# Formalisierung von Inferenzsystemen in Coq am Beispiel von Typsystemen für Curry

Niels Bunkenburg

29.09.2016

Arbeitsgruppe für Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

# Motivation

Curry

Coq

# Curry

- Syntax ähnlich zu Haskell
- Nichtdeterminismus

```
(?)   :: a -> a -> a
x ? _ = x
_ ? y = y
```

- Freie Variablen

```
> 1 + 1 == x where x free
  {x = (-_x2)} False
  {x = 0} False
  {x = 1} False
  {x = 2} True
  {x = (2 * _x3 + 1)} False
  {x = (4 * _x4)} False
  {x = (4 * _x4 + 2)} False
```

# Coq - Aussagen

- Aussagen in Coq: `Prop`

```
1 + 1 = 2
False -> False
forall (X : Type) (l : list X), [] ++ l = l
fun (x : nat) => x <> 2
```

- Induktiv definierte Aussagen

```
Inductive inInd {X : Type} : X -> list X -> Prop :=
  | head : forall n l, inInd n (n :: l)
  | tail : forall n l e, inInd n l -> inInd n (e :: l).

Example e : inInd 2 [1;2;4].
Proof.
  apply tail. (* inInd 2 [2; 4] *)
  apply head.
Qed.
```

# Was ist Typisierung?

- **Typ**: Menge von Werten, die Eigenschaften und Bedeutung der Elemente bestimmt.
- **Ausdruck**: Kombination von Literalen, Variablen, Operatoren und Funktionen.
- **Kontext**: Enthält Informationen über Variablen und das Programm.
- **Typisierung**: In einem Kontext $\Gamma$ wird einem Ausdruck $e$ ein Typ $\tau$ zugewiesen, notiert als $\Gamma \vdash e :: \tau$.

Beispiele:

- $\Gamma \vdash 2 :: \texttt{Int}$
- $\Gamma, x \mapsto \texttt{Int} \vdash \texttt{x + 2} :: \texttt{Int}$

# Inferenzsysteme

- Inferenzsystem: Menge von Inferenzregeln
- Inferenzregel: $\dfrac{p_1 \ldots p_n}{c}$ wo $p_i$ Prämissen und $c$ Konklusion
- Notation für Implikation $p_1 \to \cdots \to p_n \to c$
- Darstellung von Inferenzregeln in Coq:

$$\frac{}{\texttt{In x (x :: l)}}\ \texttt{head} \qquad\qquad \frac{\texttt{In x l}}{\texttt{In x (y :: l)}}\ \texttt{tail}$$

```
Inductive inInd {X : Type} : X -> list X -> Prop :=
  | head : forall x l, inInd x (x :: l)
  | tail : forall x y l, inInd x l -> inInd x (y :: l).
```

# Formalisierung von Curry

Vorgehensweise:

1. Syntax und Kontext in Coq darstellen.
2. Typisierungsregeln in induktive Aussagen umwandeln.
3. Code umwandeln und Eigenschaften beweisen.

Sprachen:

- CuMin (Curry Minor): Vereinfachte Teilsprache von Curry.
- FlatCurry: Zwischensprache, die in Curry Compilern benutzt wird.

# CuMin – Syntax in BNF

Syntax von CuMin in Backus-Naur-Form:

$$
\begin{aligned}
P &::= D; P \mid D \\
D &::= f :: \kappa\tau; f\,\overline{x_n} = e \\
\kappa &::= \forall^{\epsilon}\alpha.\kappa \mid \forall^{*}\alpha.\kappa \mid \epsilon \\
\tau &::= \alpha \mid \mathsf{Bool} \mid \mathsf{Nat} \mid [\tau] \mid (\tau, \tau') \mid \tau \rightarrow \tau' \\
e &::= x \mid f_{\overline{\tau_m}} \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid n \mid e_1 + e_2 \mid e_1 \stackrel{\circ}{=} e_2 \\
&\quad \mid (e_1, e_2) \mid \mathsf{case}\ e\ \mathsf{of}\ \langle (x, y) \rightarrow e_1 \rangle \\
&\quad \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{True} \rightarrow e_1;\ \mathsf{False} \rightarrow e_2 \rangle \\
&\quad \mid \mathsf{Nil}_{\tau} \mid \mathsf{Cons}(e_1, e_2) \mid \mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{Nil} \rightarrow e_1;\ \mathsf{Cons}(x, y) \rightarrow e_2 \rangle \\
&\quad \mid \mathsf{failure}_{\tau} \mid \mathsf{anything}_{\tau}
\end{aligned}
$$

