Lecture Notes on Products

15-814: Types and Programming Languages
Frank Pfenning

Lecture 7
Tuesday, September 25, 2019

1 Introduction

Types capture fundamental programming abstractions. If a type system and its underlying programming language is well-designed, we can then build complex data representations and computational mechanisms from a few primitives. The most fundamental is that of a function, captured in the type $\tau_1 \to \tau_2$. As a next step we look for ways to *aggregate* data. The simplest is *pairs*, which are captured by the type $\tau_1 \times \tau_2$. By iterating pairs we can then assemble tuples with elements of arbitrary types.

2 Constructing Pairs

Fundamentally, for each new type we introduce we must be able to *construct* element of the type. For example, $\lambda x.\, e$ constructs element of the function type $\tau_1 \to \tau_2$. To construct new elements of the type $\tau_1 \times \tau_2$ we use the almost universal notation $\langle e_1, e_2 \rangle$. The typing rule is straightforward

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ \mathsf{pair}$$

This is the only rule for pairs, so we maintain the property that the rules are syntax-directed.

Next we should consider the *dynamics*, that is, which are the new values of type $\tau_1 \times \tau_2$ and how do we evaluate pairs. In this lecture we consider

eager pairs, that is, a pair is only a value if both components are. *Lazy pairs* are the subject of Exercise 1.

$$\frac{e_1\ \mathit{val} \quad e_2\ \mathit{val}}{\langle e_1, e_2\rangle\ \mathit{val}}\ \mathsf{val/pair}$$

We then assume that we can *observe* the components of a pair. So, at the current extent of our language we can observe the Booleans and, inductively, pairs of observable type.

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \mathsf{bool} \mid \tau_1 \times \tau_2 \\ \text{Observable Types} & o & ::= & \mathsf{bool} \mid o_1 \times o_2 \end{array}$$

To *evaluate* a pair we decided on evaluating from left to right: it preserves single-step determinacy and it is consistent with other constructs like function applications that are also evaluate from left to right.

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle}\ \mathsf{step/pair}_1 \qquad \frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{\langle v_1, e_2\rangle \mapsto \langle v_1, e_2'\rangle}\ \mathsf{step/pair}_2$$

In writing this rule we are starting a convention where expressions known to be values are denoted by v instead of e.

3 Destructing Pairs

Constructing pairs is only one side of the coin. We also need to be able to access the components of a pair. There seem to be two natural choices: (1) to have a first and second projection function, and (2) decompose a pair with a *letpair*-like construct (from the pure λ-calculus in Lecture L2.4) that gives access to both components. It turns out, projections as a primitive are more suitable for lazy pairs, while a *letpair* construct matches eager pairs. We formulate it here as a *case* expression, because it turns out that several other destructors can also be written in this way, leading to a more uniform language.

$$\mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e')$$

The crucial operational rule just deconstructs a pair of values.

$$\frac{v_1\ \mathit{val} \quad v_2\ \mathit{val}}{\mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto [v_1/x_2][v_2/x_2]e_3}\ \mathsf{step/case/pair}$$

We also need a second rule to reduce the subject of the case-expression until it becomes a value.

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto \mathsf{case}\ e_0'\ (\langle x_1, x_2\rangle \Rightarrow e_3)}\ \mathsf{step/case/pair}_0$$

In the typing rule, we know the subject of the case-expression should be a pair and the body should be the same type as the whole expression.

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{\Gamma \vdash \mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e') : \tau'}\ \mathsf{case/pair}$$

Note how x_1 and x_2 are added to the context in the second premise because they may appear in e'.

We are of course obligated to check that our language properties are preserved under this extension, which we will do shortly. Meanwhile, let's write two small programs, verifying that the projections can indeed be defined.

$$\begin{array}{lcl} \mathit{fst} & : & (\alpha \times \beta) \to \alpha \\ \mathit{fst} & = & \lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow x) \\ \mathit{snd} & : & (\alpha \times \beta) \to \beta \\ \mathit{snd} & = & \lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow y) \end{array}$$

Because the typing is parametric in the variables α and β, the two projections also have types $(\tau \times \sigma) \to \tau$ and $(\tau \times \sigma) \to \sigma$, respectively, for arbitrary τ and σ.

4 Summary

Here is a summary of our language so far, and the new rules defining it.

