL

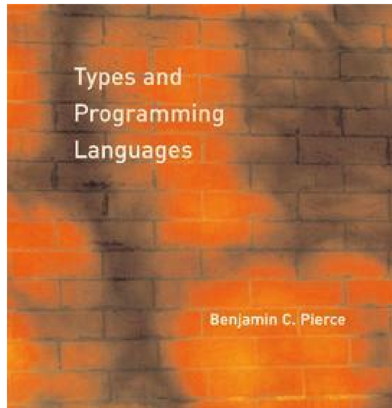


Practical Foundations of Programming Languages
by Robert Harper

```
www.cambridge.org/us/
academic/subjects/
computer-science/
programming-languages\
-and-applied-logic/
practical-foundations-\
programming-languages\
-2nd-edition
```

…oder über die Suchmaschine eurer Wahl!

Leseempfehlung (II): TAPL



Practical Foundations of Programming Languages
by Benjamin C. Pierce

`https://www.cis.upenn.edu/~bcpierce/tapl/`

...oder über die Suchmaschine eurer Wahl!

„Lese“empfehlung (III): OPLSS

Oregon Programming Language Summer School
University of Oregon

`https://www.cs.uoregon.edu/research/summerschool/summer16/`

…oder über die Suchmaschine eurer Wahl!

Programmiersprachentheorie 100.5

Was ist eine Programmiersprache?

Programmiersprachen sind formale (künstliche) Sprachen.
Besonders relevant sind die Aspekte der *Syntax* und der *Semantik*.

Was ist eine Programmiersprache?

Programmiersprachen sind formale (künstliche) Sprachen.
Besonders relevant sind die Aspekte der *Syntax* und der *Semantik*.

Die **Syntax** einer Sprache beschäftigt sich mit der Frage, welche Ausdrücke in einer Sprache als „gültig“ oder „zulässig“ gewertet werden. Die **Semantik** legt fest, was ein Ausdruck „bedeutet“, bzw. wozu er ausgewertet wird.

Was ist eine Programmiersprache?

Programmiersprachen sind formale (künstliche) Sprachen.
Besonders relevant sind die Aspekte der *Syntax* und der *Semantik*.

Die **Syntax** einer Sprache beschäftigt sich mit der Frage, welche Ausdrücke in einer Sprache als „gültig“ oder „zulässig“ gewertet werden. Die **Semantik** legt fest, was ein Ausdruck „bedeutet“, bzw. wozu er ausgewertet wird.

In der Regel ist der semantische Teil etwas trickreicher und/oder interessanter. Präzision ist aber in beiden Fällen wichtig, damit wir später genaue (mathematische) Aussagen treffen können.

Syntax by example (I)

Ein Weg, die Syntax einer Sprache festzulegen, ist eine formale Grammatik (vgl. Theoretische Informatik).

Hier ein Beispiel einer sehr simplen Sprache (ohne Typen):

```
t ::= true                    -- Truth constant
      false                   -- Falsity constant
      if t then t else t      -- Conditional
      0                       -- Zero constant
      succ t                  -- Successor
      pred t                  -- Predecessor
      isZero t                -- Zero test
```

Syntax by example (I)

Ein Weg, die Syntax einer Sprache festzulegen, ist eine formale Grammatik (vgl. Theoretische Informatik).

Hier ein Beispiel einer sehr simplen Sprache (ohne Typen):

```
t ::= true                  -- Truth constant
      false                 -- Falsity constant
      if t then t else t    -- Conditional
      0                     -- Zero constant
      succ t                -- Successor
      pred t                -- Predecessor
      isZero t              -- Zero test
```

Einige dubiose Programme (z.B. „`if 0 then true else 0`“) werden hier explizit *nicht* ausgeschlossen. Um solche „Fehler“ auszuschließen werden später Typen interessant.

Syntax by example (II)

Wir können die gleiche Aussage auch mit anderen Mitteln treffen. Zum Beispiel auch induktiv:

Die Menge der *Terme* unserer Sprache ist definiert als die kleinste Menge $\mathcal{T}$, sodass gilt:

- $\{\texttt{true}, \texttt{false}, \texttt{0}\} \subseteq \mathcal{T}$
- Falls $t_1 \in \mathcal{T}$, dann $\{\texttt{succ}\ t_1, \texttt{pred}\ t_1, \texttt{isZero}\ t_1\} \subseteq \mathcal{T}$
- Falls $t_1, t_2, t_3 \in \mathcal{T}$, dann $\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \in \mathcal{T}$

Syntax by example (II)

Wir können die gleiche Aussage auch mit anderen Mitteln treffen. Zum Beispiel auch induktiv:

Die Menge der *Terme* unserer Sprache ist definiert als die kleinste Menge $\mathcal{T}$, sodass gilt:

- $\{\texttt{true}, \texttt{false}, \texttt{0}\} \subseteq \mathcal{T}$
- Falls $t_1 \in \mathcal{T}$, dann $\{\texttt{succ }t_1, \texttt{pred }t_1, \texttt{isZero }t_1\} \subseteq \mathcal{T}$
- Falls $t_1, t_2, t_3 \in \mathcal{T}$, dann $\texttt{if }t_1\texttt{ then }t_2\texttt{ else }t_3 \in \mathcal{T}$

Quiz: Warum heben wir hervor, dass $\mathcal{T}$ die *kleinste* Menge sein soll, sodass diese Aussgen gelten?

