Supplementary Notes
Lecture 17: Bidirectional Typing

15-312: Foundations of Programming Languages
Joshua Dunfield (joshuad@cs.cmu.edu)

24 October 2002

1 Type Inference: The Good, The Bad

So far (as formulated in Assignments 2 and 4, for instance), the typing problem has always been:

> Given a context Γ and term e, produce its type τ (or fail if it has no type).

This is the same problem as type inference in core SML (SML without modules). It's possible to annotate any SML expression with a type; this is often highly desirable, given SML/NJ's suboptimal type error reporting, since it tends to produce type error messages in which the claimed error site actually *is* the error site. But it is *never* necessary (again, *without* modules—we've already seen how existential types make type inference impossible). We haven't really tried to do this in MinML; while the form of the problem is the same, we have always required certain types to be explicitly annotated, in particular, on function declarations. As we expanded the language, this became increasingly annoying; the type annotation on **raise**, for example, seems particularly gratuitous.

It's well understood how to do full type inference, SML-style: generate constraints and unify the variables. Why haven't we done this, since it would allow us to get rid of the type annotations on **raise** and so forth? There are two reasons:

1. It's somewhat complicated. It would probably be a full programming assignment, and there are more interesting things to do.

2. After a while, it stops working.

What do I mean by that? If you add existentials to your language, you can't infer all types. If you add sums, you can't infer all types—what is the type of

$$\texttt{inl}(5)$$

? If you add something more exotic, like intersection types or refinement types, you can't infer all types. Most relevantly, **you can't do subtyping**. Consider the subsumption typing rule

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma \leq \tau}{\Gamma \vdash e : \tau} \text{ (Sub)}$$

What does this say? To infer type τ for e, we must infer type σ for e and show $\sigma \leq \tau$. Right away there's a problem: there's nothing to keep us from recursing forever. But hey, infinite recursion is silly. We can just agree to not apply the subsumption rule twice in a row. (Why is this complete?)

Unfortunately, that was the least of our problems. We have $e : \sigma$. But we have to show $\sigma \leq \tau$. What is τ? We have no idea. We have to *guess* a τ that will make the rest of the typing derivation work.[1]

Life would be so much easier if we were *given* the type, instead of having to infer it.

2 No Type Inference: The Ugly

Instead of inferring types, let's check them. The problem of typing becomes

> Given a context Γ, a term e, and a type τ, return **true** iff e checks against τ.

Now everything is very easy. There's just one little problem: if we do this everywhere, we have to write so many type annotations that the language becomes unusable.

```
(2 : int) + (2 : int) : int
```

3 Some Type Inference

In practice, languages use some mixture of inferred types and checked types. C[2] and Java are examples: types have to be given with all functions

[1] This is similar to the problem with implementing the transitivity rule, discussed in recitation.

[2] Of course, C's type system is almost meaningless, but that's beside the point.

and variable declarations, but not for things like `2 + 2`. SML is another example: inference suffices for the core language but not the module language, since modules correspond to existential types. Saying that structure FOO ascribes to signature Foo amounts to writing an annotation on an existential type.

Likewise, intersection types $\sigma \,\&\, \tau$ can't be inferred because in the rule

$$\frac{\Gamma \vdash v : \sigma \qquad \Gamma \vdash v : \tau}{\Gamma \vdash v : \sigma \,\&\, \tau} \text{ (\&Intro)}$$

one has to guess both σ and τ (and in fact there are an infinite number of intersection types for any well-typed term: $\sigma \,\&\, \sigma$, $(\sigma \,\&\, \sigma) \,\&\, \sigma$, ...).

We would like a system that is

1. Practical (not just decidable, but efficient too);
2. Usable (needing only a reasonable number of type annotations, easy to understand, predictable, good error reporting, ...?);
3. Aesthetically pleasing.

4 Bidirectional Typing

The idea is to have both inference and checking judgments in the same system. In the inference judgment

$$\Gamma \vdash e \uparrow \tau$$

one is given Γ and e and must produce a τ (or fail). In the checking judgment

$$\Gamma \vdash e \downarrow \tau$$

one is given everything—Γ, e, and τ—and must simply return true iff the judgment is derivable.

