15-312 Lecture on Binding and Scope

Binders

One of the motives of studying programming languages as mathematical objects is to identify notions and structures that occur, maybe with a different syntax, over and over across languages. We will see many throughout the course. One of the most fundamental and universal is the concept of *binder*, the ability to define a symbolic name so that every occurrence within a certain area of the program (its scope) share the same meaning (unless this name is redefined). Binders are found in every non trivial language, not only in Computer Science, but also in Logic, in Mathematics and in many other fields. Within a programming language binders are everywhere:

- defining a function introduces a binder for its name,
- its formal parameters are binders,
- a type declaration binds one and often more identifiers,
- declaring an object creates all kinds of binders,
- a module specification introduces even more binders, ...

Binding is one truly fundamental concept in Computer Science.

As an example throughout this notes, let's take a standard ML-like "`let`" construct:

$$\texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \texttt{ end}$$

which binds the name x to the expression E_1 within the body E_2. Intuitively (and this will be made precise later), x is a shortcut for E_1 anywhere it appears inside E_2.

Characteristics of Binders

The objects bound by a binder are called *names* or *identifiers*, sometimes *variable*. In the above "`let`" construct, x is a name. The portion of code where this name is bound is called the *scope* of the binder. Here it is the subexpression E_2. This means that every occurrence of x within E_2 refers to the x bound by this "`let`" and therefore to the expression E_1 (unless there is another "`let`" inside E_2 that binds this same x — we will see this later).

α-equivalence

Of course, the actual string that constitutes the name x in `let` x `=` E_1 `in` E_2 `end` has no importance: the only thing that matters is that all the places in E_2 where we want to have this shortcut to E_1 are identified by this same string. Indeed, (almost) any other string would do: we can change the name of our variables in a program to our preference. Indeed,

```
let x = 3 in 4*x + 4 end
```

and

```
let three = 3 in 4*three + 4 end
```

are essentially the same expression. This is called α-*renaming* and the two expressions are said to be α-*equivalent*.

The only thing to be careful of is that the name we choose does not clash with other names introduced before. For example, consider the following expression, which differs from the above only by the fact that we have factored out the "`4`" as the let-bound name "`y`":

```
let x = 3 in let y = 4 in y*x + y end end
```

We can change the name "`y`" to anything we want, except for "`x`", without altering the meaning of this expression. If we were to choose "`x`", we would obtain

```
let x = 3 in let x = 4 in x*x + x end end
```

and that would change the binding of the occurrence of "`x`" that was there before from "`3`" to "`4`". These two expressions are not α-equivalent because they do not have the same meaning (and not even the same value). That occurrence of "`x`" is said to have been *captured* during the renaming. Note that the second expression is legal: the value of x bound by the second `let` shadows the value associated to it by the first — the problem is that it does not have the same meaning as expression above it.

In summary, the actual string used for a bound name does not matter as long as we can distinguish it from any other name in use within the scope of the binder. Two expressions are α-equivalent if they differ at most by the name of their bound variable and no names that are distinct in one are identified in the other.

α-renaming is a very common operation, and we will assume it implicitly at the level of abstract syntax whenever convenient.

Substitution

Another fundamental operation on names is the *substitution* of a name with an expression wherever it occurs in another expression. Indeed, evaluating

```
let x = 3 in let y = 4 in y*x + y end end
```

proceeds by substituting every occurrence of the name "`x`" with "`3`" in the subexpression `let y = 4 in y*x + y end`, yielding:

```
let y = 4 in y*3 + y end
```

which will itself be evaluated by substituting "y" with "4" in `y*3 + y` obtaining `4*3 + 4`.

There is the danger of variable capture also when performing substitution. Assume that deep inside an expression there is the subexpression

```
let y = 1 in x + y end
```

(with "x" bound by some outer "`let`"). Say now that during evaluation "x" gets substituted with "`y*y`" (with this "y" coming from a "`let`" even further out). A naïve substitution would result in

```
let y = 1 in y*y + y end
```

which has changed the meaning of the original sub-expression by having the outer occurrence of "y" in "`y*y`" captured by the inner binder.

