Chapter 1

Typechecking

So far, our interpretation of the typing judgment $\Gamma \vdash e : A$ has been declarative in the sense that given a triple of $\Gamma$, $e$, and $A$, the judgment answers either "yes" (meaning that $e$ has type $A$ under $\Gamma$) or "no" (meaning that $e$ does not have type $A$ under $\Gamma$). While the declarative interpretation is enough for proving type safety of the simply typed $\lambda$-calculus, it does not lend itself well to an implementation of the type system, which takes a pair of $\Gamma$ and $e$ and decides a type for $e$ under $\Gamma$, if one exists. That is, an implementation of the type system requires not a declarative interpretation but an algorithmic interpretation of the typing judgment $\Gamma \vdash e : A$ such that given $\Gamma$ and $e$ as input, the interpretation produces $A$ as output.

This chapter discusses two implementations of the type system. The first employs an algorithmic interpretation of the typing judgment, and is purely synthetic in that given $\Gamma$ and $e$, it synthesizes a type $A$ such that $\Gamma \vdash e : A$. The second mixes an algorithmic interpretation with a declarative interpretation, and achieves what is called bidirectional typechecking. It is both synthetic and analytic in that depending on the form of a given expression $e$, it requires either only $\Gamma$ to synthesize a type $A$ such that $\Gamma \vdash e : A$, or both $\Gamma$ and $A$ to confirm that $\Gamma \vdash e : A$ holds.

1.1 Purely synthetic typechecking

Let us consider a direct implementation of the type system, or equivalently the judgment $\Gamma \vdash e : A$. We introduce a function $\text{typing}$ with the following invariant:

$$\text{typing}(\Gamma, e, A) = \begin{cases} \text{okay} & \text{if } \Gamma \vdash e : A \text{ holds.} \\ \text{fail} & \text{if } \Gamma \vdash e : A \text{ does not hold.} \end{cases}$$

Since $\Gamma$, $e$, and $A$ are all given as input, we only have to translate each typing rule in the direction from the conclusion to the premise(s) (i.e., bottom-up), as illustrated in the pseudocode below:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}, \quad \text{Var} \quad \leftrightarrow \quad \text{typing}(\Gamma, x, A) = \begin{cases} \text{okay} & \text{if } x : A \in \Gamma \text{ then okay else fail} \\ \text{fail} & \text{if } \Gamma \vdash e : A \text{ does not hold.} \end{cases}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A. e : A \rightarrow B}, \quad \rightarrow l \quad \leftrightarrow \quad \text{typing}(\Gamma, \lambda x : A. e, A \rightarrow B) = \text{typing}(\Gamma', e, B) \text{ where } \Gamma' = \Gamma, x : A$$

It is not obvious, however, how to translate the rule $\rightarrow E$ because both premises require a type $A$ which does not appear in the conclusion:

$$\frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash e \ e' : B}, \quad \rightarrow E \quad \leftrightarrow \quad \text{typing}(\Gamma, e \ e', B) = \begin{cases} \text{okay} & \text{if typing } (\Gamma, e, A \rightarrow B) = \text{okay} \\ \text{and also typing } (\Gamma, e', A) = \text{okay} \text{ then okay else fail} \\ \text{where } A = ? \end{cases}$$

Therefore, in order to return okay, typing $(\Gamma, e \ e', B)$ must “guess” a type $A$ such that both typing $(\Gamma, e, A \rightarrow B)$ and typing $(\Gamma, e', A)$ return okay. The problem of guessing such a type $A$ from $e$ and $e'$ involves the problem of deciding the type of a given expression (e.g., deciding type $A$ of expression $e'$). Thus we need to
be able to decide the type of a given expression anyway, and are led to interpret the typing judgment \( \Gamma \vdash e : A \) algorithmically so that given \( \Gamma \) and \( e \) as input, an algorithmic interpretation of the judgment produces \( A \) as output.

We introduce a new judgment \( \Gamma \vdash e \triangleright A \), called an algorithmic typing judgment, to express the algorithmic interpretation of the typing judgment \( \Gamma \vdash e : A \):

\[
\Gamma \vdash e \triangleright A \iff \text{under typing context } \Gamma, \text{ the type of expression } e \text{ is inferred as } A
\]

That is, an algorithmic typing judgment \( \Gamma \vdash e \triangleright A \) synthesizes type \( A \) (output) for expression \( e \) (input). Algorithmic typing rules (i.e., inference rules for algorithmic typing judgments) are as given follows:

\[
\Gamma \vdash x : A \in \Gamma \quad \text{Var}_a \quad \Gamma, x : A \vdash B \quad \rightarrow_a \quad \Gamma \vdash e \triangleright B \quad \Gamma \vdash e' \triangleright B \\
\rightarrow a \quad \Gamma \vdash e \triangleright A \rightarrow B \quad \Gamma \vdash e' \triangleright C \quad A = C \quad \rightarrow a
\]

