CPSC 311: Definition of Programming Languages: Bidirectional Typing
25-bidirectional

Joshua Dunfield
University of British Columbia

December 8, 2015

Remark. Portions of these notes were adapted from my McGill lecture notes. A few "ML-isms" remain, such as a distinction between "expressions" and "declarations"; these are syntactically distinct in SML, and were also syntactically distinct in the language that was developed in the McGill course (a "tiny" version of SML). I chose to keep this distinction, since mainstream languages often make a similar distinction; for example, in C and Java, local variable declarations are distinct from statements and expressions.

In [Typed] Fun, the closest thing to a declaration is a with expression; in versions that have with*, each binding in a with* could be considered a declaration.

Also, these notes consistently use "variable" where I might have wanted to use "identifier", for consistency with (some of) 311.

Finally, the explanation of the "subsumption rule" may seem odd, because at McGill, I introduced bidirectional typing *before* subtyping.

1 Introduction

A claimed advantage of SML, OCaml, Haskell, and other languages with type systems in the Hindley-Milner tradition is *type inference*: one doesn't need to declare types, the compiler will figure them out. Actually, one *does* need to write types in certain situations, such as module interfaces and some uses of references. Moreover, there are drawbacks to not having to put in type annotations; programmers are deprived of a form of high-grade documentation ("high-grade" because it is formal and machine-checked, unlike English comments which are vague when not outright wrong). There's also the minor problem that more advanced, precise type systems—those that can statically check array accesses, data structure invariants, etc., etc.—*require* (at least some) annotations, as type inference is undecidable! Last but not least, without type annotations, there is no record of the programmer's intent except the declarations themselves, and so type error messages often fail to highlight the genuine source of the error.

At the other extreme, we could require a type annotation on every variable declaration (as is required in many mainstream languages—though recent versions of such languages, e.g. C++, have started to adopt some form of type inference). This is quite tedious, since the type must be written even when self-evident.

The technique of *bidirectional typechecking* lies between the extremes of type inference and mainstream typechecking. Type annotations are required for *some* expressions, and therefore on some declarations, particularly function declarations where the documentation aspect of type annotations is especially important. Unlike type inference, which works fine for type systems roughly as powerful as SML's but then "flames out", bidirectional typechecking is a good foundation for powerful, precise type systems that can check more program properties (such as, again, array accesses). It seems only a matter of time before it is widely used in practice, though as with so much of academic programming languages research, the time involved may well be measured in decades.

2 Two directions of information

The basic idea: Instead of persisting in trying to figure out the type of an expression on its own (knowing only the types of variables, and maybe not even all of those), as type inference does, we alternate between figuring out or *synthesizing* types and *checking* expressions against types already given.

In terms of *judgments*, bidirectionality replaces the standard typing judgment

$$\Gamma \vdash e : A \qquad \text{“under assumptions in the context } \Gamma\text{, the expression } e \text{ has type } A\text{”}$$

with two different judgments:

$$\Gamma \vdash e \Rightarrow A \quad \text{read “under assumptions in } \Gamma\text{, the expression } e \text{ synthesizes type } A\text{”}$$
$$\Gamma \vdash e \Leftarrow A \quad \text{read “under assumptions in } \Gamma\text{, the expression } e \text{ checks against type } A\text{”}$$

It looks like the only difference is in the direction of the arrow... The real difference is in which parts of the judgment are *inputs* and which are *outputs*. When we want to derive $\Gamma \vdash e \Rightarrow A$, we only know Γ and e: the point is to figure out the type A *from* e, as in type inference. But when deriving $\Gamma \vdash e \Leftarrow A$, we already know A, and just need to make sure that e does conform to (check against) the type A.

Remark. As we did with type inference, we assume that lam and rec expressions do *not* include a function type (for lam) or body type (for rec).

3 Typing rules

In formulating the rules for deriving bidirectional typing judgments, we are guided by two observations:

(1) we can't use information we don't have;

(2) we should try to use information we do have.

