In this lecture we introduce the topic of *scope* in the context of the λ-calculus and define translations from λ-CBV to FL for the two most common scoping rules, *static* and *dynamic* scoping.

1 Overview

Until now, we could look at a program as written and immediately determine where any variable was bound. This was possible because the λ-calculus uses *static scoping* (also known as *lexical scoping*).

The *scope* of a variable is where that variable can be mentioned and used. In static scoping, the places where a variable can be used are determined by the lexical structure of the program. An alternative to static scoping is *dynamic scoping*, in which a variable is bound to the most recent (in time) value assigned to that variable.

The difference becomes apparent when a function is applied. In static scoping, any free variables in the function body are evaluated in the context of the defining occurrence of the function; whereas in dynamic scoping, any free variables in the function body are evaluated in the context of the function call. The difference is illustrated by the following program:

$$\begin{array}{l} \mathsf{let}\ d = 2\ \mathsf{in} \\ \mathsf{let}\ f = \lambda x.\, x + d\ \mathsf{in} \\ \mathsf{let}\ d = 1\ \mathsf{in} \\ f\,2 \end{array}$$

In OCaml, which uses lexical scoping, the block above evaluates to 4:

1. The outer d is bound to 2.
2. The f is bound to $\lambda x.\, x + d$. Since d is statically bound, this is will always be equivalent to $\lambda x.\, x + 2$ (the value of d cannot change, since there is no variable assignment in this language).
3. The inner d is bound to 1.
4. When evaluating the expression $f\,2$, free variables in the body of f are evaluated using the environment in which f was defined. In that environment, d was bound to 2. We get $2 + 2 = 4$.

If the block is evaluated using dynamic scoping, it evaluates to 3:

1. The outer d is bound to 2.
2. The f is bound to $\lambda x.\, x + d$. The occurrence of d in the body of f is not locked to the outer declaration of d.
3. The inner d is bound to 1.
4. When evaluating the expression $f\,2$, free variables in the body of f are evaluated using the environment of the call, in which d is 1. We get $2 + 1 = 3$.

Dynamically scoped languages are quite common and include many interpreted scripting languages. Examples of languages with dynamic scoping are (in roughly chronological order): early versions of LISP, APL, PostScript, TeX, and Perl. Early versions of Python also had dynamic scoping before they realized their error.

Dynamic scoping does have some advantages:

- Certain language features are easier to implement.
- It becomes possible to extend almost any piece of code by overriding the values of variables that are used internally by that piece.

These advantages, however, come with a price:

- Since it is impossible to determine statically what variables will be accessible at a particular point in a program, the compiler cannot determine where to find the correct value of a variable, necessitating a more expensive variable lookup mechanism. With static scoping, variable accesses can be implemented more efficiently, as array accesses.
- Implicit extensibility makes it very difficult to keep code modular: the true interface of any block of code becomes the entire set of variables used by that block.

2 Scope and the Interpretation of Free Variables

Scoping rules are all about how to evaluate free variables in a program fragment. With static scope, free variables of a function $\lambda x.\, e$ are interpreted according to the syntactic context in which the term $\lambda x.\, e$ occurs. With dynamic scope, free variables of $\lambda x.\, e$ are interpreted according to the environment in effect when $\lambda x.\, e$ is applied. These are not the same in general.

We can demonstrate the difference by defining two translations $\mathcal{S}[\![\cdot]\!]$ and $\mathcal{D}[\![\cdot]\!]$ for static and dynamic scoping, respectively. These translations will convert λ-CBV with the corresponding scoping rule into FL.

For both translations, we use environments to capture the interpretation of names. An *environment* is simply a partial function with finite domain from variables x to values.

$$\rho : \mathit{Var} \rightharpoonup \mathit{Val}$$

Let $\rho[v/x]$ denote the environment that is identical to ρ except that its value at x is v:

$$\rho[v/x](y) \quad \triangleq \quad \begin{cases} \rho(y), & y \neq x, \\ v, & y = x. \end{cases}$$

The set of all environments is denoted Env.

