Bidirectional typechecking

Joshua Dunfield

November 9, 2012

(These are lecture notes from McGill University COMP 302, March 2010, very slightly revised November 2012.)

1 Introduction

A claimed advantage of SML 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). This is quite tedious, since the type must be written even when completely obvious.

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 is very simple. 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 known.

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

$\Gamma \vdash e : \tau$ "under assumptions in the context Γ, the expression e has type τ"

with two different judgments:

$\Gamma \vdash e \Rightarrow \tau$ read "under assumptions in Γ, the expression e synthesizes type τ"

$\Gamma \vdash e \Leftarrow \tau$ read "under assumptions in Γ, the expression e checks against type τ"

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 $\Gamma \vdash e \Rightarrow \tau$, we only know Γ and e: the point is to figure out the type τ *from* e, as in type inference. But when deriving $\Gamma \vdash e \Leftarrow \tau$, we already know τ, and just need to make sure that e does conform to (check against) the type τ.

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

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

$$\frac{}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x \Rightarrow \tau}\ \text{T-VAR}$$

3.1 Functions

Now let's look at functions. If we apply a function, we must have that function. But if we create a function (by writing fn x => e), we don't (yet) have the function. So the rule for applications $e_1\ e_2$ can reasonably expect the function type to be synthesized *from* the function e_1. On the other hand, in the rule for fn x => e we don't yet know what the domain or range of the function should be, so (following observation (1)) we check fn x => e against a type that is (somehow) already known.

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau \text{ -> } \tau' \qquad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau'}\ \text{T-APP} \qquad \frac{\Gamma, x{:}\tau \vdash e \Leftarrow \tau'}{\Gamma \vdash (\text{fn } x \text{ => } e) \Leftarrow (\tau \text{ -> } \tau')}\ \text{T-FN}$$

These rules work well in most situations. When applying a function, if the function being applied is just a variable, variables synthesize their type (rule T-VAR) so we can indeed synthesize the type of the function e_1 in T-APP. Or, if the function being applied is itself a function application, as in

$$(\texttt{twice f})\ \texttt{x}$$

(where `twice`, which applies its first argument to its second argument twice, has type `(int -> int) -> int -> int`) that *also* synthesizes its type (rule T-APP, applied to `twice f`), so again we can successfully apply T-APP. We can also successfully type

$$\texttt{(twice (fn y => y * y)) x}$$

because in T-APP, we check the argument $e_2 = \texttt{fn y => y * y}$ against the domain $\tau = \texttt{int -> int}$, satisfiying T-FN which checks against a known type. Here is the derivation, where

$$\Gamma = \texttt{twice:((int -> int) -> int -> int), x:int}$$

$$\dfrac{\dfrac{}{\Gamma \vdash \texttt{twice} \Rightarrow \texttt{((int -> int) -> int -> int)}}\,\text{T-VAR} \quad \dfrac{\dfrac{\vdots}{\Gamma, \texttt{y:int} \vdash \texttt{y * y} \Rightarrow \texttt{int}}\,\text{T-SUB}}{\Gamma \vdash \texttt{fn y => y * y} \Leftarrow \texttt{(int -> int)}}\,\text{T-FN}}{\dfrac{\Gamma \vdash \texttt{twice (fn y => y * y)} \Rightarrow \texttt{(int -> int)}}{\Gamma \vdash \texttt{(twice (fn y => y * y)) x} \Rightarrow \texttt{int}}\,\text{T-APP} \quad \dfrac{}{\Gamma \vdash \texttt{x} \Rightarrow \texttt{int}}\,\text{T-VAR}}\,\text{T-APP}$$

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

$$\texttt{(fn y => y * y) 5}$$

won't typecheck because T-APP demands that the function synthesize, and T-FN (our only rule for `fn y => y * y`) doesn't synthesize:

$$\dfrac{\cdot \vdash \texttt{fn y => y * y} \not\Rightarrow \quad \ldots}{\cdot \vdash \texttt{(fn y => y * y) 5} \not\Rightarrow}\,\text{T-APP}$$

3.2 "Subsumption"