Syntax by example (III)

Eine dritte Möglichkeit ist die Festlegung über *Inferenzregeln*, die uns später noch häufiger begenen werden. Einträge über der Linie werden hier *Prämissen* (engl.: premises) und Einträge unter der Linie *Schlussfolgerungen* (engl.: conclusions) genannt.

$$\frac{}{\texttt{true} \in \mathcal{T}} \qquad \frac{}{\texttt{false} \in \mathcal{T}} \qquad \frac{}{\texttt{0} \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T}}{\texttt{succ}\ \ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{pred}\ \ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{isZero}\ \ t_1 \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T} \qquad t_2 \in \mathcal{T} \qquad t_3 \in \mathcal{T}}{\texttt{if}\ \ t_1\ \ \texttt{then}\ \ t_2\ \ \texttt{else}\ \ t_3 \in \mathcal{T}}$$

Syntax by example (III)

Eine dritte Möglichkeit ist die Festlegung über *Inferenzregeln*, die uns später noch häufiger begenen werden. Einträge über der Linie werden hier *Prämissen* (engl.: premises) und Einträge unter der Linie *Schlussfolgerungen* (engl.: conclusions) genannt.

$$\frac{}{\texttt{true} \in \mathcal{T}} \qquad \frac{}{\texttt{false} \in \mathcal{T}} \qquad \frac{}{\texttt{0} \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T}}{\texttt{succ}\ \ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{pred}\ \ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{isZero}\ \ t_1 \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T} \qquad t_2 \in \mathcal{T} \qquad t_3 \in \mathcal{T}}{\texttt{if}\ \ t_1\ \ \texttt{then}\ \ t_2\ \ \texttt{else}\ \ t_3 \in \mathcal{T}}$$

Quiz: Warum müssen wir hier nicht gegen das Szenario von zusätzlichen, ungewollten Elementen absichern?

How do you say "semantics"?

Es gibt drei prominente Arten, anzugeben, was ein gültiger Ausdruck in einer Sprache bedeuten soll:

How do you say "semantics"?

Es gibt drei prominente Arten, anzugeben, was ein gültiger Ausdruck in einer Sprache bedeuten soll:

- **Operational Semantics:** Hier wird eine simple abstrakte Maschine zur Auswertung (big step / small step) der Ausdrücke definiert. Die Bedeutung eines Ausdruckes ist der finale Zustand der Maschine.

How do you say "semantics"?

Es gibt drei prominente Arten, anzugeben, was ein gültiger Ausdruck in einer Sprache bedeuten soll:

- **Operational Semantics:** Hier wird eine simple abstrakte Maschine zur Auswertung (big step / small step) der Ausdrücke definiert. Die Bedeutung eines Ausdruckes ist der finale Zustand der Maschine.
- **Denotational Semantics:** Hier wird jedem Ausdruck der Sprache ein mathematisches Objekt (z.B. eine Zahl oder Funktion) als Bedeutung zugeordnet. Vorteil: erlaubt abstrakteres Schließen als OpSem.

How do you say "semantics"?

Es gibt drei prominente Arten, anzugeben, was ein gültiger Ausdruck in einer Sprache bedeuten soll:

- **Operational Semantics:** Hier wird eine simple abstrakte Maschine zur Auswertung (big step / small step) der Ausdrücke definiert. Die Bedeutung eines Ausdruckes ist der finale Zustand der Maschine.
- **Denotational Semantics:** Hier wird jedem Ausdruck der Sprache ein mathematisches Objekt (z.B. eine Zahl oder Funktion) als Bedeutung zugeordnet. Vorteil: erlaubt abstrakteres Schließen als OpSem.
- **Axiomatic Semantics:** Hier werden statt Verhalten direkt Gesetze definiert, die für Terme gelten. Die Bedeutung eines Ausdrucks ist, was über ihn bewiesen werden kann.

Statics and Dynamics

Ein weiter Blickwinkel auf Programmiersprachen ist die Unterscheidung zwischen *statics* und *dynamics*. Eine Sprache heißt genau dann „sicher“, wenn statisch wohlgeformte Programme auch dynamisch „well-behaved“ sind.

Statics and Dynamics

Ein weiter Blickwinkel auf Programmiersprachen ist die Unterscheidung zwischen *statics* und *dynamics*. Eine Sprache heißt genau dann „sicher", wenn statisch wohlgeformte Programme auch dynamisch „well-behaved" sind.

Die **statische** Phase besteht aus Parsing und Typechecking und setzt sich aus Regeln zusammen, mit denen eins feststellen kann, ob ein Ausdruck für einen bestimmten Typen wohlgeformt ist.

Die **dynamische** Phase beschreibt, wie Programme ausgeführt werden. Unter anderem wird hier darüber geredet, welche Ausdrücke der Sprache auch *Werte* (engl: values) (vollständig ausgewertete Ausdrücke) sind ($e\ \texttt{val}$) und ob und wie Ausrücke, die keine Werte sind, in andere überführt werden ($e \mapsto e'$).

Typsicherheit

Type safety in a nutshell

Typsicherheit wird oft (zu Recht) als eine extrem wichtige Eigenschaft von Programmiersprachen hervorgehoben.
Hier machen wir die Bedeutung präzise:

Type Safety means *progress* + *preservation*!

Type safety in a nutshell

Typsicherheit wird oft (zu Recht) als eine extrem wichtige Eigenschaft von Programmiersprachen hervorgehoben.
Hier machen wir die Bedeutung präzise:

Type Safety means *progress* + *preservation*!

