Lecture Notes on Sums

15-814: Types and Programming Languages
Frank Pfenning

Lecture 9
Tuesday, September 30, 2020

1 Introduction

In this lecture we continue to build up our small functional language, isolating fundamental building blocks for constructing data and functions. We begin with the unit type 1 with just a single value, the unit element. After investigating some elementary properties of the unit we introduce *disjoint sums* which is the second form of data aggregation after products (whose values are pairs). With those type constructors in hand, we can represent a variety of interesting types with a finite number of elements, but not yet types with infinitely elements except opaquely through functional representations. This gap will be addressed in the next lecture by introducing *recursive types*.

2 The Unit Type

Even though it may not look particularly useful initially, we now introduce the unit type 1, inhabited by exactly one value $\langle\,\rangle$. It is also the nullary version of (eager) pairs (think: a pair $\langle v_1, v_2\rangle$ has two components while $\langle\,\rangle$ has zero).

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

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.

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

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

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

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

It is easy to verify that our theorems continue to hold, and that $\cdot \vdash e : 1$ and e *val* imply 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). We will also see that it is actually quite important in concert with disjoint sums.

3 Type Isomorphisms

Intuitively, 1 should be the nullary product, which we might hope to express with something like $\tau \times 1 = \tau$. But "=" does not make any sense here: these type are different because the are inhabited by different terms. Instead, what we want to say is the the type are *isomorphic*, written as $\tau \times 1 \cong \tau$. Again, intuitively speaking, two types are isomorphic if they have the same information contents. In the general case, we say that $\tau \cong \sigma$ if there are two functions,

Forth : $\tau \to \sigma$ and *Back* : $\sigma \to \tau$ such that they compose to the identity in both directions. Writing it out explicitly:

$$\begin{array}{lcll} \mathit{Forth} & : & \tau \to \sigma & \\ \mathit{Back} & : & \sigma \to \tau & \\ \mathit{Back} \circ \mathit{Forth} & = & \lambda x.\, x & : \ \tau \to \tau \\ \mathit{Forth} \circ \mathit{Back} & = & \lambda y.\, y & : \ \sigma \to \sigma \end{array}$$

When comparing functions (or expressions in general) we have to decide which form of equality to use. The simple β- or even $\beta\eta$-equivalence we used before does not apply here for two reasons: (1) we have many other types besides functions, and (2) we have decided that functions are opaque, so we should not try to analyze their structure. The latter observation pushes us in the direction of an *extensional equality*: two functions are equal if they return equal results when applied to the same argument. This is based on the idea that the structure of functions cannot be observed, but their behavior on arguments can. Because we are in a call-by-value language, this means we have to verify their behavior when applied to arbitrary values of the correct type. On the other hand, types like (eager) products are observable, so we can just compare their components directly. We write $v \sim v' : \tau$ if two expressions are extensionally equal, presupposing that v and v' are closed values of type τ. For general expressions, we write $e \approx e' : \tau$ which is defined by evaluating e and e' to a value and comparing the results.

Expressions: $e \approx e' : \tau$ iff $e \mapsto^* v$, $e' \mapsto^* v'$ with v, v' values, and $v \sim v' : \tau$, or neither e nor e' evaluate to value.

As a side remark, in our current language all well-typed expressions have a value, so the final condition is vacuous but will become relevant during the next language. This definition means we now have to compare *values*. For functions, this will refer back to the definition on expressions, but only at a smaller type.

Functions: $v \sim v' : \tau_1 \to \tau_2$ iff for all $v_1 : \tau_1$ we have $v\ v_1 \approx v'\ v_1 : \tau_2$.

Pairs: $v \sim v' : \tau_1 \times \tau_2$ iff $v = \langle v_1, v_2 \rangle$, $v' = \langle v_1', v_2' \rangle$ and $v_1 \sim v_1' : \tau_1$ and $v_2 \sim v_2' : \tau_2$.

Units: $v \sim v' : 1$ iff $v = \langle\,\rangle$ and $v' = \langle\,\rangle$ (which is always the case, by the canonical forms theorem).

We will later have occasion to revisit and slightly revise this definition, but it is adequate for now.

Returning to the specific example of $\tau \cong \tau \times 1$, let's verify this isomorphism. We define

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

To check that $\mathit{Back} \circ \mathit{Forth} \approx \lambda x.\, x : \tau \to \tau$ we apply both sides to an arbitrary value $v : \tau$ and calculate

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

So the two functions are extensionally equal.