The proper way to do so is to α-rename the inner bound name to something else, for example

```
let z = 1 in x + z end
```

and then perform the substitution, which will produce

```
let z = 1 in y*y + z end
```

This is called *capture-avoiding substitution*. Every substitution can be made capture-avoiding by appropriately renaming the expression where the substitution occurs.

Substitution will be a pervasive operation at the level of abstract syntax and we will introduce a specific notation to denote it.

Abstract Binding Trees

Assume we have updated the arithmetic expression lexer from previous lectures to also tokenize `let` expressions. The token stream grammar is updated to

$$\begin{array}{llcl} \textit{Expressions} & E & ::= & \mathsf{num}[n] \mid E + E \mid E * E \\ & & \mid & \mathsf{id}[x] \mid \mathsf{let}[x]\ E\ \mathsf{in}\ E\ \mathsf{end} \end{array}$$

Note that lexing now shall recognize names (denoted x here) according to whatever syntax defines them. It also defines four new tokens: $\mathsf{id}[x]$ represents name x within an expression, $\mathsf{let}[x]$ collects the `let` keyword and the identifier it introduces, in and end are just delimiters.

The validity of a token stream as an expression is best described by the general judgment

$$\vdash_\Sigma\ t\ \mathsf{exp}$$

where the parameter set Σ collects names as they are introduced through the let token. A name x found in an expression is accepted only if a note of it has previously been made in Σ (i.e., if it was introduced by a let). Here, we shall assume that identifiers

have already been renamed apart so that no clashes occur (or alternatively that Σ is ordered and the rightmost occurrence of a name is used).

This grammar is then transliterated to the following deductive system:

$$\frac{}{\vdash_\Sigma \ \mathsf{num}[n] \ \ \mathsf{exp}}\ \mathsf{num}[\mathbf{n}] \qquad \frac{\vdash_\Sigma \ t_1 \ \ \mathsf{exp} \quad \vdash_\Sigma \ t_2 \ \ \mathsf{exp}}{\vdash_\Sigma \ t_1\hat{}+\hat{}t_2 \ \ \mathsf{exp}}\ \mathsf{plus} \qquad \frac{\vdash_\Sigma \ t_1 \ \ \mathsf{exp} \quad \vdash_\Sigma \ t_2 \ \ \mathsf{exp}}{\vdash_\Sigma \ t_1\hat{}*\hat{}t_2 \ \ \mathsf{exp}}\ \mathsf{times}$$

$$\frac{}{\vdash_{\Sigma,x} \ x \ \ \mathsf{exp}}\ \mathbf{id}[\mathbf{x}] \qquad \frac{\vdash_\Sigma \ t_1 \ \ \mathsf{exp} \quad \vdash_{\Sigma,x} \ t_2 \ \ \mathsf{exp}}{\vdash_\Sigma \ \mathsf{let}[x] \ \hat{}t_1\hat{} \ \mathsf{in} \ \hat{}t_2\hat{} \ \mathsf{end} \ \ \mathsf{exp}}\ \mathbf{let}[\mathbf{x}]$$

In rule $\mathbf{id}[\mathbf{x}]$, a name x is accepted only if it occurs in the parameter set (which we expressed as the standard abbreviation Σ, x). Rule $\mathbf{let}[\mathbf{x}]$ describes how a name ends up in the parameter set Σ: when parsing a token string $\mathsf{let}[x] \ \hat{}t_1\hat{} \ \mathsf{in} \ \hat{}t_2\hat{} \ \mathsf{end}$, the subexpression t_1, to which x is meant to be bound, is parsed as usual. If this expression is well-formed, it is expected, or at least likely, that x occurs in t_2. We describe the fact that x can legally occur in t_2 by parsing it relative to a parameter set that extends Σ with x.

