Bidirectional Typing Rules: A Tutorial

David Raymond Christiansen

17 October, 2013

A type system for a programming language is not the same thing as an efficient algorithm to check that a term inhabits some type or a means of synthesizing a type for some term. For example, the typing rules for ML-style let-polymorphism are quite different from the union-find algorithm that can efficiently infer types for practical programs. Indeed, we may not even be able to translate typing rules into an algorithm straightforwardly, by treating each rule as a recursive function where premises are called to check the conclusion.

Only systems that are *syntax-directed* can be straightforwardly translated in this way, and many actual programming languages are not described in a syntax-directed manner, as this is not always the easiest kind of system to understand. A *syntax-directed* system is one in which each typing judgment has a unique derivation, determined entirely by the syntactic structure of the term in question. Syntax-directed type systems allow the typing rules to be considered as pattern-match cases in a recursive function over the term being checked.

Bidirectional typing rules are one technique for making typing rules syntax-directed (and therefore providing an algorithm for typechecking) while maintaining a close correspondence to the original presentation. The technique has a number of advantages: the resulting system is quite understandable and produces good error messages, relatively few type annotations are required by programmers, and the technique seems to scale well when new features are added to the language. In particular, bidirectional systems support a style of type annotation where an entire term is annotated once at the top level, rather than sprinkling annotations throughout the code.

While bidirectional typing is quite straightforward, there is currently a lack of instructional materials explaining the concept in operational terms. The intended audience for this tutorial are people who have some familiarity with type systems and programming languages, but who haven't been able to build up the basic intuitions about bidirectional typing rules. Thus, Section 1

of this tutorial demonstrates these concepts in a simple setting, explaining where and why bidirectional rules are useful. Section 3 lists a selection of academic works and lecture notes that either use or develop bidirectional type systems, chosen for their value as tutorials.

1 Bidirectional Rules

There are two primary ways in which a typing rule might not be syntax-directed: it might be ambiguous in the conclusion, requiring an implementation to guess which rule to apply, or it might contain a premise that is not determined by the conclusion. This section presents a version of the simply-typed λ-calculus that does not have explicit type annotations on variable binders, which makes the system not syntax-directed. Then, it demonstrates how to convert that system to a bidirectional type system.

1.1 Simply-Typed Lambda-Calculus with Booleans

For a straightforward example, we translate the simply-typed λ-calculus with Booleans to the bidirectional presentation. First, the traditional presentation:

Terms:

t	::=	$x, y, z, \ldots$	*Variables*
	|	$t\ t$	*Application*
	|	$\lambda\, x\, .\, t$	*Abstraction*
	|	$\mathsf{true} \mid \mathsf{false}$	*Boolean constants*
	|	$\textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t$	*Conditional expressions*

Types:

τ	::=	Bool	*Boolean type*
	|	$\tau \to \tau$	*Function type*

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ (\text{T-Var})$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda\, x\, .\, t : \tau_1 \to \tau_2}\ (\text{T-Abs})$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2}\ (\text{T-App})$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ (\text{T-True})$$

$$\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}} \text{(T-False)}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Bool} \qquad \Gamma \vdash t_2 : \tau \qquad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 : \tau} \text{(T-If)}$$

If we attempt to read the above rules algorithmically, we consider the context Γ in each conclusion and the term being checked to be arguments to some function, and either failure or the type in question as a result. In pseudocode, for instance, T-If might look like this:

```
inferType ctx (If t1 t2 t3) =
  case (inferType ctx t1, inferType ctx t2, inferType ctx t3) of
    (Some BoolT, Some ty2, Some ty3) =>
      if ty2 = ty3
        then Some ty2
        else None
    _ => None
```

However, we run into problems when we need to translate T-Abs:

```
inferType ctx (Lam x t) =
  case (inferType ((x, ???)::ctx) t) of
    Some ty2 => Some (Fun ??? ty2)
    None     => None
```

What should we placed where we have written `???` as a placeholder? The rule T-Abs requires that we invent a type τ_1. This works well when we are dealing with humans and human creativity, but it is not particularly convenient for mechanization. A bidirectional type system is far from the only way to fix this problem. The simplest, perhaps, is to simply annotate every λ-expression with the type of its argument. However, this can be quite noisy in real programming languages. Additionally, one can find alternative algorithms, such as Damas and Milner's Algorithm W (Damas and Milner, 1982). Bidirectional type checking is useful when you want a straightforward means of checking something closer to a surface syntax of a real programming language.

