Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Sub-structural type systems; Algebraic structures
Section and Practice Problems

Apr 6–10, 2020

---

1 Sub-structural type systems

Recall the linear lambda calculus from Lecture 17.

(a) Suppose we extended the language with a negation operation $\mathsf{not}\ e$. Intuitively, e needs to have boolean type, and the expression evaluates e to a value v and then evaluates to the negation of the boolean represented by v. The result of the negation is unrestricted (i.e., it is not linear). Write a typing rule for the negation operator.

Answer:
$$\frac{\Gamma \vdash e : q\ \textbf{\textit{bool}}}{\Gamma \vdash \textit{not}\, e : \textit{un}\ \textbf{\textit{bool}}}$$

Note that the expression e can be either linear or unrestricted.

(b) Suppose we extended the language with a conjunction operator, $e_1\ \mathsf{and}\ e_2$. Intuitively, e_1 and e_2 both need to have boolean type, and the expression evaluates to the conjunction of the booleans. The boolean value that is the result of the conjunction is unrestricted (i.e., it is not linear). Write a typing rule for the conjunction operator.

Answer:
$$\frac{\Gamma_1 \vdash e_1 : q\ \textbf{\textit{bool}} \qquad \Gamma_2 \vdash e_2 : q'\ \textbf{\textit{bool}}}{\Gamma \vdash e_1\ \textit{and}\ e_2 : \textit{un}\ \textbf{\textit{bool}}}\ \Gamma = \Gamma_1 \circ \Gamma_2$$

Note that the sub expressions can be either linear or unrestricted, and they do not need to be the same. Note that we need to split the context Γ so that linear variables in Γ are appropriately split between the computation of the subexpressions.

(c) Let context $\Gamma = x : \mathsf{lin}\ (\mathsf{lin}\ \textbf{bool} \times \mathsf{un}\ \textbf{bool})$ and expression $e \equiv \mathsf{split}\ x\ \mathsf{as}\ y, z\ \mathsf{in}\ \mathsf{if}\ y\ \mathsf{then}\ z\ \mathsf{else}\ \mathsf{not}\ z$.

Is e well-typed for context Γ? That is, does $\Gamma \vdash e : \mathsf{un}\ \textbf{bool}$ hold? If so, give the derivation.

Answer: *Yes, it is well typed. The linear data structure (i.e., the pair) contains an unrestricted value, but that is fine. It is the reverse that is a problem, i.e., when an unrestricted data structure contains a linear value. (Apologies for the tiny font. Zoom in on a PDF viewer to see the details of the derivation.)*

$$\textsc{T-Split}\ \frac{\textsc{T-Var}\ \dfrac{}{\Gamma \vdash x : \textit{lin}\ (\textit{lin}\ \textbf{\textit{bool}} \times \textit{un}\ \textbf{\textit{bool}})} \qquad \textsc{T-If}\ \dfrac{\textsc{T-Var}\ \dfrac{}{y : \textit{lin}\ \textbf{\textit{bool}}, z : \textit{un}\ \textbf{\textit{bool}} \vdash y : \textit{lin}\ \textbf{\textit{bool}}} \quad \textsc{T-Var}\ \dfrac{}{z : \textit{un}\ \textbf{\textit{bool}} \vdash z : \textit{un}\ \textbf{\textit{bool}}} \quad \textsc{T-Not}\ \dfrac{\textsc{T-Var}\ \dfrac{}{z : \textit{un}\ \textbf{\textit{bool}} \vdash z : \textit{un}\ \textbf{\textit{bool}}}}{z : \textit{un}\ \textbf{\textit{bool}} \vdash \textit{not}\ z : \textit{un}\ \textbf{\textit{bool}}}}{y : \textit{lin}\ \textbf{\textit{bool}}, z : \textit{un}\ \textbf{\textit{bool}} \vdash \textit{if}\ y\ \textit{then}\ z\ \textit{else not}\ z : \textit{un}\ \textbf{\textit{bool}}}}{\Gamma \vdash e : \textit{un}\ \textbf{\textit{bool}}}$$

