Bidirectional Type Checking

Eduardo Bonelli

January 31, 2019

Bidirectional Type-Checking

Implementing Bidirectional Type-Checking

Bidirectional Type-Checking

Type System $\neq$ Type-Checking Algorithm

- BDT is a technique that assigns an operational reading to the typing rules in a type system
- This allows one to deduce a type-checking algorithm from a type system or else get very close to one
- Benefit: it requires type declarations on (some) expressions but not on all bound variables
  - Somewhere in between inferring types and having to declare the types of all the bound variables

Example of Non-Syntax Directed Typing Rules

$$\sigma, \tau \quad ::= \quad \mathbb{B} \,|\, \mathbb{N} \,|\, \sigma \to \tau$$

- Consider the mode (where $+$ is an input and $-$ an output)

$$+ \rhd + : -$$

- Dropping annotations in abstractions leads to (T-Abs) not having an obvious algorithmic reading

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x : \sigma}\;(\text{T-Var}) \qquad \frac{}{\Gamma \rhd \mathit{true} : \mathbb{B}}\;(\text{T-True}) \qquad \frac{}{\Gamma \rhd \mathit{false} : \mathbb{B}}\;(\text{T-False})$$

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x.M : \sigma \to \tau}\;(\text{T-Abs}) \qquad \frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\,N : \tau}\;(\text{T-App})$$

$$\frac{\Gamma \rhd M : \mathbb{B} \quad \Gamma \rhd P : \sigma \quad \Gamma \rhd Q : \sigma}{\Gamma \rhd \mathit{if}\ M\ \mathit{then}\ P\ \mathit{else}\ Q : \sigma}\;(\text{T-If})$$

Bidirectional Type-Checking - Main Idea

- split typing judgement

  $\Gamma \rhd M : \sigma$ "under assumptions in the context Γ, the expression M has type σ"

- into two new judgements:

  $\Gamma \rhd M \Rightarrow \sigma$ "under assumptions in Γ, the expression M synthesizes type σ"

  $\Gamma \rhd M \Leftarrow \sigma$ "under assumptions in Γ, the expression M checks against type σ"

We alternate between synthesizing types and checking expressions against types already known.

Bidirectional Type-Checking - Main Idea

$\Gamma \triangleright M \Rightarrow \sigma$ “under assumptions in Γ, the expression M synthesizes type σ”

$\Gamma \triangleright M \Leftarrow \sigma$ “under assumptions in Γ, the expression M checks against type σ”

Note difference in terms of code:

```
let rec  synthesize gamma m = (* Synthesizing *)
    ...
and check gamma m sigma = (* Checking *)
    ...
```

Next: revisit each typing rule to see if is should be formulated in checking mode or synthesizing mode

Syntax of our Language – $\mathrm{LC}^{b}_{\Leftrightarrow}$

$$
\begin{array}{rcll}
M, N, P, Q & ::= & x & \\
& \mid & \mathit{true} & \\
& \mid & \mathit{false} & \\
& \mid & \mathit{if}\ \ M\ \ \mathit{then}\ P\ \mathit{else}\ Q & \\
& \mid & \lambda x.M & \\
& \mid & M\,N & \\
& \mid & M : \sigma & \qquad\qquad \text{local type declaration}
\end{array}
$$

Note: no type declarations on bound variables

Bidirectional Type-Checking Rules (1/7)

$$\frac{\Gamma, x : \sigma \rhd M??}{\Gamma \rhd \lambda x.M??}\ (\textsc{T-Abs})$$

In order to apply the typing rule for abstraction “backwards” we have to be given the type of the bound variable and the codomain

$$\frac{\Gamma, x : \sigma \rhd M \Leftarrow \tau}{\Gamma \rhd \lambda x.M \Leftarrow \sigma \to \tau}\ (\textsc{BT-Abs})$$

Hence the rule for abstractions is in “checking mode”

Bidirectional Type-Checking Rules (2/7)

