Local Definitions

Every model of computation is usually supplemented with a *definition mechanism*. A definition associates a name or identifier with a concept, where we use the word "concept" informally. Technically, the terminology used for the name (or whatever is being defined) is *definiendum*, and the concept in terms of which it is being defined is called the *definiens*.

In the context of our programming languages, a value definition gives a name (the definiendum) to an expression or the value to which it evaluates or calculates (the definiens). A function definition gives a name to a mapping from input values to outputs, which may be specified in mathematics or, in this course, as a program in a programming language. A type definition gives a name to a type or a type construction. And so on.

Definitions are useful, since instead of having to trot out the entire concept, one can refer to it by the given name. In the case of functions, we can provide arguments to the named entity, which gives rise to a fairly concise expression such as "`fibonacci(6)`" instead of describing the function that computes the fibonacci numbers, and then saying that this function is applied to the number 6.

But we do not want to remember everything to which we give a name. We would then end up with a huge space of names, and would have trouble recalling what concept each of these names stands for. Very often we want to define a name *temporarily*, use it and then forget that definition. Ordinary mathematics is replete with such usages: "let x be an integer greater than 0" or "let $d = b^2 - 4ac$" in a larger expression. In other words, we often localise the definition to a limited *scope*, outside of which the name is not associated with anything (or if it is, we revert to whatever earlier association that name previously had).

Such locally-scoped names are a way of managing large programs, while being able to focus attention on that part of the program which is of current interest.

Let us use the following syntax for definitions:

$$d \in Defs \ ::= \ \mathbf{def}\ x = e_1$$

A possible data type definition in OCaml to represent an abstract syntax for definitions is

```
type defs = Adef of string * exp ;;
```

which we will expand as we proceed.

This simple definition can be used as a temporary or local definition within an expression:

$$\mathbf{let}\ \mathbf{def}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{ni}$$

(the abstract syntax keywords **let**, **in** and **ni** are only delimiting markers to indicate a definition $\mathbf{def}\ x = e_1$ is *local* to the *scope* of the expression e_2.

Such a scoping discipline is called "*lexical scoping*" or "*static scoping*", since one can readily determine the scope of a definition by inspecting the text of the program. If one writes the expressions out across multiple lines and uses indentation, the scopes of definitions become quite clear to see; one can conceptually draw bounding boxes around the scopes, and get a clear sense of what variables are (relatively) "*global*" to a let-expression, and what are "local".

There is a well-established *principle* via which we will introduce such expressions into the language (See below). But first let us formalise the typing rules and denotational and operational semantics for this sort of expression.

$$(\textbf{LetT})\ \frac{\Gamma \vdash \underline{e_1} : \tau_1 \qquad \Gamma[\underline{x} : \tau_1] \vdash \underline{e_2} : \tau_2}{\Gamma \vdash \underline{\text{let def } x = e_1 \text{ in } e_2 \text{ ni}} : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \underline{\text{let def } x = e_1 \text{ in } e_2 \text{ni}} : \tau_2}$$

The typing rule (**LetT**) details how a let-expression with a simple local definition is type-checked. We first type-check, with respect to type assumption Γ, the expression e_1, and suppose its type is τ_1. Then, taking the extension (augmentation) of the type assumption Γ with the binding of variable x to the type τ_1, we type-check the “body” of the let-expression, namely e_2, and suppose this has type τ_2. Then, the whole let-expression, in the context of type assumption Γ has type τ_2.

The denotational definition is extended with the following case:

$$eval[\![\underline{\text{let def } x = e_1 \text{ in } e_2 \text{ ni}}\ ]\!]\ \rho = eval[\![\ \underline{e_2}\ ]\!]\ \rho'$$
$$\text{where } \rho' = \rho[x \mapsto eval[\![\ \underline{e_1}\ ]\!]\ \rho]$$

$$eval[\text{let } x = e_1 \text{ in } e_2 \text{ni}]\rho = eval[e_2]\rho'$$

That is, evaluate expression e_2 using a valuation that is the valuation ρ extended (augmented) with the binding of the variable x to a value v which is equal to $eval[\![\ \underline{e_1}\ ]\!]\ \rho$.

