Magnolia

Implementing System F with Anonymous Sums and Products

Andreas Halkjær From
Introduction

What foundation should we build a functional language on in 2018? Bidirectional typing seems a promising start: Used by Scala, PureScript. Known for its easy scalability to advanced features such as rank-n, higher-kindled and sized types.

Starting point:

Complete and Easy Bidirectional Type Checking for Higher-Rank Polymorphism by Joshua Dunfield & Neelakantan R. Krishnaswami

We will add sums and products manually, based on a common notion of rows of types.

Finally we will look briefly at Bob Harper’s Abstract Binding Trees that aid in the implementation.
Magnolia

OCaml + Jane Street Core. 3100 lines + tests.

- OCamllex
- Menhir
- Elaborator
- Type checker:
  - Complete and Easy Bidirectional Type Checking for Higher-Rank Polymorphism by Joshua Dunfield & Neelakantan R. Krishnaswami
  - Sums & products
  - Strictly-positive recursive types & catamorphisms
  - Elaborate to explicit type abstraction/instantiation
- prettiest, based on Jean-Philippe Bernardy’s A Pretty But Not Greedy Printer (Functional Pearl). ICFP 2017
- Interpreter
Example Code

```haskell
alias listF A R = [nil: {} | cons: {head: A, tail: R}]
alias list A = mu R. listF A R

let nil : forall A. list A
    = fold (. nil {})

let cons : forall A. A -> list A -> list A
    | head tail = fold (. cons {head, tail})

let my-list : list int
    = cons 1 (cons 2 (cons 3 (cons 4 nil)))

let main : int
    = cata [ nil -> 0
              | cons {tail: n} -> n + 1 ]
       my-list

==> 4
```
Contents

Part 1:
• Complete and Easy
  • Existential Type Variables
  • Universal Quantifiers

Part 2:
• Data Types
  • Rows
  • Types
  • Terms
  • Recursive Types (briefly)

Part 3:
• Abstract Binding Trees
  • Operators
  • Arities
  • Matching
  • Example
Part I

Complete and Easy
\[ e ::= () \mid x \mid \lambda x. e \mid e\, e \]

\[ \sigma, \tau ::= \bot \mid \sigma \rightarrow \tau \]

**Judgment** \( \Gamma \vdash e : \tau \)

\[
\begin{align*}
\Gamma & \vdash () : \bot \\
\Gamma & \vdash x : \tau \quad (x : \tau \in \Gamma)
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : \sigma & \vdash e : \tau \\
\Gamma & \vdash \lambda x. e : \sigma \rightarrow \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash e_1 : \sigma \rightarrow \tau \\
\Gamma & \vdash e_2 : \sigma \\
\Gamma & \vdash e_1\, e_2 : \tau
\end{align*}
\]
Complete and Easy

Bidirectional Typing

Insight: Context matters. Two judgments!

**Synthesis** \( \Gamma \vdash e \Rightarrow \tau \)  

**Checking** \( \Gamma \vdash e \Leftarrow \tau \)

\[
\begin{align*}
\text{Well-formedness } \Gamma \vdash \tau & \\
\Gamma \vdash () & \Rightarrow 1 \\
\Gamma \vdash x & \Rightarrow \tau
\end{align*}
\]

Switching:

\[
\begin{align*}
\Gamma \vdash \tau & \\
\Gamma \vdash e & \Leftarrow \tau \\
\Gamma \vdash (e : \tau) & \Rightarrow \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e & \Rightarrow \sigma \\
\sigma & \equiv \tau \\
\Gamma \vdash e & \Leftarrow \tau
\end{align*}
\]

We need to know \( \sigma \) and \( \tau \) (but only once!):

\[
\begin{align*}
\Gamma, x : \sigma & \vdash e \Leftarrow \tau \\
\Gamma & \vdash \lambda x.e \Leftarrow \sigma \rightarrow \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_1 & \Rightarrow \sigma \rightarrow \tau \\
\Gamma \vdash e_2 & \Leftarrow \sigma \\
\Gamma & \vdash e_1 \ e_2 \Rightarrow \tau
\end{align*}
\]
Complete and Easy

Existential Type Variables I

Synthesis for \( \lambda \)-expressions? Existential variables + ordered output context.

\[\tau ::= \ldots | \hat{\alpha}, \hat{\beta}\]

**Synthesis** \( \Gamma \vdash e \Rightarrow \tau \vdash \Delta \)

\(\Delta\) might solve more existentials than \(\Gamma\).