4.1 Syntax

$$\begin{array}{llcll} \text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \mathsf{bool} \mid \tau_1 \times \tau_2 & \\ \text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 & (\tau_1 \to \tau_2) \\ & & \mid & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ e_2\ e_3 & (\mathsf{bool}) \\ & & \mid & \langle e_1, e_2\rangle \mid \mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e') & (\tau_1 \times \tau_2) \end{array}$$

4.2 Statics

$$\frac{e_1 \; \mathit{val} \quad e_2 \; \mathit{val}}{\langle e_1, e_2 \rangle \; \mathit{val}} \; \mathsf{val/pair}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \; \mathsf{pair} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{\Gamma \vdash \mathsf{case} \; e \; (\langle x_1, x_2 \rangle \Rightarrow e') : \tau'} \; \mathsf{case/pair}$$

4.3 Dynamics

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle} \; \mathsf{step/pair}_1 \qquad \frac{v_1 \; \mathit{val} \quad e_2 \mapsto e_2'}{\langle v_1, e_2 \rangle \mapsto \langle v_1, e_2' \rangle} \; \mathsf{step/pair}_2$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case} \; e_0 \; (\langle x_1, x_2 \rangle \Rightarrow e_3) \mapsto \mathsf{case} \; e_0' \; (\langle x_1, x_2 \rangle \Rightarrow e_3)} \; \mathsf{step/case/pair}_0$$

$$\frac{v_1 \; \mathit{val} \quad v_2 \; \mathit{val}}{\mathsf{case} \; \langle v_1, v_2 \rangle \; (\langle x_1, x_2 \rangle \Rightarrow e_3) \mapsto [v_1/x_1][v_2/x_2]e_3} \; \mathsf{step/case/pair}$$

5 Preservation and Progress, Revisited

Design of the new types and expressions are always carefully rigged so that the preservation and progress theorems continue to hold. Among other things, we make sure that each definition is self-contained. For example, we might have postulated a *primitive function* $\mathsf{pair} : \tau_1 \to (\tau_2 \to (\tau_1 \times \tau_2))$ but then the canonical forms theorem would have to be altered: not every value of function type is actually a λ-expression. Instead, we have a new *expression constructor* $\langle -, - \rangle$ and we can *define pair* as a regular function from that.

Theorem 1 (Type Preservation, new cases for $\tau_1 \times \tau_2$)
If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$

Proof: Recall the structure of the proof of type preservation. We use rule induction on the derivation of $e \mapsto e'$ and apply inversion on $\cdot \vdash e : \tau$ in order to gain enough information to assemble a derivation of e'. We exploit here that the typing rules are syntax-directed. Technically, we rely on the substitution property and so that needs to be extended as well. But since we continue to use a standard hypothetical judgment and we do not touch our notion of variable, the new cases don't require any particular attention.

The congruence cases of reduction, where we reduce a subexpression, are straightforward because we can follow this pattern mechanically. For example:

Case:

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle}\ \mathsf{step/pair}_1$$

where $e = \langle e_1, e_2\rangle$, $e' = \langle e_1', e_2\rangle$.

$\cdot \vdash \langle e_1, e_2\rangle : \tau$	Assumption
$\cdot \vdash e_1 : \tau_1$ and $\cdot \vdash e_2 : \tau_2$ where $\tau = \tau_1 \times \tau_2$.	By inversion
$\cdot \vdash e_1' : \tau_1$	By ind. hyp.
$\cdot \vdash \langle e_1', e_2\rangle : \tau_1 \times \tau_2$	By rule pair

The main case to check then is one where some "real" reduction takes place. This is when a destructor for values of a type meets a constructor.

Case:

$$\frac{v_1\ \mathit{val} \quad v_2\ \mathit{val}}{\mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto [v_1/x_1][v_2/x_2]e_3}\ \mathsf{step/case/pair}$$

where $e = \mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e_3)$ and $e' = [v_1/x_2][v_2/x_2]e_3$. In this case, we cannot apply the induction hypothesis (the premises are of a different form), but we can nevertheless apply inversion and then use the substitution property.