1.2 Bidirectional STLC with Booleans

Bidirectional checking splits the above typing judgment $\Gamma \vdash t : \tau$ into two judgments: an *inference* judgment $\Gamma \vdash t \Rightarrow \tau$ (which should be read "t's type can be inferred to be τ in context Γ") and a *checking* judgment $\Gamma \vdash t \Leftarrow \tau$ (which should be read "t can be checked to have the given type

τ in the context Γ"). In other words, we get two functions that work together: an inference function, as before, and a checking function, which receives a goal type as an additional argument. The checking function calls the inference function when it reaches a term whose type can be inferred, comparing the inferred type with the type being checked against. The inference function calls the checking function when it encounters a type annotation.

The bidirectional presentation of the simply-typed λ-calculus with Booleans requires a modification to the term syntax above. We now need a means of representing type annotations. Otherwise, the language's syntax remains the same:

Terms:

t	::=	$x, y, z, \ldots$	*Variables*
	|	$t\ t$	*Application*
	|	$\lambda\, x\, .\, t$	*Abstraction*
	|	$\mathsf{true} \mid \mathsf{false}$	*Boolean constants*
	|	$\textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t$	*Conditional expressions*
	|	$t : \tau$	*Type annotation*

Types:

τ	::=	Bool	*Boolean type*
	|	$\tau \to \tau$	*Function type*

The typing rule for variables can be straightforwardly translated to a rule in inference mode, because the inference function receives the context as an argument:

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}\ (\text{BT-VAR})$$

This corresponds to the following case of the inference function:

```
inferType ctx (Var x) = lookup x ctx
```

where `lookup : Name -> Context -> Option Ty`.

Likewise, the Boolean constants can obviously be checked in inference mode:

$$\frac{}{\Gamma \vdash \mathsf{true} \Rightarrow \mathsf{Bool}}\ (\text{BT-TRUE})$$

$$\frac{}{\Gamma \vdash \mathsf{false} \Rightarrow \mathsf{Bool}}\ (\text{BT-FALSE})$$

Now that we have the trivial cases out of the way, let's turn our attention to the interaction between inference and checking modes. Obviously, if we can infer the type of a term, then we can check it against a type. Thus, we have the following rule:

$$\frac{\Gamma \vdash t \Rightarrow \tau}{\Gamma \vdash t \Leftarrow \tau}\ (\text{BT-CheckInfer})$$

This corresponds to code like:

```
checkType ctx t ty = case inferType ctx t of
                       Some ty' => if ty == ty'
                                   then Some ty
                                   else None
                       None     => None
```

As we saw with the Boolean constants, if we can determine a type uniquely by just inspecting a term, then we can infer that type. Obviously, we can determine the type of a type annotation by simply reading the annotation. Thus, typing annotations provide a means of switching from checking mode to inference mode, as follows:

$$\frac{\Gamma \vdash t \Leftarrow \tau}{\Gamma \vdash t : \tau \Rightarrow \tau}\ (\text{BT-Ann})$$

This corresponds to code like:

```
inferType ctx (Ann t ty) = checkType ctx t ty
```

Conditional expressions must be checked against some result type, as we cannot know the resulting type by examining the conditional. Because we are in checking mode, we have received the type of the expression as an argument. Since this is known, we use checking mode for all three premises, to reduce the potential for a required type annotation:

$$\frac{\Gamma \vdash t_1 \Leftarrow \mathsf{Bool} \qquad \Gamma \vdash t_2 \Leftarrow \tau \qquad \Gamma \vdash t_3 \Leftarrow \tau}{\Gamma \vdash \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \Leftarrow \tau}\ (\text{T-If})$$

This corresponds to the following case:

```
checkType ctx (If t1 t2 t3) ty =
  case (checkType ctx t1 BoolTy,
        checkType ctx t2 ty,
        checkType ctx t3 ty) of
    (Some BoolTy, Some ty2, Some ty3) => Some ty
    _ => None
```

A consequence of this is that conditional expressions will often require type annotations.

