

# INVERSION & STRUCTURAL RULES

Alex Kavvos

Reading: PFPL, §4.3

## 1 Inversion

The nature of the typing rules for the language is such that the shape of a term places strong restrictions on its type. For example, we cannot imagine that the type of a term of the form  $\text{plus}(e_1; e_2)$  is  $\text{Str}$ . After all, the only rule that allows us to derive a typing judgement of the form  $\Gamma \vdash \text{plus}(e_1; e_2) : \tau$  forces  $\tau \stackrel{\text{def}}{=} \text{Num}$ .

Such facts about type systems are often called **inversion lemmata**. They are formally stated as follows.

**Lemma 1** (Inversion). Suppose  $\Gamma \vdash e : \tau$ .

1. If  $e = \text{plus}(e_1; e_2)$  then it must be that

- $\tau = \text{Num}$
- $\Gamma \vdash e_1 : \text{Num}$
- $\Gamma \vdash e_2 : \text{Num}$

2. ...

The lemma has a case for each construct of the language; you fill in the rest.

In practice, the proof of the inversion lemma is *by inspection* ('look at the rules, there can't be another way!'). More formally, the lemma can be shown by induction on the typing derivation.

## 2 Weakening

Suppose that  $x : \sigma \vdash e : \tau$ , i.e. that  $e$  computes a value of type  $\tau$  if  $x$  is of type  $\sigma$ . It is reasonable to expect that for any **fresh variable**  $y$  (i.e. a variable that doesn't already occur in the term  $e$ ), the typing judgement  $x : \sigma, y : \rho \vdash e : \tau$  should hold as well, no matter what the type  $\rho$  is. In other words: assuming random free variables that we do not use at all should not influence the type of a program. This property is called **weakening**, and it holds in the majority of programming languages.

The reason it holds is that we may systematically thread a fresh variable across a derivation. For example,

$$\frac{\frac{x : \text{Num} \vdash x : \text{Num} \quad x : \text{Num} \vdash \text{num}[1] : \text{Num}}{x : \text{Num} \vdash \text{plus}(x; \text{num}[1]) : \text{Num}} \quad \frac{}{x : \text{Num}, y : \text{Num} \vdash y : \text{Num}}}{x : \text{Num} \vdash \text{let}(\text{plus}(x; \text{num}[1]); y. y) : \text{Num}}$$

can be systematically transformed by adding the binding  $z : \text{Str}$  everywhere to obtain the derivation

$$\frac{\frac{x : \text{Num}, z : \text{Str} \vdash x : \text{Num} \quad x : \text{Num}, z : \text{Str} \vdash \text{num}[1] : \text{Num}}{x : \text{Num}, z : \text{Str} \vdash \text{plus}(x; \text{num}[1]) : \text{Num}} \quad \frac{}{x : \text{Num}, z : \text{Str}, y : \text{Num} \vdash y : \text{Num}}}{x : \text{Num}, z : \text{Str} \vdash \text{let}(\text{plus}(x; \text{num}[1]); y. y) : \text{Num}}$$

Formally, we state and prove by induction on the typing derivation that

**Lemma 2** (Weakening). If  $\Gamma \vdash e : \tau$  and  $x$  is fresh then  $\Gamma, x : \sigma \vdash e : \tau$ .

### 3 Substitution

We have read a judgement  $x : \sigma \vdash e : \tau$  as saying that  $e : \tau$  if we assume that  $x$  stands for a program of type  $\sigma$ .

A term  $\vdash e : \sigma$  whose context is empty is called a **closed term**; that means the program  $e$  has no free variables.

If we are given a closed term  $\vdash e : \sigma$ , and a term  $x : \sigma \vdash u : \tau$ , then we should somehow be able to ‘plug-in’, or **substitute**, the term  $e : \sigma$  for the free variable  $x : \sigma$  in  $u$ . This is the same process that we know from mathematics as e.g. plugging in  $x \stackrel{\text{def}}{=} 5$  in the expression  $x^2 + 3x + 1$  to obtain  $5^2 + 3 * 5 + 1$ .