$\cdot \vdash \mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e_3) : \tau$	Assumption
$\cdot \vdash \langle v_1, v_2\rangle : \tau_1 \times \tau_2$ and $x_1 : \tau_1, x_2 : \tau_2 \vdash e_3 : \tau$ for some τ_1 and τ_2	By inversion
$\cdot \vdash v_1 : \tau_1$ and $\cdot \vdash v_2 : \tau_2$	By inversion
$x_1 : \tau_1 \vdash [v_2/x_2]e_3 : \tau$	By substitution (Theorem L6.1)
$\cdot \vdash [v_1/x_1][v_2/x_2]e_3 : \tau$	By substitution (Theorem L6.1)

□

In preparation for the progress theorem, we extend the canonical forms theorem. The latter is a bit different than the other theorems in that for every extension of our language by a new form of type, we need to add a case that characterizes the values of the new type.

Theorem 2 (Canonical Forms)
Assume v val. Then

(i) If $\cdot \vdash v : \tau_1 \to \tau_2$ then $v = \lambda x.\, e'$ for some x and e'.

(ii) If $\cdot \vdash v : \mathsf{bool}$ then $v = \mathsf{true}$ or $v = \mathsf{false}$.

(iii) If $\cdot \vdash v : \tau_1 \times \tau_2$ then $v = \langle v_1, v_2 \rangle$ for some v_1 val and v_2 val.

Proof: We consider each case for v *val* and then invert on the typing derivation in each case. □

Theorem 3 (Progress, new cases for $\tau_1 \times \tau_2$)
If $\cdot \vdash e : \tau$ then either $e \mapsto e'$ for some e' or e val.

Proof: By rule induction on $\cdot \vdash e : \tau$. The rules where we reduce pairs are straightforward, as before, so we only write out the case construct.

Case:

$$\frac{\cdot \vdash e_0 : \tau_1 \times \tau_2 \quad x_1 : \tau_1, x_2 : \tau_2 \vdash e_2 : \tau}{\cdot \vdash \mathsf{case}\ e_0\ (\langle x_1, x_2 \rangle \Rightarrow e_3) : \tau}\ \mathsf{case/pair}$$

where $e = \mathsf{case}\ e_0\ (\langle x_1, x_2 \rangle \Rightarrow e_3)$.

Either $e_0 \mapsto e_0'$ for some e_0 for e_0 *val* — By ind. hyp.

$e_0 \mapsto e_0'$ — First subcase
$\mathsf{case}\ e_0\ (\langle x_1, x_2 \rangle \Rightarrow e_3) \mapsto \mathsf{case}\ e_0'\ (\langle x_1, x_2 \rangle \Rightarrow e_3)$ By rule $\mathsf{step/case/pair}_0$

e_0 *val* — Second subcase
$e_0 = \langle v_1, v_2 \rangle$ for some v_1 *val* and v_2 *val*
By the canonical forms (Theorem 2)
$\mathsf{case}\ e_0\ (\langle x_1, x_2 \rangle \Rightarrow e_3) \mapsto [v_1/x_1][v_2/x_2]e_3$ — By rule $\mathsf{step/case/pair}$

□

6 Examples

Now that we know our statics (typing rules) and dynamics (value and evaluation rules) make sense and are consistent, we can write some examples. First, pairing as a function.

$$\begin{array}{lcl} \mathit{pair} & : & \alpha \to (\beta \to (\alpha \times \beta)) \\ \mathit{pair} & = & \lambda x.\, \lambda y.\, \langle x, y \rangle \end{array}$$

The next example illustrates and important technique and therefore has a name: *Currying*, after the logician Haskell Curry. Instead of a function taking a pair as an argument we can take the two arguments in succession. And vice versa! We express this by saying that two types are *isomorphic*, written as $\tau \cong \sigma$.

$$(\tau \times \sigma) \to \rho \cong \tau \to (\sigma \to \rho)$$

But what does a type isomorphism mean? We say $\tau \cong \tau'$ if there are two functions

$$\begin{array}{lcl} \mathit{Forth} & : & \tau \to \tau' \\ \mathit{Back} & : & \tau' \to \tau \end{array}$$

such that

$$\mathit{Back} \circ \mathit{Forth} = \lambda x.\, x = \mathit{Forth} \circ \mathit{Back}$$

where we are somewhat loose at this point what we mean by function equality.