We can move between the two judgments as follows. First we introduce a new syntactic form, the type annotation, written

$$e : \tau \qquad \texttt{Anno}(e, \tau)$$

If we need to infer a type for a term, but no type can be inferred (for example, if the term is a `pack`), the user has to give a type annotation. Then we can check against the annotation. The rule is

$$\frac{\Gamma \vdash e \downarrow \tau}{\Gamma \vdash (e : \tau) \uparrow \tau} \text{ (Anno)}$$

So we can move from inferring a type for an annotated term to checking the term against a type. How can we move in the other direction? If we can infer type τ for e, e certainly should check against τ.

$$\frac{\Gamma \vdash e \uparrow \tau}{\Gamma \vdash e \downarrow \tau} \text{ (AlmostSub)}$$

However, the subsumption rule subsumes[3] this rule. The bidirectional subsumption rule is

$$\frac{\Gamma \vdash e \uparrow \sigma \qquad \sigma \leq \tau}{\Gamma \vdash e \downarrow \tau} \text{ (Sub)}$$

Now (AlmostSub) is derivable from (Sub) using reflexivity of subtyping.

These are the only rules where we move between inference and checking of the entire term e; in the other rules, we will variously check or infer *sub*terms of the term e whose type is being inferred or checked against.

In general, for each syntactic form in MinML, we have a rule concluding $\ldots \uparrow \tau$ or $\ldots \downarrow \tau$. In some cases, it's easy to see which is appropriate. The types of free variables can always be found in the context Γ, so we can always infer a type for a variable, and we can always infer a type for a `num`.

$$\frac{\Gamma = \Gamma_1, x{:}\tau, \Gamma_2}{\Gamma \vdash x \uparrow \tau} \text{ (Var)} \qquad \frac{}{\Gamma \vdash \texttt{num}(\bar{k}) \uparrow \texttt{int}} \text{ (Int)}$$

To type a function (without type annotations), we might try

$$\frac{\Gamma, x{:}\sigma \vdash e \downarrow \tau}{\Gamma \vdash \texttt{fun}\, f(x)\, \texttt{is}\, e\, \texttt{end} \uparrow \sigma \to \tau} \text{ (bad-Fun)}$$

But this would require us to guess both σ and τ. What we want is

$$\frac{\Gamma, f{:}\sigma{\to}\tau, x{:}\sigma \vdash e \downarrow \tau}{\Gamma \vdash \texttt{fun}\, f(x)\, \texttt{is}\, e\, \texttt{end} \downarrow \sigma \to \tau} \text{ (Fun)}$$

(Note about new vs. old syntax for annotating functions.) The rule for lambdas is just the same but without f:

$$\frac{\Gamma, x{:}\sigma \vdash e \downarrow \tau}{\Gamma \vdash \lambda x.\, e \downarrow \sigma \to \tau} \text{ (Lam)}$$

To type an application $e_1 e_2$:

$$\frac{\Gamma \vdash e_1 \uparrow \sigma \to \tau \qquad \Gamma \vdash e_2 \downarrow \sigma}{\Gamma \vdash e_1 e_2 \uparrow \tau} \text{ (App)}$$

Example: $(\lambda x.\, x * 2)$ `:` $\texttt{int} \to \texttt{int}$

[3] Ow.

$$\frac{x{:}\texttt{int} \vdash x * 2 \downarrow \texttt{int}}{\vdash \lambda x.\, x * 2 \downarrow \texttt{int} \to \texttt{int}}$$

We type the body of the λ as follows: the body is an application, so we want to use the (App) rule, but (App) is an inference judgment and we are trying to derive a checking judgment. So we use the subsumption rule.

$$\frac{\dfrac{x{:}\texttt{int} \vdash x * 2 \uparrow \qquad\qquad \leq \texttt{int}}{x{:}\texttt{int} \vdash x * 2 \downarrow \texttt{int}}}{\vdash \lambda x.\, x * 2 \downarrow \texttt{int} \to \texttt{int}}$$

Then we use a rule for $*$ (write it!), which infers its result and checks its arguments, analogous to (App).

