Lecture Notes on Progress

15-814: Types and Programming Languages
Frank Pfenning

Lecture 8
Thursday, September 24, 2020

1 Introduction

We start by short exploration of the consequences of making the structure of functions opaque and then focus on proving *progress*, one of the key properties connecting typing and evaluation. This in turn requires the *canonical forms theorem*, which is a new form of representation theorem (such as we have proved for Booleans, represented in the typed λ-calculus).

Let's reiterate the critical properties we care about for now:

Preservation. If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$.

Progress. For every expression $\cdot \vdash e : \tau$ either $e \mapsto e'$ for some e' or e *value*.

Finality of Values. There is no $\cdot \vdash e : \tau$ such that $e \mapsto e'$ for some e' and e *value*.

Sequentiality. If $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$.

Since we already proved preservation for ordinary reduction in some detail for the simply-typed λ-calculus, in this lecture we focus on the progress theorem so we can understand the structure of its proof.

2 Observing Functional Values

As we have emphasized, we assume we cannot directly observe the structure of functions when they are outcome of computation. Instead, we can probe

such functions by applying them to argument and observing the results. As an example, consider our language with parametric polymorphism and Booleans, and our usual representation of natural numbers as their iterators:

$$\mathit{nat} : \forall\alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$$

If we have an expression $\cdot \vdash e : \mathit{nat}$ such that e *value* we know it will have the form $\Lambda\alpha.\, e'$ for some e', but we cannot observe e'. Moreover, e' may not even be a value, even though e is. Nevertheless, we can test, for example, if the value e is zero or positive. Consider

$$\cdot \vdash e\ [\mathsf{bool}] : (\mathsf{bool} \to \mathsf{bool}) \to \mathsf{bool} \to \mathsf{bool}$$

and

$$\cdot \vdash e\ [\mathsf{bool}]\ (\lambda b.\, \mathsf{false})\ \mathsf{true} : \mathsf{bool}$$

If this expression evaluates to true then e "represents" zero, and if it evaluates to false then e "represents" some positive number. We put "represents" in quotes here because, for example, e may not be equal to $\Lambda\alpha.\, \lambda s.\, \lambda z.\, z$. Instead, it *behaves* like this function when applied to a type τ, and two arguments of type $\tau \to \tau$ and τ in this order. We just have to keep in mind that this computation takes place when we observe e, and not when e is originally evaluated.

A small item of notation: we write $e \hookrightarrow v$ to express that e *evaluates* to the value v. This presupposes that $\cdot \vdash e : \tau$ for some τ and ensures that v *value*. Formally, it is defined by

$$\frac{v\ \mathit{value}}{v \hookrightarrow v}\ \mathsf{eval/val} \qquad \frac{e \mapsto e' \quad e' \hookrightarrow v}{e \hookrightarrow v}\ \mathsf{eval/step}$$

It is also possible to define evaluation directly as a so-called *big-step evaluation* judgment as compared to the *small-step evaluation* we have defined so far (see Exercise 1).

From now on we will often write v for an expression we know to be a value, but at least for the moment we will not automatically imply this from the notation, that is, we will still write v *value* where we are not already assured that v is indeed a value.

3 Progress

The progress property is intended to rule out intuitively meaningless expressions that neither reduce nor constitute a value. For example, the ill-typed expression if $(\lambda x.\, x)$ false true cannot take a step since the subject $(\lambda x.\, x)$ is a value but the whole expression is not a value and cannot take a step. Similarly, the expression if b false true is well-typed in the context with b : bool, but it cannot take a step nor is it a value. Therefore, it is clear that the assumptions that e is closed that that e has a valid type are both needed for this theorem. It may be helpful to refer to the summary of the judgments inference rules while reading this proof.

Theorem 1 (Progress)
If $\cdot \vdash e : \tau$ *then either* $e \mapsto e'$ *for some* e' *or* e *value.*

Proof: There are not many candidates for the structure of this proof. We have e and we have a typing for e. From that scant information we need obtain evidence that e can step or is a value. So we try the rule induction on $\cdot \vdash e : \tau$.

Case:

$$\frac{x_1 : \tau_1 \vdash e_2 : \tau_2}{\cdot \vdash \lambda x_1.\, e_2 : \tau_1 \to \tau_2}\ \mathsf{tp/lam}$$

where $e = \lambda x_1.\, e_2$. Then we have

$\lambda x_1.\, e_2$ *value*	By rule val/lam

It is fortunate we don't need the induction hypothesis, because it cannot be applied! That's because the context of the premise is not empty, which is easy to miss. So be careful!

