CPSC 311: Bidirectional typing: implementation and polymorphism **(DRAFT)** (“lec-bidir-poly”)

Joshua Dunfield
University of British Columbia

November 23, 2016

Overview: These notes cover the following.

- Starting with the rules on the last page of (the updated version of) lec-bidir.pdf, we replace pos, int, and rat, and also add subtyping—just by changing the second premise of Check-sub from $A = B$ to A `<:` B.
- Polymorphism in languages like SML.
- First steps in adding polymorphism to Fun.

Reminder: Bidirectional typing replaces $\Gamma \vdash e : A$ with two different judgments:

$\Gamma \vdash e \Rightarrow A$ read “under assumptions in Γ, the expression e synthesizes type A”

$\Gamma \vdash e \Leftarrow A$ read “under assumptions in Γ, the expression e checks against type A”

The difference between these judgments 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, kind of like we did in `typeof`. 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.

1 Typing implemented by `bidir-2.rkt`

Types: pos, int, rat, bool, $A1 * A2$, $A1 \to A2$

$$\frac{\Gamma(x) = A}{\Gamma \vdash (\mathsf{Id}\ x) \Rightarrow A}\ \text{Synth-var} \qquad \frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}\ \text{Check-sub} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (\mathsf{Anno}\ e\ A) \Rightarrow A}\ \text{Synth-anno}$$

$$\frac{\Gamma \vdash e1 \Rightarrow A \to B \qquad \Gamma \vdash e2 \Leftarrow A}{\Gamma \vdash (\mathsf{App}\ e1\ e2) \Rightarrow B}\ \text{Synth-app} \qquad \frac{x : A1, \Gamma \vdash e \Leftarrow A2}{\Gamma \vdash (\mathsf{Lam}\ x\ e) \Leftarrow A1 \to A2}\ \text{Check-lam}$$

$$\frac{u : A, \Gamma \vdash e \Leftarrow A}{\Gamma \vdash (\mathsf{Rec}\ u\ e) \Leftarrow A}\ \text{Check-rec} \qquad \frac{\Gamma \vdash e1 \Rightarrow A1 \qquad x : A1, \Gamma \vdash e2 \Leftarrow A}{\Gamma \vdash (\mathsf{Let}\ x\ e1\ e2) \Leftarrow A}\ \text{Check-let}$$

$$\frac{n \in \mathbb{Z} \qquad n \geq 0}{\Gamma \vdash (\mathsf{Num}\ n) \Rightarrow \mathsf{pos}}\ \text{Synth-pos} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash (\mathsf{Num}\ n) \Rightarrow \mathsf{int}}\ \text{Synth-int} \qquad \frac{n \in \mathbb{Q}}{\Gamma \vdash (\mathsf{Num}\ n) \Rightarrow \mathsf{rat}}\ \text{Synth-rat}$$

$$\frac{op : A1 * A2 \to B \qquad \Gamma \vdash e1 \Leftarrow A1 \qquad \Gamma \vdash e2 \Leftarrow A2}{\Gamma \vdash (\mathsf{Binop}\ op\ e1\ e2) \Rightarrow B}\ \text{Synth-binop}$$

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

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

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

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

$$\frac{\Gamma \vdash e1 \Rightarrow A1 \qquad \Gamma \vdash e2 \Rightarrow A2}{\Gamma \vdash (\mathsf{Pair}\ e1\ e2) \Rightarrow (A1 * A2)}\ \text{Synth-pair}$$

2 What is polymorphism?

In a language with polymorphism (*poly* = many; *morph* = form), some features of the language can operate with *multiple types*. "Some features" and "can operate with" are deliberately vague: there are many kinds of polymorphism, and a given language might allow one kind for some language fatures, under some circumstances, and another kind of polymorphism in others.

3 Kinds of polymorphism

In 1967, Christopher Strachey (who made important contributions to programming language semantics, *and* designed a key ancestor of C) distinguished two kinds of polymorphism:

- parametric polymorphism, and
- *ad hoc* polymorphism.

A further kind of polymorphism, perhaps the kind you've used the most, is *subtype polymorphism*, also called *inclusion polymorphism*. For example, if you have a pair of type $\mathsf{pos} * \mathsf{pos}$, you should be able to pass it to a function of type $(\mathsf{rat} * \mathsf{rat}) \to \mathsf{bool}$.

3.1 Examples of parametric polymorphism

In parametric polymorphism, types include *type variables* that can be *instantiated*.

(see poly.sml)

To understand these types, we should really write the *quantifiers* that SML (implicitly) puts around these types. For example, `identity_function` has type

$$\forall\alpha.\,(\alpha \to \alpha) \qquad \text{“for all types } \alpha, \ldots\text{”}$$

That is, any code that calls `identity_function` can provide something of any type it chooses, and will (if evaluation results in a value!) get back something of that same type.

```
identity_function 5;
identity_function (1, 2);
```

In the first line above, 5 has SML type `int`, so SML *instantiates* α with `int`, resulting in the type

$$(\mathtt{int} \to \mathtt{int})$$

Applying a function of type $(\mathtt{int} \to \mathtt{int})$ to an `int` results in an `int`, so `identity_function 5` has type `int`.