We will need a mechanism to represent environments in the target language. For this purpose we need to code variables and environments as values of the target language. We write "x" to represent the code of the variable x. The exact nature of the coding is unimportant, as long as it is possible to look up the value of a variable given its code and update an environment with a new binding. Thus the only requirement is that there be definable methods lookup and update in the target language such that for any variable x and value v, if ρ' is a representation of the environment ρ, then

$$\mathsf{lookup}\,\rho'\,\text{"}x\text{"} \quad = \quad \begin{cases} \rho(x), & \text{if } x \in \mathsf{dom}\,\rho, \\ \mathsf{error}, & \text{if } x \notin \mathsf{dom}\,\rho \end{cases}$$

and $\mathsf{update}\,\rho'\,v\,\text{"}x\text{"}$ is a representation of $\rho[v/x]$.

For example, we might represent variables as integers and environments as a functions that take an integer input n and return the value associated with the variable whose code is n, or error if there is no such binding. With this encoding, the empty environment would be $\lambda n.\,\mathsf{error}$, and the environment that binds only the variable y to 2 could be represented as

$$\lambda n.\,\mathsf{if}\ n = \text{“}y\text{”}\ \mathsf{then}\ 2\ \mathsf{else}\ \mathsf{error}.$$

With this encoding we could define

$$\mathsf{lookup} \triangleq \lambda \rho n.\,\rho(n) \qquad\qquad \mathsf{update} \triangleq \lambda \rho v n.\,\lambda m.\,\mathsf{if}\ m = n\ \mathsf{then}\ v\ \mathsf{else}\ \rho(m).$$

Given such an encoding, let Env' denote the set of all representations of environments in the target language. The meaning of a language expression e is relative to the environment in which e occurs. Therefore, the meaning $[\![e]\!]$ is a function from environments to expressions in the target language,

$$[\![e]\!] : Env' \to \mathsf{FL},$$

where FL represents the set of all target language expressions. Thus $[\![e]\!]\rho$ is an expression in the target language FL involving values and environments that can be evaluated under the usual FL rules to produce a value.

3 Static Scoping

The translation for static scoping is:

$$\begin{aligned}
\mathcal{S}[\![x]\!]\,\rho &\triangleq \mathsf{lookup}\,\rho\,\text{“}x\text{”}\\
\mathcal{S}[\![e_1\,e_2]\!]\,\rho &\triangleq (\mathcal{S}[\![e_1]\!]\,\rho)\,(\mathcal{S}[\![e_2]\!]\,\rho)\\
\mathcal{S}[\![\lambda x.\,e]\!]\,\rho &\triangleq \lambda v.\,\mathcal{S}[\![e]\!]\,(\mathsf{update}\,\rho\,v\,\text{“}x\text{”}).
\end{aligned}$$

There are a couple of things to notice about this translation. It eliminates all of the variable names from the source program and replaces them with new names that are bound immediately at the same level. All λ-terms are closed, so there is no longer any role for the scoping mechanism of the target language to decide what to do with free variables.

4 Dynamic Scoping

The translation for dynamic scoping is:

$$\begin{aligned}
\mathcal{D}[\![x]\!]\,\rho &\triangleq \mathsf{lookup}\,\rho\,\text{“}x\text{”}\\
\mathcal{D}[\![e_1\,e_2]\!]\,\rho &\triangleq (\mathcal{D}[\![e_1]\!]\,\rho)\,(\mathcal{D}[\![e_2]\!]\,\rho)\,\rho \qquad (1)\\
\mathcal{D}[\![\lambda x.\,e]\!]\,\rho &\triangleq \lambda v\tau.\,\mathcal{D}[\![e]\!]\,(\mathsf{update}\,\tau\,v\,\text{“}x\text{”}) \qquad (2)
\end{aligned}$$

In (2), we have thrown out the lexical environment ρ and replaced it with a parameter τ. Thus the translation of a function no longer expects just a single argument v; it also expects to be provided with an environment τ describing the variable bindings at the call site. This environment is passed in to the function when it is called, as shown in (1).

