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

Curry-Howard isomorphism; Existential types; Sub-structural type systems Section and Practice Problems

Mar 31-Apr 1, 2016

---

1 Curry-Howard isomorphism

The following logical formulas are tautologies, i.e., they are true. For each tautology, state the corresponding type, and come up with a term that has the corresponding type.

For example, for the logical formula $\forall\phi.\phi \implies \phi$, the corresponding type is $\forall X.\ X \to X$, and a term with that type is $\Lambda X.\ \lambda x\!:\!X.\ x$. Another example: for the logical formula $\tau_1 \wedge \tau_2 \implies \tau_1$, the corresponding type is $\tau_1 \times \tau_2 \to \tau_1$, and a term with that type is $\lambda x\!:\!\tau_1 \times \tau_2.\ \#1\ x$.

You may assume that the lambda calculus you are using for terms includes integers, functions, products, sums, universal types and existential types.

(a) $\forall\phi.\ \forall\psi.\ \phi \wedge \psi \implies \psi \vee \phi$

Answer: *The corresponding type is*

$$\forall X.\ \forall Y.\ X \times Y \to Y + X$$

A term with this type is

$$\Lambda X.\ \Lambda Y.\ \lambda x\!:\!X \times Y.\ \mathsf{inl}_{Y+X}\ \#2\ x$$

(b) $\forall\phi.\ \forall\psi.\ \forall\chi.\ (\phi \wedge \psi \implies \chi) \implies (\phi \implies (\psi \implies \chi))$

Answer: *The corresponding type is*

$$\forall X.\ \forall Y.\ \forall Z.\ (X \times Y \to Z) \to (X \to (Y \to Z))$$

A term with this type is

$$\Lambda X.\ \Lambda Y.\ \Lambda Z.\lambda f\!:\!X \times Y \to Z.\ \lambda x\!:\!X.\ \lambda y\!:\!Y.\ f\ (x, y)$$

Note that this term uncurries the function. It is the opposite of the currying we saw in class.

(c) $\exists\phi.\ \forall\psi.\ \psi \implies \phi$

Answer: *The corresponding type is*

$$\exists X.\ \forall Y.\ Y \to X$$

A term with this type is

$$\mathsf{pack}\ \{\mathbf{int}, \Lambda Y.\ \lambda y\!:\!Y.\ 42\}\ \mathsf{as}\ \exists X.\ \forall Y.\ Y \to X$$

(d) $\forall\psi.\ \psi \implies (\forall\phi.\ \phi \implies \psi)$

Answer: *The corresponding type is*

$$\forall Y.\ Y \to (\forall X.\ X \to Y)$$

A term with this type is

$$\Lambda Y.\ \lambda a\!:\!Y.\ \Lambda X.\ \lambda x\!:\!X.\ a$$

Primitive propositions in logic correspond

(e) $\forall \psi.\ (\forall \phi.\ \phi \implies \psi) \implies \psi$

Answer: *A corresponding type is*

$$\forall Y.\ (\forall X.\ X \to Y) \to Y$$

A term with this type is

$$\Lambda Y.\ \lambda f\!:\!\forall X.\ X \to Y.\ f\ [\textbf{\textit{int}}]\ 42$$

2 Existential types

(a) Write a term with type $\exists C.\ \{\ \mathit{produce} : \textbf{int} \to C, \mathit{consume} : C \to \textbf{bool}\ \}$. Moreover, ensure that calling the function *produce* will produce a value of type C such that passing the value as an argument to *consume* will return true if and only if the argument to *produce* was 42. (Assume that you have an integer comparison operator in the language.)

Answer:

*In the following solution, we use **int** as the witness type, and implement produce using the identity function, and implement consume by testing whether the value of type C (i.e., of witness type **int**) is equal to 42.*

$\textit{pack}\ \{\textbf{\textit{int}}, \{\ \mathit{produce} = \lambda a\!:\!\textbf{\textit{int}}.\ a,\ \mathit{consume} = \lambda a\!:\!\textbf{\textit{int}}.\ a = 42\ \}\}$
$\textit{as}\ \exists C.\ \{\ \mathit{produce} : \textbf{\textit{int}} \to C, \mathit{consume} : C \to \textbf{\textit{bool}}\ \}$

(b) Do the same as in part (a) above, but now use a different witness type.

Answer: *Here's another solution where instead we use **bool** as the witness type, and implement produce by comparing the integer argument to 42, and implement consume as the identity function.*

$\textit{pack}\ \{\textbf{\textit{bool}}, \{\ \mathit{produce} = \lambda a\!:\!\textbf{\textit{int}}.\ a = 42,\ \mathit{consume} = \lambda a\!:\!\textbf{\textit{bool}}.\ a\ \}\}$
$\textit{as}\ \exists C.\ \{\ \mathit{produce} : \textbf{\textit{int}} \to C, \mathit{consume} : C \to \textbf{\textit{bool}}\ \}$

(c) Assuming you have a value v of type $\exists C.\ \{\ \mathit{produce} : \textbf{int} \to C, \mathit{consume} : C \to \textbf{bool}\ \}$, use v to "produce" and "consume" a value (i.e., make sure you know how to use the unpack $\{X, x\} = e_1$ in e_2 expression.

Answer:
$\textit{unpack}\ \{D, r\} = v\ \textit{in}$
$\textit{let}\ d = r.\mathit{produce}\ 19\ \textit{in}$
$r.\mathit{consume}\ d$

3 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, need 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: $$\frac{\Gamma \vdash e : q\ \textbf{\textit{bool}}}{\Gamma \vdash \mathit{not}\, e : \mathit{un}\ \textbf{\textit{bool}}}\ \Gamma = \Gamma_1 \circ \Gamma_2$$

Note that the expression 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: $$\frac{\Gamma_1 \vdash e_1 : q\ \textbf{\textit{bool}} \qquad \Gamma_2 \vdash e_2 : q'\ \textbf{\textit{bool}}}{\Gamma \vdash e_1\ \mathit{and}\ e_2 : \mathit{un}\ \textbf{\textit{bool}}}\ \Gamma = \Gamma_1 \circ \Gamma_2$$

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.)*

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

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

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

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

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

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.