Case:

$$\frac{x : \tau \in (\cdot)}{\cdot \vdash x : \tau}$$

This case is impossible because there is not declaration for x in the empty context.

Case:

$$\frac{\cdot \vdash e_1 : \tau_2 \to \tau \quad \cdot \vdash e_2 : \tau_2}{\cdot \vdash e_1\, e_2 : \tau}$$

where $e = e_1\, e_2$. At this point we apply the induction hypothesis to e_1. If it reduces, so does $e = e_1\, e_2$. If it is a value, then we apply the induction hypothesis to e_2. If is reduces, so does $e_1\, e_2$. If not, we have a redex. In more detail:

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

- $e_1 \mapsto e_1'$ — Subcase
  $e = e_1\, e_2 \mapsto e_1'\, e_2$ — by rule $\mathsf{step/app}_1$
- e_1 *value* — Subcase
  Either $e_2 \mapsto e_2'$ for some e_2' or e_2 *value* — By ind.hyp.
  - $e_2 \mapsto e_2'$ — Sub^{2}case
    $e_1\, e_2 \mapsto e_1\, e_2'$ — By rule $\mathsf{step/app}_2$ since e_1 *value*
  - e_2 *value* — Sub^{2}case
    $e_1 = \lambda x.\, e_1'$ and $x : \tau_2 \vdash e_1' : \tau$ — By "inversion"

We pause here to consider this last step. We know that $\cdot \vdash e_1 : \tau_2 \to \tau$ and e_1 *value*. By considering all cases for how both of these judgments can be true at the same time, we see that e_1 must be a λ-abstraction. This is often summarized in a *canonical forms theorem* which we state after this proof. Finishing this sub^{2}case:

$e = (\lambda x.\, e_1')\, e_2 \mapsto [e_2/x]e_1'$ — By rule $\mathsf{step/app/lam}$ since e_2 *value*

Case:

$$\frac{}{\cdot \vdash \mathsf{true} : \mathsf{bool}}$$

where $e = \mathsf{true}$. Then $e = \mathsf{true}$ *value* by rule $\mathsf{val/true}$.

Case: Typing of false. As for true.

Case:

$$\frac{\cdot \vdash e_1 : \mathsf{bool} \quad \cdot \vdash e_2 : \tau \quad \cdot \vdash e_3 : \tau}{\cdot \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau}$$

where $e = \mathsf{if}\ e_1\ e_2\ e_3$.

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

- $e_1 \mapsto e_1'$ — Subcase
  $e = \mathsf{if}\ e_1\ e_2\ e_3 \mapsto \mathsf{if}\ e_1'\ e_2\ e_3$ — By rule step/if
- e_1 *value* — Subcase
  $e_1 = \mathsf{true}$ or $e_1 = \mathsf{false}$
  By considering all cases for $\cdot \vdash e_1 : \mathsf{bool}$ and e_1 *value*
  - $e_1 = \mathsf{true}$ — Sub^{2}case
    $e = \mathsf{if}\ \mathsf{true}\ e_2\ e_3 \mapsto e_2$ — By rule step/if/true
  - $e_1 = \mathsf{false}$ — Sub^{2}case
    $e = \mathsf{if}\ \mathsf{false}\ e_2\ e_3 \mapsto e_3$ — By rule step/if/false

Cases: For rules tp/tplam and tp/tpapp see Exercise 2.

□

This completes the proof. The complex inversion steps can be summarized in the *canonical forms theorem* that analyzes the shape of well-typed values. It is a form of the representation theorem for Booleans we proved in an earlier lecture for the simply-typed λ-calculus.

Theorem 2 (Canonical Forms)

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

(ii) *If* $\cdot \vdash v : \forall \alpha.\, \tau$ *then* $v = \Lambda \alpha.\, e$.

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

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

4 Type Preservation*

This proof was not done in lecture, but is presented here for completeness. In a future lecture we will reexamine the proof of this theorem.

We already know that the rules should satisfy the substitution property (Theorem L5.6). We can easily check the new cases in the proof because substitution remains compositional. For example, $[e'/x](\mathsf{if}\ e_1\ e_2\ e_3) = \mathsf{if}\ ([e'/x]e_1)\ ([e'/x]e_2)\ ([e'/x]e_3)$. However, some new properties are needed for parametric polymorphism, so we make them explicit here and generalize the previous theorem.

Theorem 3 (Substitution Property)

(i) If $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau, \Gamma' \vdash e' : \tau'$ *then* $\Gamma, \Gamma' \vdash [e/x]e' : \tau'$.