The second observation leads to our first typing rule, for variables. First, we should define (as a BNF grammar) the form of Γ, which represents contexts (sometimes called, confusingly, environments) of typing assumptions.

Γ	$::=$	$\emptyset$	Empty context
	$\mid$	$x{:}A, \Gamma$	Context Γ plus the assumption that variable x is of type A

And now, the rule for typing variables. It says that if we know x is supposed to have type A, because $x{:}A$ is in the context of assumptions, then x synthesizes type A.

$$\frac{(x{:}A) \in \Gamma}{\Gamma \vdash (\mathsf{id}\ x) \Rightarrow A}\ \text{Synth-var}$$

3.1 Functions

If we apply a function, we need the type of the function. But if we create a function (by writing $(\mathsf{lam}\ x\ e)$), we don't (yet) know the function type. So the rule for applications $e1\ e2$ can reasonably expect the function type to be synthesized *from* the function $e1$. On the other hand, in the rule for $(\mathsf{lam}\ x\ e)$ we don't yet know what the domain or range of the function should be, so (following observation (1)) we check $(\mathsf{lam}\ x\ e)$ against a type that is (somehow) already known.

$$\frac{\Gamma \vdash e1 \Rightarrow A \rightarrow A' \qquad \Gamma \vdash e2 \Leftarrow A}{\Gamma \vdash (\mathsf{app}\ e1\ e2) \Rightarrow A'}\ \text{Synth-app} \qquad \frac{\Gamma, x{:}A \vdash e \Leftarrow A'}{\Gamma \vdash (\mathsf{lam}\ x\ e) \Leftarrow (A \rightarrow A')}\ \text{Check-lam}$$

In most situations, these rules work well: When applying a function, if the function being applied is just a variable, we can synthesize its type (rule Synth-var) so we can indeed synthesize the type of the function $e1$ in Synth-app. Or, if the function being applied is itself a function application, as in

$$\big(\mathsf{app}\ (\mathsf{app}\ (\mathsf{id}\ twice)\ (\mathsf{id}\ f))\ (\mathsf{id}\ x)\big)$$

(where $twice$, which applies its first argument to its second argument twice, has type $(\mathsf{Num} \rightarrow \mathsf{Num}) \rightarrow \mathsf{Num} \rightarrow \mathsf{Num})$ that *also* synthesizes its type (rule Synth-app, applied to twice `f`), so again we can successfully apply Synth-app. We can also successfully type

$$\big(\mathsf{app}\ (\mathsf{app}\ (\mathsf{id}\ twice)\ (\mathsf{lam}\ y\ (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y))))\ (\mathsf{id}\ x)\big)$$

because in Synth-app, we check the argument $e2 = (\mathsf{lam}\ y\ (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y)))$ against the domain $A = (\mathsf{Num} \rightarrow \mathsf{Num})$, which is the type that Check-lam checks the lam against. Here is the derivation, where

$$\Gamma = twice{:}\underbrace{((\mathsf{Num} \rightarrow \mathsf{Num}) \rightarrow \mathsf{Num} \rightarrow \mathsf{Num})}_{Atwice}, x{:}\mathsf{Num}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash twice \Rightarrow Atwice}\ \text{Synth-var} \quad \dfrac{\dfrac{\begin{matrix}\vdots \\ y{:}\mathsf{Num}, \Gamma \vdash (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y)) \Rightarrow \mathsf{Num} \quad \mathsf{Num}{=}\mathsf{Num}\end{matrix}}{y{:}\mathsf{Num}, \Gamma \vdash (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y)) \Leftarrow \mathsf{Num}}\ \text{Check-sub}}{\Gamma \vdash (\mathsf{lam}\ y\ (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y))) \Leftarrow (\mathsf{Num} \rightarrow \mathsf{Num})}\ \text{Check-lam}}{\Gamma \vdash (\mathsf{app}\ (\mathsf{id}\ twice)\ (\mathsf{lam}\ y\ (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y)))) \Rightarrow (\mathsf{Num} \rightarrow \mathsf{Num})}\ \text{Synth-app} \quad \dfrac{\dfrac{}{\Gamma \vdash (\mathsf{id}\ x) \Rightarrow \mathsf{Num}}\ \text{Synth-var} \quad \mathsf{Num}{=}\mathsf{Num}}{\Gamma \vdash (\mathsf{id}\ x) \Leftarrow \mathsf{Num}}\ \text{Check-sub}}{\Gamma \vdash (\mathsf{app}\ (\mathsf{app}\ (\mathsf{id}\ twice)\ (\mathsf{lam}\ y\ (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y))))\ (\mathsf{id}\ x)) \Rightarrow \mathsf{Num}}\ \text{Synth-app}$$