$$\frac{\dfrac{\dfrac{x{:}\texttt{int} \vdash x{:}\texttt{int} \quad x{:}\texttt{int} \vdash 2{:}\texttt{int}}{x{:}\texttt{int} \vdash x * 2 \uparrow \texttt{int}} \qquad \texttt{int} \leq \texttt{int}}{x{:}\texttt{int} \vdash x * 2 \downarrow \texttt{int}}}{\vdash \lambda x.\, x * 2 \downarrow \texttt{int} \to \texttt{int}}$$

Example:

$$\texttt{fun}\ \mathit{mapdouble}(\ell)\ \texttt{is}\ \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell\ \texttt{end} : \texttt{intlist} \to \texttt{intlist}$$

Let $\Gamma = \mathit{intmap} : (\texttt{int} \to \texttt{int}) \to \texttt{intlist} \to \texttt{intlist},\ \ \ell : \texttt{intlist}$.

$$\frac{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \uparrow \qquad\qquad \leq \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \downarrow \texttt{intlist}}$$

The next step is to use (App). The function is $\mathit{intmap}\ (\lambda x.\, x * 2)$ and we're applying it to ℓ, so we need to infer a type for $\mathit{intmap}\ (\lambda x.\, x * 2)$.

$$\frac{\dfrac{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2) \uparrow \qquad \Gamma \vdash \ell \downarrow}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \uparrow} \qquad\qquad \leq \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \downarrow \texttt{intlist}}$$

But $\mathit{intmap}\ (\lambda x.\, x * 2)$ is also an application, so we use (App) again.

$$\frac{\dfrac{\dfrac{\Gamma \vdash \mathit{intmap} \uparrow \qquad \Gamma \vdash (\lambda x.\, x * 2) \downarrow}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2) \uparrow} \qquad \Gamma \vdash \ell \downarrow}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \uparrow} \qquad\qquad \leq \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \downarrow \texttt{intlist}}$$

Now we have to infer a type for *intmap*, but we can always infer a type for a variable by using (Var). The first argument to *intmap* has type $\texttt{int} \to \texttt{int}$, so we must check $\lambda x.\, x * 2$ against $\texttt{int} \to \texttt{int}$—which we just did in the preceding example.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathit{intmap} \uparrow \ldots \qquad \Gamma \vdash (\lambda x.\, x * 2) \downarrow \texttt{int} \to \texttt{int}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2) \uparrow \texttt{intlist} \to \texttt{intlist}} \qquad \Gamma \vdash \ell \downarrow}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \uparrow} \qquad \leq \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \downarrow \texttt{intlist}}$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathit{intmap} \uparrow \ldots \quad \Gamma \vdash (\lambda x.\, x * 2) \downarrow \texttt{int} \to \texttt{int}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2) \uparrow \texttt{intlist} \to \texttt{intlist}} \quad \Gamma \vdash \ell \downarrow \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \uparrow \texttt{intlist}} \qquad \texttt{intlist} \leq \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \downarrow \texttt{intlist}}$$

4.1 Let

$$\dfrac{\Gamma \vdash e_1 \uparrow \sigma \qquad \Gamma, x{:}\sigma \vdash e_2 \downarrow \tau}{\Gamma \vdash \texttt{let}(e_1, x.e_2) \downarrow \tau}\ \text{(Let)}$$

4.2 Sums

So it seems to work nicely for functions. Let's look at sums, which were annoying without bidirectional typing because (for example) $\texttt{inl}(5)$ doesn't have a unique type. Since it doesn't have a unique type, we need to check it against a type rather than try to infer a type.

$$\dfrac{\Gamma \vdash e \downarrow \tau_1}{\Gamma \vdash \texttt{inl}(e) \downarrow \tau_1 + \tau_2}\ \text{(Inl)} \qquad \dfrac{\Gamma \vdash e \downarrow \tau_2}{\Gamma \vdash \texttt{inr}(e) \downarrow \tau_1 + \tau_2}\ \text{(Inr)}$$

$$\dfrac{\Gamma \vdash e \uparrow \tau_1 + \tau_2 \qquad \Gamma, x_1{:}\tau_1 \vdash e_1 \downarrow \sigma \qquad \Gamma, x_2{:}\tau_2 \vdash e_2 \downarrow \sigma}{\Gamma \vdash \texttt{case}(e, x_1.e_1, x_2.e_2) \downarrow \sigma}\ \text{(Case)}$$

A peculiarity of bidirectional typing is that it doesn't work for many contrived programs. For example, $\texttt{case}(\texttt{inl}(5), x_1.0, x_2.1)$ `:` $\texttt{int}$ is not well-typed unless we annotate $\texttt{inl}(5)$. But no real program would create a sum and immediately take it apart. In practice, one almost always does a case on a variable, or on some function application—in which cases we can infer the type, and need no annotation.