We program the *Forth* and *Back* functions in a type-directed manner. We show the process only once, but we recommend thinking about coding in this general style. We have

$$\mathit{Forth} : ((\tau \times \sigma) \to \rho) \to (\tau \to (\sigma \to \rho))$$

We see this function takes three arguments in succession: first a function of type $(\tau \times \sigma) \to \rho$, then a value of type τ followed by a value of type σ. So we start the code with three λ-abstractions, followed by an as yet unknown body.

$$\mathit{Forth} = \lambda f.\, \lambda x.\, \lambda y.\, \boxed{}$$

where

$$\begin{array}{lcl} f & : & (\tau \times \sigma) \to \rho \\ x & : & \tau \\ y & : & \sigma \\ \boxed{} & : & \rho \end{array}$$

We can see that only f produces a result of type ρ, and it requires a pair of type $\tau \times \sigma$ as an argument. Fortunately, we have x and y available to form the two components of the pair. Filling everything in:

$$\begin{array}{lcl} \mathit{Forth} & : & ((\tau \times \sigma) \to \rho) \to (\tau \to (\sigma \to \rho)) \\ \mathit{Forth} & = & \lambda f.\, \lambda x.\, \lambda y.\, f\, \langle x, y \rangle \end{array}$$

Programming the other direction in a similar manner yields

$$\begin{array}{lcl} \mathit{Back} & : & (\tau \to (\sigma \to \rho)) \to ((\tau \times \sigma) \to \rho) \\ \mathit{Back} & = & \lambda g.\, \lambda p.\, \mathsf{case}\ p\ (\langle x, y \rangle \Rightarrow g\, x\, y) \end{array}$$

Let's see if we can verify that *Forth* and *Back* compose to the identity, picking an arbitrary direction first.

$$\mathit{Back} \circ \mathit{Forth} = \lambda f.\, \mathit{Back}\ (\mathit{Forth}\ f) \stackrel{?}{=} \lambda f.\, f : ((\tau \times \sigma) \to \rho) \to ((\tau \times \sigma) \to \rho)$$

To compare these two functions we apply them to an arbitrary *value* $v : (\tau \times \sigma) \to \rho$ and compare the result. We reason:

$$\begin{array}{rl}
 & (\lambda f.\, \mathit{Back}\ (\mathit{Forth}\ f))\ v \\
\mapsto & \mathit{Back}\ (\mathit{Forth}\ v) \\
= & \mathit{Back}\ ((\lambda f.\, \lambda x.\, \lambda y.\, f\ \langle x, y\rangle)\ v) \\
\mapsto & \mathit{Back}\ (\lambda x.\, \lambda y.v\ \langle x, y\rangle) \\
= & (\lambda g.\, \lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow g\ x\ y))\ (\lambda x.\, \lambda y.\, v\ \langle x, y\rangle) \\
\mapsto & \lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow (\lambda x'.\, \lambda y'.\, v\ \langle x', y'\rangle)\ x\ y) \\
\stackrel{?}{=} & v : (\tau \times \sigma) \to \rho
\end{array}$$

In the last step we renamed some variable to avoid confusion.

Again, we are comparing two functions, this time on an argument of type $\tau \times \sigma$. These two functions are the same if the return the same result if we apply them to the pair $\langle v_1, v_2\rangle$ of two values $v_1 : \tau$ and $v_2 : \tau_2$. We use values here because the type $\tau \times \sigma$ is observable, and a value of this type is a pair of two values. Then we find:

$$\begin{array}{rl}
 & (\lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow (\lambda x'.\, \lambda y'.\, v\ \langle x', y'\rangle)\ x\ y))\ \langle v_1, v_2\rangle \\
\mapsto & \mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x, y\rangle \Rightarrow (\lambda x'.\, \lambda y'.\, v\ \langle x', y'\rangle)\ x\ y) \\
\mapsto & (\lambda x'.\, \lambda y'.\, v\ \langle x', y'\rangle)\ v_1\ v_2 \\
\mapsto^2 & v\ \langle v_1, v_2\rangle \\
\stackrel{!}{=} & v\ \langle v_1, v_2\rangle
\end{array}$$

The final equality is the one we wanted to check.

Checking the other direction is left to Exercise 4

7 The Unit Type

For every binary type constructor we can ask if there is a nullary version that is its unit. So, for (eager) pairs we are looking for a type 1 such that $\tau \times 1 \cong \tau \cong 1 \times \tau$. This would hold, intuitively, if 1 where a type with exactly one element.