We actually need another rule for the example `(twice f) x`. When we check `f` against `int -> int`, we need to derive the judgment $\texttt{f} \Leftarrow \texttt{int -> int}$. But our only rule for variables is T-VAR, which (assuming $\texttt{f} : \texttt{int -> int}$) derives $\texttt{f} \Rightarrow \texttt{int -> int}$. So we need a rule that lets us show that an expression checks against a type, provided the expression synthesizes the same type. For reasons that will be made clear when we discuss subtyping, this rule is called *subsumption* and we write it with an explicit comparison between τ, the type checked against, and the synthesized type τ'.

$$\dfrac{\Gamma \vdash e \Rightarrow \tau' \qquad \tau = \tau'}{\Gamma \vdash e \Leftarrow \tau}\,\text{T-SUB}$$

3.3 Recursive expressions and typing annotations

$$\dfrac{\Gamma, \texttt{f:}\tau \vdash e \Leftarrow \tau}{\Gamma \vdash \texttt{(rec f : } \tau \texttt{ => } e\texttt{)} \Rightarrow \tau}\,\text{T-REC} \qquad \dfrac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash \texttt{(}e\texttt{ : }\tau\texttt{)} \Rightarrow \tau}\,\text{T-ANNO}$$

Annotations allow us to turn expressions that don't synthesize a type into expressions that do, by writing the type ourselves. Recall the expression `(fn y => y * y) 5`, which doesn't synthesize a

type because fn y => y * y doesn't synthesize. Add an annotation and we can readily synthesize the expression's type:

$$\dfrac{\dfrac{\dfrac{\dfrac{y{:}\mathsf{int} \vdash y * y \Leftarrow \mathsf{int}}{\cdot \vdash \mathsf{fn}\ y \Rightarrow y * y \Leftarrow \mathsf{int} \to \mathsf{int}}\ \text{T-FN}}{\cdot \vdash (\mathsf{fn}\ y \Rightarrow y * y : \mathsf{int} \to \mathsf{int}) \Rightarrow \mathsf{int} \to \mathsf{int}}\ \text{T-ANNO} \quad \dfrac{\dfrac{}{\cdot \vdash 5 \Rightarrow \mathsf{int}}\ \text{T-NUM}}{\cdot \vdash 5 \Leftarrow \mathsf{int}}\ \text{T-SUB}}{\cdot \vdash ((\mathsf{fn}\ y \Rightarrow y * y) : \mathsf{int} \to \mathsf{int})\ 5 \Rightarrow \mathsf{int}}\ \text{T-APP}$$

3.4 Primitive operations

Here we assume a judgment $\mathsf{op} : \tau \to \tau'$ where, for example, $< : \mathsf{int} * \mathsf{int} \to \mathsf{bool}$.

The typing rules work similarly to the rules for T-APP. (In fact, instead of having special rules for these operations, we could simply say they are part of a predefined context[1] containing, for example, $< : (\mathsf{int} * \mathsf{int} \to \mathsf{bool})$, and then use T-APP. However, that glosses over the way the binary operations are written: as infix operators.)

$$\dfrac{\mathsf{op} : \tau_1 * \tau_2 \to \tau \qquad \Gamma \vdash e_1 \Leftarrow \tau_1 \qquad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1\ \mathsf{op}\ e_2 \Rightarrow \tau}\ \text{T-BINARY-PRIMOP}$$

$$\dfrac{\mathsf{op} : \tau_1 \to \tau \qquad \Gamma \vdash e_1 \Leftarrow \tau_1}{\Gamma \vdash \mathsf{op}\ e_1 \Rightarrow \tau}\ \text{T-UNARY-PRIMOP}$$

3.5 Booleans

$$\dfrac{}{\Gamma \vdash \mathsf{true} \Rightarrow \mathsf{bool}}\ \text{T-TRUE} \qquad \dfrac{}{\Gamma \vdash \mathsf{false} \Rightarrow \mathsf{bool}}\ \text{T-FALSE} \qquad \dfrac{\Gamma \vdash e \Leftarrow \mathsf{bool} \qquad \Gamma \vdash e_1 \Leftarrow \tau \qquad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Leftarrow \tau}\ \text{T-IF}$$