(ii) If $\Gamma \vdash \tau$ *type and* $(\Gamma, \alpha$ *type*$, \Gamma')$ *ctx then* $(\Gamma, [\tau/\alpha]\Gamma')$ *ctx.*

(iii) If $\Gamma \vdash \tau$ *type and* Γ, α *type*$, \Gamma' \vdash \sigma$ *type then* $\Gamma, [\tau/\alpha]\Gamma' \vdash [\tau/\alpha]\sigma$ *type.*

(iv) If $\Gamma \vdash \tau$ *type and* Γ, α *type*$, \Gamma' : \tau \vdash e : \sigma$ *then* $\Gamma, [\tau/\alpha]\Gamma' \vdash [\tau/\alpha]e : [\tau/\alpha]\sigma$.

Proof: Each part by rule induction on the second given derivation. We have to exploit the fact that term variables x do not occur in types, and we need to remember our presuppositions and (silent) renaming of bound variables (both for terms and types). □

On to preservation.

Theorem 4 (Type Preservation)
If $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \tau$.

Proof: By rule induction on the derivation of $e \mapsto e'$.

In each case we apply inversion on the typing derivation to obtain typing derivations for the components of e. From these derivations we assemble a typing derivation for e'. In cases of a step involving substitution, we have to appeal to the substitution property to obtain the resulting derivation.

Case:

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app}_1$$

where $e = e_1\, e_2$ and $e' = e_1'\, e_2$.

$\cdot \vdash e_1\, e_2 : \tau$	Assumption
$\cdot \vdash e_1 : \tau_2 \to \tau$ and $\cdot \vdash e_2 : \tau_2$ for some τ_2	By inversion
$\cdot \vdash e_1' : \tau_2 \to \tau$	By ind.hyp.
$\cdot \vdash e_1'\, e_2 : \tau$	By rule app

Case:

$$\frac{v_1\ \textit{value} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto v_1\, e_2'}\ \mathsf{step/app}_2$$

where $e = v_1\, e_2$ and $e' = v_1\, e_2'$. As in the previous case, we proceed by inversion on typing.

$\cdot \vdash v_1\, e_2 : \tau$	Assumption
$\cdot \vdash v_1 : \tau_2 \to \tau$ and $\cdot \vdash e_2 : \tau_2$ for some τ_2	By inversion
$\cdot \vdash e_2' : \tau_2$	By ind.hyp.
$\cdot \vdash v_1\, e_2' : \tau$	By rule app

Case:

$$\frac{v_2\ \textit{value}}{(\lambda x.\, e_1)\, v_2 \mapsto [v_2/x]e_1}\ \mathsf{step/app/lam}$$

where $e = (\lambda x.\, e_1)\, v_2$ and $e' = [v_2/x]e_1$. Again, we apply inversion on the typing of e, this time twice. Then we have enough pieces to apply the *substitution property* (Theorem 3).

$\cdot \vdash (\lambda x.\, e_1)\, v_2 : \tau$	Assumption
$\cdot \vdash \lambda x.\, e_1 : \tau_2 \to \tau$ and $\cdot \vdash v_2 : \tau_2$ for some τ_2	By inversion
$x : \tau_2 \vdash e_1 : \tau$	By inversion
$\cdot \vdash [v_2/x]e_1 : \tau$	By the *substitution property* (Theorem 3)

Case:

$$\frac{e_1 \mapsto e_1'}{\mathsf{if}\ e_1\ e_2\ e_3 \mapsto \mathsf{if}\ e_1'\ e_2\ e_3}\ \mathsf{step/if}$$

where $e = \mathsf{if}\ e_1\ e_2\ e_3$ and $e' = \mathsf{if}\ e_1'\ e_2\ e_3$. As might be expected by now, we apply inversion to the typing of e, followed by the induction hypothesis on the type of e_1, followed by re-application of the typing rule for if.

$\cdot \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau$	Assumption
$\cdot \vdash e_1 : \mathsf{bool}$ and $\cdot \vdash e_2 : \tau$ and $\cdot \vdash e_3 : \tau$	By inversion
$\cdot \vdash e_1' : \mathsf{bool}$	By ind.hyp.
$\cdot \vdash \mathsf{if}\ e_1'\ e_2\ e_3 : \tau$	By rule tp/if

Case:

$$\frac{}{\mathsf{if}\ \mathsf{true}\ e_2\ e_3 \mapsto e_2}\ \mathsf{step/if/true}$$

where $e = \mathsf{if}\ \mathsf{true}\ e_2\ e_3$ and $e' = e_2$. This time, we don't have an induction hypothesis since this rule has no premise, but fortunately one step of inversion suffices.