These rules don't let us immediately apply a lam; for example,

$$\big(\mathsf{app}\ (\mathsf{lam}\ y\ (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y)))\ (\mathsf{num}\ 5)\big)$$

won't typecheck because Synth-app demands that the lam synthesize, and our only rule for lam, namely Check-lam, doesn't synthesize:

$$\dfrac{\emptyset \vdash (\mathsf{lam}\ y\ (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y))) \not\Rightarrow \qquad \ldots}{\emptyset \vdash \big(\mathsf{app}\ (\mathsf{lam}\ y\ (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y)))\ (\mathsf{num}\ 5)\big) \not\Rightarrow}\ \text{Synth-app}$$

This restriction is somewhat inconvenient for Fun expressions that we might write in 311, but it's not very inconvenient in practice: real code seldom applies a function immediately in this way.

3.2 "Subsumption"

We actually used an undeclared rule in the example above. When we check (id f) against Num → Num, we need to derive the judgment (id f) $\Leftarrow$ Num → Num. But our only rule for variables is Synth-var, which (assuming f : Num → Num is in Γ) derives $\Gamma \vdash (\mathsf{id}\ f) \Rightarrow \mathsf{Num} \to \mathsf{Num}$.

We need a rule that lets us show that an expression checks against a type, provided the expression synthesizes the same type. For reasons related to subtyping, this rule is called *subsumption* and we write it with an explicit comparison between A (the type checked against), and A' (the synthesized type).

$$\dfrac{\Gamma \vdash e \Rightarrow A' \qquad A' = A}{\Gamma \vdash e \Leftarrow A}\ \text{Check-sub}$$

3.3 Recursive expressions and typing annotations

$$\dfrac{\Gamma, f{:}A \vdash e \Leftarrow A}{\Gamma \vdash (\mathsf{rec}\ u\ e) \Leftarrow A}\ \text{Check-rec} \qquad \dfrac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (\mathsf{anno}\ e\ A) \Rightarrow A}\ \text{Synth-anno}$$

Annotations allow us to turn expressions that don't synthesize a type into expressions that do, by writing the type ourselves. Recall the expression (app (lam y (add (id y) (id y))) (num 5)), which doesn't synthesize a type because (lam y (add (id y) (id y))) doesn't synthesize. Add an annotation and we can readily synthesize the expression's type:

$$\dfrac{\dfrac{\dfrac{y{:}\mathsf{Num} \vdash (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y)) \Leftarrow \mathsf{Num}}{\emptyset \vdash (\mathsf{lam}\ y\ (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y))) \Leftarrow \mathsf{Num}{\to}\mathsf{Num}}\ \text{Check-lam}}{\emptyset \vdash (\mathsf{anno}\ (\mathsf{lam}\ y\ (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y)))\ \mathsf{Num}{\to}\mathsf{Num}) \Rightarrow \mathsf{Num}{\to}\mathsf{Num}}\ \text{Synth-anno} \qquad \dfrac{\dfrac{}{\emptyset \vdash 5 \Rightarrow \mathsf{Num}}\ \text{Synth-num}}{\emptyset \vdash 5 \Leftarrow \mathsf{Num}}\ \text{Check-sub}}{\emptyset \vdash (\mathsf{app}\ (\mathsf{anno}\ (\mathsf{lam}\ y\ (\mathsf{add}\ (\mathsf{id}\ y)\ (\mathsf{id}\ y)))\ \mathsf{Num}{\to}\mathsf{Num})\ (\mathsf{num}\ 5)) \Rightarrow \mathsf{Num}}\ \text{Synth-app}$$