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

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

Again both sides are equal, so both compositions are equal to the identity, witnessing the isomorphism between τ and $\tau \times 1$.

4 Disjoint Sums

Type theory is an open-ended enterprise: we are always looking to capture types of data, modes of computation, properties of programs, etc. One important building block are *type constructors* that build more complicated types out of simpler ones. The function type constructor $\tau_1 \to \tau_2$ is one example. Today we see another one: disjoint sums $\tau_1 + \tau_2$. A value of this type is either a value of type τ_1 or a value of type τ_2 *tagged with the information about which side of the sum it is.* This last part is critical and distinguishes it from the *union type* which is not tagged and much more difficult to integrate soundly into a programming language. We use $\mathbf{l}$ and $\mathbf{r}$ as *tags* or *labels* and write $\mathbf{l} \cdot e_1$ for the expression of type $\tau_1 + \tau_2$ if $e_1 : \tau_1$ and, analogously, $\mathbf{r} \cdot e_2$ if $e_2 : \tau_2$.

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathbf{l} \cdot e_1 : \tau_1 + \tau_2}\ \textsf{tp/left} \qquad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{r} \cdot e_2 : \tau_1 + \tau_2}\ \textsf{tp/right}$$

These two forms of expressions allow us to form elements of the disjoint sum. To destruct such a sum we need a case construct that discriminates based on whether element of the sum is injected on the left or on the right.

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \sigma \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \sigma}{\Gamma \vdash \textsf{case}\ e\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) : \sigma}\ \textsf{tp/cases}$$

Let's talk through this rule. The subject of the case should have type $\tau_1 + \tau_2$ since this is what we are discriminating. If the value of this type is $\mathbf{l} \cdot v_1$ then by the typing rule for the left injection, v_1 must have type τ_1. Since the variable x_1 stands for v_1 it should have type τ_1 in the first branch. Similarly, x_2 should have type τ_2 in the seond branch. Since we cannot tell until the program executes which branch will be taken, just like the conditional in the last lecture, we require that both branches have the same type σ, which is also the type of the whole case.

From this, we can also deduce the value and stepping judgments for the new constructs.

$$\frac{e\ \mathit{value}}{\mathbf{l} \cdot e\ \mathit{value}}\ \textsf{val/left} \qquad \frac{e\ \mathit{value}}{\mathbf{r} \cdot e\ \mathit{value}}\ \textsf{val/right}$$

$$\frac{e \mapsto e'}{\mathbf{l} \cdot e \mapsto l \cdot e'}\ \textsf{step/left} \qquad \frac{e \mapsto e'}{\mathbf{r} \cdot e \mapsto \mathbf{r} \cdot e'}\ \textsf{step/right}$$

$$\frac{e_0 \mapsto e_0'}{\textsf{case}\ e_0\ (\ldots \mid \ldots) \mapsto \textsf{case}\ e_0'\ (\ldots \mid \ldots)}\ \textsf{step/cases}_0$$

$$\frac{v_1\ \mathit{value}}{\textsf{case}\ (\mathbf{l} \cdot v_1)\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \ldots) \mapsto [v_1/x_1]e_1}\ \textsf{step/cases/left}$$

$$\frac{v_2\ \mathit{value}}{\textsf{case}\ (\mathbf{r} \cdot v_2)\ (\ldots \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) \mapsto [v_2/x_2]e_2}\ \textsf{step/cases/right}$$

We have carefully constructed our rules so that the new cases in the preservation and progress theorems should be straightforward.

Theorem 1 (Preservation)
If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$

Proof: Before we dive into the new case, a remark on the rule. We can see that the type of an expression $\mathbf{l} \cdot e_1$ is inherently ambiguous, even if we know that $e_1 : \tau_1$. In fact, it will have the type $\tau_1 + \tau_2$ for *every* type τ_2. In the "official" rule, therefore, we should check that τ_2 is a valid type (see Section 8).

In any case, these considerations do not affect type preservation. There, we just need to show that *any* type τ that e possesses will also be a type of e' if $e \mapsto e'$. Now, it is completely possible that e' will have *more* types than e, but that doesn't contradict the theorem.[1]

The proof of preservation proceeds as usual, by rule on induction on the step $e \mapsto e'$, applying inversion of the typing of e. We show only the new cases, because the cases for all other constructs remain exactly as before. We assume that the substitution property carries over.