$\cdot \vdash \mathsf{if}\ \mathsf{true}\ e_2\ e_3 : \tau$	Assumption
$\cdot \vdash \mathsf{true} : \mathsf{bool}$ and $\cdot \vdash e_2 : \tau$ and $\cdot \vdash e_3 : \tau$	By inversion
$\cdot \vdash e' : \tau$	Since $e' = e_2$.

Case: Rule step/if/false is analogous to the previous case.

Case:

$$\frac{e_1 \mapsto e_1'}{e_1\ [\sigma] \mapsto e_1'\ [\sigma]}\ \mathsf{step/tpapp}$$

where $e = e_1\ [\sigma]$ and $e' = e_1'\ [\sigma]$.

$\cdot \vdash e_1\ [\sigma] : \tau$	Assumption
$\cdot \vdash e_1 : \forall\alpha.\,\tau_2$ where $\tau = [\sigma/\alpha]\tau_2$	By inversion
$\cdot \vdash e_1' : \forall\alpha.\,\tau_2$	By ind. hyp
$\cdot \vdash e_1'\ [\sigma] : [\sigma/\alpha]\tau_2$	By rule tp/tpapp
$\cdot \vdash e' : \tau$	Since $e' = e_1'\ [\sigma]$ and $\tau = [\sigma/\alpha]\tau_2$

Case:

$$\frac{}{(\Lambda\alpha.\,e_2)\ [\sigma] \mapsto [\sigma/\alpha]e_2}\ \mathsf{step/tpapp/tplam}$$

where $e = (\Lambda\alpha.\,e_2)\ [\sigma]$ and $e' = [\sigma/\alpha]e_2$.

$\cdot \vdash (\Lambda\alpha.\,e_2)\ [\sigma] : \tau$	Assumption
$\cdot \vdash (\Lambda\alpha.\,e_2) : \forall\alpha.\,\tau_2$ and $\cdot \vdash \sigma\ \mathit{type}$ with $\tau = [\sigma/\alpha]\tau_2$ for some τ_2	By inversion
$\alpha\ \mathit{type} \vdash e_2 : \tau_2$	By inversion
$\cdot \vdash [\sigma/\alpha]e_2 : [\sigma/\alpha]\tau_2$	By the substitution property (Theorem 3)

□

5 Pairs

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.

5.1 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

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

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 6.

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

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.

Types	τ	$::=$	$\alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha.\, \tau \mid \mathsf{bool} \mid \tau_1 \times \tau_2$
Observable Types	o	$::=$	$\mathsf{bool} \mid o_1 \times o_2$

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

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

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

5.2 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) 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.

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

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

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

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.

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

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.

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

6 Preservation and Progress, Revisited*

This section was also not covered in lecture, but given here for completeness.

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 5 (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 tp/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:

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

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 3)
$\cdot \vdash [v_1/x_1][v_2/x_2]e_3 : \tau$	By substitution (Theorem 3)

□

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 6 (Canonical Forms)
Assume v value. Then

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

(ii) If $\cdot \vdash v : \forall \alpha.\, \tau$ then $v = \Lambda \alpha.\, e$.

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

(iv) 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.

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

Theorem 7 (Progress, new cases for $\tau_1 \times \tau_2$)
If $\cdot \vdash e : \tau$ then either $e \mapsto e'$ for some e' or e value.

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:

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

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 *value* — 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 $\mathsf{step/casep}_0$

e_0 *value* — Second subcase

$e_0 = \langle v_1, v_2\rangle$ for some v_1 *value* and v_2 *value* — By the canonical forms (Theorem 6)

$\mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto [v_1/x_1][v_2/x_2]e_3$ — By rule step/casep/pair

□

Exercises

Exercise 1 Design rules for the big-step evaluation judgment $e \hookrightarrow v$ which do not use any auxiliary judgment. In particular, you cannot refer to e *value* or $e \mapsto e'$, nor may design your own auxiliary judgments. You may restrict yourself to functions and Booleans, and you should presuppose that $\cdot \vdash e : \tau$.

(i) Show the rules.

(ii) Prove that if $e \hookrightarrow v$ with $\cdot \vdash e : \tau$ then v *value*.

(iii) Prove that if $e \hookrightarrow v$ (with $\cdot \vdash e : \tau$) then $e \mapsto^* v$.

