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 Γ, e, and A, the judgment answers either "yes" (meaning that e has type A under Γ) or "no" (meaning that e does not have type A under Γ). While the declarative interpretation is enough for proving type safety of the simply typed λ-calculus, it does not lend itself well to an implementation of the type system, which takes a pair of Γ and e and decides a type for e under Γ, 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 Γ 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 Γ 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 Γ to synthesize a type A such that $\Gamma \vdash e : A$, or both Γ 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 typing with the following invariant:

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

Since Γ, 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:

$$\begin{array}{rcl} \dfrac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \mathsf{Var} & \Leftrightarrow & \begin{array}{l} \mathsf{typing}\ (\Gamma, x, A) = \\ \quad \mathsf{if}\ x : A \in \Gamma\ \mathsf{then}\ \mathsf{okay}\ \mathsf{else}\ \mathsf{fail} \end{array} \\ \dfrac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x{:}A.\, e : A \to B}\ {\to}\mathsf{I} & \Leftrightarrow & \begin{array}{l} \mathsf{typing}\ (\Gamma, \lambda x{:}A.\, e, A \to B) = \\ \quad \mathsf{typing}\ (\Gamma', e, B)\ \mathsf{where}\ \Gamma' = \Gamma, x : A \end{array} \end{array}$$

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

$$\dfrac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B}\ {\to}\mathsf{E} \quad \Leftrightarrow \quad \begin{array}{l} \mathsf{typing}\ (\Gamma, e\ e', B) = \\ \quad \mathsf{if}\ \mathsf{typing}\ (\Gamma, e, A \to B) = \mathsf{okay} \\ \quad\quad \mathsf{andalso}\ \mathsf{typing}\ (\Gamma, e', A) = \mathsf{okay} \\ \quad \mathsf{then}\ \mathsf{okay}\ \mathsf{else}\ \mathsf{fail} \\ \quad \mathsf{where}\ A = ? \end{array}$$

Therefore, in order to return okay, $\mathsf{typing}\ (\Gamma, e\ e', B)$ must "guess" a type A such that both $\mathsf{typing}\ (\Gamma, e, A \to B)$ and $\mathsf{typing}\ (\Gamma, e', A)$ 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 the 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 Γ and e as input, an algorithmic interpretation of the judgment produces A as output.

We introduce a new judgment $\Gamma \vdash e \rhd A$, called an *algorithmic typing judgment*, to express the algorithmic interpretation of the typing judgment $\Gamma \vdash e : A$:

$$\Gamma \vdash e \rhd A \quad \Leftrightarrow \quad \textit{under typing context } \Gamma \textit{, the type of expression } e \textit{ is inferred as } A$$

That is, an algorithmic typing judgment $\Gamma \vdash e \rhd A$ synthesizes type A (output) for expression e (input) under typing context Γ (input). The inference rules for algorithmic typing judgments are as follows:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \rhd A}\ \mathsf{Var_a} \qquad \frac{\Gamma, x : A \vdash e \rhd B}{\Gamma \vdash \lambda x\!:\!A.\, e \rhd A \to B}\ {\to}\mathsf{I_a}$$

$$\frac{\Gamma \vdash e \rhd A \to B \quad \Gamma \vdash e' \rhd C \quad A = C}{\Gamma \vdash e\ e' \rhd B}\ {\to}\mathsf{E_a}$$

$$\frac{}{\Gamma \vdash \mathsf{true} \rhd \mathsf{bool}}\ \mathsf{True_a} \qquad \frac{}{\Gamma \vdash \mathsf{false} \rhd \mathsf{bool}}\ \mathsf{False_a}$$

$$\frac{\Gamma \vdash e \rhd \mathsf{bool} \quad \Gamma \vdash e_1 \rhd A_1 \quad \Gamma \vdash e_2 \rhd A_2 \quad A_1 = A_2}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rhd A_1}\ \mathsf{If_a}$$

Note that in the rule ${\to}\mathsf{E_a}$, we may not write the second premise as $\Gamma \vdash e' \rhd A$ (and remove the third premise) because type C to be inferred from Γ 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 $\mathsf{If_a}$.) A typechecking algorithm based on the algorithmic typing judgment $\Gamma \vdash e \rhd A$ is said to be purely synthetic.