In a binder-free language, we could read the abstract syntax of a grammar off the parsing derivation: we simply used the rule names as operators. This will still be possible in the presence of names and binders, but we need to introduce some machinery first. This machinery is a universal binder called an *abstractor*: it has the form

$$x.E$$

where x is a name and E is a term in abstract syntax that may mention x.[1] Any time we encounter a construct that binds a name in one of its subexpressions, its abstract syntax will have an abstractor in the corresponding argument. For example, the token stream $\mathsf{let}[x] \ \hat{}t_1\hat{} \ \mathsf{in} \ \hat{}t_2\hat{} \ \mathsf{end}$ will yield the term $\mathsf{let}(E_1, x.E_2)$, where E_1 and E_2 are the terms corresponding to t_1 and t_2 respectively.

With this new tool, we are again in a position to read the abstract syntax of a grammar directly from the inference rules that correspond to each production: we continue to use the rule names as operators and we continue to take each premise as an argument. Now however, every time a premise extends the parameter set, we record this fact by making the corresponding argument into an abstractor. Applying this idea to rule $\mathbf{let}[\mathbf{x}]$, we obtain the operator $\mathsf{let}(\_, x.\_)$. Altogether we have the following grammar our expressions.:

$$\begin{array}{lrcl} \textit{Abstract Expressions:} & E & ::= & \mathsf{num}[n] \mid \mathsf{plus}(E,E) \mid \mathsf{times}(E,E) \\ & & \mid & \mathsf{id}[x] \mid \mathsf{let}(E, x.E) \end{array}$$

The resulting terms are called *abstract binding trees* and this form of syntax is known as *higher-order abstract syntax*. As usual, we will generally omit the tags num and id.

Abstract α-equivalence and substitution

If we take an abstractor as a universal binder, we need to define α-renaming, α-equivalence and capture-avoiding substitution for it. Chapter 6 of Harper's book contains an excellent account of all these concepts: it very precisely defines these notions

[1] A more common syntax for an abstractor is $\lambda x.E$.

as a series of deductive systems. What follows are some notes on syntax and assumptions.

At this abstract level, names are usually taken to be symbolic objects rather than strings of characters: we will often use the meta-variables x, y, z for names, but we can think that they come from an infinite set. This also means that, differently from concrete names which are just strings, abstract names are not inductively defined. In a judgment, we can easily indicate that two names should be the same (we write two occurrences of x, say), but how do we say that two names, say x and y, should be different? To do so, we rely on the judgment $x\#y$ name. Because abstract names are not inductively defined, we shall assume that this judgment is given to us not by rules but as a generally infinite set of instances. This is one of the very few judgments we will encounter that are not inductively defined, maybe the only one.

Given a term E that may contain names x and y free (i.e., outside the scope of a binder for them), expression E' being obtained by swapping x for y within E is defined by the judgment $[x \leftrightarrow y]E = E'$ abt in Harper's book.[2] If x or y do not occur (free) in E, the rules given by Harper safely implements α-renaming. Given this judgment it is easy to define α-equivalence as another judgment, written $E =_\alpha E'$ abt. Given a term E which may contain the name x free, the capture-avoiding substitution of some other term E' for x in E is then denoted $[E'/x]E$ as a meta-operation, and implemented by the judgment $[E'/x]E = E''$ abt. .

In what follows, we will implicitly rely on α-renaming to avoid capture and rename variables to whatever will be most convenient as we go along. This means that every term and even judgment will be considered modulo α-equivalence.