Case:

$$\frac{e_1 \mapsto e_1'}{\mathbf{l} \cdot e_1 \mapsto \mathbf{l} \cdot e_1'}\ \textsf{step/left}$$

where $e = \mathbf{l} \cdot e_1$ and $e' = \mathbf{l} \cdot e_1'$

$\cdot \vdash \mathbf{l} \cdot e_1 : \tau_1 + \tau_2$	Assumption
$\cdot \vdash e_1 : \tau_1$	By inversion
$\cdot \vdash e_1' : \tau_1$	By ind.hyp.
$\cdot \vdash \mathbf{l} \cdot e_1' : \tau_1 + \tau_2$	By rule step/left

Case: Rule step/right: analogous to step/left.

Case: Rule step/cases$_0$: similar to the previous two cases.

Case:

$$\frac{v_1\ \mathit{value}}{\textsf{case}\ (\mathbf{l} \cdot v_1)\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \ldots) \mapsto [v_1/x_1]e_1}\ \textsf{step/cases/left}$$

where $e = \textsf{case}\ (\mathbf{l} \cdot v_1)\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \ldots)$ and $e' = [v_1/x_1]e_1$.

[1] It is an instructive exercise to construct a well-typed closed term e with $e \mapsto e'$ such that e' has more types than e.

$\cdot \vdash \mathsf{case}\ (\mathbf{l} \cdot v_1)\ (l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2) : \tau$	Assumption
$\cdot \vdash \mathbf{l} \cdot v_1 : \tau_1 + \tau_2$ and	
$x_1 : \tau_1 \vdash e_1 : \tau$, and $x_2 : \tau_2 \vdash e_2 : \tau$ for some τ_1 and τ_2	By inversion
$\cdot \vdash v_1 : \tau_1$	By inversion
$\cdot \vdash [v_1/x_1]e_1 : \tau$	By the substitution property

Case: Rule step/case/sum/r: analogous to the previous case.

□

The progress theorem proceeds by induction on the typing derivation, as usual, analyzing the possible cases. Before we do that, it is always helpful to call out the canonical forms theorem that characterizew well-typed values. New here is part (iv).

Theorem 2 (Canonical Forms) *Assume v value.*

(i) If $\cdot \vdash v : \tau_1 \to \tau_2$ then $v = \lambda x_1.\, e_2$ for some e_2.

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

(iii) If $\cdot \vdash v : 1$ then $v = \langle\,\rangle$.

(iv) If $\cdot \vdash v : \tau_1 + \tau_2$ then $v = \mathbf{l} \cdot v_1$ for some v_1 value or $v = \mathbf{r} \cdot v_2$ for some v_2 value.

Proof sketch: For each part, analyzing all the possible cases for the value and typing judgments. □

Theorem 3 (Progress)
If $\cdot \vdash e : \tau$ then either $e \mapsto e'$ for some e' or e value.

Proof: By rule induction on the given typing derivation.

Cases: For constructs pertaining to types $\tau_1 \to \tau_2$, bool, $\tau_1 \times \tau_2$, and 1 just as before since we did not change their rules.

Case:

$$\frac{\cdot \vdash e_1 : \tau_1}{\cdot \vdash \mathbf{l} \cdot e_1 : \tau_1 + \tau_2}\ \mathsf{tp/left}$$

where $e = \mathbf{l} \cdot e_1$.

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 *value* — By ind.hyp.

$e_1 \mapsto e_1'$ — Subcase
$\mathbf{l} \cdot e_1 \mapsto \mathbf{l} \cdot e_1'$ — By rule step/left

e_1 *value* — Subcase
$\mathbf{l} \cdot e_1$ *value* — By rule val/l

Case: Rule tp/right is symmetric to previous case.

Case:

$$\frac{\cdot \vdash e_0 : \tau_1 + \tau_2 \quad x_1 : \tau_1 \vdash e_1 : \tau \quad x_2 : \tau_2 \vdash e_2 : \tau}{\cdot \vdash \mathsf{case}\ e_0\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) : \tau}\ \mathsf{tp/cases}$$

where $e = \mathsf{case}\ e_0\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2)$.