Progress (Fortschritt):
Wenn $e : \tau$, dann gilt entweder e `val` oder $\exists\, e' .\, e \mapsto e'$.
Ausdrücke sind entweder Werte oder können weiter ausgewertet werden. Berechnungen „bleiben nicht hängen“.

Type safety in a nutshell

Typsicherheit wird oft (zu Recht) als eine extrem wichtige Eigenschaft von Programmiersprachen hervorgehoben.
Hier machen wir die Bedeutung präzise:

Type Safety means *progress* + *preservation*!

Progress (Fortschritt):
Wenn $e : \tau$, dann gilt entweder $e \; \texttt{val}$ oder $\exists\, e' \,.\, e \mapsto e'$.
Ausdrücke sind entweder Werte oder können weiter ausgewertet werden. Berechnungen „bleiben nicht hängen“.

Preservation (Erhaltung):
Wenn $e : \tau$ und $e \mapsto e'$, dann gilt $e' : \tau$.
Einen Ausdruck auszuwerten ändert nicht seinen Typen.

Beweise für Typsicherheit

Beweise, das eine Sprache typsicher nach dieser Definition ist, werden oft per Induktion über die Struktur von Ausdrücken geführt. Dies ist möglich, da wir präzise Definitionen darüber gegeben haben, was einen gültigen Ausdruck darstellt und was nicht bzw. wie diese ausgewertet werden.

Beispiel:

$$\frac{e_1 \;:\; \texttt{num} \qquad e_2 \;:\; \texttt{num}}{plus(e_1; e_2) \;:\; \texttt{num}}$$

Beweise für Typsicherheit

Beweise, das eine Sprache typsicher nach dieser Definition ist, werden oft per Induktion über die Struktur von Ausdrücken geführt. Dies ist möglich, da wir präzise Definitionen darüber gegeben haben, was einen gültigen Ausdruck darstellt und was nicht bzw. wie diese ausgewertet werden.

Beispiel:

$$\frac{e_1 \ : \ \texttt{num} \qquad e_2 \ : \ \texttt{num}}{\mathit{plus}(e_1; e_2) \ : \ \texttt{num}}$$

Für Beispiele und Näheres (z.B. wie mit `div`-Operatoren umgegangen werden kann) werden Kapitel 6 (PFPL) und Kapitel 8 (TAPL) wärmstens empfohlen.

System **T**

Gödel's System **T**

Ein bekannter Formalismus aus der theoretischen Informatik, genannt „System **T**" oder „Gödel's **T**", soll uns als Beispiel für den Rest der Vorlesung dienen.
In **T**, dem System der natürlichen Zahlen und Funktionstypen, werden nicht Primitive für Addition, Subtraktion etc. eingeführt, sondern werden über *primitive Rekursion* definiert werden.

Gödel's System **T**

Ein bekannter Formalismus aus der theoretischen Informatik, genannt „System **T**" oder „Gödel's **T**", soll uns als Beispiel für den Rest der Vorlesung dienen.
In **T**, dem System der natürlichen Zahlen und Funktionstypen, werden nicht Primitive für Addition, Subtraktion etc. eingeführt, sondern werden über *primitive Rekursion* definiert werden.

Alle Programme, die wir in **T** definieren können, sind immer *total*. Das bedeutet, dass sie auf jeden Fall terminieren. Eine (wohltypisierte) Endlosschleife zu schreiben ist in **T** also z.B. unmöglich.

Die Grammatik von **T**

Hier ist die formale Grammatik, also die Syntax für **T**:

$\texttt{Typ}\ \tau$	$::=$	$\texttt{Nat}$	Naturals
		$\texttt{arr}(\tau_1; \tau_2)$	Functions
$\texttt{Exp}\ e$	$::=$	x	Variable
		Z	Zero
		$S(e)$	Successor
		$\texttt{rec}\{e_0; x.y.e_1\}(e)$	Recursion
		$\texttt{lam}\{\tau\}(x.e)$	Abstraction
		$\texttt{ap}(e_1; e_2)$	Application

Die Grammatik von **T**

Hier ist die formale Grammatik, also die Syntax für **T**:

Typ τ	::=	Nat	Naturals
		$\texttt{arr}(\tau_1; \tau_2)$	Functions
Exp e	::=	x	Variable
		Z	Zero
		$S(e)$	Successor
		$\texttt{rec}\{e_0; x.y.e_1\}(e)$	Recursion
		$\texttt{lam}\{\tau\}(x.e)$	Abstraction
		$\texttt{ap}(e_1; e_2)$	Application

Im Folgenden werden wir auch eine dem Auge etwas angenehmere Notation verwenden (z.B. $\lambda(x : \tau).e$ oder $foo : \mathbb{N} \to \mathbb{N}$).

Rekursion in **T**

System **T** hat ein Konstrukt für Rekursion, in unseren zwei Notationsarten so dargestellt:

$$\texttt{rec}\{e_0; x.y.e_1\}(e)$$

$$\texttt{rec}\ e\{Z \rightsquigarrow e_0 \mid S(x)\ \texttt{with}\ y \rightsquigarrow e_1\}$$

Dieses Konstrukt wird auch der *recursor* genannt. Es stellt präzise die e-fache Anwendung von $x.y.e_1$ dar, angefangen bei e_0. Die gebundenen Variable x steht hier für den Vorgänger und y für das Ergebnis der bereits x-fach angewendeten Iteration. Beide diese Terme können in der Berechnung vorkommen.