Your rules should also be complete in the sense that if $e \mapsto^* v$ with v *value* then $e \hookrightarrow v$, but you do not need to prove this.

Exercise 2 Show cases for type abstraction and type application in the proof of progress (Theorem 1).

Exercise 3 Consider adding a new expression $\bot$ to our call-by-value language (with functions and Booleans) with the following evaluation and typing rules:

$$\frac{}{\bot \mapsto \bot}\ \mathsf{step/bot} \qquad \frac{}{\Gamma \vdash \bot : \tau}\ \mathsf{bot}$$

We do not change our notion of value, that is, $\bot$ is not a value.

1. Does preservation (Theorem L6.2) still hold? If not, provide a counterexample. If yes, show how the proof has to be modified to account for the new form of expression.

2. Does the canonical forms theorem (L6.4) still hold? If not, provide a counterexample. If yes, show how the proof has to be modified to account for the new form of expression.

3. Does progress (Theorem L6.3) still hold? If not, provide a counterexample. If yes, show how the proof has to be modified to account for the new form of expression.

Once we have nonterminating computation, we sometimes compare expressions using *Kleene equality*: e_1 and e_2 are Kleene equal ($e_1 \simeq e_2$) if they evaluate to the same value, or they both diverge (do not compute to a value). Since we assume we cannot observe functions, we can further restrict this definition: For $\cdot \vdash e_1 : \mathsf{bool}$ and $\cdot \vdash e_2 : \mathsf{bool}$ we write $e_1 \simeq e_2$ iff for all values v, $e_1 \mapsto^* v$ iff $e_2 \mapsto^* v$.

4. Give an example of two closed terms e_1 and e_2 of type bool such that $e_1 \simeq e_2$ but not $e_1 =_\beta e_2$, or indicate that no such example exists (no proof needed in either case).

Exercise 4 In our call-by-value language with functions, Booleans, and $\bot$ (see Exercise 3) consider the following specification of *or*, sometimes called "short-circuit or":

$$\begin{array}{lcl} \mathit{or}\ \mathsf{true}\ e & \simeq & \mathsf{true} \\ \mathit{or}\ \mathsf{false}\ e & \simeq & e \end{array}$$

where $e_1 \simeq e_2$ is Kleene equality from Exercise 3.

- We cannot define a *function or* : $\mathsf{bool} \to (\mathsf{bool} \to \mathsf{bool})$ with this behavior. Prove that it is indeed impossible.

- Show how to translate an expression $\mathit{or}\ e_1\ e_2$ into our language so that it satisfies the specification, and verify the given equalities by calculation.

Exercise 5 In our call-by-value language with functions, Booleans, and $\bot$ (see Exercise 3) consider the following specification of *por*, sometimes called "parallel or":

$$\begin{array}{lcl} \mathit{por}\ \mathsf{true}\ e & \simeq & \mathsf{true} \\ \mathit{por}\ e\ \mathsf{true} & \simeq & \mathsf{true} \\ \mathit{por}\ \mathsf{false}\ \mathsf{false} & \simeq & \mathsf{false} \end{array}$$

where $e_1 \simeq e_2$ is Kleene equality as in Exercises 3 and 4.

1. We cannot define a *function por* : $\mathsf{bool} \to (\mathsf{bool} \to \mathsf{bool})$ in our language with this behavior. Prove that it is indeed impossible.

2. We also cannot translate expressions *por* $e_1\ e_2$ into our language so that the result satisfies the given properties (which you do not need to prove). Instead consider adding a new primitive form of expression $\mathsf{por}\ e_1\ e_2$ to our language.

   (a) Give one or more typing rules for $\mathsf{por}\ e_1\ e_2$.

   (b) Provide one or more evaluation rules for $\mathsf{por}\ e_1\ e_2$ so that it satisfies the given specification and, furthermore, such that preservation, canonical forms, and progress continue to hold.

   (c) Show the new case(s) in the preservation theorem.

   (d) Show the new case(s) in the progress theorem.

   (e) Do your rules satisfy sequentiality? If not, provide a counterexample. If yes, just indicate that it is the case (you do not need to prove it).

Exercise 6 *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 $\tau_1 \mathbin{\&} \tau_2$. 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 7 Design the *lazy unit* $\langle\!|\, |\!\rangle$ as the nullary version of the lazy pairs of Exercise 6. 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 8 It is often stated that lazy pairs are not necessary in an eager language, because we can already define $\tau_1 \mathbin{\&} \tau_2$ and the corresponding constructors and destructors. Fill in this table.

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