Lecture Notes on Sum Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 6
September 21, 2018

1 Introduction

So far in this course we have introduced only basic constructs that exist in pretty much any programming language: functions and Booleans. There may be details of syntax and maybe some small semantics differences such as call-by-value vs. call-by-name, but any such differences can be easily explained and debated within the framework set out so far.

At this point we have a choice between several different directions in which we can extend our inquiry into the nature of programming language.

Precision of Types. We can make types more or less precise in what they say about the program. For example, we might have type containing just true and another containing just false. At the end of this spectrum would be *dependent types* so precise that they can completely specify a function.

Expressiveness of Types. We can analyze which programs can not be typed and make the type system accept more programs, as long as it remains sound.

Computational Mechanisms. So far computation in our language is *value-oriented* in that evaluating an expression returns a value, but it cannot have any effect such as mutating a store, performing input or output, raising an exception, or execute concurrently.

Level of Dynamics. The rules for computation are at a very high level of abstraction and do not talk about, for example, where data might be allocated in memory, or how functions are compiled. A language admits a range of different operational specifications at different levels of abstraction.

Equality and Reasoning. We have introduced typing rules, but no informal or formal system for reasoning about programs. This might include various definitions when we might consider programs to be equal, and rules for establishing equality. Or it might include a language for specifying programs and rules for establishing that they satisfy their specifications. Under this general heading we might also consider translations between different languages and showing their correctness.

All of these are interesting and the subject of ongoing research in programming languages. At the moment, we do not yet have enough infrastructure to make most of these questions rich and interesting. So in the next few lectures we will introduce additional types and corresponding expressions to make the language expressive enough to recover partial recursive functions over interesting forms of data such as natural numbers, lists, trees, etc.

2 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 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 l and r as *tags* or *labels* and write $l \cdot e_1$ for the expression of type $\tau_1 + \tau_2$ if $e_1 : \tau_1$ and, analogously, $r \cdot e_2$ if $e_2 : \tau_2$.

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash l \cdot e_1 : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash r \cdot e_2 : \tau_1 + \tau_2}$$

These two forms of expressions allow us to form element 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.

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\ e\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\} : \tau}$$

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 $l \cdot v_1$ 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 is 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 τ, which is also the type of the whole case.

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

$$\frac{e\ \mathit{val}}{l \cdot e\ \mathit{val}}\ \mathsf{val}/l \qquad \frac{e\ \mathit{val}}{r \cdot e\ \mathit{val}}\ \mathsf{val}/r$$

$$\frac{e \mapsto e'}{l \cdot e \mapsto l \cdot e'}\ {\mapsto}/l \qquad \frac{e \mapsto e'}{r \cdot e \mapsto r \cdot e'}\ {\mapsto}/r$$

$$\frac{e \mapsto e'}{\mathsf{case}\ e\ \{\ldots\} \mapsto \mathsf{case}\ e'\ \{\ldots\}}\ {\mapsto}/\mathsf{case}_1$$

$$\frac{v_1\ \mathit{val}}{\mathsf{case}\ (l \cdot v_1)\ \{l \cdot x_1 \Rightarrow e_1 \mid \ldots\} \mapsto [v_1/x_1]e_1}\ {\mapsto}/\mathsf{case}/l$$

$$\frac{v_2\ \mathit{val}}{\mathsf{case}\ (r \cdot v_2)\ \{\ldots \mid r \cdot x_2 \Rightarrow e_2\} \mapsto [v_2/x_2]e_2}\ {\mapsto}/\mathsf{case}/r$$

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. You can see that the type of an expression $l \cdot e_1$ 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. This is acceptable because we either use bidirectional type checking, in which case both $\tau_1 + \tau_2$ and $l \cdot e_1$ are given to use, or we use some form of type inference that will determine the most general type for an expression.

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:

$$\frac{e_1 \mapsto e_1'}{l \cdot e_1 \mapsto l \cdot e_1'} \mapsto/l$$

where $e = l \cdot e_1$ and $e' = l \cdot e_1'$

$\cdot \vdash l \cdot e_1 : \tau_1 + \tau_2$	Assumption
$\cdot \vdash e_1 : \tau_1$	By inversion
$\cdot \vdash e_1' : \tau_1$	By ind.hyp.
$\cdot \vdash l \cdot e_1' : \tau_1 + \tau_2$	By rule

Case: Rule $\mapsto/r$: analogous to $\mapsto/l$.