Because a function can be applied in different and unpredictable locations that can vary at runtime, it is difficult in general to come up with an efficient representation of dynamic environments.

5 Correctness of the Static Scoping Translation

That static scoping is the scoping discipline of λ-CBV is captured in the following theorem.

Theorem 1. *For any λ-CBV expression e and $\rho \in Env$ such that $FV(e) \subseteq \mathsf{dom}\,\rho$, let $\rho' \in Env'$ be a representation of ρ. Then $\mathcal{S}[\![e]\!]\,\rho'$ is $\beta\eta$-equivalent to $e\{\rho(y)/y \mid y \in \mathrm{Var}\}$.*

Proof. By structural induction on e. For variables and applications,

$$\begin{array}{rcl}
\mathcal{S}[\![x]\!]\,\rho' & = & \mathsf{lookup}\,\rho'\,\text{“}x\text{”} \quad = \quad \rho(x) \quad = \quad x\{\rho(y)/y \mid y \in \mathrm{Var}\} \\
\mathcal{S}[\![e_1\,e_2]\!]\,\rho' & = & (\mathcal{S}[\![e_1]\!]\,\rho')\,(\mathcal{S}[\![e_2]\!]\,\rho') \\
 & =_{\beta\eta} & (e_1\{\rho(y)/y \mid y \in \mathrm{Var}\})\,(e_2\{\rho(y)/y \mid y \in \mathrm{Var}\}) \\
 & = & (e_1\,e_2)\{\rho(y)/y \mid y \in \mathrm{Var}\}.
\end{array}$$

For a λ-abstraction $\lambda x.\,e$, for any value v, since $\mathsf{update}\,\rho'\,v\,\text{“}x\text{”}$ is a representation for $\rho[v/x]$, by the induction hypothesis

$$\begin{array}{rcl}
\mathcal{S}[\![e]\!]\,(\mathsf{update}\,\rho'\,v\,\text{“}x\text{”}) & =_{\beta\eta} & e\{\rho[v/x](y)/y \mid y \in \mathrm{Var}\} \\
 & = & e\{\rho(y)/y \mid y \in \mathrm{Var} - \{x\}\}\{v/x\} \\
 & =_{\beta} & (\lambda x.\,e\{\rho(y)/y \mid y \in \mathrm{Var} - \{x\}\})\,v \\
 & = & ((\lambda x.\,e)\{\rho(y)/y \mid y \in \mathrm{Var}\})\,v.
\end{array} \tag{3}$$

Then

$$\begin{array}{rcll}
\mathcal{S}[\![\lambda x.\,e]\!]\,\rho' & = & \lambda v.\,\mathcal{S}[\![e]\!]\,(\mathsf{update}\,\rho'\,v\,\text{“}x\text{”}) & \\
 & = & \lambda v.\,((\lambda x.\,e)\{\rho(y)/y \mid y \in \mathrm{Var}\})\,v & \text{by (3)} \\
 & =_{\eta} & (\lambda x.\,e)\{\rho(y)/y \mid y \in \mathrm{Var}\}. &
\end{array}$$

□

The pairing of a function $\lambda x.\,e$ with an environment ρ is called a *closure*. The theorem above says that $\mathcal{S}[\![\cdot]\!]$ can be implemented by forming a closure consisting of the term e and an environment ρ that determines how to interpret the free variables of e. By contrast, in dynamic scoping, the translated function does not record the lexical environment, so closures are not needed.

6 Closure Conversion

In implementations of OCaml, Scheme, or other functional languages with static scoping, functions $\lambda x.\,e$ are paired with the environments ρ in which they were defined. The pair $\langle \lambda x.\,e,\, \rho\rangle$ is called a *closure*. The environment ρ tells how to evaluate the free variables of $\lambda x.\,e$. Usually the environment is a partial function from variables to values that is defined on (at least) all the free variables of $\lambda x.\,e$. It is obtained from the syntactic context in which $\lambda x.\,e$ appears.