# CuMin – Syntax in Coq

Darstellung der Syntax von CuMin in Coq:

```
Inductive ty : Type :=
  | TVar  : id -> ty
  | TBool : ty
  | TNat  : ty
  | TList : ty -> ty
  | TPair : ty -> ty -> ty
  | TFun  : ty -> ty -> ty.

Inductive func_decl : Type :=
  | FDecl : id -> list quantifier -> ty ->
            list id -> tm -> func_decl.
```

# Beispiel CuMin

Formale Definition:

$$\text{fst} :: \forall^{*}\alpha.\forall^{*}\beta.(\alpha, \beta) \rightarrow \alpha \qquad \text{one} :: \text{Nat}$$

$$\text{fst}\ p = \text{case}\ p\ \text{of}\ \langle (u, v) \rightarrow u \rangle \qquad \text{one} = \text{fst}_{Nat,Bool}\ (1, \text{True})$$

Coq Definition:

```
Definition fst := FDecl (Id 0)
  [for_all (Id 1) tag_star; for_all (Id 2) tag_star]
  (TFun (TPair (TVar (Id 1)) (TVar (Id 2))) (TVar (Id 1)))
  [Id 3]
  (tcasep (tvar (Id 3)) (Id 4) (Id 5) (tvar (Id 4))).

Definition one := tapp (tfun (Id 0) [TNat; TBool])
  (tpair (tsucc tzero) ttrue).
```

# CuMin – Datentypen und Kontext

Inferenzregeln für Datentypen:

$$\Gamma, \alpha^* \vdash \alpha \in \mathsf{Data} \qquad \Gamma \vdash \mathsf{Bool} \in \mathsf{Data} \qquad \Gamma \vdash \mathsf{Nat} \in \mathsf{Data}$$

$$\frac{\Gamma \vdash \tau \in \mathsf{Data}}{\Gamma \vdash [\tau] \in \mathsf{Data}} \qquad \frac{\Gamma \vdash \tau \in \mathsf{Data} \quad \Gamma \vdash \tau' \in \mathsf{Data}}{\Gamma \vdash (\tau, \tau') \in \mathsf{Data}}$$

Informationen, die im Kontext enthalten sind:

```
Definition program := list func_decl.

Inductive context : Type :=
  | con: (partial_map id tag) -> (partial_map id ty) ->
        context.
```

# CuMin – formales Typsystem

$$\Gamma,\ x \mapsto \tau \vdash x :: \tau \qquad \Gamma \vdash \text{True} :: \text{Bool}$$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash e_1\ e_2 :: \tau_2}$$

$$\frac{\Gamma \vdash e :: (\tau_1, \tau_2) \quad \Gamma, l \mapsto \tau_1, r \mapsto \tau_2 \vdash e_1 :: \tau}{\Gamma \vdash \text{case}\ e\ \text{of}\ \langle (l, r) \to e_1 \rangle :: \tau}$$

$$\frac{(f :: \forall^{v_1} \alpha_1. \cdots \forall^{v_m} \alpha_m.\tau; f\,\overline{x_n} = e) \in P}{\Gamma \vdash f_{\overline{\tau_m}} :: \tau[\overline{\tau_m / \alpha_m}]} \bigstar$$

$\bigstar$ Falls für alle $i$ mit $v_i = *$ gilt $\Gamma \vdash \tau_i \in \text{Data}$.

# CuMin – Typsystem in Coq

```
Reserved Notation "Gamma '|-' t ':::' T" (at level 40).
Inductive has_type : context -> tm -> ty -> Prop :=
  | T_Var   : forall Gamma x T,
               (type_con Gamma) x = Some T ->
               Gamma |- tvar x ::: T
  | T_App   : forall Gamma e1 e2 T1 T2,
                Gamma |- e1 ::: (TFun T1 T2) ->
                Gamma |- e2 ::: T1 ->
                Gamma |- (tapp e1 e2) ::: T2
  | T_CaseP : forall Gamma e e1 l r T T1 T2,
                Gamma |- e ::: (TPair T1 T2) ->
                let Omega := (type_update Gamma l T1)
                in (type_update Omega r T2) |- e1 ::: T ->
                Gamma |- (tcasep e l r e1) ::: T
where "Gamma '|-' t ':::' T" := (has_type Gamma t T).
```

```
T_Fun: forall Gamma id tys T,
         let result := lookup_func Prog id in
         let fdecl  := fromOption default_fd result in
         specialize_func fdecl tys = Some T ->
         Forall (is_data_type Gamma)
                (fd_to_star_tys fdecl tys) ->
         Gamma |- (tfun id tys) ::: T
```

# FlatCurry im Vergleich zu CuMin

Syntax:

- Weniger Ausdrücke und Typen, dafür allgemeinere Form.
- Applikation von Funktionen auf mehr als ein Argument gleichzeitig.
- Let Ausdrücke mit beliebig vielen Bindungen.

$\rightarrow$ Komplexere Typisierungsregeln

# Zusammenfassung