Once we reach a variable, we should already know its type

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x \Rightarrow \sigma}\,(\textsc{BT-Var})$$

Hence the rule for variables is in "synthesis mode"

Bidirectional Type-Checking Rules (3/7)

We infer the type of constants

$$\frac{}{\Gamma \rhd \mathit{true} \Rightarrow \mathbb{B}} \text{(BT-True)}$$

$$\frac{}{\Gamma \rhd \mathit{false} \Rightarrow \mathbb{B}} \text{(BT-False)}$$

Hence the rule for variables is in "synthesis mode"

Bidirectional Type-Checking Rules (4/7)

In the rule

$$\frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\,N : \tau}\ (\textsc{T-App})$$

- We should somehow know the type of the function in advance if we wish to apply it
- Moreover, σ does not occur in the type of the conclusion (i.e. τ); hence there is no way we could check M against $\sigma \to \tau$ since we don't have it.
- This leads to

$$\frac{\Gamma \rhd M \Rightarrow \sigma \to \tau \quad \Gamma \rhd N \Leftarrow \sigma}{\Gamma \rhd M\,N \Rightarrow \tau}\ (\textsc{BT-App})$$

Bidirectional Type-Checking Rules (5/7)

$$\frac{\Gamma \rhd M \Leftarrow \mathbb{B} \quad \Gamma \rhd P \Leftarrow \sigma \quad \Gamma \rhd Q \Leftarrow \sigma}{\Gamma \rhd \mathit{if}\ \ M\ \ \mathit{then}\ \ P\ \ \mathit{else}\ \ Q \Leftarrow \sigma}\,(\mathrm{BT\text{-}ITE})$$

- The type σ must be given to check that both branches have the same type
- Inferring the types of branches would require computing a least upper bound (which we avoid)

Bidirectional Type-Checking Rules (6/7)

$$\frac{\Gamma \triangleright M \Leftarrow \sigma}{\Gamma \triangleright M : \sigma \Rightarrow \sigma}\ (\textsc{BT-TDecl})$$

$$\frac{\Gamma \triangleright M \Rightarrow \tau \quad \tau = \sigma}{\Gamma \triangleright M \Leftarrow \sigma}\ (\textsc{BT-ChkInf})$$

Summary of Rules (7/7)

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x \Rightarrow \sigma}\ (\text{BT-Var})$$

$$\frac{\Gamma, x : \sigma \rhd M \Leftarrow \tau}{\Gamma \rhd \lambda x.M \Leftarrow \sigma \to \tau}\ (\text{BT-Abs}) \qquad \frac{\Gamma \rhd M \Rightarrow \sigma \to \tau \quad \Gamma \rhd N \Leftarrow \sigma}{\Gamma \rhd M\,N \Rightarrow \tau}\ (\text{BT-App})$$

$$\frac{}{\Gamma \rhd \mathit{true} \Rightarrow \mathbb{B}}\ (\text{BT-True}) \qquad \frac{}{\Gamma \rhd \mathit{false} \Rightarrow \mathbb{B}}\ (\text{BT-False})$$

$$\frac{\Gamma \rhd M \Leftarrow \mathbb{B} \quad \Gamma \rhd P \Leftarrow \sigma \quad \Gamma \rhd Q \Leftarrow \sigma}{\Gamma \rhd \mathit{if}\ M\ \mathit{then}\ P\ \mathit{else}\ Q \Leftarrow \sigma}\ (\text{BT-ITE})$$

$$\frac{\Gamma \rhd M \Leftarrow \sigma}{\Gamma \rhd M : \sigma \Rightarrow \sigma}\ (\text{BT-TDecl}) \qquad \frac{\Gamma \rhd M \Rightarrow \tau \quad \tau = \sigma}{\Gamma \rhd M \Leftarrow \sigma}\ (\text{BT-ChkInf})$$

Example 1

Note that abstractions can only be checked in "checker mode"

$$\emptyset \triangleright \lambda y.y + y \Leftarrow \mathbb{N} \rightarrow \mathbb{N}$$