Just as the variables bound in function abstractions would require annotations in the syntax-directed unidirectional simply-typed λ-calculus, functions must be checked against some type. This prevents us from having to guess the type τ_1 of the argument:

$$\frac{\Gamma, x : \tau_1 \vdash t \Leftarrow \tau_2}{\Gamma \vdash \lambda\, x\, .\, t \Leftarrow \tau_1 \to \tau_2}\ (\text{BT-Abs})$$

This allows a style reminiscent of programs in a language like Haskell, where type annotations are written for a whole function at a time, rather than interspersed across the variable bindings. For example, Boolean negation can be written as the function:

$$(\lambda\, b\, .\, \textbf{if}\ b\ \textbf{then}\ \mathsf{false}\ \textbf{else}\ \mathsf{true}) : \mathsf{Bool} \to \mathsf{Bool}$$

which corresponds closely to the Haskell:

```
not :: Bool -> Bool
not b = if b then False else True
```

The single annotation of the entire term puts the system into checking mode, after which the conditional does not need to be annotated. This is witnessed by the following derivation:

$$\dfrac{\dfrac{\dfrac{(b : \mathsf{Bool}) \in \Gamma, b : \mathsf{Bool}}{\Gamma, b : \mathsf{Bool} \vdash b \Leftarrow \mathsf{Bool}} \quad \dfrac{}{\Gamma, b : \mathsf{Bool} \vdash \mathsf{false} \Rightarrow \mathsf{Bool}}}{\Gamma, b : \mathsf{Bool} \vdash \mathsf{false} \Leftarrow \mathsf{Bool}} \quad \dfrac{}{\Gamma, b : \mathsf{Bool} \vdash \mathsf{true} \Rightarrow \mathsf{Bool}}}{\Gamma, b : \mathsf{Bool} \vdash \mathsf{true} \Leftarrow \mathsf{Bool}}}{\dfrac{\Gamma, b : \mathsf{Bool} \vdash \textbf{if}\ b\ \textbf{then}\ \mathsf{false}\ \textbf{else}\ \mathsf{true} \Leftarrow \mathsf{Bool}}{\dfrac{\Gamma \vdash \lambda\, b\, .\, \textbf{if}\ b\ \textbf{then}\ \mathsf{false}\ \textbf{else}\ \mathsf{true} \Leftarrow \mathsf{Bool} \to \mathsf{Bool}}{\Gamma \vdash (\lambda\, b\, .\, \textbf{if}\ b\ \textbf{then}\ \mathsf{false}\ \textbf{else}\ \mathsf{true}) : \mathsf{Bool} \to \mathsf{Bool} \Rightarrow \mathsf{Bool} \to \mathsf{Bool}}}$$

The final rule that remains to be translated to a bidirectional style is the rule for function application. The basic intuition to have is that we must somehow know the type of a function ahead of time, whether it be through an explicit annotation or by having it the context. In real programming languages, the types of library functions will be available in the context, so this requirement may not be particularly onerous.

Because we want to enforce that the type of the function expression is somehow known in advance, we check it in inference mode. The result of this check will be an arrow type, which gives us a type to check the argument against. Once we have this argument type, we can use it to check that the argument (t_2 in the rule below) has the correct type. And, because τ_2 was also recovered from the function, the conclusion judgment can be in inference mode.

$$\frac{\Gamma \vdash t_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 \Leftarrow \tau_1}{\Gamma \vdash t_1\, t_2 \Rightarrow \tau_2}\ (\text{BT-App})$$