We write the resulting term as  $u[e/x]$ . This is **not a construct of the programming language**. Rather, it is some notation we use to signify the substitution of one expression in another (as above). We sometimes call such things **metatheoretic operations**.<sup>1</sup> Formally, substitution is defined by induction on pre-terms.

$$\begin{aligned} z[e/x] &\stackrel{\text{def}}{=} \begin{cases} e & \text{if } z \equiv x \\ z & \text{if } z \not\equiv x \end{cases} & \text{let}(e_1; y. e_2)[e/x] &\stackrel{\text{def}}{=} \text{let}(e_1[e/x]; y. e_2[e/x]) \\ (\text{num}[n])[e/x] &\stackrel{\text{def}}{=} \text{num}[n] & \text{plus}(e_1; e_2)[e/x] &\stackrel{\text{def}}{=} \text{plus}(e_1[e/x]; e_2[e/x]) \\ (\text{str}[s])[e/x] &\stackrel{\text{def}}{=} \text{str}[s] & \text{cat}(e_1; e_2)[e/x] &\stackrel{\text{def}}{=} \text{cat}(e_1[e/x]; e_2[e/x]) \end{aligned}$$

The missing cases for  $\text{times}(e_1; e_2)$  and  $\text{len}(e_1)$  are analogous.

The fact the term  $\text{let}(e_1; y. e_2)$  binds  $y$  in  $e_2$  is treacherous! Consider what should happen in the following cases.

$$\text{let}(x + \text{num}[1]; y. x + y)[y/x] \qquad \text{let}(y; y. \text{len}(y))[\text{str}[\text{hi}]/y]$$

In the first case, we risk **variable capture**; we must first  $\alpha$ -rename the term to  $\text{let}(x + \text{num}[1]; z. x + z)$ . In the second case, we risk **referential clash**; we must first  $\alpha$ -rename the term to  $\text{let}(y; z. \text{len}(z))$ .

Bound variables are always a source of trouble. It is common to adopt the **Barendregt convention**:<sup>2</sup> when substituting we assume that everything has been silently  $\alpha$ -renamed in a way that is advantageous for us, and that will not cause the two problems exemplified above. Thus, when writing  $u[e/x]$  we will assume that the variable  $x$  does not occur bound anywhere in the term  $u$ , because we can  $\alpha$ -rename it if it does.

Substitution interacts very well with typing. The following is perhaps the most important result in this unit.

**Lemma 3** (Substitution). If  $\Gamma \vdash e : \tau$  and  $\Gamma, x : \tau \vdash u : \sigma$ , then  $\Gamma \vdash u[e/x] : \sigma$ .

*Proof.* By induction on the derivation of  $\Gamma, x : \tau \vdash u : \sigma$ . We show only the most involved case, viz. that of LET.

If the derivation of  $\Gamma, x : \tau \vdash u : \sigma$  ends with LET, then we know that, for some type  $\sigma_1$  it has the form

$$\frac{\frac{\vdots}{\Gamma, x : \tau \vdash e_1 : \sigma_1} \quad \frac{\vdots}{\Gamma, x : \tau, y : \sigma_1 \vdash e_2 : \sigma_2}}{\underbrace{\Gamma, x : \tau \vdash}_{\text{"}\Gamma\text{"}} \underbrace{\text{let}(e_1; y. e_2) :}_{\text{"}u\text{"}} \underbrace{\sigma_2}_{\text{"}\sigma\text{"}}} \text{LET}$$

By the **induction hypothesis (IH)** applied to the assumption  $\Gamma \vdash e : \tau$  and the derivation of  $\Gamma, x : \tau \vdash e_1 : \sigma_1$  we obtain a derivation of  $\Gamma \vdash e_1[e/x] : \sigma_1$ . By the assumption  $\Gamma \vdash e : \tau$  and **weakening** we get  $\Gamma, y : \sigma_1 \vdash e : \tau$ . We can then apply the **IH** to that and the second subtree to obtain a derivation of  $\Gamma, y : \sigma_1 \vdash e_2[e/x] : \sigma_2$ .

Using a single instance of the LET rule we can put these two together to obtain a derivation

$$\frac{\frac{\vdots}{\Gamma \vdash e_1[e/x] : \sigma_1} \quad \frac{\vdots}{\Gamma, y : \sigma_1 \vdash e_2[e/x] : \sigma_2}}{\Gamma \vdash \text{let}(e_1[e/x]; y. e_2[e/x]) : \sigma_2} \text{LET}$$

But  $(\text{let}(e_1; y. e_2))[e/x]$  is by definition of substitution exactly the term  $\text{let}(e_1[e/x]; y. e_2[e/x])$  in this derivation, so we have shown the result!  $\square$

<sup>1</sup>This term has its origins in logic. We are studying a little programming language which we call our **theory**. Anything we prove about it this programming language using maths and our minds is a **metatheoretic** statement, i.e. a statement *about* the theory itself.

<sup>2</sup>Originally coined by Dutch logician Henk Barendregt (b. 1947). Called the **identification convention** in PFPL.