Let Arguments Go First

Ningning Xie, Bruno C. d. S. Oliveira
The University of Hong Kong

ESOP 2018, Thessaloniki, Greece
2018-04-17

Background

Bi-Directional Type Checking

- Well known in the folklore of type system for a long time
- Popularized by Pierce and Turner's work*
- Can support many type system features: refinements, indexed types, intersections and unions, contextual modal types, object-oriented subtyping, …

* Benjamin C Pierce and David N Turner. Local type inference. TOPLAS, 22(1):1–44, 2000.

Bi-Directional Type Checking

Two Modes in type-checking

- Inference (synthesis): e synthesizes A

$$\Gamma \vdash e \Rightarrow A$$

- Checked: check e against A

$$\Gamma \vdash e \Leftarrow A$$

- Which constructs should be in which mode?

Bi-Directional Type Checking

- A recipe from Dunfield and Pfenning*
  - Introduction rules $\Leftarrow$

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B}$$

  - Elimination rules $\Rightarrow$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \text{APP}$$

Notice here how type information flows from functions to arguments

* Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. POPL' 04, 2004.

Bi-Directional Type Checking

- Consider designing rules for pairs

$$\frac{\Gamma \vdash e_1 \Leftarrow A \qquad \Gamma \vdash e_2 \Leftarrow B}{\Gamma \vdash (e_1, e_2) \Leftarrow (A, B)} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash (e_1, e_2) \Rightarrow (A, B)}$$

- (1, 2) cannot type-check
  …unless you write it as (1, 2) : (int, int)
- …unless you have also **an inference rule for pairs**
- Rules **scales up** with the typing rules

Contributions

- **A variant of bi-directional type checking**
  with an application mode
  type information propagates from arguments to functions
- **A new design for type inference of higher-ranked types**
  generalizes the Hindley-Milner type system
  supports syntactic sugar for polymorphic let
- **A System-F like calculus**
  compatible with type application
  encoding type declaration

Except the algorithm system, most parts are formalized in Coq

Contributions

- **A variant of bi-directional type checking**
  with an application mode
  type information propagates from arguments to functions
- **A new design for type inference of higher-ranked types**
  generalizes the Hindley-Milner type system
  supports syntactic sugar for polymorphic let
- **A System-F like calculus**
  compatible with type application
  encoding type declaration

Application Mode

Application Mode

$$(\lambda x.\ x)\ 1$$

- cannot type-check
  …unless you write it as $((\lambda x.\ x) : \mathsf{int} \to \mathsf{int})\ 1$
- …unless you have also an inference rule for lambdas
  (with some type inference)
- **…WAIT!** What if the type of the argument is accounted for in inferring the function?

Application Mode

$$(\lambda x.\ x)\ 1$$

The argument 1 has type int
Can the function accept arguments of int
Let's assume x : int
We have return type int
The type of function is int → int

Application Mode

Instead of…

$$\frac{\Gamma \vdash e_1 \;\Rightarrow\; A \to B \qquad \Gamma \vdash e_2 \;\Leftarrow\; A}{\Gamma \vdash e_1\ e_2 \;\Rightarrow\; B}\ \text{APP}$$

An alternative idea is to **push** the type of the arguments into the typing of the function

$$\frac{\Gamma \vdash e_2 \;\Rightarrow\; A \qquad \Gamma \mid \Psi, A \vdash e_1 \;\Rightarrow\; A \to B}{\Gamma \mid \Psi \vdash e_1\ e_2 \;\Rightarrow\; B}$$

Application Mode

- Application context Ψ is a stack that tracks the type of the arguments
- Lambda expressions can now make use of the application context

$$\frac{\Gamma, x : A \mid \Psi \vdash e \;\Rightarrow\; B}{\Gamma \mid \Psi, A \vdash \lambda x.\; e \;\Rightarrow\; A \to B}\;\textsc{Lam}$$

- $(\lambda x.\; x)\; 1$

Application Mode

Two Modes in type-checking

- Inference (synthesis): e synthesizes A

  $$\Gamma \vdash e \Rightarrow A$$

- Application: under application ctx Ψ, e synthesizes A

  $$\Gamma \mid \Psi \vdash e \Rightarrow A$$

Application Mode

Interpretation

$$\Gamma \vdash e \Rightarrow A$$

$$\Gamma \mid \Psi \vdash e \Rightarrow A$$

- Recipe?