Note how the context $y : \textit{lin}\ \textbf{\textit{bool}}, z : \textit{un}\ \textbf{\textit{bool}}$ is split into $y : \textit{lin}\ \textbf{\textit{bool}}, z : \textit{un}\ \textbf{\textit{bool}}$ (which is used to type check the the conditional expression y) and the context $z : \textit{un}\ \textbf{\textit{bool}}$ (which is used to check the two branches). The linear resource $y : \textit{lin}\ \textbf{\textit{bool}}$ goes into only one of the contexts, whereas the unrestricted resource $z : \textit{un}\ \textbf{\textit{bool}}$ goes into both.

(d) Consider the execution of the following (well-typed) expression, starting from the empty store. What does the final store look like? (In particular, recall that locations containing linear values are removed once the linear value is used.)

$$(\mathsf{lin}\ \lambda x\!:\!\mathsf{lin}\ (\mathsf{un}\ \mathbf{bool} \times \mathsf{un}\ \mathbf{bool}).\ \mathsf{split}\ x\ \mathsf{as}\ y, z\ \mathsf{in}\ y)\ (\mathsf{lin}\ (\mathsf{un}\ \mathsf{false}, \mathsf{un}\ \mathsf{true}))$$

Answer: *As the program executes, it creates locations for each value. Let's suppose that these are the locations created for each value:*

$$\begin{aligned}
&\ell_1 \mapsto \mathit{lin}\ \lambda x\!:\!\mathit{lin}\ (\mathit{un}\ \mathbf{bool} \times \mathit{un}\ \mathbf{bool}).\ \mathit{split}\ x\ \mathit{as}\ y, z\ \mathit{in}\ y\\
&\ell_2 \mapsto \mathit{un\ false}\\
&\ell_3 \mapsto \mathit{un\ true}\\
&\ell_4 \mapsto \mathit{lin}\ (\ell_2, \ell_3)
\end{aligned}$$

As each linear value is used, it is removed from the store. The type system ensures that every linear value is used exactly once, and thus, every linear value is removed from the store by the end of the execution. Thus, the store at the end of execution contains only locations ℓ_2 and ℓ_3, the locations that contain unrestricted values.

2 Algebraic structures

(a) Show that the option type, with *map* defined as in the lecture notes (Lecture 18, Section 2.2) satisfy the functor laws.

Answer: *The functor laws are:*

$$\begin{aligned}
\forall f \in A \to B, g \in B \to C.\ (\mathit{map}\ f) \fatsemi (\mathit{map}\ g) &= \mathit{map}\ (f \fatsemi g) && \textit{Distributivity}\\
\mathit{map}\ (\lambda a\!:\!A.\ a) &= (\lambda a\!:\!T_A.\ a) && \textit{Identity}
\end{aligned}$$

The definition of map for the option type is

$$\mathit{map} \equiv \lambda f\!:\!\tau_1 \to \tau_2.\ \lambda a\!:\!\tau_1\ \mathbf{option}.\ \mathit{case}\ a\ \mathit{of}\ \lambda x\!:\!\mathbf{unit}.\ \mathit{none} | \lambda v\!:\!\tau_1.\ \mathit{some}\ (f\ v)$$

To show distributivity, we need to show that for all functions $f : \tau_1 \to \tau_2$ and $g : \tau_2 \to \tau_3$ we have that $\lambda x\!:\!\tau_1\ \mathbf{option}.\ (\mathit{map}\ g)\ ((\mathit{map}\ f)\ x)$ is equivalent to map $(\lambda x\!:\!\tau_1.\ g\ (f\ x))$

Recall that $\fatsemi$ indicates function composition. So the function $f \fatsemi g$ can can be expressed as $\lambda x\!:\!\tau_1.\ g\ (f\ x)$, and the function $(\mathit{map}\ f) \fatsemi (\mathit{map}\ g)$ can be expressed as $\lambda x\!:\!\tau_1\ \mathbf{option}.\ (\mathit{map}\ g)\ ((\mathit{map}\ f)\ x)$.