Inductive Definitions over Abstract Binding Trees

Abstract binding trees can participate in inductive definitions in the same way as the abstract syntax trees we have seen so far. The presence of abstractors has however the effect that these inductive definitions typically rely on general judgments (judgments that are both hypothetical and parametric — see Harper's book, Chapters 7–8). Let's build on our example and define what it means for two (abstract) expressions to be equal. We will do so by means of the judgment

$$\Gamma \vdash_\Sigma \ E_1 = E_2 \ \ \mathsf{exp}$$

where Σ records the names that have been encountered so far in abstractors, and Γ lists assumptions of the form $x = y$ exp regarding the equality of names x and y

[2] Where Harper uses the token abt — for "abstract binding tree" in a judgment, we have used exp — for expression. In general, we use some mnemonic string for the non-terminal in the grammatical production corresponding to a rule rather than a catch-all form such as abt.

encountered before. This judgment is defined by the following set of rules:[3]

$$\frac{}{\Gamma \vdash_\Sigma\ n = n\ \ \mathsf{exp}}\ {=}\mathbf{num} \qquad \frac{}{\Gamma, x = y\ \ \mathsf{exp} \vdash_{\Sigma,x,y}\ x = y\ \ \mathsf{exp}}\ {=}\mathbf{id}$$

$$\frac{\Gamma \vdash_\Sigma\ E_1 = E_2\ \ \mathsf{exp} \quad \Gamma \vdash_\Sigma\ E_1' = E_2'\ \ \mathsf{exp}}{\Gamma \vdash_\Sigma\ \mathsf{plus}(E_1, E_1') = \mathsf{plus}(E_2, E_2')\ \ \mathsf{exp}}\ {=}+$$

$$\frac{\Gamma \vdash_\Sigma\ E_1 = E_2\ \ \mathsf{exp} \quad \Gamma \vdash_\Sigma\ E_1' = E_2'\ \ \mathsf{exp}}{\Gamma \vdash_\Sigma\ \mathsf{times}(E_1, E_1') = \mathsf{times}(E_2, E_2')\ \ \mathsf{exp}}\ {=}*$$

$$\frac{\Gamma \vdash_\Sigma\ E_1 = E_2\ \ \mathsf{exp} \quad \Gamma, x = y\ \ \mathsf{exp} \vdash_{\Sigma,x,y}\ E_1' = E_2'\ \ \mathsf{exp}}{\Gamma \vdash_\Sigma\ \mathsf{let}(E_1, x.E_1') = \mathsf{let}(E_2, y.E_2')\ \ \mathsf{exp}}\ {=}\mathbf{let}$$

We prove properties of general judgments in pretty much the same way as we did for categorical judgments: a typical proof proceeds by cases on the last rule that has been applied in one or more given derivations and derives an induction hypothesis from its premises. The main difference is that we now have to consider objects that consist not only of a simple judgment form (like say $E\ \ \mathsf{exp}$), but also of hypotheses (Γ) and parameters (Σ) and that we make sure that whatever a rule does to these is taken into account when applying the induction hypothesis.

Concretely, let's prove that equality over expressions is associative:

> *Given derivations* $\mathcal{D} :: \Gamma \vdash_\Sigma\ E_1 = E_2\ \ \mathsf{exp}$ *and* $\mathcal{E} :: \Gamma \vdash_\Sigma\ E_2 = E_3\ \ \mathsf{exp}$, *there exists a derivation of* $\mathcal{F} :: \Gamma \vdash_\Sigma\ E_1 = E_3\ \ \mathsf{exp}$.[4]

Proof: Because of the extreme regularity of the rules for equality (there is actually a deep principle here), this proof can proceed by induction on any of E_1, E_2, E_3, $\mathcal{D}$ or $\mathcal{F}$. Let's conduct it by induction on E_1. We distinguish cases based on the productions for abstract expressions — there are 5 possibilities:

1. Case $E_1 = n$ (recall that we are omitting the tags for conciseness). There is exactly one rule for $\Gamma \vdash_\Sigma\ E_1 = E_2\ \ \mathsf{exp}$ that mentions a numeral between "$\vdash_\Sigma$" and "$=$": it is $= \mathbf{num}$. Therefore $\mathcal{D}$ must have the form

$$\mathcal{D} = \frac{}{\Gamma \vdash_\Sigma\ n = n\ \ \mathsf{exp}}\ {=}\mathbf{num}$$