**Checking** \( \Gamma \vdash e \Leftarrow \tau \vdash \Delta \)

\[
\begin{align*}
\Gamma \vdash () & \Rightarrow 1 \vdash \Gamma \\
\Gamma \vdash x & \Rightarrow \tau \vdash \Gamma \\
\Gamma \vdash e & \Leftarrow \tau \vdash \Delta \\
\Gamma, x : \sigma & \vdash e \Leftarrow \tau \vdash \Delta, x : \sigma, \Theta
\end{align*}
\]

\[
\begin{align*}
\sigma & \equiv \tau \\
\Gamma & \vdash e \Rightarrow \tau \vdash \Delta \\
\Gamma \vdash e & \Leftarrow \tau \vdash \Delta \\
\Gamma, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} & \vdash e \Leftarrow \hat{\beta} \vdash \Delta, x : \hat{\alpha}, \Theta
\end{align*}
\]

Remember to enforce scope:

\[
\begin{align*}
\Gamma, x : \sigma & \vdash e \Leftarrow \tau \vdash \Delta, x : \sigma, \Theta \\
\Gamma & \vdash \lambda x.e \Leftarrow \sigma \rightarrow \tau \vdash \Delta \\
\Gamma, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} & \vdash e \Leftarrow \hat{\beta} \vdash \Delta, x : \hat{\alpha}, \Theta \\
\Gamma & \vdash \lambda x.e \Rightarrow \hat{\alpha} \rightarrow \hat{\beta} \vdash \Delta
\end{align*}
\]
Existential Type Variables II

Application of an existential type variable? Extra judgment!

Application $\Gamma \vdash \sigma \cdot e \Rightarrow \tau \vdash \Delta$

$$
\frac{
\Gamma \vdash e_1 \Rightarrow \sigma \vdash \Theta \quad \Theta \vdash [\Theta] \sigma \cdot e_2 \Rightarrow \tau \vdash \Delta
}{
\Gamma \vdash e_1 \ e_2 \Rightarrow \tau \vdash \Delta
}
$$

Context substitution $[\Theta] \tau$ substitutes existentials in $\tau$ for solutions from $\Theta$.

$$
\frac{
\Gamma \vdash e \Leftarrow \sigma \vdash \Delta
}{
\Gamma \vdash \sigma \rightarrow \tau \cdot e \Rightarrow \tau \vdash \Delta
}
\quad
\frac{
\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash e \Leftarrow \hat{\alpha}_1 \vdash \Delta
}{
\Gamma[\hat{\alpha}] \vdash \hat{\alpha} \cdot e \Rightarrow \hat{\alpha}_2 \vdash \Delta
}
$$

The holed context $\Gamma[\hat{\alpha}]$ is short for $\Gamma_l, \hat{\alpha}, \Gamma_r$.

$\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2]$ plugs the hole with something else.
Checking against an existential type variable? Subtyping!

\[ \boxed{\Gamma \vdash e \Rightarrow \sigma \vdash \Theta \quad \Theta \vdash [\Theta]\sigma \leq [\Theta]\tau \vdash \Delta} \]

\[ \Gamma \vdash e \Leftarrow \tau \vdash \Delta \]

Common-sense rules (omitted) + instantiation:

\[ \boxed{\Gamma \vdash \hat{\alpha} \leq\tau \vdash \Delta} \quad \Gamma \vdash \tau \leq \hat{\alpha} \vdash \Delta} \]

\[ \hat{\alpha} \notin \text{FV}(\tau) \quad \Gamma[\hat{\alpha}] \vdash \hat{\alpha} \leq\tau \vdash \Delta} \]

\[ \Gamma[\hat{\alpha}] \vdash \hat{\alpha} \leq\tau \vdash \Delta} \]

\[ \hat{\alpha} \notin \text{FV}(\tau) \quad \Gamma[\hat{\alpha}] \vdash \tau \leq \hat{\alpha} \vdash \Delta} \]

\[ \Gamma[\hat{\alpha}] \vdash \tau \leq \hat{\alpha} \vdash \Delta} \]
\[
\Gamma \vdash \tau \\
\Gamma, \hat{\alpha}, \Gamma' \vdash \hat{\alpha} \triangleq \tau \dashv \Gamma, \hat{\alpha} = \tau, \Gamma'
\]

\[
\Gamma[\hat{\alpha}][\hat{\beta}] \vdash \hat{\alpha} \triangleq \hat{\beta} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]
\]