when $\rho' = \rho[x \mapsto eval[\![e_1]\!]]$

The operational specification is extended with the inference rule:

$$(\textbf{CalcLet})\ \frac{\gamma \vdash \underline{e_1} \Longrightarrow \underline{a_1} \qquad \gamma[\underline{x} \mapsto \underline{a_1}] \vdash \underline{e_2} \Longrightarrow \underline{a_2}}{\gamma \vdash \underline{\text{let def } x = e_1 \text{ in } e_2 \text{ ni}} \Longrightarrow \underline{a_2}}$$

The operational semantics rule (**CalcLet**) details how a let-expression with a simple local definition is calculated with respect to a table γ. We first calculate, with respect to table γ, the expression e_1, and suppose its answer is a_1. Then, taking the extension (augmentation) of the table γ with the binding of variable x to the answer a_1, we calculate the expression e_2, and suppose this yields answer a_2. Then, the whole let-expression, calculated in the context of table γ yields answer a_2.

Notice the similarity in shape between the operational rule (**CalcLet**) and the typing rule (**LetT**) — this will be important for extending the Type Preservation result.

Principle of Qualification

The Principle of Qualification states that *any* (abstract) syntactic category can be enriched by qualifying its elements with a local definition.

This principle gives us the essence of block-structured languages. We will start with qualified expressions, namely the the let-expressions such as those of the form seen above. Note that in such a qualified expression, while the expression e_1 may use any variable (name/indentifier) that has been previously defined, the body of the let-expression, viz e_2 can additionally use the defined variable x.

Exercise: Extend the OCaml definition of the data type `exp` to incorporate let-expressions of the form defined above.
Exercise: Extend the syntactic functions `ht` and `size`.
Exercise: Extend the definition of the definitional interpreter `eval` (in OCaml) to support let-expressions of the form shown above.
Exercise: Extend the type checking relation `hastype` (in Prolog) to incorporate let expressions of the simple form defined above.
Exercise: Extend the big-step operational semantics (coded in Prolog) to incorporate let-expressions of the simple form shown above.

How should the Stack Machine be modified? In particular, consider how we should represent the table component. Since the table will be temporarily extended, we can use a stack as a data structure — so instead of a monolithic table, we form our table by stacking up mini-tables, where new bindings are placed at the top of the stack, and popped off when we need to exit the scope of a definition.

Notice that the definition *binds* a name to a value/answer temporarily, and this binding is not visible outside the scope of the let-expression. So it makes sense to refine the concept of the syntactic function *vars*. Instead we will talk about *binding* occurrences of variables — to the left of the definitional equals — and occurrences of these variables in the scope of this definition which are *bound* (not “bounded’! *Bound* is the past-tense of *bind* here) by this binding occurrence. Occurrences of variables which are not bound are called *free*. So we define the set of free variables in an expression, i.e., those variables which have at least one free occurrence in an expression. Likewise we also can define the set of variables being defined in a definition.

Exercise: Refine the definition of *vars* to define a function *fv* that returns the set of free variables in a given expression.

Structuring Definitions

Notice that the block structure of a let-expression can itself appear in the scope of a let-expression. For example:
let def $x = e_1$ **in let def** $y = e_2$ **in let def** $z = e_3$ **in** e_4 **ni ni ni**
where the scopes can be better seen as follows:

```
        let def x = e1
         in
             let def y = e2
              in
                  let def z = e3
                   in
                       e4
                   ni
              ni
         ni
```

Observe that the expression e_1 can legitimately use only variables that were defined in the environment *outside* the whole expression, whereas the expression e_2 can use all those variables as well as the variable x, and the expression e_3 can additionally use those the variable y, while expression e_4 can use all the original (“global”) variable from the context as well as the defined variables x, y, z.

Now, we might easily see that such cascading forms are a nuisance to write, and we might instead say that we may provide structure within the structure of definitions itself. After all, we can see that the cascaded definitions are a form of sequential definitions, which we can write as $d_1; d_2$. In such sequentially structured definitions, the definition d_1 is "processed" and then in the augmented environment (whether type assumption or table) obtained after processing it, the definition d_2 is processed.

Let $dv(d)$ denote the variables defined in a definition d. Clearly $dv(\mathbf{def}\ x = e) = \{x\}$. Now, $dv(d_1; d_2) = dv(d_1) \cup dv(d_2)$.