[3]This example is admittedly artificial for several reasons: first, we are simply specifying the α-equivalence of two expression, and so, since we are operating modulo α-equivalence, it could be defined more succinctly by the single rule

$$\frac{}{E = E}$$

Second, even if we wanted to traverse the terms, α-equivalence, which we are using in rule $= \mathbf{let}$ to make sure names are different, could be used more economically to make them the same, which would allow us to avoid introducing the context Γ altogether.

[4]To be fully precise, we should conclude this statement with the phrase "*for every* Γ, Σ, E_1, E_2, *and* E_3". Qualifiers such as these, that can easily be inferred from the context, are often omitted.

(this technique, of identifying which rule must have been applied in a derivation based on the knowledge of a piece of the judgment it derives, is called *inversion* — it is a very general tools when doing proofs over derivations.)

The form of $\mathcal{D}$ entails that $E_2 = n$. Knowing this, we apply the exact same technique again and obtain that $\mathcal{E}$ is again an application of $= \mathbf{num}$ (indeed, $\mathcal{E} = \mathcal{D}$) and that $E_3 = n$.

Then, we trivially build a derivation of $\Gamma \vdash_\Sigma \ E_1 = E_3 \ \ \mathsf{exp}$ (where again $E_1 = E_3 = n$) by setting $\mathcal{F} = \mathcal{D}$.

2. Case $E_1 = x$. By inversion, we obtain that

$$\mathcal{D} = \frac{}{\Gamma, x = y \ \ \mathsf{exp} \vdash_{\Sigma,x,y} \ x = y \ \ \mathsf{exp}} =\mathbf{id} \qquad E_2 = y$$

$$\mathcal{E} = \frac{}{\Gamma, y = z \ \ \mathsf{exp} \vdash_{\Sigma,y,z} \ y = z \ \ \mathsf{exp}} =\mathbf{id} \qquad E_3 = z,$$

Then, we can define $\mathcal{F}$ to be yet another instance of rule $= \mathbf{id}$ that refers to names x and z:

$$\mathcal{E} = \frac{}{\Gamma, x = z \ \ \mathsf{exp} \vdash_{\Sigma,x,z} \ x = z \ \ \mathsf{exp}} =\mathbf{id}$$

3. Case $E_1 = \mathsf{plus}(E_1', E_1'')$. Then, by inversion

$$\mathcal{D} = \frac{\begin{array}{cc}\mathcal{D}' & \mathcal{D}'' \\ \Gamma \vdash_\Sigma \ E_1' = E_2' \ \ \mathsf{exp} & \Gamma \vdash_\Sigma \ E_1'' = E_2'' \ \ \mathsf{exp}\end{array}}{\Gamma \vdash_\Sigma \ \mathsf{plus}(E_1', E_1'') = \mathsf{plus}(E_2', E_2'') \ \ \mathsf{exp}} =+ \qquad E_2 = \mathsf{plus}(E_2', E_2'')$$

$$\mathcal{E} = \frac{\begin{array}{cc}\mathcal{E}' & \mathcal{E}'' \\ \Gamma \vdash_\Sigma \ E_2' = E_3' \ \ \mathsf{exp} & \Gamma \vdash_\Sigma \ E_2'' = E_3'' \ \ \mathsf{exp}\end{array}}{\Gamma \vdash_\Sigma \ \mathsf{plus}(E_2', E_2'') = \mathsf{plus}(E_3', E_3'') \ \ \mathsf{exp}} =+ \qquad E_3 = \mathsf{plus}(E_3', E_3'')$$