Rekursion in **T**

System **T** hat ein Konstrukt für Rekursion, in unseren zwei Notationsarten so dargestellt:

$$\texttt{rec}\{e_0; x.y.e_1\}(e)$$

$$\texttt{rec}\ e\{Z \leadsto e_0 \mid S(x)\ \texttt{with}\ y \leadsto e_1\}$$

Dieses Konstrukt wird auch der *recursor* genannt. Es stellt präzise die e-fache Anwendung von $x.y.e_1$ dar, angefangen bei e_0. Die gebundenen Variable x steht hier für den Vorgänger und y für das Ergebnis der bereits x-fach angewendeten Iteration. Beide diese Terme können in der Berechnung vorkommen.

Quiz: Wenn wir Rekursion haben, ist **T** dann trotzdem total?

Ein paar „Programme“ in System **T**

Verdoppelung einer natürlichen Zahl:

$\mathit{double} : (\mathbb{N} \to \mathbb{N})$
$\mathit{double} = \lambda(a : \mathbb{N}).\texttt{rec}\ a\{Z \leadsto Z \mid S(b)\ \texttt{with}\ c \leadsto S(S(c))\}$

Ein paar „Programme" in System **T**

Verdoppelung einer natürlichen Zahl:

$double : (\mathbb{N} \to \mathbb{N})$
$double = \lambda(a : \mathbb{N}).\mathtt{rec}\ a\{Z \rightsquigarrow Z \mid S(b)\ \mathtt{with}\ c \rightsquigarrow S(S(c))\}$

Addition via Rekursion über das zweite Argument:

$add : (\mathbb{N} \to (\mathbb{N} \to \mathbb{N}))$
$add = \lambda(a : \mathbb{N}).\lambda(b : \mathbb{N}).\mathtt{rec}\ b\{Z \rightsquigarrow a \mid S(c)\ \mathtt{with}\ d \rightsquigarrow S(d)\}$

Ein paar „Programme“ in System **T**

Verdoppelung einer natürlichen Zahl:

$$\mathit{double} : (\mathbb{N} \to \mathbb{N})$$
$$\mathit{double} = \lambda(a : \mathbb{N}).\mathtt{rec}\ a\{Z \leadsto Z \mid S(b)\ \mathtt{with}\ c \leadsto S(S(c))\}$$

Addition via Rekursion über das zweite Argument:

$$\mathit{add} : (\mathbb{N} \to (\mathbb{N} \to \mathbb{N}))$$
$$\mathit{add} = \lambda(a : \mathbb{N}).\lambda(b : \mathbb{N}).\mathtt{rec}\ b\{Z \leadsto a \mid S(c)\ \mathtt{with}\ d \leadsto S(d)\}$$

Funktionsverkettung für $\mathbb{N} \to \mathbb{N}$, nicht polymorph:

$$\circ : ((\mathbb{N} \to \mathbb{N}) \to ((\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})))$$
$$\circ = \lambda(f : (\mathbb{N} \to \mathbb{N})).\lambda(g : (\mathbb{N} \to \mathbb{N})).\lambda(x : \mathbb{N}).f(g(x))$$

Code: Types as Expressions

Der erste Schritt von der Tafel zum Prozessor ist in unserem Fall, Typen in `Haskell` anzugeben, die zu Typen und Ausdrücken in System **T** korrespondieren. Das ist glücklicherweise einfach.

```
-- Haskell type of Syste T types
data Typ =  Nat                         -- Natural numbers
         | Arrow Typ Typ                -- Functions
         deriving Eq

-- Haskell type of System T expressions
data Exp = Z                            -- Zero
         | Succ Exp                     -- Successor
         | Var String                   -- Variables
         | Lambda Typ Exp Exp           -- Lambda
         | Rec Exp Exp Exp Exp Exp      -- Recursor
         | Ap  Exp Exp                  -- Application
         deriving Eq
```

Typing rules for System **T**

Die Regeln für Typzuweisung in System **T** lauten wie folgt:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

Typing rules for System **T**

Die Regeln für Typzuweisung in System **T** lauten wie folgt:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash Z : \mathit{Nat}} \qquad \frac{\Gamma \vdash e : \mathit{Nat}}{\Gamma \vdash S(e) : \mathit{Nat}}$$

Typing rules for System **T**

Die Regeln für Typzuweisung in System **T** lauten wie folgt:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash Z : Nat} \quad \frac{\Gamma \vdash e : Nat}{\Gamma \vdash S(e) : Nat}$$

$$\frac{\Gamma x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}\{\tau_1\}(x.e) : \texttt{arr}(\tau_1, \tau_2)} \quad \frac{\Gamma \vdash e_1 : \texttt{arr}(\tau_1, \tau_2) \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \texttt{ap}(e1, e2) : \tau_2}$$

Typing rules for System **T**

Die Regeln für Typzuweisung in System **T** lauten wie folgt:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash Z : \mathit{Nat}} \qquad \frac{\Gamma \vdash e : \mathit{Nat}}{\Gamma \vdash S(e) : \mathit{Nat}}$$

$$\frac{\Gamma x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{lam}\{\tau_1\}(x.e) : \texttt{arr}(\tau_1, \tau_2)} \qquad \frac{\Gamma \vdash e_1 : \texttt{arr}(\tau_1, \tau_2) \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \texttt{ap}(e1, e2) : \tau_2}$$

$$\frac{\Gamma \vdash e : \mathit{Nat} \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : \mathit{Nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \texttt{rec}\{e0, x.y.e1\} : \tau}$$