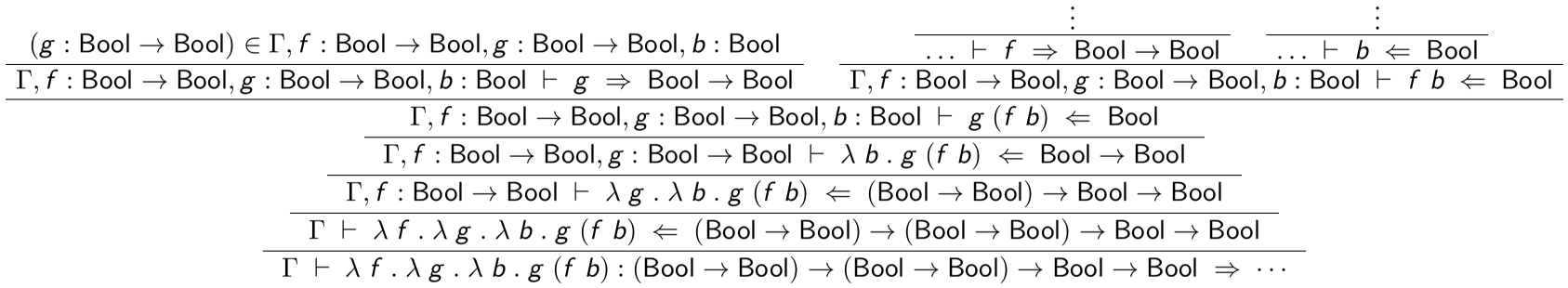

Figure 1: Bidirectional typing derivation for Boolean function composition

To demonstrate how this works in concert with the rule for application, Figure 1 demonstrates a typing derivation for composition of Boolean functions:

$$\lambda\, f\, .\, \lambda\, g\, .\, \lambda\, b\, .\, g\ (f\ b) : (\mathsf{Bool} \to \mathsf{Bool}) \to (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool}$$

Remember that the derivation, like the bidirectional typing rules, should be read bottom-to-top and left-to-right. This means that the results of previous "function calls" are available to later "calls", which is why we have the value to fill out the metavariable with when we check the type of a function argument against the domain of the inferred type of the function.

2 Discussion

While Section 1 presented some of the advantages of a bidirectional type system, there are certainly trade-offs. Perhaps the most serious is that variable substitution no longer works for typing derivations. In particular, the rules provided in Section 1.1 allow a variable in a derivation to be replaced by the derivation for a term of the same type. This is not the case in the bidirectional system of Section 1.2, because variables are checked using inference mode, while many interesting constructs for the language must be checked in checking mode.

Note that, while the introduction to this tutorial initially motivated bidirectional type systems as a means of making a system syntax-directed, it is not the case that all bidirectional type systems are syntax-directed. It is easy enough to create a bidirectional system with two separate inference rules for the same syntactic category of term, after all! Instead, bidirectional typing rules should be seen as a way to make the system *more* syntax-directed that can possibly take it all the way.

The presentation of the λ-calculus in this tutorial skipped evaluation semantics. Presumably, the type annotations in the bidirectional system would simply have no effect — that is, there might be a small-step rule such as:

$$\frac{}{t : \tau \ \longrightarrow\ t}$$

or a big-step rule such as:

$$\frac{t \Downarrow t'}{t : \tau \ \Downarrow\ t'}$$

Sometimes, explicit type annotations will need to be within a term rather than at the top level. In particular, explicit function abstractions that are being applied directly may require a type annotation. For example, the following term:

$$(\lambda\ b\ .\ \textbf{if}\ b\ \textbf{then}\ \mathsf{false}\ \textbf{else}\ \mathsf{true})\ \mathsf{true} : \mathsf{Bool}$$

cannot be type checked. When we begin to construct the derivation, we reach a point where the type of an un-annotated λ-abstraction must be inferred:

