Definitions vs Functions

Landin in his papers also introduced the concept of *syntactic sugar* — namely, intuitive and readable syntactic notations used for convenience, which can be expressed using the more elementary syntactic forms already in the language. We shall discuss some common instances of syntactic sugar below.

For example, OCaml allows the user to write lists using square brackets and infix separators — e.g., `[ ]` and `[1; 2; 3]` which are more readable than the same lists written in terms of defined constructors, e.g., `Nil` and `Cons(1, Cons(2, Cons(3, Nil) ) ).` Likewise we use an infix operator `@` for the list `append` function. This sweetening is at the level of surface syntax — where the lexical analyser and parse can be employed to make writing programs easier for the programmer.

For functions, we can write shorthand definitions

$$\textbf{def}\ f\ x = e$$

instead of

$$\textbf{def}\ f = \lambda x\,.\,e$$

Or more generally:

$$\textbf{def}\ f\ x_1\ \ldots\ x_n = e$$

instead of the far less readable

$$\textbf{def}\ f = \lambda x_1\,.\,(\ldots(\lambda x_n\,.\,e)\ldots)$$

We also follow conventions (like BODMAS) that allow us to get rid of pesky parenthesis. For instance, we will write

$$e_1\ e_2\ \ldots\ e_n$$

instead of the (left-associative) fully parenthesised version

$$(\ \ldots\ (e_1\ e_2)\ \ldots\ e_n)$$

[Note that the left-associativity of application in expressions implies a right-associative parenthesisation on the types. For example, the type of e_1 above will be of the form $\tau_2 \rightarrow (\ \ldots\ (\tau_n \rightarrow \tau)\ldots)$, where the types of the arguments e_i are (respectively) τ_i and the result type is τ.]

Such sugaring also can be done on abstract syntax. Recall that the *Principle of Qualification* allowed us to introduce the concept of local definitions within expressions, namely "let-expressions" of the form **let def** $x = e_1$ **in** e_2 **ni**, where a local variable x is temporarily and locally defined to take the value of expression e_1 within the expression e_2. The same principle allows us to write increasingly complex expressions such as

$$\begin{array}{l}
\textbf{let def}\ x_1 = e_1 \\
\textbf{in} \\
\quad \textbf{let def}\ x_2 = e_2 \\
\quad \textbf{in} \\
\qquad \vdots\ \vdots\ \vdots \\
\qquad\quad \textbf{in}\ e_n \\
\qquad\quad \textbf{ni} \\
\qquad \vdots\ \vdots\ \vdots \\
\quad \textbf{ni}
\end{array}$$

ni

This sequence of nested let expressions is rather ugly to read. Instead it is simpler to consider the idea of sequential definitions, and instead write a more readable and concise let-expression:

let def $x_1 = e_1$; **def** $x_2 = e_2$; ...
in e_n
ni

with the intention that both expressions mean the same.

Exercise: Verify using both the denotational semantics for let-expressions and definitions that for all variables x_1, x_2 and expressions e_1, e_2, e, the expressions

let def $x_1 = e_1$
in
  let def $x_2 = e_2$
  in e_n
  ni
ni

and

let def $x_1 = e_1$; **def** $x_2 = e_2$
in e_n
ni

have the same meaning.

Program Equivalence

What we see above is that the convenience of writing more readable text should be supported by a notion of programs having the same mathematical denotation and/or the same operational behaviour.

Definition: We say that two expressions (programs) e, e' are *denotationally equivalent* if for all A-valuations $\rho : \mathcal{X} \to A$ (with respect to a signed algebra $\mathcal{A} = \langle A, \ldots \rangle$) $eval[\![e]\!]\rho = eval[\![e']\!]\rho$.

Definition: We say that two expressions (programs) e, e' are *operationally equivalent* , written $e \approx e'$, if for all tables $\gamma : \mathcal{X} \to Ans$, and answers $a \in Ans$, $\gamma \vdash e \Longrightarrow a$ if and only if $\gamma \vdash e' \Longrightarrow a$. We shall soon see that for functions, this is much too restrictive, since there may be many different representations (especially as closures) of the same mathematical function. So we may want a more ramified approach:

- For expressions e, e' of base types (e.g., `unit, bool, int`), $e \approx e'$ if whenever $\gamma \vdash e \Longrightarrow a$ then $\gamma \vdash e' \Longrightarrow a$, and conversely.
- For function expressions, i.e., expressions of type $\tau_1 \to (\ \ldots\ (\tau_n \to \tau)\ldots)$ we can say that $e \approx e'$ if for *all* arguments $e_1, \ldots, e_n$ such that each argument expression e_i has type τ_i, we have for the two base-type expressions $e\ e_1\ \ldots\ e_n \approx e'\ e_1\ \ldots\ e_n$.