The equivalence between two judgments $\Gamma \vdash e \rhd A$ 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 $\Gamma \vdash e \rhd A$ in the sense that if an algorithmic typing judgment infers type A for expression e under typing context Γ, then A is indeed the type for e under Γ. 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 $\Gamma \vdash e \rhd A$ in the sense that for any well-typed expression e under typing context Γ, 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* $\Gamma \vdash e \rhd A$*, then* $\Gamma \vdash e : A$*.*

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

Theorem 1.3. $\Gamma \vdash e : A$ *if and only if* $\Gamma \vdash e \rhd A$*.*

Proof. Follows from Lemmas 1.1 and 1.2. □

1.2 Bidirectional typechecking

In the simply typed λ-calculus, every variable in a λ-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:

$$(\lambda f\!:\!\mathsf{bool} \to \mathsf{bool}.\, f\ \mathsf{true})\ \lambda x\!:\!\mathsf{bool}.\, x$$

The type of the first subexpression $\lambda f\!:\!\mathsf{bool} \to \mathsf{bool}.\, f\ \mathsf{true}$ is $(\mathsf{bool} \to \mathsf{bool}) \to \mathsf{bool}$, so the whole expression typechecks only if the second subexpression $\lambda x\!:\!\mathsf{bool}.\, x$ has type $\mathsf{bool} \to \mathsf{bool}$ (according to the rule ${\to}\mathsf{E}$).

Then the type annotation for variable x becomes redundant because it must have type bool anyway if $\lambda x\!:\!\mathsf{bool}.\, x$ is to have type $\mathsf{bool} \rightarrow \mathsf{bool}$. 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 λ-abstractions with no type annotation (*i.e.*, $\lambda x.\, e$ as in the untyped λ-calculus), but also requiring certain expressions to be explicitly annotated with their types. Thus bidirectional typechecking assumes a modified definition of abstract syntax:

$$\begin{array}{lrcl} \mathsf{expression} & e & ::= & x \mid \lambda x.\, e \mid e\; e \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\; e\; \mathsf{then}\; e\; \mathsf{else}\; e \mid (e : A) \end{array}$$

A λ-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 λ-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 Γ. That is, Γ, e, and 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 Γ. That is, only Γ 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 λ-abstraction $\lambda x.\, e$ first:

$$\frac{\cdots}{\Gamma \vdash \lambda x.\, e \Downarrow A \rightarrow B} \rightarrow\!\mathsf{I_b} \qquad or \qquad \frac{\cdots}{\Gamma \vdash \lambda x.\, e \Uparrow A \rightarrow B} \rightarrow\!\mathsf{I_b}$$

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:

$$\frac{\Gamma, x : A \vdash e \Uparrow B}{\Gamma \vdash \lambda x.\, e \Uparrow A \rightarrow B} \rightarrow\!\mathsf{I_b}$$

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

$$\frac{\cdots}{\Gamma \vdash e\; e' \Downarrow B} \rightarrow\!\mathsf{E_b} \qquad or \qquad \frac{\cdots}{\Gamma \vdash e\; e' \Uparrow B} \rightarrow\!\mathsf{E_b}$$

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:

$$\frac{\Gamma \vdash e \Downarrow A \rightarrow B \quad \Gamma \vdash e' \Uparrow A}{\Gamma \vdash e\; e' \Downarrow B} \rightarrow\!\mathsf{E_b}$$

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

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Downarrow A} \mathsf{Var_b}$$

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:

$$\frac{\Gamma \vdash e \Downarrow A}{\Gamma \vdash e \Uparrow A} \; \Downarrow\!\Uparrow_{\mathsf{b}}$$

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:

$$\frac{\Gamma \vdash e \Uparrow A}{\Gamma \vdash (e : A) \Downarrow A} \; \Uparrow\!\Downarrow_{\mathsf{b}}$$

The rule $\Uparrow\!\Downarrow_{\mathsf{b}}$ 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:

$$\begin{array}{rrcl} \textsf{intro expression} & I & ::= & \lambda x.\, I \mid E \\ \textsf{elim expression} & E & ::= & x \mid E\; I \mid (I : A) \end{array}$$

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 $\rightarrow$ introduction rule $\rightarrow\!\mathsf{I}$. 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 $\rightarrow$ elimination rule $\rightarrow\!\mathsf{E}$, 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 λ-calculus. As in λ-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.