By two applications of the induction hypothesis on $\mathcal{D}'$ and $\mathcal{E}'$, and on $\mathcal{D}''$ and $\mathcal{E}''$ respectively, there are derivations $\mathcal{F}' :: \Gamma \vdash_\Sigma \ E_1' = E_3' \ \ \mathsf{exp}$ and $\mathcal{F}'' :: \Gamma \vdash_\Sigma \ E_1'' = E_3'' \ \ \mathsf{exp}$. Then, we can obtain the desired derivation $\mathcal{F}$ by simply combining $\mathcal{F}'$ and $\mathcal{F}''$ using rule $= +$:

$$\mathcal{F} = \frac{\begin{array}{cc}\mathcal{F}' & \mathcal{F}'' \\ \Gamma \vdash_\Sigma \ E_1' = E_3' \ \ \mathsf{exp} & \Gamma \vdash_\Sigma \ E_1'' = E_3'' \ \ \mathsf{exp}\end{array}}{\Gamma \vdash_\Sigma \ \mathsf{plus}(E_1', E_1'') = \mathsf{plus}(E_3', E_3'') \ \ \mathsf{exp}} =+$$

4. Case $E_1 = \mathsf{times}(E_1', E_1'')$. Done in the exact same way.

5. Case $E_1 = \mathsf{let}(E_1', x.E_1'')$. By inversion, we obtain

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \qquad\qquad \mathcal{D}'' \\ \Gamma \vdash_\Sigma \ E_1' = E_2' \ \ \mathsf{exp} \quad \Gamma, x = y \ \ \mathsf{exp} \vdash_{\Sigma,x,y} \ E_1'' = E_2'' \ \ \mathsf{exp}\end{array}}{\Gamma \vdash_\Sigma \ \mathsf{let}(E_1', x.E_1'') = \mathsf{let}(E_2', y.E_2'') \ \ \mathsf{exp}} =_{\mathbf{let}}$$

$$\mathcal{E} = \dfrac{\begin{array}{c}E_2 = \mathsf{let}(E_2', y.E_2'') \qquad E_3 = \mathsf{let}(E_3', z.E_3'') \\ \mathcal{E}' \qquad\qquad \mathcal{E}'' \\ \Gamma \vdash_\Sigma \ E_2' = E_3' \ \ \mathsf{exp} \quad \Gamma, y = z \ \ \mathsf{exp} \vdash_{\Sigma,y,z} \ E_2'' = E_3'' \ \ \mathsf{exp}\end{array}}{\Gamma \vdash_\Sigma \ \mathsf{let}(E_2', y.E_2'') = \mathsf{let}(E_3', z.E_3'') \ \ \mathsf{exp}} =_{\mathbf{let}}$$

Then, by induction hypothesis on $\mathcal{D}'$ and $\mathcal{E}'$, we obtain that there exists a derivation $\mathcal{F}'$ of $\Gamma \vdash_\Sigma \ E_1' = E_3' \ \ \mathsf{exp}$, and similarly, by induction hypothesis on $\mathcal{D}''$ and $\mathcal{E}''$ we obtain a derivation $\mathcal{F}''$ of $\Gamma, x = z \ \ \mathsf{exp} \vdash_{\Sigma,x,z} \ E_1'' = E_3'' \ \ \mathsf{exp}$. We now simply apply rule $=\mathbf{let}$ once more:

$$\mathcal{F} = \dfrac{\begin{array}{c}\mathcal{F}' \qquad\qquad \mathcal{F}'' \\ \Gamma \vdash_\Sigma \ E_1' = E_3' \ \ \mathsf{exp} \quad \Gamma, x = z \ \ \mathsf{exp} \vdash_{\Sigma,x,z} \ E_1'' = E_3'' \ \ \mathsf{exp}\end{array}}{\Gamma \vdash_\Sigma \ \mathsf{let}(E_1', x.E_1'') = \mathsf{let}(E_3', z.E_3'') \ \ \mathsf{exp}} =_{\mathbf{let}}$$

which proves our result.

□