When a function $\lambda x.\,e$ is called on some argument a, the argument is bound to the formal parameter x, and this new binding is appended to the environment ρ in the closure, then the body is evaluated in that environment.

The process of converting a function to take an environment as an extra argument is called *closure conversion*. In this lecture we will see why this works by proving a theorem that shows how closures adequately represent static scoping.

7 Closures and Environments

Formally, a *value* of λ-CBV is a closed CBV-irreducible λ-term, which is just a closed term of the form $\lambda x.\, e$. Let *Val* denote the set of λ-CBV values, and let Λ denote the set of all closed λ-terms (not necessarily CBV-irreducible).

Now we define a new language whose values are closures $\langle \lambda x.\, e,\, \rho\rangle$, where $\lambda x.\, e$ is permitted to have free variables, provided they are all in the domain of ρ. Because the definitions of closures and environments depend on each other, we must define them by mutual induction. We denote the set of closures and the set of environments by Cl and *Env*, respectively. They are defined to be the smallest sets such that

$$\begin{aligned} \mathit{Env} &= \{\text{partial functions } \rho : \mathit{Var} \to \mathsf{Cl} \text{ with finite domain}\}, \\ \mathsf{Cl} &= \{\langle \lambda x.\, e,\, \rho\rangle \mid \rho \in \mathit{Env},\ \mathit{FV}(\lambda x.\, e) \subseteq \mathsf{dom}\,\rho\}. \end{aligned}$$

This definition may seem circular, but actually it does have a basis. We have $\bot \in \mathit{Env}$, where $\bot$ is the null environment with domain $\varnothing$, therefore $\langle \lambda x.\, e,\, \bot\rangle \in \mathsf{Cl}$, where $\lambda x.\, e$ is closed. Once we know that Cl and *Env* are nonempty, we can form some nonnull environments $\rho : \mathit{Var} \to \mathsf{Cl}$ and closures $\langle \lambda x.\, e,\, \rho\rangle$ where $\lambda x.\, e$ is not closed, provided ρ is defined on all free variables of $\lambda x.\, e$. This allows us to form even more closures and environments, and so on. After countably many steps, we reach a fixpoint.

Permitting environments to be partial functions is essential, but the restriction to finite domain is not. We need to allow partial functions so that the induction will get off the ground, i.e. $\mathsf{Cl} \neq \varnothing$. The restriction to finite domain makes the monotone map in the definition finitary, which ensures that the construction closes at ω. Without this restriction we would have to use transfinite induction.

More concretely, define

$$\begin{aligned} \mathsf{Cl}_0 &\stackrel{\triangle}{=} \varnothing \\ \mathit{Env}_n &\stackrel{\triangle}{=} \{\text{partial functions } \rho : \mathit{Var} \to \mathsf{Cl}_n \text{ with finite domain}\} \\ \mathsf{Cl}_{n+1} &\stackrel{\triangle}{=} \{\langle \lambda x.\, e,\, \rho\rangle \mid \rho \in \mathit{Env}_n,\ \mathit{FV}(\lambda x.\, e) \subseteq \mathsf{dom}\,\rho\} \\ \mathit{Env} &\stackrel{\triangle}{=} \bigcup_{n\geq 0} \mathit{Env}_n \\ \mathsf{Cl} &\stackrel{\triangle}{=} \bigcup_{n\geq 0} \mathsf{Cl}_n. \end{aligned}$$

Note that

$$\begin{aligned} \mathit{Env}_0 &= \{\bot\} \\ \mathsf{Cl}_1 &= \{\langle \lambda x.\, e,\, \bot\rangle \mid \lambda x.\, e \text{ is closed}\}. \end{aligned}$$