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

$e_0 \mapsto e_0'$ — Subcase
$e = \mathsf{case}\ e_0\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2)$
$\quad \mapsto \mathsf{case}\ e_0'\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2)$ — By rule step/cases$_0$

e_0 *value* — Subcase
$e_0 = \mathbf{l} \cdot e_0'$ for some e_0' *value*
or $e_0 = \mathbf{r} \cdot e_0'$ for some e_0' *value* — By canonical forms (Theorem 2)

$e_0 = \mathbf{l} \cdot e_0'$ and e_0' *value* — Sub^{2}case
$e = \mathsf{case}\ (\mathbf{l} \cdot e_0')\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \ldots) \mapsto [e_0'/x_1]e_1$
By rule step/cases/left

$e_0 = \mathbf{r} \cdot e_0'$ and e_0' *value* — Sub^{2}case
$e = \mathsf{case}\ (\mathbf{r} \cdot e_0')\ (\ldots \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) \mapsto [e_0'/x_2]e_2$
By rule step/cases/right

□

5 Examples of Sums

Once we have sums and the unit type from the previous lecture, we can now *define* the Boolean type.

$$
\begin{array}{lcl}
\mathit{bool} & \triangleq & 1+1 \\[4pt]
\mathit{true} & \triangleq & \mathbf{l}\cdot\langle\,\rangle \\
\mathit{false} & \triangleq & \mathbf{r}\cdot\langle\,\rangle \\[4pt]
\mathit{if}\ e_0\ e_1\ e_2 & \triangleq & \mathsf{case}\ e_0\ (\mathbf{l}\cdot x_1 \Rightarrow e_1 \mid \mathbf{r}\cdot x_2 \Rightarrow e_2) \\
 & & (\text{provided } x_1 \notin \mathsf{FV}(e_1) \text{ and } x_2 \notin \mathsf{FV}(e_2))
\end{array}
$$

The provisos on the last definition are important because we don't want to accidentally capture a free variable in e_1 or e_2 during the translation.

Using 1 we can define other types. For example

$$\mathit{option}\ \tau \quad = \quad \tau + 1$$

represents an optional value of type τ. Its values are $\mathbf{l}\cdot v$ for $v : \tau$ (we have a value) or $\mathbf{r}\cdot\langle\,\rangle$ (we have not value of type τ).

A more interesting example would be the natural numbers:

$$
\begin{array}{lcl}
\mathit{nat} & = & 1+(1+(1+\cdots)) \\
\overline{0} & = & \mathbf{l}\cdot\langle\,\rangle \\
\overline{1} & = & \mathbf{r}\cdot(\mathbf{l}\cdot\langle\,\rangle) \\
\overline{2} & = & \mathbf{r}\cdot(\mathbf{r}\cdot(\mathbf{l}\cdot\langle\,\rangle)) \\
\mathit{succ} & = & \lambda n.\,\mathbf{r}\cdot n
\end{array}
$$

Unfortunately, "$\cdots$" is not really permitted in the definition of types. We could define it *recursively* as

$$\mathit{nat} \quad = \quad 1 + \mathit{nat}$$

but supporting this style of recursive type definition is not straightforward. So natural numbers, if we want to build them up from simpler components rather than as a primitive, require a unit type, sums, and recursive types.

6 The Empty Type

We have the singleton type 1, a type with two elements, $1+1$, so can we also have a type with no elements? Yes! We'll call it 0 because it will satisfy that $0 + \tau \cong \tau$. There are no constructors and no values of this type, so the e *value* judgment is not extended.

If we think of 0 as a nullary sum, we expect there still to be a destructor. But instead of two branches it has zero branches!

$$\frac{\Gamma \vdash e_0 : 0 \quad \Gamma \vdash \tau \ \mathit{type}}{\Gamma \vdash \mathsf{case}\ e_0\ (\,) : \tau}\ \mathsf{tp/casez}$$

Computation also makes some sense with a congruence rule reducing the subject, but the case can never be reduced.

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\,) \mapsto \mathsf{case}\ e_0'\ (\,)}\ \mathsf{step/casez}_0$$

Progress and preservation extend somewhat easily, and the canonical forms property is extended with

(v) If $\cdot \vdash v : 0$ then we have a contradiction.

The empty type has somewhat limited uses precisely because there is no value of this type. However, there may still be expression e such that $\cdot \vdash e : 0$ if we have explicitly nonterminating expressions. Such terms can appear the subject of a case where they reduce forever by the only rule. We can also ask, for example, what would be functions from $0 \to 0$. We find:

$$\begin{array}{lcl} \lambda x.\, x & : & 0 \to 0 \\ \lambda x.\, \mathsf{case}\ x\ (\,) & : & 0 \to 0 \\ \lambda x.\, \bot & : & 0 \to 0 \end{array}$$

where $\bot$ is introduced in Exercise L8.3.