Function arrow is contravariant:

\[
\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash \sigma \triangleq \hat{\alpha}_1 \dashv \Theta \quad \Theta \vdash \hat{\alpha}_2 \triangleq [\Theta] \tau \dashv \Delta \\
\Gamma[\hat{\alpha}] \vdash \hat{\alpha} \triangleq \sigma \to \tau \dashv \Delta
\]

Symmetric rules for right instantiation.
Monotypes $\sigma, \tau ::= 1 | \alpha | \hat{\alpha} | \sigma \rightarrow \tau$

Types $A, B, C ::= \tau | A \rightarrow B | \forall \alpha.A$

Existentials only stand in for monotypes.

Checking a polymorphic type:

$$\Gamma, \alpha \vdash e \iff A \vdash \Delta, \alpha, \Theta$$

$$\Gamma \vdash e \iff \forall \alpha.A \vdash \Delta$$

No synthesis rule.

Applying a polymorphic type instantiates it:

$$\Gamma, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]A \bullet e \Rightarrow C \vdash \Delta$$

$$\Gamma \vdash \forall \alpha.A \bullet e \Rightarrow C \vdash \Delta$$
How to answer \( \forall \alpha. A <: B \)?

Can we instantiate quantifier suitably:

\[
\Gamma, \uparrow \hat{\alpha}, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]A <: B \vdash \Delta, \uparrow \hat{\alpha}, \Theta \\
\Gamma \vdash \forall \alpha. A <: B \vdash \Delta
\]

We may solve existential so use a marker, \( \uparrow \hat{\alpha} \).

What about \( A <: \forall \beta. B \)?

Is \( A \) a subtype of \( B \) for arbitrary \( \beta \):

\[
\Gamma, \beta \vdash A <: B \vdash \Delta, \beta, \Theta \\
\Gamma \vdash A <: \forall \beta. B \vdash \Delta
\]
∀β. B :≤ ˆα?

Instantiate quantifier with fresh existential.

\[
\begin{align*}
\Gamma[\hat{\alpha}], \triangleright_\beta, \hat{\beta} \vdash [\hat{\beta}/\beta]B & :≤ \hat{\alpha} \dashv \Delta, \triangleright_\beta, \Theta \\
\Gamma[\hat{\alpha}] \vdash \forall \beta. B & :≤ \hat{\alpha} \dashv \Delta
\end{align*}
\]

ˆα :≤ ∀β. B?

Make ˆα a subtype of B for arbitrary β:

\[
\begin{align*}
\Gamma[\hat{\alpha}], \beta \vdash \hat{\alpha} & :≤ B \dashv \Delta, \beta, \Theta \\
\Gamma[\hat{\alpha}] \vdash \hat{\alpha} & :≤ \forall \beta. B \dashv \Delta
\end{align*}
\]
Let

\[ e ::= \ldots \mid \text{let } x = e \text{ in } e \]

\[
\frac{\Gamma \vdash e_1 \Rightarrow \sigma \vdash \Theta, x : \sigma \vdash e_2 \Rightarrow A \vdash \Delta, x : \sigma, \Delta'}
{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow A \vdash \Delta}
\]

\[
\frac{\Gamma \vdash e_1 \Rightarrow \sigma \vdash \Theta, x : \sigma \vdash e_2 \Leftarrow A \vdash \Delta, x : \sigma, \Delta'}
{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Leftarrow A \vdash \Delta}
\]

No generalisation to preserve cut-elimination property.
E.g. in let \( \text{id} = \lambda x. x \) in \( \text{id} () \), \( \text{id} \) will have type \( () \to () \).
Part II

Data Types
Data Types

Rows

Only monotypes in rows.

