CPSC 311: Bidirectional typing **(DRAFT)** ("lec-bidir")

Joshua Dunfield
University of British Columbia

November 22, 2016

Remark. Parts 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 Let expression; in versions that have Let*, each binding in a Let* could be considered a declaration.

Also, these notes consistently use "variable" rather than "identifier".

Finally, the explanation of the "subsumption rule" is a little out of place, because at McGill, I introduced bidirectional typing *before* subtyping.

1 Introduction

When we started doing typing, we encountered the rule

$$\frac{x : A, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{Lam}\ x\ e) : A \to B}\ \text{?Type-lam}$$

which we couldn't implement, because to make the recursive call to `typeof`, we needed to extend the typing context Γ (`tc`) with $x : A$, but we didn't know what A was.

As a workaround, we added the type A to the syntax of Lam:

$$\frac{x : A, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{Lam}\ x\ A\ e) : A \to B}\ \text{?Type-lam}$$

Here, the A in $(\mathsf{Lam}\ x\ A\ e)$ is a *type annotation*. Many typed languages do not force you to write A in this situation. One technique for doing that is *type inference*, used in ML, OCaml, Haskell, and other languages (even C++, which now has an `auto` keyword). In these languages, you still need to write types in some places, such as module interfaces and some uses of references.

Not having to write type annotations has some drawbacks. 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 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 it's obvious.

Bidirectional typing lies between the extremes of type inference and mainstream type checking. 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 relatively simple type systems but then "flames out", bidirectional typing 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 main idea: Instead of persisting in trying to figure out the type of an expression on its own as `typeof` does, we alternate between figuring out or *synthesizing* types and *checking* expressions against types we already know.

In terms of judgments, bidirectionality replaces the 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:

$\Gamma \vdash e \Rightarrow A$ 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.

3 Typing rules

Two ideas will help us design the rules for deriving bidirectional typing judgments:

(1) We can't use information we don't have.

(2) We should 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.

$$\begin{array}{rcll} \Gamma & ::= & \emptyset & \text{Empty context} \\ & | & x : A, \Gamma & \text{Context } \Gamma \text{ plus the assumption that variable } x \text{ is of type } A \end{array}$$

And now, the rule for typing variables. It says that if we have assumed x to have type A, because $x : A$ is given in Γ, then x synthesizes type A.

$$\frac{\Gamma(x) = A}{\Gamma \vdash (\mathsf{Id}\ x) \Rightarrow A}\ \text{Synth-var}$$

3.1 Functions

If we Apply a function, we need the type of the function. But when we create a function by writing (Lam x e), we don't (yet) know that type! So the rule for applications e1 e2 needs to synthesize the function type *from* the function e1.

On the other hand, in the rule for (Lam x e) we don't yet know what the domain or range of the function should be, so (following observation (1)) we check (Lam x e) against a type that is (somehow) already known.

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

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.

$$\frac{\dfrac{(\ldots)(g) = \mathsf{bool} \to \mathsf{num}}{g : \mathsf{bool} \to \mathsf{num}, \emptyset \vdash (\mathsf{Id}\ g) \Rightarrow \mathsf{bool} \to \mathsf{num}}\ \text{Synth-var} \qquad g : \mathsf{bool} \to \mathsf{num}, \emptyset \vdash (\mathsf{Bfalse}) \Leftarrow \mathsf{bool}}{g : \mathsf{bool} \to \mathsf{num}, \emptyset \vdash (\mathsf{App}\ (\mathsf{Id}\ g)\ (\mathsf{Bfalse})) \Rightarrow \mathsf{num}}\ \text{Synth-app}$$

Or, if the function being applied is itself a function application, as in

$$\big(\mathsf{App}\ (\mathsf{App}\ (\mathsf{Id}\ \mathit{twice})\ (\mathsf{Id}\ \mathsf{f}))\ (\mathsf{Id}\ \mathsf{x})\big)$$