Note that in the rule \( \rightarrow a \), we may not write the second premise as \( \Gamma \vdash e' \triangleright A \) (and remove the third premise) because type \( C \) to be inferred from \( \Gamma \) and \( e' \) is unknown in general and must be explicitly compared with type \( A \) as is done in the third premise. (Similarly for types \( A_1 \) and \( A_2 \) in the rule \( \text{if}_a \).) A typechecking algorithm based on the algorithmic typing judgment \( \Gamma \vdash e \triangleright A \) is said to be purely synthetic.

The equivalence between two judgments \( \Gamma \vdash e \triangleright A \) and \( \Gamma \vdash e : A \) is stated in Theorem 1.3, whose proof uses Lemmas 1.1 and 1.2. Lemma 1.1 proves soundness of \( \Gamma \vdash e \triangleright A \) in the sense that if an algorithmic typing judgment infers type \( A \) for expression \( e \) under typing context \( \Gamma \), then \( A \) is indeed the type for \( e \) under \( \Gamma \). In other words, if an algorithmic typing judgment gives an answer, it always gives a correct answer and is thus “sound.” Lemma 1.2 proves completeness of \( \Gamma \vdash e \triangleright A \) in the sense that for any well-typed expression \( e \) under typing context \( \Gamma \), there exists an algorithmic typing judgment inferring its type. In other words, an algorithmic typing judgment covers all possible cases of well-typed expressions and is thus “complete.”

**Lemma 1.1 (soundness).** If \( \Gamma \vdash e \triangleright A \), then \( \Gamma \vdash e : A \).

**Proof.** By rule induction on the judgment \( \Gamma \vdash e \triangleright A \). \[ \square \]

**Lemma 1.2 (completeness).** If \( \Gamma \vdash e : A \), then \( \Gamma \vdash e \triangleright A \).

**Proof.** By rule induction on the judgment \( \Gamma \vdash e : A \). \[ \square \]

**Theorem 1.3.** \( \Gamma \vdash e : A \) if and only if \( \Gamma \vdash e \triangleright A \).

**Proof.** Follows from Lemmas 1.1 and 1.2. \[ \square \]

### 1.2 Bidirectional typechecking

In the simply typed \( \lambda \)-calculus, every variable in a \( \lambda \)-abstraction is annotated with its type (e.g., \( \lambda x : A. e \)). While it is always good to know the type of a variable for the purpose of typechecking, a typechecking algorithm may not need the type annotation of every variable which sometimes reduces code readability. As an example, consider the following expression which has type bool:

\[
(\lambda f : \text{bool} \to \text{bool}. f \text{ true}) \lambda x : \text{bool}. x
\]

The type of the first subexpression \( \lambda f : \text{bool} \to \text{bool}. f \text{ true} \) is \( \text{bool} \to \text{bool} \), and the whole expression typechecks only if the second subexpression \( \lambda x : \text{bool}. x \) has type \( \text{bool} \to \text{bool} \) (according to the rule \( \rightarrow E \)). Then the type annotation for variable \( x \) becomes redundant because it must have type \( \text{bool} \) anyway if \( \lambda x : \text{bool}. x \) is to have type \( \text{bool} \to \text{bool} \). This example illustrates that not every variable in a well-typed expression needs to be annotated with its type.
A bidirectional typechecking algorithm takes a different approach by allowing $\lambda$-abstractions with no type annotation (i.e., $\lambda x. e$ as in the untyped $\lambda$-calculus), but also requiring certain expressions to be explicitly annotated with their types. Thus bidirectional typechecking assumes a modified definition of abstract syntax:

$$
\text{expression } e ::= x | \lambda x. e | e \; e | \text{true} | \text{false} \; | \text{if } e \; \text{then } e \; \text{else } e \; | (e : A)
$$

A $\lambda$-abstraction $\lambda x. e$ does not annotate its formal argument with a type. (It is okay to permit $\lambda x : A. e$ in addition to $\lambda x. e$, but it does not expand the set of well-typed expressions under bidirectional typechecking.) $(e : A)$ explicitly annotates expression $e$ with type $A$, and plays the role of variable $x$ bound in a $\lambda$-abstraction $\lambda x : A. e$. Specifically it is $(e : A)$ that feeds type information into a bidirectional typechecking algorithm whereas it is $\lambda x : A. e$ that feeds type information into an ordinary typechecking algorithm.