8 Iterated Substitution

Let $\mathcal{C}$ denote the set of pairs $\langle e,\, \rho\rangle$, where e is any λ-term (not necessarily closed or CBV-irreducible) and ρ an environment such that $\mathit{FV}(e) \subseteq \mathsf{dom}\,\rho$. Every such pair gives rise to a closed λ-term obtained by "iterated substitution". This is given by a map $F : \mathcal{C} \to \Lambda$ defined inductively as follows:

$$F(\langle e,\, \rho\rangle) \stackrel{\triangle}{=} e\{F(\rho(y))/y,\ y \in \mathsf{dom}\ \rho\}.$$

Again, this may seem like a circular definition, but it's not.

Lemma F is well-defined on $\mathcal{C}$ and takes values in Λ. On inputs in Cl, F takes values in *Val*.

Proof. Induction on the stage of definition of $\rho \in \mathit{Env}$. Consider first $\langle e,\, \rho\rangle \in \mathcal{C}$ with $\rho \in \mathit{Env}_0$. Then $\rho = \bot$, e is closed, and

$$F(\langle e,\, \bot\rangle) \quad \stackrel{\triangle}{=} \quad e\{F(\bot(y))/y,\ y \in \mathsf{dom}\,\bot\} \quad = \quad e.$$

If $\rho \in \mathit{Env}_{n+1}$, then ρ takes values in Cl_{n+1}. Then for all $y \in \mathsf{dom}\,\rho$, $\rho(y) = \langle \lambda x.\, d,\, \sigma\rangle$ for some $\sigma \in \mathit{Env}_n$, $FV(\lambda x.\, d) \subseteq \mathsf{dom}\,\sigma$. By the induction hypothesis, $F(\rho(y)) \in \Lambda$. Then

$$F(\langle e,\, \rho\rangle) \quad = \quad e\{F(\rho(y))/y,\ y \in \mathsf{dom}\,\rho\} \quad \in \quad \Lambda.$$

F takes values in *Val* on inputs in Cl, because

$$F(\langle \lambda x.\, e,\, \rho\rangle) \quad = \quad (\lambda x.\, e)\{F(\rho(y))/y,\ y \in \mathsf{dom}\,\rho\},$$

which is closed and CBV-irreducible, thus a value of λ-CBV. $\square$

9 λ-Cl

The terms of λ-Cl consist of the elements of $\mathcal{C}$. The values of λ-Cl are elements of Cl. We now give a set of evaluation rules defining a big-step SOS for a binary relation $\Downarrow\, \subseteq \mathcal{C} \times \mathsf{Cl}$. The statement $\langle e,\, \rho\rangle \Downarrow v$ should be interpreted as: When e is evaluated in the environment ρ, the result is v. Given this informal meaning, these rules below reflect the usual evaluation rules for functional expressions in Scheme or ML.

$$\langle x,\, \sigma\rangle \Downarrow \sigma(x) \qquad \langle \lambda x.\, e,\, \rho\rangle \Downarrow \langle \lambda x.\, e,\, \rho\rangle \qquad \frac{\langle e_1,\, \sigma\rangle \Downarrow \langle \lambda x.\, e,\, \tau\rangle, \quad \langle e_2,\, \sigma\rangle \Downarrow u, \quad \langle e,\, \tau[u/x]\rangle \Downarrow v}{\langle e_1\, e_2,\, \sigma\rangle \Downarrow v}.$$

Note that the rule for λ-abstractions is just the identity relation! This rule says that evaluating $\lambda x.\, e$ in the environment ρ results in the closure $\langle \lambda x.\, e,\, \rho\rangle$.

The third rule is the usual rule for evaluation of a function application: to evaluate $e_1\, e_2$ in the environment σ, first evaluate the function e_1 in environment σ to get a closure $\langle \lambda x.\, e,\, \tau\rangle$, then evaluate the argument e_2 in environment σ, then bind the value of the argument to the formal parameter x in the environment of the closure τ and evaluate the body of the function in that environment.