We can also approach this from the purely formalistic perspective. A pair is a tuple with two elements, so an element of 1 should be a tuple with

0 elements. We write this tuple as $\langle\,\rangle$ and type it with

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1}\ \mathsf{unit} \qquad \frac{}{\langle\,\rangle\ \mathit{val}}\ \mathsf{val/unit}$$

With pairs, there is a single destructor thats extracts two components, so for the unit type there is also a single destructor that extracts zero components.

$$\frac{\Gamma \vdash e : 1 \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') : \tau'}\ \mathsf{case/unit}$$

In the dynamics, we only reduce the new version of the case construct, since the unit element is already a value.

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\langle\,\rangle \Rightarrow e_1) \mapsto \mathsf{case}\ e_0'\ (\langle\,\rangle \Rightarrow e_1)}\ \mathsf{step/case/unit}_0$$

$$\frac{}{\mathsf{case}\ \langle\,\rangle\ (\langle\,\rangle \Rightarrow e_1) \mapsto e_1}\ \mathsf{step/case/unit}$$

It is easy to verify that our theorems continue to hold, and that $\cdot \vdash e : 1$ and $e\ \mathit{val}$ implies that $e = \langle\,\rangle$ (as an extension of the canonical forms theorem).

The unit type is not as useless as it might appear. In C, the unit type is called `void` and indicates that a function does not return a value. In a functional language with effects, you will often see code such as

```
let val () = print(v)
```

to execute an effect and return the only value of type 1 (called `unit` in Standard ML).

Let's quickly verify that $\tau \cong \tau \times 1$. We have

$$\begin{array}{lcl} \mathit{Forth} & : & \tau \to (\tau \times 1) \\ \mathit{Forth} & = & \lambda x.\, \langle x, \langle\,\rangle\rangle \\ \\ \mathit{Back} & : & (\tau \times 1) \to \tau \\ \mathit{Back} & = & \lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow x) \end{array}$$

We first check that $\mathit{Back}\,(\mathit{Forth}\,v) = v$ for an arbitrary value $v : \tau$

$$\begin{array}{lcl} & & \mathit{Back}\,(\mathit{Forth}\,v) \\ & = & \mathit{Back}\,((\lambda x.\, \langle x, \langle\,\rangle\rangle)\,v) \\ & \mapsto & \mathit{Back}\,\langle v, \langle\,\rangle\rangle \\ & = & (\lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow x))\,\langle v, \langle\,\rangle\rangle \\ & \mapsto & \mathsf{case}\ \langle v, \langle\,\rangle\rangle\ (\langle x, y\rangle \Rightarrow x) \\ & \mapsto & v \end{array}$$

For the other direction, we exploit that, by the canonical forms theorem, a value of type $v : \tau \times 1$ must have the form $v = \langle v', \langle\,\rangle\rangle$:

$$
\begin{array}{ll}
 & \mathit{Forth}\,(\mathit{Back}\,\langle v', \langle\,\rangle\rangle) \\
= & \mathit{Forth}\,((\lambda p.\,\mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow x))\,\langle v', \langle\,\rangle\rangle) \\
\mapsto & \mathit{Forth}\,(\mathsf{case}\ \langle v', \langle\,\rangle\rangle\ (\langle x, y\rangle \Rightarrow x)) \\
\mapsto & \mathit{Forth}\,v' \\
= & (\lambda x.\,\langle x, \langle\,\rangle\rangle)\,v' \\
\mapsto & \langle v', \langle\,\rangle\rangle \\
\stackrel{!}{=} & v
\end{array}
$$

8 Checking and Synthesis

Extending our rules for checking and synthesizing types is not straightforward, since the new case constructs require some thought. We also have to revise our notion of *neutral expression* to account for the possibility of case. The constructor is easy, in that we check it against a type as usual.

$$\frac{\Gamma \vdash e_1 \Leftarrow \tau_1 \quad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle \Leftarrow \tau_1 \times \tau_2}\ \mathsf{chk/pair} \qquad \frac{}{\Gamma \vdash \langle\,\rangle \Leftarrow 1}\ \mathsf{chk/unit}$$

Synthesis goes from synthesizing a type for the subject of a case to the types of the variables (which always synthesize their type), but the whole construct is checked.

$$\frac{\Gamma \vdash e \Rightarrow \tau_1 \times \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e' \Leftarrow \tau'}{\Gamma \vdash \mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e') \Leftarrow \tau'}\ \mathsf{chk/case/pair}$$

$$\frac{\Gamma \vdash e \Rightarrow 1 \quad \Gamma \vdash e' \Leftarrow \tau'}{\Gamma \vdash \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') \Leftarrow \tau'}\ \mathsf{chk/case/unit}$$