Code: Typechecking

Ein Typchecker für unsere Sprache ist eine Funktion, die für einen Ausdruck (mit gewissen Kontexten) einen korrekten Typen findet (falls möglich). Dieser Vorgang heißt *Typinferenz*.

```
typecheck :: Gamma -> Context -> Exp -> Either String Typ
typecheck _ _ Z           = Right Nat
typecheck g c (Succ e)    = case typecheck g c e of
  (Left err)  -> Left err
  (Right Nat) -> Right Nat
  (Right tau) -> Left $ "Type error for expr. " ++ show e ++
                      "\n Expected: Nat, Found: " ++ show tau
typecheck g c w@(Var v) = case find (\(trm, _) -> trm == w) g of
  (Nothing)      -> Left $ "Could not find a type for " ++ show w
  (Just (x,tau)) -> Right tau
typecheck g c (Lambda tau x e) = typecheck ((x, tau):g) c e
                                 >>= \sigma -> pure (Arrow tau sigma)
...
```

System **T**: Dynamics (I)

Die geschlossenen Werte von **T** sind wie folgt definiert:

$$\frac{}{Z \ \texttt{val}}$$

$$\frac{[e \ \texttt{val}]}{S(e) \ \texttt{val}}$$

$$\frac{}{\lambda\{\tau\}(x.e) \ \texttt{val}}$$

System $\mathbf{T}$: Dynamics (I)

Die geschlossenen Werte von **T** sind wie folgt definiert:

$$\frac{}{Z \ \mathtt{val}}$$

$$\frac{[e \ \mathtt{val}]}{S(e) \ \mathtt{val}}$$

$$\frac{}{\lambda\{\tau\}(x.e) \ \mathtt{val}}$$

Geklammerte Ausdrücke können für eine *eager evaluation* hinzu genommen oder für *lazy evaluation* weggelassen werden.

Code: value function

Eine Funktion, die uns für einen gegebenen Ausdruck zurück gibt, ob er vollständig ausgewertet wurde, ist schnell geschrieben:

```
val :: Exp -> Bool
val Z                = True
val (Succ e)         = val e  -- bzw. konstant "True"
val (Lambda _ _ _) = True
val _                = False
```

System **T**: Dynamics (II)

Die sieben Übergangsregeln für **T** sind wie folgt:

$$\left[\frac{e \mapsto e'}{S(e) \mapsto S(e')}\right]$$

System **T**: Dynamics (II)

Die sieben Übergangsregeln für **T** sind wie folgt:

$$\left[\frac{e \mapsto e'}{S(e) \mapsto S(e')}\right]$$

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e1; e2) \mapsto \mathtt{ap}(e_1'; e2)}$$

System **T**: Dynamics (II)

Die sieben Übergangsregeln für **T** sind wie folgt:

$$\left[\frac{e \mapsto e'}{S(e) \mapsto S(e')}\right]$$

$$\left[\frac{e_1 \ \texttt{val} \qquad e_2 \mapsto e_2'}{\texttt{ap}(e_1; e_2) \mapsto \texttt{ap}(e_1; e_2')}\right]$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e1; e2) \mapsto \texttt{ap}(e_1'; e2)}$$

System **T**: Dynamics (II)

Die sieben Übergangsregeln für **T** sind wie folgt:

$$\left[\frac{e \mapsto e'}{S(e) \mapsto S(e')}\right]$$

$$\left[\frac{e_1 \ \texttt{val} \qquad e_2 \mapsto e_2'}{\texttt{ap}(e_1; e_2) \mapsto \texttt{ap}(e_1; e_2')}\right]$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e1; e2) \mapsto \texttt{ap}(e_1'; e2)}$$

$$\frac{[e_2 \ \texttt{val}]}{\texttt{ap}(\texttt{lam}\{\tau\}(x.e); e_2) \mapsto [e_2/x] \ e}$$

System $\mathbf{T}$: Dynamics (II)

Die sieben Übergangsregeln für **T** sind wie folgt:

$$\left[\frac{e \mapsto e'}{S(e) \mapsto S(e')}\right]$$

$$\left[\frac{e_1 \ \texttt{val} \qquad e_2 \mapsto e_2'}{\texttt{ap}(e_1; e_2) \mapsto \texttt{ap}(e_1; e_2')}\right]$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e1; e2) \mapsto \texttt{ap}(e_1'; e2)}$$

$$\frac{[e_2 \ \texttt{val}]}{\texttt{ap}(\texttt{lam}\{\tau\}(x.e); e_2) \mapsto [e_2/x] \ e}$$

Auch hier können geklammerte Ausdrücke für eine *eager evaluation* hinzu genommen oder für *lazy evaluation* weggelassen werden.

System **T**: Dynamics (III)

Es bleiben drei Übergangsregeln, die immer gelten, egal ob *lazy* oder *eager* evaluiert wird:

$$\frac{e \mapsto e'}{\texttt{rec}\{e0; x.y.e1\}(e) \mapsto \texttt{rec}\{e0; x.y.e1\}(e')}$$

System **T**: Dynamics (III)

Es bleiben drei Übergangsregeln, die immer gelten, egal ob *lazy* oder *eager* evaluiert wird:

$$\frac{e \mapsto e'}{\texttt{rec}\{e0; x.y.e1\}(e) \mapsto \texttt{rec}\{e0; x.y.e1\}(e')}$$

$$\frac{}{\texttt{rec}\{e_0; x.y.e_1\}(Z) \mapsto e_0}$$

System **T**: Dynamics (III)