(where *twice*, which applies its first argument to its second argument twice, has type (num → num) → num → 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

$$\big(\mathsf{App}\ (\mathsf{App}\ (\mathsf{Id}\ \mathit{twice})\ (\mathsf{Lam}\ \mathsf{y}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{y})\ (\mathsf{Id}\ \mathsf{y}))))\ (\mathsf{Id}\ \mathsf{x})\big)$$

because in Synth-app, we check the argument e2 = (Lam y (Add (Id y) (Id y))) against the domain A = (num → num), which is the type that Check-lam checks the lam against. Here is the derivation, where

$$\Gamma = \mathit{twice} : \underbrace{((\mathsf{num} \to \mathsf{num}) \to \mathsf{num} \to \mathsf{num})}_{A\mathit{twice}}, \mathsf{x} : \mathsf{num}$$

$$\frac{\dfrac{\Gamma \vdash \mathit{twice} \Rightarrow A\mathit{twice}}{\Gamma \vdash (\mathsf{App}\ (\mathsf{Id}\ \mathit{twice})\ (\mathsf{Lam}\ \mathsf{y}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{y})\ (\mathsf{Id}\ \mathsf{y})))) \Rightarrow (\mathsf{num} \to \mathsf{num})}\ \text{Synth-var} \qquad \dfrac{\dfrac{\vdots}{\mathsf{y} : \mathsf{num}, \Gamma \vdash (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{y})\ (\mathsf{Id}\ \mathsf{y})) \Rightarrow \mathsf{num} \quad \mathsf{num}{=}\mathsf{num}}\ \text{Check-sub}}{\mathsf{y} : \mathsf{num}, \Gamma \vdash (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{y})\ (\mathsf{Id}\ \mathsf{y})) \Leftarrow \mathsf{num}}\ \text{Check-lam}}{\Gamma \vdash (\mathsf{Lam}\ \mathsf{y}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{y})\ (\mathsf{Id}\ \mathsf{y}))) \Leftarrow (\mathsf{num} \to \mathsf{num})}\ \text{Synth-app} \qquad \frac{\dfrac{\Gamma \vdash (\mathsf{Id}\ \mathsf{x}) \Rightarrow \mathsf{num}}{\Gamma \vdash (\mathsf{Id}\ \mathsf{x}) \Leftarrow \mathsf{num}}\ \text{Synth-var} \quad \mathsf{num}{=}\mathsf{num}}{\Gamma \vdash (\mathsf{Id}\ \mathsf{x}) \Leftarrow \mathsf{num}}\ \text{Check-sub}$$

$$\Gamma \vdash (\mathsf{App}\ (\mathsf{App}\ (\mathsf{Id}\ \mathit{twice})\ (\mathsf{Lam}\ \mathsf{y}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{y})\ (\mathsf{Id}\ \mathsf{y}))))\ (\mathsf{Id}\ \mathsf{x})) \Rightarrow \mathsf{num}\ \text{Synth-app}$$

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

$$\big(\mathsf{App}\ (\mathsf{Lam}\ \mathsf{y}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{y})\ (\mathsf{Id}\ \mathsf{y})))\ (\mathsf{Num}\ 5)\big)$$

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

$$\frac{\emptyset \vdash (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y))) \not\Rightarrow \qquad \ldots}{\emptyset \vdash \big(\mathsf{App}\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))\ (\mathsf{Num}\ 5)\big) \not\Rightarrow}\ \text{Synth-app}$$

This restriction is inconvenient for the small examples we often use in 311, but it's not very inconvenient in practice: real code seldom applies a Lam immediately in this way.