A larger example is `map_list`, which has the polymorphic type

$$\forall\alpha.\,\big(\forall\beta.\,(\alpha \to \beta) \to (\alpha\ \mathtt{list}) \to (\beta\ \mathtt{list})\big)$$

This type says: if you pick types α and β (which, like meta-variables in typing rules, might or might not be *different* types), and pass (first) a function of type $\alpha \to \beta$ and (second) a list whose elements all have type α, then the value returned by calling `map_list` (if that call returns at all) will be a list whose elements are of type β.

(illustrate with `map_list` `make_pair` from poly.sml)

The reason this is called *parametric* polymorphism is that the types α and β don't matter: the implementation of `map_list` doesn't care what types you instantiate α and β with. In fact, in SML it is *impossible* for `map_list` to know which types α and β have been instantiated with!

If you try to do something that depends on α having a particular type, SML will infer a "less polymorphic" type instead:

```
val unpoly_map_list = fn : (bool -> 'b) -> bool list -> 'b list
```

The fact that a parametrically polymorphic function *cannot* inspect its argument's type means that we can prove "parametricity properties", such as:

> If a function has type $\forall\alpha.\ (\alpha \rightarrow \alpha)$, and it is applied to a value v of some type A, and that application evaluates to a value, then the resulting value *is exactly* v.

Or, suppose a function has type $\forall\alpha.\ \big((\alpha * \alpha) \rightarrow \alpha\big)$. It could return the first part of the pair, or the second part. Could it do anything else?

Turning the question around (sideways?): What functions *besides* `map_list` have `map_list`'s type?

3.2 Examples of ad hoc polymorphism

A common form of *ad hoc* polymorphism is *operator overloading*: in many languages, the + operator works on more than one type of argument. For example, in SML, + works on both `int`s and `real`s (though not on `string`, and not on one `int` and one `real`).

3.3 Polymorphism in untyped languages

Is Racket polymorphic? The answer depends on whether we take "type" in the (vague) definition above to mean a static type (perhaps defined through typing rules), or whether we consider it more informally, so that, say, `3` and `#false` in Racket are of different types, even though Racket has no type system to stop you from compiling a program like `(+ 3 #false)`.

- If we require "type" to mean a static type, then Racket is not polymorphic because, in a sense, it has *only one type*: the type of "s-expressions", which includes numbers, `#true` and `#false`, functions (`lambda`), lists, and everything else.

  This claim is sometimes phrased as "dynamic 'typing' is *really* just *unityping*", a "unityped" language being a (statically) typed language with only one (*uni*-) type. Thus, Carnegie Mellon University's Bob Harper:

> "Dynamic typing is but a special case of static typing, one that limits, rather than liberates... Something can hardly be *opposed* to that of which it is but a trivial special case." (from a 2011 blog post)

- If we say that *any* precise organization of code and/or data into subcategories is "typing", then `#true` and `#false` can be called "booleans", `(`**lambda**` (x) x)` can be called a "function", and so on. Then Racket is certainly polymorphic, because many functions that you can write in Racket—for example, `(`**lambda**` (x) x)`—work on many different kinds of Racket "types".

4 Polymorphism in Fun

To add polymorphism to Fun, we need to add two new forms of type:

- *Polymorphic types*

$$\forall \alpha.\, B$$

  which are read "for all [types] α, ...".

- *Type variables* α, b, c, etc.

These new forms are meant to be used together. For example, the following will be the type of the identity function $(\mathsf{Lam}\ x\ (\mathsf{Id}\ x))$.

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

The main idea is that, whenever an expression has a polymorphic type, it can be *instantiated* by "plugging in" different types. For example, if we plug in bool for α in the type above, we get

$$\mathsf{bool} \to \mathsf{bool}$$

Similar to substitution-based evaluation, which substitutes a value v for $(\mathsf{Id}\ x)$ throughout an expression e

$$[v/x]e$$

we are substituting the type bool for the type variable α throughout a type:

$$\begin{aligned} &[A/\alpha]B \\ [\mathsf{bool}/\alpha](\alpha \to \alpha) &= ([\mathsf{bool}/\alpha]\alpha) \to ([\mathsf{bool}/\alpha]\alpha) \\ &= \mathsf{bool} \to \mathsf{bool} \end{aligned}$$

We'll also need a new form of assumption in our contexts:

$$\begin{array}{lcll} \Gamma & ::= & \emptyset & \text{Empty context} \\ & \mid & x : A, \Gamma & \text{Context } \Gamma \text{ plus the assumption that variable } x \text{ is of type } A \\ & \mid & \alpha\ \mathsf{type}, \Gamma & \text{Context } \Gamma \text{ plus the assumption that } \alpha \text{ is any type} \end{array}$$

4.1 Typing rules

$$\frac{\alpha\ \mathsf{type}, \Gamma \vdash e \Leftarrow A}{\Gamma \vdash (\mathsf{All}\ \alpha\ e) \Leftarrow \forall \alpha.\, A}\ \text{Check-all} \qquad \frac{\Gamma \vdash e \Rightarrow \forall b.\, B1}{\Gamma \vdash (\mathsf{At}\ e\ A) \Rightarrow [A/b]B1}\ \text{Synth-at}$$