  Whether the expression can be applied or not

  - Expressions can be applied:
    variables, lambdas, applications, eliminations of pairs,…
  - Expressions that cannot be applied:
    literals, pairs,…

Application Mode

Interpretation

$$\Gamma \vdash e \Rightarrow A$$

$$\Gamma \mid \Psi \vdash e \Rightarrow A$$

- What if the application context is empty?

…it is not applied to any arguments
…we know nothing about the expression
…we should infer it!

  - We can model inference mode as a particular case of the application mode

Application Mode

Interpretation

$$\Gamma \vdash e \Rightarrow A$$

$$\Gamma \mid \Psi \vdash e \Rightarrow A$$

- What if we have a possibly non-empty application context?

$$e : A \to B \to C$$

- Inference: e
- Checked : e, A → B → C
- Application Mode : e $\quad \Psi = \emptyset \quad \Psi = A \quad \Psi = A, B$

finer grain notion leads to partial type checking

Application Mode

Interpretation

$$\Gamma \vdash e \Rightarrow A$$

$$\Gamma \mid \Psi \vdash e \Rightarrow A$$

- Is Inference mode + Application mode better/worse than Bi-Directional type checking?
  - No one is conservative over the other.

    $(\lambda x.\ x)\ 1$

    $(\lambda x : \mathsf{int} \to \mathsf{int}. \lambda y : \mathsf{bool}.\ y)\ (\lambda y.\ y)\ \mathsf{True}$
  - But it does open paths to design choices

Application Mode

Benefits

$$\Gamma \vdash e \Rightarrow A$$

$$\Gamma \mid \Psi \vdash e \Rightarrow A$$

- Local constraint solver for function variables.

  Type system with implicit polymorphism and/or static overloading needs type information about the arguments when type-checking function variables.

  id 3, (==) True False, …

  Type system employs an **application subtyping** $\Psi \vdash A \leq B$

Application Mode

Benefits

$$\Gamma \vdash e \Rightarrow A$$

$$\Gamma \mid \Psi \vdash e \Rightarrow A$$

- Declaration desugaring for lambda abstractions.

  Annotations are never needed for applied lambdas

  $(\lambda x.\ x)\ 1$

  It enables **let sugar**

  let x = e_1 **in** $e_2 \rightsquigarrow (\lambda \mathtt{x}.\ e_2)\ e_1$

Contributions

- **A variant of bi-directional type checking**
  with an application mode
  type information propagates from arguments to functions
- **A new design for type inference of higher-ranked types**
  generalizes the Hindley-Milner type system
  supports syntactic sugar for polymorphic let
- **A System-F like calculus**
  compatible with type application
  encoding type declaration

Application 1 for the Application Mode

Application 1

- Type Inference of (implicit, predicative) Higher-Ranked Types
  - Enables expressiveness power of System F.
  - Undecidable.
  - Hindley-Milner*(henceforth HM) with let generalization has rank 1 types.

* Luis Damas and Robin Milner. Principal type-schemes for functional programs. POPL ' 82, 1982.
* J. Roger Hindley. The principal type-scheme of an object in combinatory logic. Transactions of the American Mathematical Society, 146:29–60, 1969.