Es bleiben drei Übergangsregeln, die immer gelten, egal ob *lazy* oder *eager* evaluiert wird:

$$\frac{e \mapsto e'}{\texttt{rec}\{e0; x.y.e1\}(e) \mapsto \texttt{rec}\{e0; x.y.e1\}(e')}$$

$$\frac{}{\texttt{rec}\{e_0; x.y.e_1\}(Z) \mapsto e_0}$$

$$\frac{S(e)\ \texttt{val}}{\texttt{rec}\{e_0; x.y.e_1\}(S(e)) \mapsto [e, \texttt{rec}\{e_0; x.y.e_1\}(e)/x, y]\ e_1}$$

Code: Evaluator

Hier sind die Regel für *eager* evaluation in Code gegossen:

```
eval :: Exp -> Exp
eval (Succ e)           = Succ (eval e)
eval ap@(Ap e1 e2)
  | not (val e1)        = eval $ (Ap (eval e1) e2)
  | not (val e2)        = eval $ (Ap e1 (eval e2))
  | otherwise           = case e1 of
    (Lambda tau x e) -> eval $ replace e2 x e
    _                -> ap
eval (Rec e0 x y e1 q)
  | not (val q)         = eval (Rec e0 x y e1 (eval q))
  | otherwise           = case q of
    (Z)      -> eval e0
    (Succ e) -> eval $ replace (Rec e0 x y e1 e) y (replace e x e1)
eval e                  = e
```

Was im Code noch fehlt…

Wir haben hier die wichtigsten Teile einer Implementation von System **T** in `Haskell` besprochen, es fehlt nur noch etwas „Kleber“ zwischen den einzelnen Abschnitten:

Was im Code noch fehlt…

Wir haben hier die wichtigsten Teile einer Implementation von System **T** in `Haskell` besprochen, es fehlt nur noch etwas „Kleber“ zwischen den einzelnen Abschnitten:

- Eine REPL (Read - Evaluate - Print - Loop)

Was im Code noch fehlt...

Wir haben hier die wichtigsten Teile einer Implementation von System **T** in `Haskell` besprochen, es fehlt nur noch etwas „Kleber“ zwischen den einzelnen Abschnitten:

- Eine REPL (Read - Evaluate - Print - Loop)
- Ein Parser für Ausdrücke, Eingaben, Dateien...

Was im Code noch fehlt...

Wir haben hier die wichtigsten Teile einer Implementation von System **T** in `Haskell` besprochen, es fehlt nur noch etwas „Kleber“ zwischen den einzelnen Abschnitten:

- Eine REPL (Read - Evaluate - Print - Loop)
- Ein Parser für Ausdrücke, Eingaben, Dateien...
- Eine Funktion, die zwei Ausdrücke disjunkt halten kann, um *variable capture* zu vermeiden

Was im Code noch fehlt…

Wir haben hier die wichtigsten Teile einer Implementation von System **T** in `Haskell` besprochen, es fehlt nur noch etwas „Kleber“ zwischen den einzelnen Abschnitten:

- Eine REPL (Read - Evaluate - Print - Loop)
- Ein Parser für Ausdrücke, Eingaben, Dateien…
- Eine Funktion, die zwei Ausdrücke disjunkt halten kann, um *variable capture* zu vermeiden
- Eine präzise Implementation von `replace`

Was im Code noch fehlt…

Wir haben hier die wichtigsten Teile einer Implementation von System **T** in `Haskell` besprochen, es fehlt nur noch etwas „Kleber" zwischen den einzelnen Abschnitten:

- Eine REPL (Read - Evaluate - Print - Loop)
- Ein Parser für Ausdrücke, Eingaben, Dateien…
- Eine Funktion, die zwei Ausdrücke disjunkt halten kann, um *variable capture* zu vermeiden
- Eine präzise Implementation von `replace`
- …

Eine (fast) vollständige Implementation (inkl. Summentypen, Produkttypen, nativen Booleans...) findet ihr unter folgender URL:

`https://github.com/jbetzend/goedelT`

Theorie: Undefinierbarkeit (I)

Ein interessantes Resultat über **T** ist, dass gezeigt werden kann, dass Funktionen über die natürlichen Zahlen gibt, die in **T** nicht definierbar sind. Um dies zu zeigen, nutzen wir die Vorgehensweise der *Diagonalisierung* und das Verfahren der *Gödelisierung*.

Theorie: Undefinierbarkeit (I)

Ein interessantes Resultat über **T** ist, dass gezeigt werden kann, dass Funktionen über die natürlichen Zahlen gibt, die in **T** nicht definierbar sind. Um dies zu zeigen, nutzen wir die Vorgehensweise der *Diagonalisierung* und das Verfahren der *Gödelisierung*.

Gödelisierung ist der Prozess, jedem (geschlossenen) Ausdruck in **T** eine *eindeutige* natürliche Zahl zuzuweisen. Eine Zahl kann so eine Aussage repräsentieren. Dieser Gedanke ist uns heute nicht fremd, war zu Gödels Zeiten allerdings revolutionär!

Theorie: Undefinierbarkeit (II)

Ein AST (Abstract Syntax Tree) a hat die Form $\sigma(a_1, \ldots, a_k)$, wobei σ ein Operator der Arität k ist. Wir können alle Operatoren der Sprache so durchnummerieren, dass jeder Operator einen Index $i \in \mathbb{N}$ hat. Sei nun m der Index von σ.