Of course, we will want our notions of Soundness of the operational semantics with respect to the denotational meaning, (and conversely, the notion of Completeness) to cohere with

the notion of program equivalence: two operationally equivalent programs should have the same denotation (i.e, the same denotational value). Ideally, we would also like to have the converse result — that if two expressions are denotationally equivalent, they are also operationally equivalent.

It is perhaps better to subscript the equivalence symbol with the (common) type of the expressions. Then we get a family of (type-indexed, i.e., logical) equivalences:

- For closed expressions e, e' of base type τ_0 (e.g., `unit, bool, int`), $e \approx_{\tau_0} e'$ if whenever $\gamma \vdash e \Longrightarrow a$ then $\gamma \vdash e' \Longrightarrow a$, and conversely.
- For closed expressions e, e' of function type $\tau \to \tau'$: $e \approx_{\tau \to \tau'} e'$ if for all closed expressions $e_1 : \tau$, $e\ e_1 \approx_{\tau'} e'\ e_1$

This formulation should preferably be generalised to expressions with free variables, and including agreement conditions on the table with respect to typing assumptions.

Exercise (**): Check if the generalisation "For closed expressions e, e' of function type $\tau \to \tau'$: $e \approx_{\tau \to \tau'} e'$ if for all equivalent closed expressions $e_1, e_1' : \tau$, $e\ e_1 \approx_{\tau'} e'\ e_1'$

Correspondence between definitions and function call

Let us now revisit the let-form with simple definitions

let def $x = e_1$ **in** e_2 **ni**

(introduced via the *Principle of Qualification* and see how it compares with the notion of function invocation (call) which was introduced via the *Principle of Abstraction* and first-class functions — in particular the expression $(\lambda x\,.\,e_2)\ e_1$.

Recall the typing rule for the let-expression

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

Now let us look at how the typing rules play out for the expression $(\lambda x\,.\,e_2)\ e_1$.
First, from the assumptions, we have $\Gamma \vdash \underline{e_1} : \tau_1$. From the other assumption, namely $\Gamma[\underline{x} : \tau_1] \vdash \underline{e_2} : \tau_2$, we can conclude — by rule **(AbsT)** — $\Gamma \vdash \underline{\lambda x\,.\,e_2} : \tau_1 \to \tau_2$.
Combining these two types judgments — via rule **(AppT)** — we get $\Gamma \vdash \underline{(\lambda x\,.\,e_2)\ e_1} : \tau_2$.

Now let us look at the operational semantics for both these expressions. For the let-expression, we have — via **(CalcLet)**:

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

Now looking at the operational rules for the expression $(\lambda x\,.\,e_2)\ e_1$, we have — via **(CalcAbs)** —

$$\gamma \vdash \underline{\lambda x \,.\, e_2} \Longrightarrow \langle\langle \lambda x \,.\, e_2,\, \gamma \rangle\rangle$$

Now from the assumptions we have
$\gamma \vdash \underline{e_1} \Longrightarrow \underline{a_1}$, and
$\gamma[\underline{x} \mapsto \underline{a_1}] \vdash \underline{e_2} \Longrightarrow \underline{a_2}$

Now via (**CalcApp**), we get from these three assumptions,
$\gamma \vdash \underline{(\lambda x \,.\, e_2)\ e_1} \Longrightarrow \underline{a_2}$

Thus we see that both these expressions are operationally equivalent.

Principle of Correspondence
The Principle of Correspondence states that for *every* definition mechanism there is a corresponding parameter-passing mechanism and conversely for *every* parameter-passing mechanism there is a corresponding definition mechanism.

The implications of this principle are many:
- At the typing level, there is a very precise correspondence with the let-form that uses local definitions and the function call — with the locally defined variable corresponding to the formal parameter, with the scope of the let expression corresponding to the body of the function; and the defining expression corresponding to the actual argument.
- At the operational level, the correspondence is even more precise: the calculation of each of the key sub-expressions is in exact correspondence. The let-expression binds the formal parameter to an answer that is exactly the result obtained from calculating the actual argument. And then the calculation of the scope of the let-expression — with respect to the environment where the variable is bound — corresponds exactly to the calculation of the result from the function body.
- The let-expression form is more efficient, in that we do not build a closure only to immediately deconstruct it.

At a practical level, the correspondence is even more important:
- We should be able to treat formal parameters of a function exactly the same way that we treat local variables. Both are bound, and the binding occurrence of the parameter of an abstraction behaves exactly the same way as a variable introduced in a local definition.
- If we know how to implement a local definition mechanism (use a stack!), we know how to implement function call (use a stack!).
- If we decide to use a particular parameter passing mechanism, we should use a similar definition mechanism: it would be unwise to use eager evaluation in let-expressions while trying a lazy approach to function call. And conversely.

Exercise: We have earlier outlined the specification a *lazy* evaluation mechanism for function call. What should the corresponding operational rule for a simple let form be? Write operational and denotational specifications.

.