$$\begin{array}{rrcll} \textsf{intro expression} & I & ::= & \lambda x.\, I & \rightarrow\!\mathsf{I_b} \\ & & & \mid (I, I) & \times\mathsf{I_b} \\ & & & \mid \mathsf{inl}\; I & +\mathsf{I_{Lb}} \\ & & & \mid \mathsf{inr}\; I & +\mathsf{I_{Rb}} \\ & & & \mid \mathsf{case}\; E\; \mathsf{of}\; \mathsf{inl}\; x.\, I \mid \mathsf{inr}\; x.\, I & +\mathsf{E_b} \\ & & & \mid () & \mathsf{Unit_b} \\ & & & \mid \mathsf{abort}\; E & \mathsf{Abort_b} \\ & & & \mid \mathsf{fix}\; x.\, I & \mathsf{Fix_b} \\ & & & \mid E & \Downarrow\!\Uparrow_{\mathsf{b}} \\ \textsf{elim expression} & E & ::= & x & \mathsf{Var_b} \\ & & & \mid E\; I & \rightarrow\!\mathsf{E_b} \\ & & & \mid \mathsf{fst}\; E & \times\mathsf{E_{1b}} \\ & & & \mid \mathsf{snd}\; E & \times\mathsf{E_{2b}} \\ & & & \mid (I : A) & \Uparrow\!\Downarrow_{\mathsf{b}} \end{array}$$

Typing rules for bidirectional typechecking are as follows:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Downarrow A} \; \mathsf{Var_b} \qquad \frac{\Gamma, x : A \vdash I \Uparrow B}{\Gamma \vdash \lambda x.\, I \Uparrow A \rightarrow B} \; \rightarrow\!\mathsf{I_b} \qquad \frac{\Gamma \vdash E \Downarrow A \rightarrow B \quad \Gamma \vdash I \Uparrow A}{\Gamma \vdash E\; I \Downarrow B} \; \rightarrow\!\mathsf{E_b}$$

$$\frac{\Gamma \vdash I_1 \Uparrow A_1 \quad \Gamma \vdash I_2 \Uparrow A_2}{\Gamma \vdash (I_1, I_2) \Uparrow A_1 \times A_2} \times\mathsf{I_b} \quad \frac{\Gamma \vdash E \Downarrow A_1 \times A_2}{\Gamma \vdash \mathsf{fst}\ E \Downarrow A_1} \times\mathsf{E_{1b}} \quad \frac{\Gamma \vdash E \Downarrow A_1 \times A_2}{\Gamma \vdash \mathsf{snd}\ E \Downarrow A_2} \times\mathsf{E_{2b}}$$

$$\frac{\Gamma \vdash I \Uparrow A_1}{\Gamma \vdash \mathsf{inl}\ I \Uparrow A_1 + A_2} +\mathsf{I_{Lb}} \quad \frac{\Gamma \vdash I \Uparrow A_2}{\Gamma \vdash \mathsf{inr}\ I \Uparrow A_1 + A_2} +\mathsf{I_{Rb}}$$

$$\frac{\Gamma \vdash E \Downarrow A_1 + A_2 \quad \Gamma, x_1 : A_1 \vdash I_1 \Uparrow C \quad \Gamma, x_2 : A_2 \vdash I_2 \Uparrow C}{\Gamma \vdash \mathsf{case}\ E\ \mathsf{of}\ \mathsf{inl}\ x_1.\, I_1 \mid \mathsf{inr}\ x_2.\, I_2 \Uparrow C} +\mathsf{E_b}$$

$$\frac{}{\Gamma \vdash () \Uparrow \mathsf{unit}} \mathsf{Unit_b} \quad \frac{\Gamma \vdash E \Downarrow \mathsf{void}}{\Gamma \vdash \mathsf{abort}\ E \Uparrow C} \mathsf{Abort_b}$$

$$\frac{\Gamma, x : A \vdash I \Uparrow A}{\Gamma \vdash \mathsf{fix}\ x.\, I \Uparrow A} \mathsf{Fix_b}$$

$$\frac{\Gamma \vdash E \Downarrow A}{\Gamma \vdash E \Uparrow A} \Downarrow\Uparrow_{\mathsf{b}} \quad \frac{\Gamma \vdash I \Uparrow A}{\Gamma \vdash (I : A) \Downarrow A} \Uparrow\Downarrow_{\mathsf{b}}$$

1.3 Exercises

Exercise 1.4. Give typing rules for true, false, and if e then e_1 else e_2 under bidirectional typechecking.

Exercise 1.5. $(\lambda x.\, x)\ ()$ has type unit. This expression, however, does not typecheck against unit under bidirectional typechecking. Write as much of a derivation $\cdot \vdash (\lambda x.\, x)\ () \Uparrow \mathsf{unit}$ as you can, and indicate with an asterisk (*) where the derivation gets stuck.

Exercise 1.6. 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.