Dann ist die *Gödelzahl* (von a):

$$\ulcorner a \urcorner = 2^m \cdot 3^{n_1} \cdot 5^{n_2} \cdot \cdots \cdot p_k^{n_k}$$

Hier ist p_k die k-te Primzahl ($p_0 = 2, p_1 = 3, \ldots$) und $n_1, n_2, \ldots$ sind die wiederum die Gödelzahlen von $a_1, a_2, \ldots$. Dieser Vorgang bildet jeden AST auf eine natürliche Zahl ab. Außerdem können wir dank dem Fundamentalsatz der Arithmetik jede Zahl n als einzigartigen AST „parsen“.

Theorie: Undefinierbarkeit (III)

Nun können wir eine (mathematische, wohldefinierte) Funktion $f_{univ} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ definieren, sodass $\forall\, e : \texttt{Nat} \to \texttt{Nat}$ gilt, dass $f_{univ}(\ulcorner e \urcorner)(m) = n$ iff $e(\overline{m}) \equiv \overline{n} : \texttt{Nat}$.

Theorie: Undefinierbarkeit (III)

Nun können wir eine (mathematische, wohldefinierte) Funktion $f_{univ} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ definieren, sodass $\forall\, e : \mathtt{Nat} \to \mathtt{Nat}$ gilt, dass $f_{univ}(\ulcorner e \urcorner)(m) = n$ iff $e(\overline{m}) \equiv \overline{n} : \mathtt{Nat}$.

Diese Funktion heißt die *universelle* Funktion von **T** weil sie das Verhalten aller $e : \mathtt{Nat} \to \mathtt{Nat}$ spezifiziert.

Theorie: Undefinierbarkeit (III)

Nun können wir eine (mathematische, wohldefinierte) Funktion $f_{univ} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ definieren, sodass $\forall\, e : \texttt{Nat} \to \texttt{Nat}$ gilt, dass $f_{univ}(\ulcorner e \urcorner)(m) = n$ iff $e(\overline{m}) \equiv \overline{n} : \texttt{Nat}$.

Diese Funktion heißt die *universelle* Funktion von **T** weil sie das Verhalten aller $e : \texttt{Nat} \to \texttt{Nat}$ spezifiziert.

Mit der universellen Funktion können wir eine weitere Funktion δ definieren, genannt die *Diagonalfunktion*.

$$\delta : \mathbb{N} \to \mathbb{N}$$

$$\delta(m) = f_{univ}(m)(m)$$

Theorie: Undefinierbarkeit (III)

Nun können wir eine (mathematische, wohldefinierte) Funktion $f_{univ} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ definieren, sodass $\forall\, e : \texttt{Nat} \to \texttt{Nat}$ gilt, dass $f_{univ}(\ulcorner e \urcorner)(m) = n$ iff $e(\overline{m}) \equiv \overline{n} : \texttt{Nat}$.

Diese Funktion heißt die *universelle* Funktion von **T** weil sie das Verhalten aller $e : \texttt{Nat} \to \texttt{Nat}$ spezifiziert.

Mit der universellen Funktion können wir eine weitere Funktion δ definieren, genannt die *Diagonalfunktion*.

$$\delta : \mathbb{N} \to \mathbb{N}$$

$$\delta(m) = f_{univ}(m)(m)$$

δ ist also so gewählt, dass $\delta(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \equiv \overline{n} : \texttt{Nat}$.

Theorie: Undefinierbarkeit (IV)

Wir zeigen nun, dass f_{univ} nicht in **T** definierbar ist. Angenommen sie wäre definierbar durch den Ausdruck e_{univ}, dann wäre δ ebenfalls definierbar durch den Ausdruck

$$e_\delta = \lambda(m : \texttt{Nat}).e_{univ}(m)(m)$$

Theorie: Undefinierbarkeit (IV)

Wir zeigen nun, dass f_{univ} nicht in **T** definierbar ist. Angenommen sie wäre definierbar durch den Ausdruck e_{univ}, dann wäre δ ebenfalls definierbar durch den Ausdruck

$$e_\delta = \lambda(m : \texttt{Nat}).e_{univ}(m)(m)$$

In diesem Fall würden aber folgende Gleichungen gelten:

$$e_\delta(\overline{\ulcorner e \urcorner}) \equiv e_{univ}(\overline{\ulcorner e \urcorner})(\overline{\ulcorner e \urcorner}) \equiv e(\overline{\ulcorner e \urcorner})$$

Theorie: Undefinierbarkeit (V)

Dann können wir allerdings auch einen neuen Ausdruck e_{Δ} wie folgt definieren

$$e_{\Delta} = \lambda(x : \texttt{Nat}).S(e_{\delta}(x))$$

Theorie: Undefinierbarkeit (V)

Dann können wir allerdings auch einen neuen Ausdruck e_Δ wie folgt definieren

$$e_\Delta = \lambda(x : \texttt{Nat}).S(e_\delta(x))$$

und daraus schlussfolgern, dass

$$e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) \equiv S(e_\delta(\overline{\ulcorner e_\Delta \urcorner})) \equiv S(e_\Delta(\overline{\ulcorner e_\Delta \urcorner}))$$

Theorie: Undefinierbarkeit (V)

Dann können wir allerdings auch einen neuen Ausdruck e_Δ wie folgt definieren

$$e_\Delta = \lambda(x : \texttt{Nat}).S(e_\delta(x))$$

und daraus schlussfolgern, dass

$$e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) \equiv S(e_\delta(\overline{\ulcorner e_\Delta \urcorner})) \equiv S(e_\Delta(\overline{\ulcorner e_\Delta \urcorner}))$$

Dank Terminierungsgarantie wissen wir, dass es ein n geben muss, sodass $e_\Delta(\overline{\ulcorner e_\Delta \urcorner})1 \equiv \overline{n}$. Damit haben wir allerdings $\overline{n} \equiv S(\overline{n})$ und das ist unmöglich. □