Application 1

- Type Inference of Higher-Ranked Types
  - GHC rejects:

    ```
    (λf. (f 1, f 'c')) (λx. x)
    ```

  - Rewriting according to bi-directional guideline

    ```
    ((λf. (f 1, f 'c')) : (∀a. a → a) → (Int, Char)) (λx . x)
    ```

  - …Wait! If we generalize the identity function and propagate it into the function…

* a state-of-the-art compiler for Haskell

Application 1

- Type Inference of Higher-Ranked Types

  Apply the application mode to higher-ranked type system*
  with generalization on applications

* Martin Odersky and Konstantin L¨aufer. Putting type annotations to work. POPL ’ 96, 1996.
* Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. Journal of func- tional programming, 17(01):1–82, 2007.
* Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. ICFP ’ 13, 2013.

Application 1

- Type Inference of Higher-Ranked Types

  1. Sugars for HM style polymorphic let expression

  2. Conservative over the HM type system

  3. Comparison with existing literatures

Application 1

System	Types	Impred	Let	Annotations
ML^F	flexible and rigid	yes	yes	on polymorphically used parameters
HML	flexible F-types	yes	yes	on polymorphic parameters
FPH	boxy F-types	yes	yes	on polymorphic parameters and some let bindings with higher-ranked types
Peyton Jones et al. (2007)	F-types	no	yes	on polymorphic parameters
Dunfield et al. (2013)	F-types	no	no	on polymorphic parameters
this paper	F-types	no	sugar	on polymorphic parameters that are not applied

Application 1

- Type Inference of Higher-Ranked Types

  1. Sugars for HM style polymorphic let expression
  2. Conservative over the HM type system
  3. Comparison with existing literatures
  4. Interesting metatheory studies about the application mode formalized in Coq.
  5. An algorithm; a translation to System F

Contributions

- **A variant of bi-directional type checking**
  with an application mode
  type information propagates from arguments to functions
- **A new design for type inference of higher-ranked types**
  generalizes the Hindley-Milner type system
  supports syntactic sugar for polymorphic let
- **A System-F like calculus**
  compatible with type application
  encoding type declaration

Application 2 for the Application Mode

Application 2

"It is possible, of course, to come up with examples where it would be beneficial to synthesize the argument types first and then use the resulting information to avoid type annotations in the function part of an application expression....Unfortunately this refinement does not help infer the type of polymorphic functions. For example, we cannot uniquely determine the type of x in the expression `(fun[A](x) e) [Int] 3`" *

* Benjamin C Pierce and David N Turner. Local type inference. TOPLAS, 22(1):1–44, 2000.

Application 2

- A Variant of System F with
  More Expressive Type Applications

  $(\Lambda$a. λx : a. x + 1) Int

  …not typeable in traditional System F

  …using application mode, we can verify $a = \mathsf{Int}$

  Use application context to track type equalities
  introduced by type application

Application 2

- A Variant of System F with
  More Expressive Type Applications

  ```
  (fun[A](x) e) [Int] 3
  ```

  x:A or x:Int

Application 2

- A Variant of System F with
  More Expressive Type Applications

  1. Sugar for type synonyms **type** a = A **in** e $\rightsquigarrow$ (Λa. e) A

  2. Preserve System F type abstraction

     (Λa. λx : a. x + 1) Int

     let inc = Λa. λx : a. x + 1 **in** inc Int e

Application 2

- A Variant of System F with
  More Expressive Type Applications

  1. Sugar for type synonyms **type** a = A **in** e $\rightsquigarrow$ (Λa. e) A

  2. Preserve System F type abstraction

  3. Metatheory studies formalized in Coq.
     type safety, uniqueness of typing

Discussion

Discussion

- Combine application and checked mode
- Additional constructs: pairs
- Encoding declarations in dependent type systems*
- Related work

See our full paper if you are interested!

* Paula Severi and Erik Poll. Pure type systems with definitions. Logical Foundations of Computer Science, pages 316–328, 1994.

Thanks !

Let Arguments Go First

Ningning Xie, Bruno C. d. S. Oliveira
The University of Hong Kong

ESOP 2018, Thessaloniki, Greece
2018-04-17