3.4 Primitive operations

The typing rules for add and sub work similarly to the rules for Synth-app, if we consider the operator being applied to be a function whose type is somehow known, and the two expressions $e1$ and $e2$ its arguments.

$$\dfrac{\Gamma \vdash e1 \Leftarrow \mathsf{Num} \qquad \Gamma \vdash e2 \Leftarrow \mathsf{Num}}{\Gamma \vdash (\mathsf{add}\ e1\ e2) \Rightarrow \mathsf{Num}}\ \text{Synth-add} \qquad \dfrac{\Gamma \vdash e1 \Leftarrow \mathsf{Num} \qquad \Gamma \vdash e2 \Leftarrow \mathsf{Num}}{\Gamma \vdash (\mathsf{sub}\ e1\ e2) \Rightarrow \mathsf{Num}}\ \text{Synth-sub}$$

3.5 Booleans

$$\frac{}{\Gamma \vdash (\mathsf{btrue}) \Rightarrow \mathsf{Bool}}\ \text{Synth-btrue} \qquad \frac{}{\Gamma \vdash (\mathsf{bfalse}) \Rightarrow \mathsf{Bool}}\ \text{Synth-bfalse}$$

$$\frac{\Gamma \vdash e \Leftarrow \mathsf{Bool} \qquad \Gamma \vdash e1 \Leftarrow A \qquad \Gamma \vdash e2 \Leftarrow A}{\Gamma \vdash (\mathsf{ite}\ e\ e1\ e2) \Leftarrow A}\ \text{Check-ite}$$

3.6 Pairs

$$\frac{\Gamma \vdash e1 \Leftarrow A1 \qquad \Gamma \vdash e2 \Leftarrow A2}{\Gamma \vdash (\mathsf{pair}\ e1\ e2) \Leftarrow (A1 * A2)}\ \text{Check-pair}$$

$$\frac{\Gamma \vdash e \Rightarrow (A1 * A2) \qquad x1 : A1, x2 : A2, \Gamma \vdash eBody \Leftarrow B}{\Gamma \vdash (\mathsf{pair\text{-}case}\ e\ x1\ x2\ eBody) \Leftarrow B}\ \text{Check-pair-case}$$

3.7 with-expressions

In the expression

$$(\mathsf{with}\ x\ (\mathsf{app}\ (\mathsf{id}\ fact)\ (\mathsf{num}\ 5))\ (\mathsf{pair}\ (\mathsf{id}\ x)\ (\mathsf{id}\ x)))$$

we should be able to figure out (assuming our context Γ contains the typing *fact* : $\mathsf{Num} \rightarrow \mathsf{Num}$) that $(\mathsf{id}\ x)$ has type Num and therefore $(\mathsf{pair}\ (\mathsf{id}\ x)\ (\mathsf{id}\ x))$ checks against $\mathsf{Num} * \mathsf{Num}$.

$$\frac{\Gamma \vdash e1 \Rightarrow A \qquad x{:}A, \Gamma \vdash e2 \Leftarrow B}{\Gamma \vdash (\mathsf{with}\ x\ e1\ e2) \Leftarrow B}\ \text{Check-with}$$

3.8 Adding more convenience

The rules above can be criticized for requiring too many annotations. For example, even if the body of a with does synthesize a type, Check-with refuses to utilize that fact, and demands that the type of the body be given already. The same criticism applies to Check-ite, and even Check-pair: the rules above can derive

$$\Gamma \vdash (\mathsf{pair}\ (\mathsf{num}\ 3)\ (\mathsf{num}\ 5)) \Leftarrow (\mathsf{Num} * \mathsf{Num})$$

but not

$$\Gamma \vdash (\mathsf{pair}\ (\mathsf{num}\ 3)\ (\mathsf{num}\ 5)) \Rightarrow (\mathsf{Num} * \mathsf{Num})$$

This is less of a problem in practice than it might appear: many withs *can* be checked, such as a with that is the body of a lam; many pairs are passed as arguments to functions, where their types will be checked.