3.6 Let-expressions and declarations

In the expression

$$\mathsf{let\ val}\ x = \mathit{fact}\ 5\ \mathsf{in}\ (x, x)\ \mathsf{end}$$

we should be able to figure out (assuming our context Γ contains the typing $\mathit{fact} : \mathsf{int} \to \mathsf{int}$) that x has type int and therefore (x, x) has type $\mathsf{int} * \mathsf{int}$. That is, the declaration

$$\mathsf{val}\ x = \mathit{fact}\ 5$$

should produce the assumption $x{:}\mathsf{int}$, which we then use to synthesize a type for the body (x, x).

The judgment for declarations will be

$$\Gamma \vdash \mathit{decs} \Rightarrow \Gamma'$$

read "under assumptions Γ, the declarations decs produce the assumptions Γ'". For the example above, we would derive

$$\Gamma \vdash (\mathsf{val}\ x = \mathit{fact}\ 5) \Rightarrow (x{:}\mathsf{int})$$

[1] Predefined contexts go by many names: predefined environment, standard environment, standard basis, prelude, built-in environment, ...

The rules T-LET and T-LET-SYN use the judgment form for declarations in their premises. Here's part of the derivation for the above example.

$$\dfrac{\dfrac{\Gamma \vdash (\mathit{fact}\ 5) \Rightarrow \mathsf{int}}{\Gamma \vdash (\mathsf{val}\ x = \mathit{fact}\ 5) \Rightarrow (x{:}\mathsf{int})}\ \textsc{T-BY-VAL} \qquad \Gamma, x{:}\mathsf{int} \vdash (x, x) \Rightarrow \mathsf{int} * \mathsf{int}}{\Gamma \vdash (\mathsf{let}\ \mathsf{val}\ x = \mathit{fact}\ 5\ \mathsf{in}\ (x, x)\ \mathsf{end}) \Rightarrow (\mathsf{int} * \mathsf{int})}\ \textsc{T-LET-SYN}$$

Notice how the $(x{:}\mathsf{int})$ produced from the declaration $(\mathsf{val}\ x = \mathit{fact}\ 5)$ is added to the assumptions when we synthesize a type for (x, x). Also notice that the $x{:}\mathsf{int}$ is not somehow added to the conclusion of T-LET-SYN—because x is not in scope outside the let-expression.

$$\dfrac{\Gamma \vdash \mathit{decs} \Rightarrow \Gamma' \qquad \Gamma, \Gamma' \vdash e \Leftarrow \tau}{\Gamma \vdash \mathsf{let}\ \mathit{decs}\ \mathsf{in}\ e\ \mathsf{end} \Leftarrow \tau}\ \textsc{T-LET} \qquad \dfrac{\Gamma \vdash \mathit{decs} \Rightarrow \Gamma' \qquad \Gamma, \Gamma' \vdash e \Rightarrow \tau}{\Gamma \vdash \mathsf{let}\ \mathit{decs}\ \mathsf{in}\ e\ \mathsf{end} \Rightarrow \tau}\ \textsc{T-LET-SYN}$$

$$\dfrac{\Gamma \vdash \mathit{dec}_1 \Rightarrow \Gamma_1 \qquad \Gamma, \Gamma_1 \vdash \mathit{decs} \Rightarrow \Gamma_2}{\Gamma \vdash \mathit{dec}_1\ \mathit{decs} \Rightarrow \Gamma_1, \Gamma_2}\ \textsc{T-DECS}$$

$$\dfrac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (\mathsf{val}\ x = e) \Rightarrow (x : \tau)}\ \textsc{T-BY-VAL} \qquad \dfrac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (\mathsf{name}\ x = e) \Rightarrow (x : \tau)}\ \textsc{T-BY-NAME}$$

$$\dfrac{\Gamma \vdash e \Rightarrow (\tau_1 * \cdots * \tau_n)}{\Gamma \vdash (\mathsf{val}\ (x_1, \ldots, x_n) = e) \Rightarrow (x_1 : \tau_1), \ldots, (x_n : \tau_n)}\ \textsc{T-BY-VAL-TUPLE}$$