There are at least two reasons why the first premise of the chk/case/pair rule *synthesized* a type for e. First, if we allowed $\Gamma \vdash e \Leftarrow \tau_1 \times \tau_2$, then case $\langle v_1, v_2\rangle$ $(\langle x_1, x_2\rangle \Rightarrow e')$ would be allowed, but this is a redex and we only want to check normal forms. The second reason has to do with the algorithmic interpretation of the rules. If we look at the conclusion of the rule, we can read off Γ, e, x_1, x_2, e' and τ' but *neither* τ_1 *nor* τ_2. Therefore we cannot invoke the checking judgment $\Gamma \vdash e \Leftarrow \tau_1 \times \tau_2$ because that requires all components of the judgment (including τ_1 and τ_2 to be known).

Exercises

Exercise 1 *Lazy pairs*, constructed as $\langle\!|e_1, e_2|\!\rangle$, are an alternative to the eager pairs $\langle e_1, e_2\rangle$. Lazy pairs are typically available in "lazy" languages such as Haskell. The key differences are that a lazy pair $\langle\!|e_1, e_2|\!\rangle$ is always a value, whether its components are or not. In that way, it is like a λ-expression, since $\lambda x.\, e$ is always a value. The second difference is that its destructors are $\mathsf{fst}\ e$ and $\mathsf{snd}\ e$ rather than a new form of case expression.

We write the type of lazy pairs as $\tau_1 \& \tau_2$. In this exercise you are asked to design the rules for lazy pairs and check their correctness.

1. Write out the new rule(s) for $e\ \mathit{val}$.
2. State the typing rules for new expressions $\langle\!|e_1, e_2|\!\rangle$, $\mathsf{fst}\ e$, and $\mathsf{snd}\ e$.
3. Give evaluation rules for the new forms of expressions.

Instead of giving the complete set of new proof cases for the additional constructs, we only ask you to explicate a few items. Nevertheless, you need to make sure that the progress and preservation continue to hold.

4. State the new clause in the canonical forms theorem.
5. Show one case in the proof of the preservation theorem where a destructor is applied to a constructor.
6. Show the case in the proof of the progress theorem analyzing the typing rule for $\mathsf{fst}\ e$.

Exercise 2 Design the *lazy unit* $\langle\!|\,|\!\rangle$ as the nullary version of the lazy pairs of Exercise 1. We write this type as $\top$. Give the rules for values, typing, and evaluation, being careful to preserve their origins as "*lazy pairs with zero components*". Prove or refute that $1 \cong \top$.

Exercise 3 It is often stated that lazy pairs are not necessary in an eager language, because we can already define $\tau_1 \& \tau_2$ and the corresponding constructors and destructors. Fill in this table.

$\tau_1 \& \tau_2$	$\triangleq$	$(1 \to \tau_1) \times (1 \to \tau_2)$
$\langle\!|e_1, e_2|\!\rangle$	$\triangleq$	
$\mathsf{fst}\ e$	$\triangleq$	
$\mathsf{snd}\ e$	$\triangleq$	

Explain with some counterexamples why we cannot just define $\tau_1 \& \tau_2 \triangleq \tau_1 \times \tau_2$. It may be helpful to refer to Exercise L6.2.

Exercise 4 Verify that the composition $\mathit{Forth} \circ \mathit{Back} = \lambda g.\, g$ where *Forth* and *Back* coerce from a curried function to its tupled counterpart.

$$\begin{array}{lcl} \mathit{Forth} & : & ((\tau \times \sigma) \to \rho) \to (\tau \to (\sigma \to \rho)) \\ \mathit{Forth} & = & \lambda f.\, \lambda x.\, \lambda y.\, f\, \langle x, y \rangle \end{array}$$

$$\begin{array}{lcl} \mathit{Back} & : & (\tau \to (\sigma \to \rho)) \to ((\tau \times \sigma) \to \rho) \\ \mathit{Back} & = & \lambda g.\, \lambda p.\, \mathsf{case}\ p\ (\langle x, y \rangle \Rightarrow g\, x\, y) \end{array}$$

For equality of functions, use the simple call-by-value extensionality principle that $f = g : \tau_1 \to \tau_2$ if for every value $v : \tau_1$ we have $f\, v = g\, v : \tau_2$.