Case: Rule $\mapsto/\mathsf{case}_1$: similar to the previous two cases.

Case:

$$\frac{v_1 \ \mathit{val}}{\mathsf{case}\ (l \cdot v_1)\ \{l \cdot x_1 \Rightarrow e_1 \mid \ldots\} \mapsto [v_1/x_1]e_1} \mapsto/\mathsf{case}/l$$

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

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

[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.

Case: Rule $\mapsto/\mathsf{case}/r$: analogous to $\mapsto/r$.

□

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 (iii).

Lemma 2 (Canonical Forms)

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

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

(iii) If $\cdot \vdash v : \tau_1 + \tau_2$ and v val then $v = l \cdot v_1$ for some v_1 val or $v = r \cdot v_2$ for some v_2 val.

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 val.

Proof: By rule induction on the given typing derivation.

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

Case:

$$\frac{\cdot \vdash e_1 : \tau_1}{\cdot \vdash l \cdot e_1 : \tau_1 + \tau_2}$$

where $e = l \cdot e_1$.

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 *val*	By ind.hyp.
$e_1 \mapsto e_1'$	Subcase
$l \cdot e_1 \mapsto l \cdot e_1'$	By rule $\mapsto/l$
e_1 *val*	Subcase
$l \cdot e_1$ *val*	By rule val/l

Case: Typing of $r \cdot e_2$: analogous to previous case.

Case:

$$\frac{\cdot \vdash e_0 : \tau_1 + \tau_2 \quad \cdot, x_1 : \tau_1 \vdash e_1 : \tau \quad \cdot, x_2 : \tau_2 \vdash e_2 : \tau}{\cdot \vdash \mathsf{case}\ e_0\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\} : \tau}$$

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

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

$e_0 \mapsto e_0'$ — Subcase
$e = \mathsf{case}\ e_0\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\}$
$\mapsto \mathsf{case}\ e_0'\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\}$ — By rule $\mapsto/\mathsf{case}_1$

e_0 *val* — Subcase
$e_0 = l \cdot e_0'$ for some e_0' *val*
or $e_0 = r \cdot e_0'$ for some e_0' *val* — By the canonical forms property (4)

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

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

□

3 The Unit Type 1

In order to use sums, it is helpful to have a unit type, written 1, that has exactly one element $\langle\,\rangle$. If we had such a type, we could define *bool* $= 1 + 1$ and bool would no longer be primitive. $1 + 1$ contains exactly two values, namely $l \cdot \langle\,\rangle$ and $r \cdot \langle\,\rangle$.

We have one form "constructing" the unit value and a corresponding case-like elimination, except that there is only on branch.

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1} \qquad \frac{\Gamma \vdash e_0 : 1 \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \mathsf{case}\ e_0\ \{\langle\,\rangle \Rightarrow e_1\} : \tau}$$

There is not much going on as far as the operational semantics is concerned.

$$\frac{}{\langle\,\rangle\ \mathsf{val}}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{case}\ e_1\ \{\langle\,\rangle \Rightarrow e_1\} \mapsto \mathsf{case}\ e_1'\ \{\langle\,\rangle \Rightarrow e_1\}} \qquad \frac{}{\mathsf{case}\ \langle\,\rangle\ \{\langle\,\rangle \Rightarrow e_1\} \mapsto e_1}$$

Preservation and progress continue to hold, and are proved following the pattern of the previous cases. We just restate the canonical forms lemma.

Lemma 4 (Canonical Forms)

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

(ii) If $\cdot \vdash v :$ bool and v val then $v =$ true or $v =$ false.

(iii) If $\cdot \vdash v : \tau_1 + \tau_2$ and v val then $v = l \cdot v_1$ for some v_1 val or $v = r \cdot v_2$ for some v_2 val.

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

4 Using the Unit Type

As indicated in the previous section, we can now *define* the Boolean type using sums and unit. We have:

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

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. Recommended question to think about: could we define a function $\mathit{if}_\tau : (1+1) \to \tau \to \tau \to \tau$ for arbitrary τ that implements the case construct?