7 More Isomorphisms

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 with the following type isomorphism:[2]

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

[2]In lecture, we only discussed the existence of this isomorphism without providing or checking the function witnessing it.

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

$$Forth : ((\tau \times \sigma) \to \rho) \to (\tau \to (\sigma \to \rho))$$

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.

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

where

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

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:

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

Programming the other direction in a similar manner yields

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

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

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

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

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

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:

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

The final equality is the one we wanted to check. Checking the other direction is left to Exercise 3.

For the purpose of reasoning about type isomorphisms we extend our notion of extensional equality by adding a case for sums.

Sums: $v \sim v' : \tau_1 + \tau_2$ iff either $v = \mathbf{l} \cdot v_1$, $v' = \mathbf{l} \cdot v_1'$ and $v_1 \sim v_1' : \tau_1$ or $v = \mathbf{r} \cdot v_2$, $v' = \mathbf{r} \cdot v_2'$ and $v_2 \sim v_2' : \tau_2$

One of the properties that is easy to check is that $\tau + \sigma \cong \sigma + \tau$. We can speculate some other isomorphism, based on an kind of arithmetic interpretation of the types. For example, $\times$ might distribute over $+$:

$$\tau \times (\sigma + \rho) \quad \overset{?}{\cong} \quad (\tau \times \sigma) + (\tau \times \rho)$$

Some strange ones pop up if we think of $\sigma \to \tau$ as τ^σ. The reason to even conjecture this is because we have already checked that $\rho \to (\sigma \to \tau) \cong (\rho \times \sigma) \to \tau$ which could be written as $(\tau^\sigma)^\rho \cong \tau^{\sigma \times \rho}$.

$$\begin{array}{rcl} 2 \to \tau & \overset{?}{\cong} & \tau \times \tau \\ 1 \to \tau & \overset{?}{\cong} & \tau \\ 0 \to \tau & \overset{?}{\cong} & 1 \end{array}$$

While odd, these are not ridiculous. Consider the first one, and recall that $1 + 1 \cong \mathsf{bool}$. In one direction, we can apply the given function to true and false to obtain two values, in other direction we can set the given values as result of the function on true and false, respectively. Do these functions constitute an ismorphism?

An example of types that are *not* isomorphic in general would be

$$\tau \ncong \tau \times \tau$$

In order to show, that they are not always isomorphic it suffices to provide a counterexample where the cardinality of the set of values do not match. (Recall that isomorphism implies equal cardinality, but also that the functions *Forth* and *Back* are expressible in our language.) In this example, if we pick $\tau = 2$ then $v : \tau$ for two values ($\mathbf{l} \cdot \langle\,\rangle$ and $\mathbf{r} \cdot \langle\,\rangle$, to be precise) while the right-hand side contains four values. On the other hand, the isomorphism does hold for $\tau = 1$ since both sides have exactly one value ($\langle\,\rangle$ for the left-hand side and $\langle \langle\,\rangle, \langle\,\rangle \rangle$ for the right-hand side).

8 Summary

See 09-sums-rules.pdf for a summary of the rules.

Exercises

Exercise 1 Exhibit the functions *Forth* and *Back* witnessing the following isomorphisms. You do not need to prove that they constitute an ismorphism, just show the functions. We remain here in the pure language of Section 8 where every function is terminating.

(i) $\tau \times (\sigma + \rho) \cong (\tau \times \sigma) + (\tau \times \rho)$

(ii) $2 \to \tau \cong \tau \times \tau$

(iii) $1 \to \tau \cong \tau$

(iv) $0 \to \tau \cong 1$

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

Exercise 2 Many of the type isomorphisms follow arithmetic equalities, interpreting $\tau + \sigma$ as addition, $\tau \times \sigma$ as multiplication, and $\tau \to \sigma$ as exponentiation σ^τ (see Exercise 1).

But there are also differences. In arithmetic, we have an additive inverse $-a$ such that $a + (-a) = 0$. Prove that there can be no general type constructor $-\tau$ such that $\tau + (-\tau) \cong 0$.

Exercise 3 Verify that the composition *Forth* $\circ$ *Back* $\approx \lambda g.\, g$ where *Forth* and *Back* coerce from a curried function to its tupled counterpart.

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