Theorie: Undefinierbarkeit (V)

Dann können wir allerdings auch einen neuen Ausdruck e_Δ wie folgt definieren

$$e_\Delta = \lambda(x : \texttt{Nat}).S(e_\delta(x))$$

und daraus schlussfolgern, dass

$$e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) \equiv S(e_\delta(\overline{\ulcorner e_\Delta \urcorner})) \equiv S(e_\Delta(\overline{\ulcorner e_\Delta \urcorner}))$$

Dank Terminierungsgarantie wissen wir, dass es ein n geben muss, sodass $e_\Delta(\overline{\ulcorner e_\Delta \urcorner})1 \equiv \overline{n}$. Damit haben wir allerdings $\overline{n} \equiv S(\overline{n})$ und das ist unmöglich. $\square$

Durch diesen Beweis sehen wir: eine totale Sprache $\mathcal{L}$ kann nicht *universell* sein (einen Interpreter für $\mathcal{L}$ in $\mathcal{L}$ zulassen) und umgekehrt.

Zusätzliches zu System **T**

Product Types

Produkte von Typen (leer, binär, ternär,. . .) sind oft sehr nützliche Konstrukte in Programmiersprachen. Zum Beispiel erlaubt dies „multiple“Argumente/Rückgabewerte von Funktionen und sogar *primitive mutual recursion* (siehe PFPL, Kapitel 10).

Product Types

Produkte von Typen (leer, binär, ternär,...) sind oft sehr nützliche Konstrukte in Programmiersprachen. Zum Beispiel erlaubt dies „multiple"Argumente/Rückgabewerte von Funktionen und sogar *primitive mutual recursion* (siehe PFPL, Kapitel 10).

Eine Erweiterung von **T** um Produkttypen könnte so aussehen:

$\texttt{Typ}\ \tau$	$::=$	$\texttt{unit}$	Nullary Product
		$\texttt{prod}(\tau_1, \tau_2)$	Binary Product
$\texttt{Exp}\ e$	$::=$	$\texttt{triv}$	Empty tuple
		$\texttt{pair}(e_1; e_2)$	ordered pair
		$\texttt{pr}_1(e)$	left projection
		$\texttt{pr}_2(e)$	right projection

Sum Types

Summentypen sind ebenfalls sehr hilfreich, wenn es um Ausdrucksstärke geht. So kann zum Beispiel mit einer Summe unterschieden werden, ob es sich bei einem Knoten in einem Baum um ein Blatt oder eine Verzweigung handelt.

Sum Types

Summentypen sind ebenfalls sehr hilfreich, wenn es um Ausdrucksstärke geht. So kann zum Beispiel mit einer Summe unterschieden werden, ob es sich bei einem Knoten in einem Baum um ein Blatt oder eine Verzweigung handelt.

Eine Erweiterung von **T** um Summentypen könnte so aussehen:

$\texttt{Typ}\ \tau$	$::=$	$\texttt{void}$	Nullary Sum
		$\texttt{sum}(\tau_1, \tau_2)$	Binary Sum
$\texttt{Exp}\ e$	$::=$	$\texttt{abort}\{\tau\}(e)$	Abort
		$\texttt{inL}\{\tau_1, \tau_2\}(e)$	Left Injection
		$\texttt{inR}\{\tau_1, \tau_2\}(e)$	Right Injection
		$\texttt{case}(e; x_1.e_1; x_2.e_2)$	Case Analysis

Native Booleans (I)

Vielleicht der simpelste Summentyp sind die Wahrheitswerte ($\mathbb{B}$). Sie können entweder nativ dem Typsystem hinzugefügt werden oder über Summen + Unit simuliert werden.

`Typ` τ	$::=$	`Bool`	Booleans
`Exp` e	$::=$	`true`	Truth
		`false`	Falsity
		$\text{if}(e; e_1; e_2)$	conditional

Native Booleans (II)

Statics:

$$\frac{}{\Gamma \vdash \texttt{true} : \mathbb{B}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \mathbb{B}}$$

$$\frac{\Gamma \vdash e : \mathbb{B} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

Dynamics:

$$\frac{}{\texttt{true val}} \qquad \frac{}{\texttt{false val}}$$

$$\frac{}{\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \mapsto e_1}$$

$$\frac{}{\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \mapsto e_2}$$

$$\frac{e \mapsto e'}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mapsto \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

Hier könnte ihr Sprachenfeature stehen!

Es gibt noch einige Sprachfeatures, die zu **T** hinzugefügt werden könnten, um die Sprache noch ausdrucksstärker zu machen. Beispiele sind Gleichheit auf natürlichen Zahlen ($= : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$), Ungleichheit ($\neq, \leq, \geq : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$), Option Types, ...

Hier könnte ihr Sprachenfeature stehen!

Es gibt noch einige Sprachfeatures, die zu **T** hinzugefügt werden könnten, um die Sprache noch ausdrucksstärker zu machen. Beispiele sind Gleichheit auf natürlichen Zahlen ($= : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$), Ungleichheit ($\neq, \leq, \geq : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$), Option Types, ...

Challenge: Schreibt ein Programm in **T** (oder „**T**++") für den ggT (größter gemeinsamer Teiler, $\mathtt{ggT} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$). Welche Umformulierungen des klassischen euklidischen Algoritmus' sind dafür notwendig?

Fragen?

Muffin-Button

http://zombiegirl01.deviantart.com/