

THE SIMPLY-TYPED λ -CALCULUS: SUMS AND PRODUCTS

Alex Kavvos

Reading: PFPL §10.1, 11.1

The language of numbers and strings we have been studying so far has very limited expressivity. We will now proceed to radically expand its capabilities. As a result, it will increasingly resemble a realistic functional programming language. The full language we will study this week is known as the **simply-typed λ -calculus**.

First, we will show how to add facilities that can express the following Haskell data types and programs.

```
("hello", "world") :: (Str, Str)
data EitherNumStr = Left Num | Right Str
```

1 Products

Product types allow the programmer to form tuples. **Binary products** allow us to write functions that return not one, but two values. The **unit type** (or **nullary product**) allows us to write functions that return nothing.¹

We extend the syntax chart of Lecture 3 by adding the following new types and pre-terms:

types	$\tau ::= \dots$	
	$\tau_1 \times \tau_2$	product type
	$\mathbf{1}$	unit type
pre-terms	$e ::= \dots$	
	$\langle e_1, e_2 \rangle$	pair constructor
	$\pi_1(e)$	first projection
	$\pi_2(e)$	second projection

The **statics** of product types are given by adding the following typing rules.

UNIT	PROD	PROJ-1	PROJ-2
$\frac{}{\Gamma \vdash \langle \rangle : \mathbf{1}}$	$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$	$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1}$	$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_2}$

The **dynamics** of product types are given by adding the following rules.

VAL-UNIT	VAL-PAIR	D-PROJ-TUPLE-1	D-PROJ-TUPLE-2
$\frac{}{\langle \rangle \text{ val}}$	$\frac{}{\langle e_1, e_2 \rangle \text{ val}}$	$\frac{}{\pi_1(\langle e_1, e_2 \rangle) \mapsto e_1}$	$\frac{}{\pi_2(\langle e_1, e_2 \rangle) \mapsto e_2}$
	D-PROJ-1		D-PROJ-2
	$\frac{e \mapsto e'}{\pi_1(e) \mapsto \pi_1(e')}$		$\frac{e \mapsto e'}{\pi_2(e) \mapsto \pi_2(e')}$

For example, the following typing judgements hold.

$$\begin{aligned} &\vdash \langle \langle \rangle, \langle \text{str}[\text{'hello'}], \text{str}[\text{'world'}] \rangle \rangle : \mathbf{1} \times (\text{Str} \times \text{Str}) \\ &\vdash \pi_1(\langle \langle \rangle, \langle \text{str}[\text{'hello'}], \text{str}[\text{'world'}] \rangle \rangle) : \mathbf{1} \\ &p : (\text{Num} \times \text{Num}) \times \text{Num} \vdash \langle \pi_1(\pi_1(p)), \langle \pi_2(\pi_1(p)), \pi_2(p) \rangle \rangle : \text{Num} \times (\text{Num} \times \text{Num}) \end{aligned}$$

¹In Haskell the binary product of two types `a` and `b` is written `(a, b)`. The unit type is written `()`, and has the unique value `()`.

2 Sums

Sum types express choices between values of different types. **Binary sums** allow us to write programs that pattern match on a variable. The **void type** (or **empty type**, or **nullary sum**) offers no choice at all.²

We further extend the syntax chart given above by adding the following new types and pre-terms:

types	$\tau ::= \dots$	
	$\tau_1 + \tau_2$	sum type
	$\mathbf{0}$	void type
pre-terms	$e ::= \dots$	
	$\text{abort}(e)$	abort
	$\text{inl}(e)$	left injection
	$\text{inr}(e)$	right injection
	$\text{case}(e; x. e_1; y. e_2)$	case analysis

The **statics** of sums are given by adding the following rules.

$\frac{\text{ABORT} \quad \Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \text{abort}(e) : \tau}$	$\frac{\text{INL} \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inl}(e) : \tau_1 + \tau_2}$	$\frac{\text{INR} \quad \Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inr}(e) : \tau_1 + \tau_2}$
$\frac{\text{CASE} \quad \Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case}(e; x. e_1; y. e_2) : \tau}$		

The **dynamics** of sums are given by adding the following rules.

$\frac{\text{VAL-INL}}{\text{inl}(e) \text{ val}}$	$\frac{\text{VAL-INR}}{\text{inr}(e) \text{ val}}$	$\frac{\text{D-ABORT-1} \quad e \mapsto e'}{\text{abort}(e) \mapsto \text{abort}(e')}$
$\frac{\text{D-CASE-INL}}{\text{case}(\text{inl}(e); x. e_1; y. e_2) \mapsto e_1[e/x]}$	$\frac{\text{D-CASE-INR}}{\text{case}(\text{inr}(e); x. e_1; y. e_2) \mapsto e_2[e/y]}$	
$\frac{\text{D-CASE-1} \quad e \mapsto e'}{\text{case}(e; x. e_1; y. e_2) \mapsto \text{case}(e'; x. e_1; y. e_2)}$		

The definition of substitution is the one in Lecture 4, but extended with the following clauses.

$$\begin{aligned} \langle e_1, e_2 \rangle [e/x] &\stackrel{\text{def}}{=} \langle e_1[e/x], e_2[e/x] \rangle & \pi_i(u) [e/x] &\stackrel{\text{def}}{=} \pi_i(u[e/x]) \\ \text{inl}(u) [e/x] &\stackrel{\text{def}}{=} \text{inl}(u[e/x]) & \text{inr}(u) [e/x] &\stackrel{\text{def}}{=} \text{inr}(u[e/x]) \\ \text{case}(u; z. e_1; v. e_2) [e/x] &\stackrel{\text{def}}{=} \text{case}(u[e/x]; z. e_1[e/x]; v. e_2[e/x]) \end{aligned}$$

Notice that z and v are bound in e_1 and e_2 respectively, so the Barendregt convention applies.

For example, the following typing judgements hold.

$$\begin{aligned} &\vdash \text{inl}(\text{num}[4]) : \text{Num} + \text{Str} \\ x : \text{Str} + (\text{Str} \times \text{Num}) &\vdash \text{case}(x; y. y; z. \pi_1(z)) : \text{Str} \\ x : \text{Str} + \text{Num} &\vdash \text{case}(x; y. \text{inr}(y); z. \text{inl}(z)) : \text{Num} + \text{Str} \end{aligned}$$

²In Haskell the binary sum of two types is given by the declaration `data Either a b = Left a | Right b`. The void type can be defined by the declaration `data Empty`, but it is less useful.