$$\begin{aligned}
&\lambda x\!:\!\tau_1\ \mathbf{option}.\ (\mathit{map}\ g)\ ((\mathit{map}\ f)\ x)\\
&\quad = \lambda x\!:\!\tau_1\ \mathbf{option}.\ (\mathit{map}\ g)\ ((\lambda a\!:\!\tau_1\ \mathbf{option}.\ \mathit{case}\ a\ \mathit{of}\ \lambda y\!:\!\mathbf{unit}.\ \mathit{none} | \lambda v\!:\!\tau_1.\ \mathit{some}\ (f\ v))\ x) && \textit{expand map } f\\
&\quad = \lambda x\!:\!\tau_1\ \mathbf{option}.\ (\mathit{map}\ g)\ (\mathit{case}\ x\ \mathit{of}\ \lambda y\!:\!\mathbf{unit}.\ \mathit{none} | \lambda v\!:\!\tau_1.\ \mathit{some}\ (f\ v)) && \textit{by } \beta\textit{-equivalence}\\
&\quad = \lambda x\!:\!\tau_1\ \mathbf{option}.\ (\lambda b\!:\!\tau_2\ \mathbf{option}.\ \mathit{case}\ b\ \mathit{of}\ \lambda z\!:\!\mathbf{unit}.\ \mathit{none} | \lambda w\!:\!\tau_2.\ \mathit{some}\ (g\ w))\\
&\qquad\qquad (\mathit{case}\ x\ \mathit{of}\ \lambda y\!:\!\mathbf{unit}.\ \mathit{none} | \lambda v\!:\!\tau_1.\ \mathit{some}\ (f\ v)) && \textit{expand map } g\\
&\quad = \lambda x\!:\!\tau_1\ \mathbf{option}.\ \mathit{let}\ b = \mathit{case}\ x\ \mathit{of}\ \lambda y\!:\!\mathbf{unit}.\ \mathit{none} | \lambda v\!:\!\tau_1.\ \mathit{some}\ (f\ v)\ \mathit{in}\\
&\qquad\qquad \mathit{case}\ b\ \mathit{of}\ \lambda z\!:\!\mathbf{unit}.\ \mathit{none} | \lambda w\!:\!\tau_2.\ \mathit{some}\ (g\ w) && \textit{rewrite as let expression}\\
&\quad = \lambda x\!:\!\tau_1\ \mathbf{option}.\ \mathit{case}\ x\ \mathit{of}\ \lambda y\!:\!\mathbf{unit}.\ \mathit{none} | \lambda v\!:\!\tau_1.\ \mathit{some}\ (g\ (f\ v)) && \textit{simplifying nested cases}\\
&\quad = \lambda x\!:\!\tau_1\ \mathbf{option}.\ \mathit{case}\ x\ \mathit{of}\ \lambda t\!:\!\mathbf{unit}.\ \mathit{none} | \lambda v\!:\!\tau_1.\ \mathit{some}\ ((\lambda y\!:\!\tau_1.\ g\ (f\ y))\ v)\\
&\quad = \mathit{map}\ (\lambda y\!:\!\tau_1.\ g\ (f\ y)) && \textit{un-expanding map } (\lambda y\!:\!\tau_1.\ g\ (f\ y))
\end{aligned}$$

*To show identity, we need to show that map $\lambda a:\tau.\, a$ is equivalent to $\lambda a:\tau$ **option**. a.*

map $\lambda a:\tau.\, a$

$= \lambda b:\tau$ **option**. *case* b *of* $\lambda x:$ **unit**. *none*$|\lambda v:\tau.$ *some* $((\lambda a:\tau.\, a)\ v)$ — *expand map* $(\lambda a:\tau.\, a)$

$= \lambda b:\tau$ **option**. *case* b *of* $\lambda x:$ **unit**. *none*$|\lambda v:\tau.$ *some* $\ v$ — *by* β*-equivalence*

$= \lambda b:\tau$ **option**. b

(b) Consider the list type, τ **list**. Define functions *return* and *bind* for the list monad that satisfy the monad laws. Check that they satisfy the monad laws.