Using 1 we can define other types. For example

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

represents an optional value of type τ. Its values are $l \cdot v$ for $v : \tau$ (we have a value) or $r \cdot \langle\,\rangle$, where $r \cdot \langle\,\rangle$ (we have no value).

A more interesting examples would be the natural numbers:

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

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. We will probably use explicit *recursive types* to define

$$\mathit{nat} \quad = \quad \rho\alpha.\, 1 + \alpha$$

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.

5 The Empty Type 0

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 (in a way we do not make precise) that $0 + \tau \simeq \tau$. There are no constructors and no values of this type, so the e *val* judgment is not extended.

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

$$\frac{\Gamma \vdash e_0 : 0}{\Gamma \vdash \mathsf{case}\ e_0\ \{\,\} : \tau}$$

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

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ \{\,\} \mapsto \mathsf{case}\ e_0'\ \{\,\}}$$

Progress and preservation extend somewhat easily, and the canonical forms property is extended with *(v) If $\cdot \vdash v : 0$ and v val 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:

$$\begin{array}{lcl} \lambda x.\, x & : & 0 \to 0 \\ \lambda x.\, \mathsf{case}\ x\ \{\,\} & : & 0 \to 0 \\ \lambda x.\, \mathit{loop} & : & 0 \to 0 \end{array}$$

assume we can define a looping term and give it type 0.

6 Summary

We present a brief summary of the language of types and expressions we have defined so far.

$$\begin{array}{llcl} \text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid 0 \mid 1 \\ \text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \\ & & \mid & l \cdot e \mid r \cdot e \mid \mathsf{case}\ e_0\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\} \\ & & \mid & \mathsf{case}\ e_0\ \{\,\} \\ & & \mid & \langle\,\rangle \mid \mathsf{case}\ e_0\ \{\langle\,\rangle \Rightarrow e_1\} \end{array}$$

Functions.

$$\frac{\Gamma, x_1 : \tau_2 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1.\, e_2 : \tau_1 \to \tau_2} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

$$\frac{}{\lambda x.\, e\ \mathit{val}}$$

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2} \qquad \frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto v_1\, e_2'}$$

$$\frac{v_2\ \mathit{val}}{(\lambda x.\, e_1)\, v_2 \mapsto [v_2/x]e_1}$$

Disjoint Sums.

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash l \cdot e_1 : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash r \cdot e_2 : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\ e\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\} : \tau}$$

$$\frac{e\ \mathit{val}}{l \cdot e\ \mathit{val}} \qquad \frac{e\ \mathit{val}}{r \cdot e\ \mathit{val}}$$

$$\frac{e \mapsto e'}{l \cdot e \mapsto l \cdot e'} \qquad \frac{e \mapsto e'}{r \cdot e \mapsto r \cdot e'}$$

$$\frac{e \mapsto e'}{\mathsf{case}\ e\ \{\ldots\} \mapsto \mathsf{case}\ e'\ \{\ldots\}}$$

$$\frac{v_1\ \mathit{val}}{\mathsf{case}\ (l \cdot v_1)\ \{l \cdot x_1 \Rightarrow e_1 \mid \ldots\} \mapsto [v_1/x_1]e_1}$$

$$\frac{v_2\ \mathit{val}}{\mathsf{case}\ (r \cdot v_2)\ \{\ldots \mid r \cdot x_2 \Rightarrow e_2\} \mapsto [v_2/x_2]e_2}$$

Unit Type.

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1} \qquad \frac{\Gamma \vdash e_0 : 1 \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \mathsf{case}\ e_0\ \{\langle\,\rangle \Rightarrow e_1\} : \tau}$$

$$\frac{}{\langle\,\rangle\ \mathsf{val}}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{case}\ e_1\ \{\langle\,\rangle \Rightarrow e_1\} \mapsto \mathsf{case}\ e_1'\ \{\langle\,\rangle \Rightarrow e_1\}} \qquad \frac{}{\mathsf{case}\ \langle\,\rangle\ \{\langle\,\rangle \Rightarrow e_1\} \mapsto e_1}$$

Empty Type.

$$\frac{\Gamma \vdash e_0 : 0}{\Gamma \vdash \mathsf{case}\ e_0\ \{\,\} : \tau}$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ \{\,\} \mapsto \mathsf{case}\ e_0'\ \{\,\}}$$