Component-wise well-formedness:

\[
\begin{align*}
\cdots \ & \Gamma \vdash \tau_i \ \cdots \\
\Gamma \vdash \#\{\ldots, c_i : \tau_i, \ldots\} \text{ row}
\end{align*}
\]

Component-wise subtyping:

\[
\begin{align*}
\cdots \ & \Gamma_{i-1} \vdash \tau_i < : \sigma_i \vdash \Gamma_i \ c_i = d_i \ \cdots \\
\Gamma_0 \vdash \#\{\ldots, c_i : \tau_i, \ldots\} < : \#\{\ldots, d_i : \sigma_i, \ldots\} \vdash \Gamma_n
\end{align*}
\]

Component-wise instantiation:

\[
\begin{align*}
\Gamma_0[\hat{\alpha}_1, \ldots, \hat{\alpha} = \#\{c_1 : \hat{\alpha}_1, \ldots\}] \vdash \hat{\alpha}_1 \colon= \tau_1 \vdash \Gamma_1 \ \cdots \\
\Gamma_{i-1} \vdash \hat{\alpha}_i \colon= \tau_i \vdash \Gamma_i \ \cdots \\
\Gamma_0[\hat{\alpha}] \vdash \hat{\alpha} \colon= \#\{\ldots, c_i : \tau_i, \ldots\} \vdash \Gamma_n
\end{align*}
\]

Symmetric right instantiation.
Data Types

Sum and Product Types

\[
\begin{align*}
\Gamma \vdash r \text{ row} & \quad \text{ } & \Gamma \vdash r \text{ row} \\
\Gamma \vdash [r] & \quad \text{ } & \Gamma \vdash \{r\}
\end{align*}
\]

Subtyping on rows (no fanciness).

\[
\begin{align*}
\Gamma \vdash r <: r' & \vdash \Delta & \Gamma \vdash r <: r' & \vdash \Delta \\
\Gamma \vdash [r] <: [r'] & \vdash \Delta & \Gamma \vdash \{r\} <: \{r'\} & \vdash \Delta
\end{align*}
\]
Sums

Eliminators. Collection of functions:

\[
\ldots \quad \Gamma_{i-1} \vdash e_i \iff \tau_i \to B \vdash \Gamma_i \quad \ldots \\
\Gamma_0 \vdash [\ldots, c_i \to e_i, \ldots] \iff [\ldots, c_i : \tau_i, \ldots] \to B \vdash \Gamma_n
\]

\[
\Gamma_{i-1} \vdash e_i \iff \hat{\alpha}_i \to \hat{\beta} \vdash \Gamma_i \quad \ldots \\
\Gamma_0 \vdash [\ldots, c_i \to e_i, \ldots] \Rightarrow [\ldots, c_i : \hat{\alpha}_i, \ldots] \to \hat{\beta} \vdash \Gamma_n
\]

Injection:

\[
\Gamma \vdash e \iff \tau_k \vdash \Delta \\
\Gamma \vdash c_k \cdot e \iff [\ldots, c_k : \tau_k, \ldots] \vdash \Delta
\]

No synthesis for injection (can build yourself).
Possibilities: Existential row variables, polymorphic variants.
Data Types

Products

Records:

\[ \ldots \quad \Gamma_{i-1} \vdash e_i \iff \tau_i \vdash \Gamma_i \quad \ldots \]

\[ \Gamma_0 \vdash \{ \ldots, c_i : e_i, \ldots \} \iff \{ \ldots, c_i : \tau_i, \ldots \} \vdash \Gamma_n \]

\[ \ldots \quad \Gamma_{i-1} \vdash e_i \Rightarrow A_i \vdash \Gamma_i \quad \ldots \]

\[ \Gamma_0 \vdash \{ \ldots, c_i : e_i, \ldots \} \Rightarrow \{ \ldots, c_i : A_i, \ldots \} \vdash \Gamma_n \]

Projection:

\[ \Gamma \vdash e \Rightarrow \{ \ldots, c_k : A_k, \ldots \} \vdash \Delta \]

\[ \Gamma \vdash e \cdot c_k \Rightarrow A_k \vdash \Delta \]