A bidirectional typechecking algorithm proceeds by alternating between an analysis phase, in which it “analyzes” a given expression to verify that it indeed has a given type, and a synthesis phase, in which it “synthesizes” the type of a given expression. We use two new judgments for the two phases of bidirectional typechecking:

- $\Gamma \vdash e \uparrow A$ means that we are checking expression $e$ against type $A$ under typing context $\Gamma$. That is, $\Gamma, e, A$ are all given and we are checking if $\Gamma \vdash e : A$ holds. $\Gamma \vdash e \uparrow A$ corresponds to a declarative interpretation of the typing judgment $\Gamma \vdash e : A$.

- $\Gamma \vdash e \downarrow A$ means that we have synthesized type $A$ from expression $e$ under typing context $\Gamma$. That is, only $\Gamma$ and $e$ are given and we have synthesized type $A$ such that $\Gamma \vdash e : A$ holds. $\Gamma \vdash e \downarrow A$ corresponds to an algorithmic interpretation of the typing judgment $\Gamma \vdash e : A$, and is stronger (i.e., more difficult to prove) than $\Gamma \vdash e \uparrow A$.

Now we have to decide which of $\Gamma \vdash e \uparrow A$ and $\Gamma \vdash e \downarrow A$ is applicable to a given expression $e$. Let us consider a $\lambda$-abstraction $\lambda x. e$ first:

$$
\Gamma \vdash \lambda x. e \downarrow A \rightarrow B \quad \text{or} \quad \Gamma \vdash \lambda x. e \uparrow A \rightarrow B
$$

Intuitively we cannot hope to synthesize type $A \rightarrow B$ from $\lambda x. e$ because the type of $x$ is unknown in general. For example, $e$ may not use $x$ at all, in which case it is literally impossible to infer the type of $x$! Therefore we have to check $\lambda x. e$ against a type $A \rightarrow B$ to be given in advance:

$$
\Gamma, x : A \vdash e \uparrow B
\Gamma \vdash \lambda x. e \uparrow A \rightarrow B \quad \text{without a rule.}
$$

Next let us consider an application $e \; e'$:

$$
\Gamma \vdash e \downarrow B
\Gamma \vdash e \uparrow A \rightarrow B \quad \text{without a rule.}
$$

Intuitively it is pointless to check $e \; e'$ against type $B$, since we have to synthesize type $A \rightarrow B$ for $e$ anyway. With type $A \rightarrow B$ for $e$, then, we automatically synthesize type $B$ for $e \; e'$ as well, and the problem of checking $e \; e'$ against type $B$ becomes obsolete because it is easier than the problem of synthesizing type $B$ for $e \; e'$. Therefore we synthesize type $B$ from $e \; e'$ by first synthesizing type $A \rightarrow B$ from $e$ and then verifying that $e$ has type $A$:

$$
\Gamma \vdash e \downarrow A \rightarrow B
\Gamma \vdash e \uparrow A \rightarrow B
\Gamma \vdash e \uparrow A
$$

For a variable, we can always synthesize its type by looking up a typing context:

$$
\Gamma \vdash x \downarrow A
\Gamma \vdash x \uparrow A
$$

Then how can we relate the two judgments $\Gamma \vdash e \uparrow A$ and $\Gamma \vdash e \downarrow A$? Since $\Gamma \vdash e \downarrow A$ is stronger than $\Gamma \vdash e \uparrow A$, the following rule makes sense regardless of the form of expression $e$:

$$
\Gamma \vdash e \downarrow A
\Gamma \vdash e \uparrow A
$$

May 15, 2007
The opposite direction does not make sense, but by annotating $e$ with its intended type $A$, we can relate the two judgments in the opposite direction:

$$
\Gamma \vdash e \uparrow A \\
\Gamma \vdash (e : A) \downarrow \uparrow_{\text{b}} A
$$

The rule $\uparrow \downarrow_{\text{b}}$ says that if expression $e$ is annotated with type $A$, we may take $A$ as the type of $e$ without having to guess, or "synthesize," it, but only after verifying that $e$ indeed has type $A$.