By the lemma above, the "iterated substitution" map F translates λ-Cl expressions to λ-CBV expressions and λ-Cl values to λ-CBV values. The following theorem asserts adequacy of this translation.

Theorem $\quad F(\langle e,\, \sigma\rangle) \underset{\text{cbv}}{\overset{*}{\rightarrow}} v \quad \Leftrightarrow \quad \exists w\ \langle e,\, \sigma\rangle \Downarrow w \wedge v = F(w).$

Proof. ($\Leftarrow$) We wish to show that if $\langle e,\, \sigma\rangle \Downarrow w$, then $F(\langle e,\, \sigma\rangle) \underset{\text{cbv}}{\overset{*}{\rightarrow}} F(w)$. The proof is by induction on the derivation $\langle e,\, \sigma\rangle \Downarrow w$.

For the case $e = x$, we have $\langle x,\, \sigma\rangle \Downarrow \sigma(x)$. In this case $F(\langle x,\, \sigma\rangle) = x\{F(\sigma(y))/y,\ y \in \mathsf{dom}\,\sigma\} = F(\sigma(x))$, so $F(\langle x,\, \sigma\rangle) \underset{\text{cbv}}{\overset{*}{\rightarrow}} F(\sigma(x))$.

For the case $\lambda x.\, e$, we have $\langle \lambda x.\, e,\, \sigma\rangle \Downarrow \langle \lambda x.\, e,\, \sigma\rangle$. In this case $F(\langle x,\, \sigma\rangle) \underset{\text{cbv}}{\overset{*}{\rightarrow}} F(\langle x,\, \sigma\rangle)$ trivially.

Finally, for the case $e_1\, e_2$, if $\langle e_1\, e_2,\, \sigma\rangle \Downarrow w$, for some $\lambda x.\, d$, τ, and u, we must have

- $\langle e_1,\, \sigma\rangle \Downarrow \langle \lambda x.\, d,\, \tau\rangle$;

- $\langle e_2,\, \sigma\rangle \Downarrow u$;
- $\langle d,\, \tau[u/x]\rangle \Downarrow w$.

By the induction hypothesis,

- $F(\langle e_1,\, \sigma\rangle) \xrightarrow[\text{cbv}]{*} F(\langle \lambda x.\, d,\, \tau\rangle)$;
- $F(\langle e_2,\, \sigma\rangle) \xrightarrow[\text{cbv}]{*} F(u)$;
- $F(\langle d,\, \tau[u/x]\rangle) \xrightarrow[\text{cbv}]{*} F(w)$.

Under CBV semantics, we have

$$\begin{array}{rcl}
F(\langle e_1\, e_2,\, \sigma\rangle) & = & F(\langle e_1,\, \sigma\rangle)\ F(\langle e_2,\, \sigma\rangle) \\
& \xrightarrow[\text{cbv}]{*} & F(\langle \lambda x.\, d,\, \tau\rangle)\ F(u) \\
& = & (\lambda x.\, d)\{F(\tau(y))/y,\ y \in \mathsf{dom}\,\tau\}\ F(u) \\
& = & (\lambda x.\, d\{F(\tau(y))/y,\ y \in \mathsf{dom}\,\tau,\ y \neq x\})\ F(u) \\
& \xrightarrow[\beta]{} & d\{F(\tau[u/x](y))/y,\ y \in \mathsf{dom}\,\tau,\ y \neq x\}\{F(\tau[u/x](x))/x\} \\
& = & d\{F(\tau[u/x](y))/y,\ y \in \mathsf{dom}\,\tau\} \\
& = & F(\langle d,\, \tau[u/x]\rangle) \\
& \xrightarrow[\text{cbv}]{*} & F(w).
\end{array}$$

($\Rightarrow$) Suppose $F(\langle e,\, \sigma\rangle) \xrightarrow[\text{cbv}]{*} v$. We proceed by induction on the length of the derivation.