Have to know type of \( e \) (can write own projection function).
Data Types

Recursive Types

Inspired by *Practical Foundations for Programming Languages* by Bob Harper:

\[
\tau ::= \ldots \mid \mu r. \tau \\
\]

\[
e ::= \ldots \mid \text{fold } e \mid \text{unfold } e \mid \text{cata } e e \\
\]

Also: “X marks the spot-mapping”:

\[
e ::= \ldots \mid \text{map}\{X.\tau\} \ e \ e \\
\]

e.g.

\[
\text{map}\{X.\text{list } X\}, \\
\text{map}\{X.\text{[none: {}|some: } X]\}, \\
\]
Part III

Abstract Binding Trees
\[ \ldots \ [y/x]A \ldots \]
\[ \ldots \forall x.A \ldots \]

M’colleague Bob Atkey once memorably described the capacity to put up with De Bruijn indices as a Cylon detector, the kind of reverse Turing Test that the humans in Battlestar Galactica invent, the better to recognize one another by their common inadequacies. He had a point.

— Conor McBride, “I am not a number, I am a classy hack”
Abstract Binding Trees

Operators

Using ABT library in OCaml (port of CMU’s SML library).

```ocaml
module type op =
  (* Rows *)
  | Vec of int
  | Tag of Tag.t

  (* Types *)
  | Basic of basic
  | Exi of ExiVar.t
  | Arr
  | All
  | Sum
  | Prod
  | Mu

  (* Terms *)
  | Lit of literal
  | Ann
  | App

  (* Explicit polymorphism *)
  | Lam of typed
  | Let

  (* Datatypes *)
  | Gen
  | Inst

  (* Recursive datatypes *)
  | Gen
  | Inst
  | Lit of literal
  | Ann
  | App

  (* Gen *)
  | Lam of typed
  | Let

  (* Inst *)
  | Gen
  | Inst

  (* Datatypes *)
  | Gen
  | Inst

  (* Recursive datatypes *)
  | Fold of typed
  | Unfold
  | Cata of typed
```

25 DTU Compute  Magnolia  15 March 2018
let arity op =
  match op with
  | Vec n -> List.init ~f:(const 0) n
  | Tag _ -> [0]
  | Basic _ -> []
  | Exi _ -> []
  | Arr -> [0; 0]
  | All -> [1]
  | Sum -> [0]
  | Prod -> [0]
  | Mu -> [1]
  | Lit _ -> []
  | Ann -> [0; 0]
  | App -> [0; 0]
  | Lam Untyped -> [1]
  | Lam Typed -> [1; 0]
  | Let -> [0; 1]
  | Gen -> [1]
  | Inst -> [0; 0]
  | Inj (_, Untyped) -> [0]
  | Inj (_, Typed) -> [0; 0]
  | Proj (_, Untyped) -> [0]
  | Proj (_, Typed) -> [0; 0]
...
Abstract Binding Trees

Out

Build a term \((\lambda x.x)\):

```plaintext
let x = Syntax.Var.named "x" in
Lam $$ [x ^^ (!! x)]
```

Syntax.out e returns one of:

- **VarView** \(x\)
- **AbsView** \(x.e\) (where \(x\) is fresh and free in \(e\))
- **AppView** \(op(e_1, \ldots, e_n)\) (corresponding to the arity of \(op\))

$$, out etc. throw errors on arity mismatch.

Also get: subst, aequiv.

Note: We give up on some static help from the compiler.
Also: Not fast.
Abstract Binding Trees

Example

\[
\Gamma[\hat{\alpha}], \triangleright_{\hat{\beta}}, \hat{\beta} \vdash [\hat{\beta}/\beta]B \leq \hat{\alpha} \rightarrow \Delta, \triangleright_{\hat{\beta}}, \Theta \]

\[
\Gamma[\hat{\alpha}] \vdash \forall \beta.B \leq \hat{\alpha} \rightarrow \Delta
\]

(* InstRAll *)

| AppView (All, [t]) ->

  (match Syntax.out t with
   | AbsView (b, t) ->
     let b' = fresh_exi () in
     let ctx = ctx +> Marker b' +> ExiVar b' and inst = Syntaxsubst (Exi b' $$ [])) b t in
     let%bind ctx = instr ctx ~typ:inst ~var:a in
     add_inst inst;
     return (Ctx.until (Marker b')) ctx)
   | _ -> raise Syntax.Malformed)