$$\dfrac{\dfrac{\Gamma\ \vdash\ \lambda\ b\ .\ \textbf{if}\ b\ \textbf{then}\ \mathsf{false}\ \textbf{else}\ \mathsf{true}\ \Rightarrow\ ???\qquad ???}{\Gamma\ \vdash\ (\lambda\ b\ .\ \textbf{if}\ b\ \textbf{then}\ \mathsf{false}\ \textbf{else}\ \mathsf{true})\ \mathsf{true}\ \Leftarrow\ \mathsf{Bool}}}{\Gamma\ \vdash\ (\lambda\ b\ .\ \textbf{if}\ b\ \textbf{then}\ \mathsf{false}\ \textbf{else}\ \mathsf{true})\ \mathsf{true} : \mathsf{Bool}\ \Rightarrow\ \mathsf{Bool}}$$

However, we have no rule to perform such an inference, as BT-Abs is a checking rule. We can solve this by providing specialized inference rules for certain limited forms of λ-abstractions and conditionals, and we may even be successful in the simply-typed context. While this is useful, it will not scale to more interesting type systems for which type inference is undecidable. It may, however, be possible to make a quite useful system in practice, where users only need to annotate complicated code.

The algorithms that naturally result from a bidirectional type systems are known for producing good error messages. As the type checker proceeds, it carries with it information about the type that is expected for most terms, and it can straightforwardly report where in a term the error occurs. Compared to systems that build up collections of type constraints to be solved later, it can be much more straightforward to locate the source of an error.

In a real programming language, it might be preferable to use a bidirectional system to provide a convenient surface syntax with minimal top-level annotations and good error messages, but to have the type checker produce a version of the term with full type annotations on every binding for ease of processing later.

3 Further Reading

This section does not intend to be a full survey of the academic literature on bidirectional type checking. For that, see the introduction to Dunfield and Krishnaswami (2013). Instead, this section lists particularly accessible presentations.

Bidirectional type checking was introduced to the academic literature by Pierce and Turner (1998). However, they cite the idea as "folklore" and do

not take credit for coming up with the basic idea. At the time, the technique was commonly combined with ML-style type inference. They demonstrate its application to subtyping and bounded quantification.

A set of lecture notes from Dunfield (2012) cover more ground more quickly than this tutorial, and would be a good next resource to cement the idea. Another set of lecture notes from Pfenning (2004) do a good job at building intuitions regarding the view of typing rules as a program, with input and output to and from rules. Löh, McBride, and Swierstra (2010) use bidirectional type checking in a very accessible tutorial implementation of a dependently-typed language. Remember to look at the code on the Web page for the paper. Additionally, Weirich's lecture notes and video lectures from the Oregon Programming Languages Summer School (2013) are a good resource if you learn better from videos and exercises.

Acknowledgments I would like to thank Joshua Dunfield for his in-depth comments on an earlier draft of this tutorial. Additionally, I would like to thank Anthony Cowley and Morten Fangel for catching minor mistakes.

References

Damas, Luis and Milner, Robin (1982). "Principal type-schemes for functional programs". In: *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*. Albuquerque, New Mexico: ACM, pp. 207–212.

Dunfield, Joshua (2012). Slightly revised lecture notes from McGill University COMP 302. URL: `http://www.mpi-sws.org/~joshua/bitype.pdf`.

Dunfield, Joshua and Krishnaswami, Neelakantan R. (2013). "Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism". In: *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. `arXiv:1306.6032[cs.PL]`.

Löh, Andres, McBride, Conor, and Swierstra, Wouter (2010). "A Tutorial Implementation of a Dependently Typed Lambda Calculus". In: *Fundamenta Informaticae* 102.2, pp. 177–207.

Pfenning, Frank (2004). Lecture notes for 15-312: Foundations of Programming Languages. URL: `http://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf`.

Pierce, Benjamin C. and Turner, David N. (1998). "Local Type Inference". In: *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California*. Full version in *ACM Trans-*

actions on Programming Languages and Systems (TOPLAS), 22(1), January 2000, pp. 1–44.

Weirich, Stephanie (2013). Lecture notes for Oregon Programming Languages Summer School. URL: `http://www.cs.uoregon.edu/research/summerschool/summer13/lectures/weirich_main.pdf`.