Sometimes there are situations where we wish to make multiple definitions but where we do not want to make them in a sequential order, but instead "in parallel" — that is the new definitions are both made and processed with respect to the same prior environment. Note that such a definitional form is of the shape $d_1 \parallel d_2$, which is well-formed only when $dv(d_1) \cap dv(d_2) = \{\}$, i.e., the set of defined variables of d_1, d_2 are *disjoint*. Otherwise, we could have conflicting definitions leading to inconsistency (in the typing or in the tables). Notice that $dv(d_1 \parallel d_2) = dv(d_1) \cup dv(d_2)$. Again!

Therefore, we add structure to our (abstract) syntactic category of *definitions*, extending it as follows:

$$d \in Defs \ ::= \ \mathbf{def}\ x = e_1 \ \mid \ d_1; d_2 \ \mid \ d_1 \parallel d_2$$

Static Semantics (Types)

Now let us define a relation $\Gamma \vdash \underline{d} :> \Gamma'$ similar to type-checking, but now working on the syntactic category of definitions. We call this type-elaboration or type extension. The result of type-elaborating a definition d is a type assumption Γ'. The type-elaboration of a definition d is conducted with respect to a given type assumption Γ. For reasons that should become clear very soon, we opt to present the type-elaboration relation in an incremental form — where the type-elaboration of a definition gives us the change (the "delta") in the type environment, rather than the cumulative new type environment. Note that getting the incremental form Γ' allows us to represent the cumulative new environment as the augmentation $\Gamma[\Gamma']$, as well as quickly undo and revert to the prior environment Γ. (In practice, this is implemented using a stack data structure, pushing and popping bindings)

(ADefT) $$\frac{\Gamma \vdash \underline{e_1} : \tau}{\Gamma \vdash \underline{\mathrm{def}\ x = e\ } :> \{x : \tau\}}$$

If under type assumption Γ, expression e has type τ, then type elaborating the simple definition $\underline{\mathrm{def}\ x = e\ }$ yields the (incremental) type assumption $\{x : \tau\}$.

(SeqDefT) $$\frac{\Gamma \vdash \underline{d_1} :> \Gamma_1 \qquad \Gamma[\Gamma_1] \vdash \underline{d_2} :> \Gamma_2}{\Gamma \vdash \underline{d_1; d_2} :> \Gamma_1[\Gamma_2]}$$

If under type assumption Γ, type-elaborating the definition d_1 yields the (incremental) type assumption Γ_1, and after augmenting the type assumption Γ with Γ_1, the type elaboration of the definition d_2 yields the (incremental) type assumption Γ_2, then under type assumption Γ, type-elaborating the sequential definition $d_1; d_2$ yields the (incremental) type assumption $\Gamma_1[\Gamma_2]$.

$$\textbf{(ParDefT)}\ \frac{\Gamma \vdash \underline{d_1} :> \Gamma_1 \qquad \Gamma \vdash \underline{d_2} :> \Gamma_2}{\Gamma \vdash \underline{d_1 \parallel d_2} :> \Gamma_1 \cup \Gamma_2}$$

If under type assumption Γ, type-elaborating the definition d_1 yields the (incremental) type assumption Γ_1, and under the *same* type assumption Γ, type elaborating the definition d_2 yields the (incremental) type assumption Γ_2, then under type assumption Γ, type-elaborating the parallel definition $d_1 \parallel d_2$ yields the (incremental) type assumption $\Gamma_1 \cup \Gamma_2$. Note that since by assumption of defined variables being disjoint, $dv(d_1) \cap dv(d_2) = \{\}$, we have $\Gamma_1 \cup \Gamma_2 = \Gamma_1[\Gamma_2] = \Gamma_2[\Gamma_1]$.

Denotational Semantics for processing definitions

We define a new function $elab[\![\_]\!]\_ : Defs \to [\mathcal{X} \to \mathbb{V}] \to [\mathcal{X} \to_{fin} \mathbb{V}]$

$$elab[\![\underline{\mathbf{def}\ x = e}]\!]\rho = \{\ x \mapsto eval[\![e]\!]\rho\ \}$$

$$elab[\![\underline{d_1; d_2}]\!]\rho = \rho_1[\rho_2] \text{ where } \rho_1 = elab[\![\underline{d_1}]\!]\rho \text{ and } \rho_2 = elab[\![\underline{d_2}]\!](\rho[\rho_1])$$

$$elab[\![\underline{d_1 \parallel d_2}]\!]\rho = \rho_1 \cup \rho_2 \text{ where } \rho_1 = elab[\![\underline{d_1}]\!]\rho \text{ and } \rho_2 = elab[\![\underline{d_2}]\!]\rho$$

Note that since by assumption of defined variables being disjoint, $dv(d_1) \cap dv(d_2) = \{\}$, we have $\rho_1 \cup \rho_2 = \rho_1[\rho_2] = \rho_2[\rho_1]$.