Now we can classify expressions into two kinds: intro(duction) expressions $I$ and elim(ination) expressions $E$. We always check an intro expression $I$ against some type $A$; hence $\Gamma \vdash I \uparrow A$ makes sense, but $\Gamma \vdash I \downarrow A$ is not allowed. For an elim expression $E$, we can either try to synthesize its type $A$ or check it against some type $A$; hence both $\Gamma \vdash E \downarrow A$ and $\Gamma \vdash E \uparrow A$ make sense. The mutual definition of intro and elim expressions is specified by the rules for bidirectional typechecking:

$$
\text{intro expression} \quad I ::= \lambda x. I \mid E \\
\text{elim expression} \quad E ::= x \mid E I \mid (I : A)
$$

As you might have guessed, an expression is an intro expression if its corresponding typing rule is an introduction rule. For example, $\lambda x. e$ is an intro expression because its corresponding typing rule is the $\to$ introduction rule $\to_\text{I}$. Likewise an expression is an elim expression if its corresponding typing rule is an elimination rule. For example, $e e'$ is an elim expression because its corresponding typing rule is the $\to$ elimination rule $\to_\text{E}$, although it requires further consideration to see why $e$ is an elim expression and $e'$ is an intro expression.

For your reference, we give the complete definition of intro and elim expressions by including remaining constructs of the simply typed $\lambda$-calculus. As in $\lambda$-abstractions, we do not need type annotations in left injections, right injections, abort expression, and the fixed point construct. We use a case expression as an intro expression instead of an elim expression. We use an abort expression as an intro expression because it is a special case of a case expression. Figure 1.1 shows the definition of intro and elim expressions as well as all typing rules for bidirectional typechecking.

### 1.3 Exercises

**Exercise 1.4.** Give algorithmic typing rules for the extended simply typed $\lambda$-calculus in Figure ??.

**Exercise 1.5.** Give typing rules for true, false, and if $e$ then $e_1$ else $e_2$ under bidirectional typechecking.

**Exercise 1.6.** $(\lambda x. x) ()$ has type unit. This expression, however, does not typecheck against unit under bidirectional typechecking. Write as much of a derivation $\cdot \vdash (\lambda x. x) () \uparrow \text{unit}$ as you can, and indicate with an asterisk (*) where the derivation gets stuck.

**Exercise 1.7.** Annotate some intro expression in $(\lambda x. x) ()$ with a type (i.e., convert an intro expression $I$ into an elim expression $(I : A)$), and typecheck the whole expression using bidirectional typechecking.
intro expression \[ I ::= \lambda x. I \rightarrow I b \mid (I, I) \mid \text{inl } I + I Lb \mid \text{inr } I + I Rb \mid \text{case } E \text{ of } \text{inl } x. I \mid \text{inr } x. I + E b \mid () \mid \text{Unit} b \mid \text{abort } E \mid \text{fix } x. I \mid E \Rightarrow b \] 

elim expression \[ E ::= x \mid E I \mid \text{fst } E \times E 1b \mid \text{snd } E \times E 2b \mid (I : A) \Rightarrow b \]

\[ \frac{x : A \in \Gamma}{\Gamma \vdash x \downarrow A} \quad \frac{\text{Var}_b}{\Gamma \vdash \lambda x. I \uparrow A \rightarrow B} \quad \frac{\text{Var}_b}{\Gamma \vdash E \downarrow A \rightarrow B} \quad \frac{\text{Var}_b}{\Gamma \vdash I \uparrow A} \quad \frac{\text{Var}_b}{\Gamma \vdash E \downarrow A \times A_2} \quad \frac{\text{Var}_b}{\Gamma \vdash \text{fst } E \downarrow A_1} \quad \frac{\text{Var}_b}{\Gamma \vdash E \downarrow A_1 \times A_2} \quad \frac{\text{Var}_b}{\Gamma \vdash \text{snd } E \downarrow A_2} \quad \frac{\text{Var}_b}{\Gamma \vdash I \uparrow A_1} \quad \frac{\text{Var}_b}{\Gamma \vdash I \uparrow A_2} \quad \frac{\text{Var}_b}{\Gamma \vdash \text{inl } I \uparrow A_1 + A_2} \quad \frac{\text{Var}_b}{\Gamma \vdash \text{inr } I \uparrow A_1 + A_2} \quad \frac{\text{Var}_b}{\Gamma \vdash \text{case } E \text{ of } \text{inl } x_1. I_1 \mid \text{inr } x_2. I_2 \uparrow C} + E_b \quad \frac{\text{Var}_b}{\Gamma \vdash () \uparrow \text{unit}} \quad \frac{\text{Var}_b}{\Gamma \vdash E \downarrow \text{void}} \quad \frac{\text{Var}_b}{\Gamma \vdash \text{abort } E \uparrow C} \quad \frac{\text{Var}_b}{\Gamma \vdash \text{fix } x. I \uparrow A} \quad \frac{\text{Var}_b}{\Gamma \vdash E \downarrow A} \quad \frac{\text{Var}_b}{\Gamma \vdash I \uparrow A} \quad \frac{\text{Var}_b}{\Gamma \vdash (I : A) \uparrow \psi_b} \]

Figure 1.1: Definition of intro and elim expressions with their typing rules