Answer: *We define return and bind so that they represent a set of possible values that could be produced by a computation. That is, we use lists to represent the possible values of a nondeterministic computation. There are other ways to define return and bind on lists, for example, as a stream of results produced by a computation.*

Here return will take a value of type τ and return a list that contains the value as its only element. bind will take a list of τ, take a function f from τ to lists of τ', and apply f to every element of the list (using the function map, defined in class), to get a list of lists of τ'. We then use a utility function flatten to flatten the list of lists of τ' to a list of τ'. (In the definition of flatten, a acts as an accumulator.)

return $\triangleq \lambda x:\tau.\, x :: [\,]$

bind $\triangleq \lambda xs:\tau$ **list**. $\lambda f:\tau \to \tau'$ **list**. *flatten* $(map\ f\ xs)$

flatten $\triangleq$ *let* $fl = \mu f:\tau'$ **list** $\to (\tau'$ **list**$)$ **list** $\to \tau'$ **list**.

$\lambda a:\tau'$ **list**. $\lambda x:(\tau'$ **list**$)$ **list**. *if isempty?* x *then* a *else* $f\ (append\ a\ (\mathsf{head}\ x))\ (\mathsf{tail}\ x)$ *in*

$fl\ [\,]$

We now show that the definitions above satisfy the monad laws. We must show that Left and Right Identity and Associativity hold.

bind (return x) f

$= (\lambda xs:\tau$ **list**. $\lambda f:\tau \to \tau'$ **list**. *flatten* $(map\ f\ xs))((\lambda x:\tau.\, x :: [\,])\ x)\ f$ — *expand return and bind*

$= (\lambda xs:\tau$ **list**. $\lambda f:\tau \to \tau'$ **list**. *flatten* $(map\ f\ xs))(x :: [\,])\ f$ — *application of return*

$= (\lambda f:\tau \to \tau'$ **list**. *flatten* $(map\ f\ (x :: [\,])))\ f$ — *application of bind*

$=$ *flatten* $(map\ f\ (x :: [\,]))$

$=$ *flatten* $(g :: [\,])$ — *where* $f\ x = g$

$= g$ — *as forall lists l, flatten (l :: []) = l*

Case $am = [\,]$*:*

bind [ ] return

$= [\,] = am$

Case $am = x_0 :: x_1 :: \ldots :: x_n :: []$.

bind am return

$= (\lambda xs : \tau$ ***list***$.\ \lambda f : \tau \to \tau'$ ***list***$.$ *flatten* $(map\ f\ xs))\ am\ (\lambda x.\ x :: [])$ *expand return and bind*

$=$ *flatten* $(map\ (\lambda x.\ x :: [])\ am)$

$=$ *flatten* $((x_0 :: []) :: (x_1 :: []) :: \ldots :: (x_n :: []) :: [])$

$= x_0 :: x_1 :: \ldots :: x_n :: []$

The proof of Associativity is left as an exercise for the reader.

3 Haskell

(a) Install the Haskell Platform, via `https://www.haskell.org/platform/`.

(b) Get familiar with Haskell. Take a look at `http://www.seas.harvard.edu/courses/cs152/2020sp/resources.html` for some links to tutorials.

In particular, get comfortable doing functional programming in Haskell. Write the factorial function. Write the append function for lists.

(c) Get comfortable using monads, and the bind syntax. Try doing the exercises at `https://wiki.haskell.org/All_About_Monads#Exercises` (which will require you to read the previous sections to understand `do` notation, and their previous examples).

(d) Also, look at the Haskell code provided on the section page, which includes some example Haskell code (that will likely be covered in Section).