Operational Semantic Specification for elaborating definitions

We define a relation called elaboration $\gamma \vdash \underline{d} \approx\!> \gamma'$ similar to calculation, but now working on the syntactic category of definitions. The result of elaborating a definition d is a table γ'. The elaboration of a definition d is conducted with respect to a given table γ. Again, we opt to present the elaboration relation in an incremental form — where the elaboration of a definition gives us the change (the “delta”) in the environment, rather than the cumulative new environment, since this not only allows us to build the cumulative environment whenever needed, but also to undo any changes when we wish to revert to an old environment. Again, the implementation can use stacks, pushing and popping answer bindings.

$$\textbf{(ElabADef)}\ \frac{\gamma \vdash \underline{e_1} \Longrightarrow \underline{a}}{\gamma \vdash \underline{\text{def } x = e} \approx\!> \{x \mapsto \underline{a}\}}$$

If under table γ, expression e yields answer a, then elaborating the simple definition $\underline{\text{def } x = e}$ yields the (incremental) table $\{x \mapsto a\}$.

$$\textbf{(ElabSeqDef)}\ \frac{\gamma \vdash \underline{d_1} \approx\!> \gamma_1 \qquad \gamma[\gamma_1] \vdash \underline{d_2} \approx\!> \gamma_2}{\gamma \vdash \underline{d_1; d_2} \approx\!> \gamma_1[\gamma_2]}$$

If under table γ, elaborating the definition d_1 yields the (incremental) table γ_1, and after augmenting the table γ with γ_1, the elaboration of the definition d_2 yields the (incremental) table γ_2, then under table γ, elaborating the sequential definition $d_1; d_2$ yields the (incremental) table $\gamma_1[\gamma_2]$.

$$\textbf{(ElabParDef)}\ \frac{\gamma \vdash \underline{d_1} \approx\!> \gamma_1 \qquad \gamma \vdash \underline{d_2} \approx\!> \gamma_2}{\gamma \vdash \underline{d_1 \parallel d_2} \approx\!> \gamma_1 \cup \gamma_2}$$

If under table γ, elaborating the definition d_1 yields the (incremental) table γ_1, and under the *same* table γ, type elaborating the definition d_2 yields the (incremental) table γ_2, then under table γ, elaborating the parallel definition $d_1 \parallel d_2$ yields the (incremental) table $\gamma_1 \cup \gamma_2 = \gamma_1[\gamma_2] = \gamma_2[\gamma_1]$ (since $dv(d_1) \cap dv(d_2) = \{\}$).

The Principle of Qualification on Definitions.

But Wait! We are not done with the Principle of Qualification. The Principle said that any syntactic category can be qualified with a local definition. We now have a syntactic category of *definitions*. *So definitions too can be subjected to the Principle of Qualification.*

Doing so gives us a further extension to the syntactic category of definitions:

$$d \in Defs \ ::= \ \textbf{def}\ x = e_1 \ \mid\ d_1; d_2 \ \mid\ d_1 \parallel d_2 \ \mid\ \textbf{local}\ d_1\ \textbf{in}\ d_2\ \textbf{ni}$$

Note that we ought to use the same mechanisms for all qualification (namely formalise them using augmentation, and implement them using stacks). Note also that $fv(\textbf{local}\ d_1\ \textbf{in}\ d_2\ \textbf{ni}) = fv(d_1) \cup (fv(d_2) - dv(d_1))$ and

Type-Elaboration

$$\textbf{(LocalDefT)}\ \frac{\Gamma \vdash \underline{d_1} :> \Gamma_1 \qquad \Gamma[\Gamma_1] \vdash \underline{d_2} :> \Gamma_2}{\Gamma \vdash \underline{\textbf{local}\ d_1\ \textbf{in}\ d_2\ \textbf{ni}} :> \Gamma_2}$$

If under type assumption Γ, type-elaborating the definition d_1 yields the (incremental) type assumption Γ_1, and after augmenting the type assumption Γ with Γ_1, the type elaboration of the definition d_2 yields the (incremental) type assumption Γ_2, then under type assumption Γ, type-elaborating the local definition **local** d_1 **in** d_2 **ni** yields the (incremental) type assumption Γ_2. Note that the temporary type assumption Γ_1 created from definition d_1 is omitted, since it was purely for the local usage in the definition d_2.