4 Summary of typing rules

Rules for expressions e

$$\frac{}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x \Rightarrow \tau}\ \text{T-VAR}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau \text{ -> } \tau' \qquad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau'}\ \text{T-APP} \qquad \frac{\Gamma, x{:}\tau \vdash e \Leftarrow \tau'}{\Gamma \vdash (\text{fn } x \text{ => } e) \Leftarrow (\tau \text{ -> } \tau')}\ \text{T-FN}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n \Leftarrow \tau_n}{\Gamma \vdash (e_1, \ldots, e_n) \Leftarrow (\tau_1 * \cdots * \tau_n)}\ \text{T-TUPLE} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n \Rightarrow \tau_n}{\Gamma \vdash (e_1, \ldots, e_n) \Rightarrow (\tau_1 * \cdots * \tau_n)}\ \text{T-TUPLE-SYN}$$

$$\frac{\text{op} : \tau_1 * \tau_2 \text{ -> } \tau \qquad \Gamma \vdash e_1 \Leftarrow \tau_1 \qquad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1\ \text{op}\ e_2 \Rightarrow \tau}\ \text{T-BINARY-PRIMOP}$$

$$\frac{}{\cdot \vdash n \Rightarrow \text{int}}\ \text{T-NUM} \qquad \frac{\text{op} : \tau_1 \text{ -> } \tau \qquad \Gamma \vdash e_1 \Leftarrow \tau_1}{\Gamma \vdash \text{op}\ e_1 \Rightarrow \tau}\ \text{T-UNARY-PRIMOP}$$

$$\frac{}{\Gamma \vdash \text{true} \Rightarrow \text{bool}}\ \text{T-TRUE} \qquad \frac{}{\Gamma \vdash \text{false} \Rightarrow \text{bool}}\ \text{T-FALSE} \qquad \frac{\Gamma \vdash e \Leftarrow \text{bool} \qquad \Gamma \vdash e_1 \Leftarrow \tau \qquad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Leftarrow \tau}\ \text{T-IF}$$

$$\frac{\Gamma, f{:}\tau \vdash e \Leftarrow \tau}{\Gamma \vdash (\text{rec f} : \tau \text{ => } e) \Rightarrow \tau}\ \text{T-REC} \qquad \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash (e : \tau) \Rightarrow \tau}\ \text{T-ANNO} \qquad \frac{\Gamma \vdash e \Rightarrow \tau' \qquad \tau = \tau'}{\Gamma \vdash e \Leftarrow \tau}\ \text{T-SUB}$$

$$\frac{\Gamma \vdash \text{decs} \Rightarrow \Gamma' \qquad \Gamma, \Gamma' \vdash e \Leftarrow \tau}{\Gamma \vdash \text{let decs in } e \text{ end} \Leftarrow \tau}\ \text{T-LET} \qquad \frac{\Gamma \vdash \text{decs} \Rightarrow \Gamma' \qquad \Gamma, \Gamma' \vdash e \Rightarrow \tau}{\Gamma \vdash \text{let decs in } e \text{ end} \Rightarrow \tau}\ \text{T-LET-SYN}$$

Rules for declarations

$$\frac{\Gamma \vdash \text{dec}_1 \Rightarrow \Gamma_1 \qquad \Gamma, \Gamma_1 \vdash \text{decs} \Rightarrow \Gamma_2}{\Gamma \vdash \text{dec}_1\ \text{decs} \Rightarrow \Gamma_1, \Gamma_2}\ \text{T-DECS}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (\text{val } x = e) \Rightarrow (x : \tau)}\ \text{T-BY-VAL} \qquad \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (\text{name } x = e) \Rightarrow (x : \tau)}\ \text{T-BY-NAME}$$

$$\frac{\Gamma \vdash e \Rightarrow (\tau_1 * \cdots * \tau_n)}{\Gamma \vdash (\text{val } (x_1, \ldots, x_n) = e) \Rightarrow (x_1 : \tau_1), \ldots, (x_n : \tau_n)}\ \text{T-BY-VAL-TUPLE}$$