This does not mean that all abstractions have to have type annotations, as we shall see in Example 3

Example 2

Can we derive the following judgement?

$$\emptyset \rhd (\lambda y.y + y)\,\underline{2} \Rightarrow \mathbb{N}$$

The programmer is required to add a type annotation

$$\emptyset \rhd ((\lambda y.y + y) : \mathbb{N} \to \mathbb{N})\,\underline{2} \Rightarrow \mathbb{N}$$

Example 3

$$\Gamma = \{\textit{twice} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}\}$$

Let us prove

$$\Gamma \rhd (\textit{twice}\,(\lambda y . y + y))\,\underline{2} \Rightarrow \mathbb{N}$$

Note that no type declarations at all are required in this program

Other Constructs

$$\frac{}{\Gamma \rhd \underline{n} \Rightarrow \mathbb{N}} \text{(BT-Const)}$$

$$\frac{op : \sigma_1 \to \sigma_2 \to \tau \quad \Gamma \rhd M \Leftarrow \sigma_1 \quad \Gamma \rhd N \Leftarrow \sigma_2}{\Gamma \rhd M\, op\, N \Rightarrow \tau} \text{(BT-BOp)}$$

$$\frac{\Gamma \rhd M \Leftarrow \sigma}{\Gamma \rhd inl(M) \Leftarrow \sigma + \tau} \text{(BT-Inl)} \qquad \frac{\Gamma \rhd M \Leftarrow \tau}{\Gamma \rhd inr(M) \Leftarrow \sigma + \tau} \text{(BT-Inr)}$$

$$\frac{\Gamma \rhd M \Rightarrow \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \rhd P \Leftarrow \sigma \quad \Gamma, y : \tau_2 \rhd Q \Leftarrow \sigma}{\Gamma \rhd case M of \{x \to P; y \to Q\} \Leftarrow \sigma} \text{(BT-Case)}$$

Bidirectional Type-Checking

Implementing Bidirectional Type-Checking

Towards an Implementation

- As it stands $\mathrm{LC}^{b}_{\Leftrightarrow}$ is almost syntax directed
- There is no restriction placed on M below:

$$\frac{\Gamma \triangleright M \Rightarrow \tau \quad \tau = \sigma}{\Gamma \triangleright M \Leftarrow \sigma}\,(\text{BT-}\textsc{ChkInf})$$

- However, since $\Gamma \triangleright M \Rightarrow \tau$ is not possible for abstractions, it only makes sense when M is any of the other constructs
- But for these other constructs there are no rules in checking mode

Towards an Implementation

An alternative approach, more theoretical in nature but perhaps less practical, is to add a new construct in the language to help the type system realize when it should resort to $(\textsc{BT-ChkInf})$

- Extended Syntax

$$\begin{array}{rcl} M, N, P, Q & ::= & \ldots \\ & | & \mathit{Inf}\ M \end{array}$$

- Modified typing rule

$$\frac{\Gamma \rhd M \Rightarrow \tau \quad \tau = \sigma}{\Gamma \rhd \mathit{Inf}\ M \Leftarrow \sigma}\ (\textsc{BT-ChkInf})$$

Implementation

- Details in assignment 1
- We will use the Bindlib library fo handling renaming, substitution, fresh name generation, etc.

Further Reading

- Tutorial by David Raymond Christiansen
  `davidchristiansen.dk/tutorials/bidirectional.pdf`
- Tutorial by Joshua Dunfield
  `research.cs.queensu.ca/~joshuad/bitype.pdf`
- Tutorial by Pfenning, Frank (2004). Lecture notes for 15-312: Foundations of Programming Languages.
  `www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf`
- Original paper by Pierce and Turner:

Pierce, Benjamin C. and Turner, David N. (1998). Local Type Inference. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL). ACM Transactions on Programming Languages and Systems (TOPLAS), 22(1), January 2000, pp. 1–44.