For the case $e = x$, we have $\langle x,\, \sigma\rangle \Downarrow \sigma(x)$ and

$$F(\sigma(x)) \quad = \quad x\{F(\sigma(y))/y,\ y \in \mathsf{dom}\,\sigma\} \quad = \quad F(\langle x,\, \sigma\rangle) \quad \xrightarrow[\text{cbv}]{*} \quad v.$$

By the lemma, $F(\sigma(x)) \in \mathit{Val}$, therefore $F(\sigma(x)) = v$, so we can take $w = \sigma(x)$.

For the case $\lambda x.\, e$, we have $\langle \lambda x.\, e,\, \sigma\rangle \Downarrow \langle \lambda x.\, e,\, \sigma\rangle$ and

$$F(\langle \lambda x.\, e,\, \sigma\rangle) \quad = \quad (\lambda x.\, e)\{F(\sigma(y))/y,\ y \in \mathsf{dom}\,\sigma\} \quad \in \quad \mathit{Val},$$

thus $F(\langle \lambda x.\, e,\, \sigma\rangle) = v$, so we can take $w = \langle \lambda x.\, e,\, \sigma\rangle$.

Finally, for the case $e_1\, e_2$, we have

$$F(\langle e_1\, e_2,\, \sigma\rangle) \quad = \quad F(\langle e_1,\, \sigma\rangle)\ F(\langle e_2,\, \sigma\rangle) \quad \xrightarrow[\text{cbv}]{*} \quad v.$$

Under the CBV reduction strategy, the only way this can happen is if

- $F(\langle e_1,\, \sigma\rangle) \xrightarrow[\text{cbv}]{*} \lambda x.\, d$;
- $F(\langle e_2,\, \sigma\rangle) \xrightarrow[\text{cbv}]{*} u$;
- $d\{u/x\} \xrightarrow[\text{cbv}]{*} v$

for some $\lambda x.\, d$ and $u \in \mathit{Val}$. These are all shorter derivations than the original derivation $F(\langle e,\, \sigma\rangle) \xrightarrow[\text{cbv}]{*} v$, so by the induction hypothesis there exist $\lambda x.\, e$, ρ, and $t \in \mathsf{Cl}$ such that

- $\langle e_1,\, \sigma\rangle \Downarrow \langle \lambda x.\, e,\, \rho\rangle$ and $F(\langle \lambda x.\, e,\, \rho\rangle) = \lambda x.\, d$;
- $\langle e_2,\, \sigma\rangle \Downarrow t$ and $F(t) = u$.

Then

$$\begin{array}{rcl} \lambda x.\, d & = & (\lambda x.\, e)\{F(\rho(y))/y,\ y \in \mathsf{dom}\,\rho\} \\ & = & \lambda x.\, (e\{F(\rho(y))/y,\ y \in \mathsf{dom}\,\rho,\ y \neq x\}), \end{array}$$

therefore $d = e\{F(\rho(y))/y,\ y \in \mathsf{dom}\,\rho,\ y \neq x\}$, and

$$\begin{array}{rcl} d\{u/x\} & = & e\{F(\rho(y))/y,\ y \in \mathsf{dom}\,\rho,\ y \neq x\}\{u/x\} \\ & = & e\{F(\rho(y))/y,\ y \in \mathsf{dom}\,\rho,\ y \neq x\}\{F(t)/x\} \\ & = & e\{F(\rho[t/x](y))/y,\ y \in \mathsf{dom}\,\rho,\ y \neq x\}\{F(\rho[t/x](x))/x\} \\ & = & e\{F(\rho[t/x](y))/y,\ y \in \mathsf{dom}\,\rho\} \\ & = & F(\langle e,\, \rho[t/x]\rangle). \end{array}$$

By the induction hypothesis, $\langle e,\, \rho[t/x]\rangle \Downarrow w$ and $F(w) = v$ for some w, therefore $\langle e_1\, e_2,\, \rho[t/x]\rangle \Downarrow w$. $\square$