For the other cases, we can deal with many of these problems fairly easily, by adding Synth- versions of some of the Check- rules.

$$\frac{\Gamma \vdash e1 \Rightarrow A1 \qquad \Gamma \vdash e2 \Rightarrow A2}{\Gamma \vdash (\mathsf{pair}\ e1\ e2) \Rightarrow (A1 * A2)}\ \text{Synth-pair} \qquad \frac{\Gamma \vdash e \Rightarrow (A1 * A2) \qquad x1 : A1, x2 : A2, \Gamma \vdash eBody \Rightarrow B}{\Gamma \vdash (\mathsf{pair\text{-}case}\ e\ x1\ x2\ eBody) \Rightarrow B}\ \text{Synth-pair-case}$$

$$\frac{\Gamma \vdash e1 \Rightarrow A \qquad x{:}A, \Gamma \vdash e2 \Rightarrow B}{\Gamma \vdash (\mathsf{with}\ x\ e1\ e2) \Rightarrow B}\ \text{Synth-with}$$

$$\frac{\Gamma \vdash e \Leftarrow \mathsf{Bool} \qquad \Gamma \vdash e1 \Rightarrow A \qquad \Gamma \vdash e2 \Rightarrow A}{\Gamma \vdash (\mathsf{ite}\ e\ e1\ e2) \Rightarrow A}\ \text{Synth-ite}$$

$$\frac{\Gamma \vdash e \Leftarrow \mathsf{Bool} \qquad \Gamma \vdash e1 \Rightarrow A1 \qquad \Gamma \vdash e2 \Rightarrow A2 \qquad A1 = A2}{\Gamma \vdash (\mathsf{ite}\ e\ e1\ e2) \Rightarrow A1}\ \text{Synth-ite}$$

The last two versions of Synth-ite are equivalent.

4 Scaling up

New typed languages, and new versions of typed languages, tend to accumulate more and fancier type systems. Type inference works nicely for languages that are essentially the λ-calculus with a limited form of (parametric) polymorphism. But extending type inference to support a more powerful type system often constitutes a research project in itself! For example, type inference works well for the Hindley-Milner form of polymorphism, which only allows the generic quantifiers on types on the outside:

$$\forall\alpha.\ (\alpha * \alpha) \to \alpha$$

But type inference is difficult for types like

$$\forall\beta.\ \big(\forall\alpha.\ (\alpha * \alpha) \to \alpha\big) \to \beta \to \beta$$

that have a quantifier on the inside. We don't have time to explain how such types work, or why they're useful, but they do occur (at least, occasionally) in programs.

Type inference also has a lot of trouble with subtyping. (Some of the hostility of functional programmers to object-oriented languages may be "sour grapes": most object-oriented languages have subtyping, while typed functional languages that use type inference don't, partly *because* they use type inference.)

Bidirectional typing easily supports subtyping. In fact, all we have to do is change the Check-sub rule (whose name made no sense because it didn't do any subtyping!) to use <: instead of =:

$$\frac{\Gamma \vdash e \Rightarrow A' \qquad A' = A}{\Gamma \vdash e \Leftarrow A}\ \text{Check-sub} \qquad\qquad \frac{\Gamma \vdash e \Rightarrow A' \qquad A' <: A}{\Gamma \vdash e \Leftarrow A}\ \text{Check-sub}$$

This avoids the issue of possibly trying to apply Check-sub repeatedly on the same expression: Check-sub's conclusion has $\Leftarrow$, but its premise has $\Rightarrow$, and there is no Synth- rule that uses subtyping.

Bidirectional typing also supports operator overloading, record subtyping, intersection types, and refinement types.