3.2 "Subsumption"

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

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

3.3 Recursive expressions and typing annotations

$$\frac{u : B, \Gamma \vdash e \Leftarrow B}{\Gamma \vdash (\mathsf{Rec}\ u\ e) \Leftarrow B}\ \text{Check-rec} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (\mathsf{Anno}\ e\ A) \Rightarrow A}\ \text{Synth-anno}$$

Annotations allow us to turn expressions that don't synthesize a type into expressions that do, by writing the type ourselves. Recall the expression $(\mathsf{App}\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))\ (\mathsf{Num}\ 5))$, which doesn't synthesize a type because $(\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))$ doesn't synthesize. Using an annotation, we can readily synthesize the expression's type:

$$\frac{\dfrac{\dfrac{y : \mathsf{num} \vdash (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)) \Leftarrow \mathsf{num}}{\emptyset \vdash (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y))) \Leftarrow \mathsf{num}\to\mathsf{num}}\ \text{Check-lam}}{\emptyset \vdash \big(\mathsf{Anno}\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))\ \mathsf{num}\to\mathsf{num}\big) \Rightarrow \mathsf{num}\to\mathsf{num}}\ \text{Synth-anno} \qquad \dfrac{\dfrac{}{\emptyset \vdash (\mathsf{Num}\ 5) \Rightarrow \mathsf{num}}\ \text{Synth-num} \qquad \mathsf{num} = \mathsf{num}}{\emptyset \vdash (\mathsf{Num}\ 5) \Leftarrow \mathsf{num}}\ \text{Check-sub}}{\emptyset \vdash \Big(\mathsf{App}\ \big(\mathsf{Anno}\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))\ \mathsf{num}\to\mathsf{num}\big)\ (\mathsf{Num}\ 5)\Big) \Rightarrow \mathsf{num}}\ \text{Synth-app}$$

3.4 Primitive operations

The typing rules for Add and Sub work similarly to the rules for Synth-app, if we think of the operator being applied as a function whose type is known, and the two expressions $e1$ and $e2$ as its arguments.

$$\frac{\Gamma \vdash e1 \Leftarrow \mathsf{num} \qquad \Gamma \vdash e2 \Leftarrow \mathsf{num}}{\Gamma \vdash (\mathsf{Add}\ e1\ e2) \Rightarrow \mathsf{num}}\ \text{Synth-add} \qquad \frac{\Gamma \vdash e1 \Leftarrow \mathsf{num} \qquad \Gamma \vdash e2 \Leftarrow \mathsf{num}}{\Gamma \vdash (\mathsf{Sub}\ e1\ e2) \Rightarrow \mathsf{num}}\ \text{Synth-sub}$$

3.5 Booleans

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

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

3.6 Pairs

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

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

3.7 Let

In the expression

$$\Big(\mathsf{Let}\ x\ \big(\mathsf{App}\ (\mathsf{Id}\ \mathtt{fact})\ (\mathsf{Num}\ 5)\big)\ \big(\mathsf{Pair}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)\big)\Big)$$

we should be able to figure out (assuming our context Γ contains the typing $\mathtt{fact} : \mathsf{num} \to \mathsf{num}$) that $(\mathsf{Id}\ x)$ has type num, and therefore $\big(\mathsf{Pair}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)\big)$ checks against $\mathsf{num} * \mathsf{num}$.

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

3.8 Adding more convenience

The rules above can be criticized for requiring too many annotations. For example, even if the body of a Let does synthesize a type, Check-let 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 Lets *can* be checked, such as a Let 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.

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

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

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

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

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 well for very small functional languages. But extending type inference to support a more powerful type system often constitutes a research project in itself!

In particular, type inference has a lot of trouble with subtyping. (Some of the hostility of functional programmers to object-oriented languages may be "sour grapes": typed 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 didn't make sense, because it didn't do any subtyping!) to use $<:$ instead of $=$:

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

This avoids the problem 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.

It also avoids other difficulties we've seen with Type-sub. For example, in bidirectional typing, there is no need for an `upper-bound` function.

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

5 Typing implemented by `bidir-1.rkt`

Types: num, bool, $A1 * A2$, $A1 \to A2$

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

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

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

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

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

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

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

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

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