Denotational Semantics

$elab[\![\underline{\textbf{local } d_1 \textbf{ in } d_2 \textbf{ ni}}]\!]\rho = \rho_2$ where $\rho_1 = elab[\![\underline{d_1}]\!]\rho$ and $\rho_2 = elab[\![\underline{d_2}]\!](\rho[\rho_1])$
Observe that the intermediate valuation ρ_1 is not part of the returned valuation.

$$\textbf{(ElabLocalDef)}\ \frac{\gamma \vdash \underline{d_1} \mathrel{\approx\!>} \gamma_1 \qquad \gamma[\gamma_1] \vdash \underline{d_2} \mathrel{\approx\!>} \gamma_2}{\gamma \vdash \underline{\textbf{local } d_1 \textbf{ in } d_2 \textbf{ ni}} \mathrel{\approx\!>} \gamma_2}$$

If under table γ, elaborating the definition d_1 yields the (incremental) table γ_1, and after augmenting the table γ with γ_1, the elaboration of the definition d_2 yields the (incremental) table γ_2, then under table γ, elaborating the local definition **local** d_1 **in** d_2 **ni** yields the (incremental) table γ_2.

Note how in all respects, type-elaboration, denotational semantics and elaboration, the local definition resembles sequential composition, but then in what is finally yielded, omits what the first definition produces. This may not be very easy to implement (but not too hard either) to implement in a stack machine.

Generalising Let expressions

Note that we now can incorporate — instead of the simple let-expression form with which we started this section — a general let-expression of the form **let** d **in** e_2 **ni**

The rule for typing (static semantics) is the following, which one can check is completely consistent with the earlier rule given:

$$\textbf{(Let'T)}\ \frac{\Gamma \vdash d :> \Gamma_1 \qquad \Gamma[\Gamma_1] \vdash \underline{e} : \tau}{\Gamma \vdash \underline{\textbf{let } d \textbf{ in } e \textbf{ ni}} : \tau}$$

Likewise, the denotational rule is a straightforward generalisation that conserves as a special case the definition given at the start of this section.

$$eval[\![\underline{\textbf{let } d \textbf{ in } e \textbf{ ni}}]\!]\ \rho = eval[\![\ \underline{e}\ ]\!]\ \rho' \text{ where } \rho' = \rho[\ elab[\![\ \underline{d}\ ]\!]\ \rho]$$

And finally, the big-step operational semantics also smoothly generalises:

$$\textbf{(CalcLet')}\ \frac{\gamma \vdash \underline{d} \mathrel{\approx\!>} \gamma_1 \qquad \gamma[\gamma_1] \vdash \underline{e} \Longrightarrow \underline{a}}{\gamma \vdash \underline{\textbf{let } d \textbf{ in } e \textbf{ ni}} \Longrightarrow \underline{a}}$$

Exercise: Define the OCaml definition of the data type `defs` to incorporate definitions of the form defined above.
Exercise: Define the syntactic functions `dht` and `dsize` to return the height and size of a definition in terms of the `ht` and `size` of their constituent expressions.
Exercise: Define a function `elab` (in OCaml) for definitions in terms of `eval` to process definitions.
Exercise: Define the relation `typeelab` using the type checking relation `hastype`.
Exercise: Extend the big-step operational semantics (coded in Prolog) to incorporate definitions of the form shown above

What about Soundness and Completeness of the Calculator with respect to the Definitional Interpreter?   They are preserved. Yet again!

Exercise: Prove the new induction steps in the Soundness Theorem.

Exercise: Prove the new induction steps in the Completeness Theorem.

The Type Preservation theorem also continues to hold! Surprise, surprise!

Exercise: Prove the new induction steps in the Type Preservation Theorem.

.