Moreover, whenever an expression appears as the body of a function, we check it against the (usually annotated) result type of the function. So we can return an injection from a function with no additional type annotations, beyond the type annotation for the function. And, as we saw in the *intmap* example, sometimes we don't even need to annotate the function.

Before looking at how bidirectional typing behaves with other types, let's consider the formal properties of the system.

4.3 Soundness and Completeness

When we examine bidirectionality in connection with the dynamic semantics, several questions arise. The first is: How should preservation and progress be formulated? Perhaps we could formulate preservation as

(1) If $\vdash e \uparrow \tau$ and $e \mapsto e'$ then $\vdash e' \uparrow \tau$
(2) If $\vdash e \downarrow \tau$ and $e \mapsto e'$ then $\vdash e' \downarrow \tau$

But this becomes very messy; even proving that $(\lambda x.e)v \mapsto \{v/x\}e$ preserves types is nasty. The type τ of $(\lambda x.e)v$ is inferred, so we have case (1), but the premise of (Lam) is a checking judgment, so we don't have $x{:}\sigma \vdash e \uparrow \tau$ which would lead (by substitution) to $\{v/x\}e \uparrow \tau$.

Besides, e may have type annotations. While we can certainly write a rule

$$\frac{}{(e : \tau) \mapsto e}$$

it doesn't remotely correspond to any reasonable model of computation.

The formulation

If $\vdash e \uparrow \tau$ or $\vdash e \downarrow \tau$ and $e \mapsto e'$ then $\vdash e' \uparrow \tau$ or $\vdash e' \downarrow \tau$

might be correct, but it suggests that we don't actually care about the direction. Which is indeed the case: we use bidirectionality so we can write the typechecker; it has nothing to do with running the program. Indeed, since type annotations are (in some cases) essential to bidirectional typing, and types should not matter at runtime, there seems to be a gulf between bidirectional typing and dynamic semantics.

Since we know how to state (and prove) preservation and progress for a non-bidirectional type system, why not have a non-bidirectional system as well, and show that we can get from a bidirectional typing to a typing in that system? The non-bidirectional system I have in mind is simply the bidirectional system without (Anno) and with all $\uparrow$, $\downarrow$ changed to $:$. Formally:

Theorem 1 (Soundness). *If $\vdash e \downarrow \tau$ or $\vdash e \uparrow \tau$ then $\vdash |e| : \tau$ where $|e|$ is e with all type annotations erased.*

Proving this is straightforward (after generalizing to an arbitrary context Γ). I call it *soundness* because it says that the bidirectional system is

sound with respect to the non-bidirectional system: anything derivable in the first is derivable in the second (after erasing type annotations).

Do we have *completeness*? No. As a counterexample,

$$\texttt{case}(\texttt{inl}(5), x_1.0, x_2.1)$$

is well-typed in the non-bidirectional system, but not in the bidirectional system.

4.4 Polymorphism

$$\frac{\Gamma, t\ \mathsf{type} \vdash e \downarrow \sigma}{\Gamma \vdash \mathtt{Fun}(t.e) \downarrow \forall t.\sigma}\ \text{(Typefun)} \qquad \frac{\Gamma \vdash e \uparrow \forall t.\sigma \qquad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma \vdash \mathtt{Inst}(e, \tau) \uparrow \{\tau/t\}\sigma}\ \text{(Inst)}$$

Exercise: derive

$$\vdash \mathtt{Fun}(t.\lambda x.\, x) \downarrow \forall t.t \to t$$

4.5 Other Type Constructors

See the Assignment 6 handout.