Lecture #17: Curry-Howard Isomorphism

CS 6371: Advanced Programming Languages

March 12, 2020

There is a deep relationship between computer programs and mathematical proofs often dubbed the *Curry-Howard isomorphism*. In the 1950s and 60s, as computer scientists wrestled with the question of how one might develop provably correct software, mathematicians H. Curry and W.A. Howard wrote a series of papers observing that type signatures in software are actually theorems, and computer programs are actually proofs of those theorems. This is has become a very important observation for modern high assurance software creation because it allows automated theorem proving systems to specify program correctness properties as types, and check program correctness by interpreting programs as proofs of those theorems.

The Curry-Howard Isomorphism can be seen in a simple form by studying the *type inhabitation problem* for System F, which we define as follows:

Definition (Inhabitation)**.** A System F type τ is said to be *inhabited* if there exists a System F expression e such that $\perp \vdash e : \tau$ is derivable.

Problem (Type Inhabitation)**.** *Given a System F type τ, decide whether τ is inhabited. If it is, give an example of a System F expression e having type τ.*

We will solve this problem for the System F type system presented in class:

$$\tau ::= \mathit{int} \mid \mathit{bool} \mid \mathit{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \forall\alpha.\tau \mid \alpha$$

We will additionally write type *void* as an alias for $\forall\alpha.\alpha$.

To decide inhabitation of τ, it suffices to transform τ into a first-order propositional logic sentence $I(\tau)$ using the following algorithm:

$$I(\mathit{int}) = I(\mathit{bool}) = I(\mathit{unit}) = T \tag{1}$$
$$I(\tau_1 \to \tau_2) = I(\tau_1) \Rightarrow I(\tau_2) \tag{2}$$
$$I(\tau_1 \times \tau_2) = I(\tau_1) \wedge I(\tau_2) \tag{3}$$
$$I(\tau_1 + \tau_2) = I(\tau_1) \vee I(\tau_2) \tag{4}$$
$$I(\forall\alpha.\tau) = \boldsymbol{\forall}\boldsymbol{\alpha}\,.\,I(\tau) \tag{5}$$
$$I(\alpha) = \boldsymbol{\alpha} \tag{6}$$

Equations 5 and 6 change type quantifiers $\forall$ and type variables α into propositional quantifiers $\boldsymbol{\forall}$ and propositional (boolean-valued) variables $\boldsymbol{\alpha}$. Type τ is inhabited if and only if $I(\tau)$ is a provably true statement of intuitionistic[1] propositional logic.

[1]The form of logic taught in most discrete math courses is classical propositional logic, which differs from intuitionistic propositional logic in how implication $\Rightarrow$ is defined. However, I will not give you any problems in this course that require you to know the difference between intuitionistic and classical logic, so you can treat these as classical logic formulae for the purposes of this course.

Exercise 1. *Solve the type inhabitation problem for the following type:*

$$\tau = \big(\forall\alpha.(\mathit{unit} \to ((\alpha \to \mathit{int}) \times (\mathit{int} \to \alpha)))\big) \to \big(\forall\beta.\forall\eta.(\beta + \eta)\big)$$

Solution. Applying algorithm I to τ yields the following propositional sentence:

$$\big(\boldsymbol{\forall\alpha}.(T \Rightarrow ((\boldsymbol{\alpha} \Rightarrow T) \wedge (T \Rightarrow \boldsymbol{\alpha})))\big) \Rightarrow \big(\boldsymbol{\forall\beta}.\boldsymbol{\forall\eta}.(\boldsymbol{\beta} \vee \boldsymbol{\eta})\big) \tag{7}$$

At a high level, the sentence is an implication, which is true in classical logic if the antecedent (left-hand side) is false or the consequent (right-hand side) is true. Antecedent $\boldsymbol{\forall\alpha}.(T \Rightarrow ((\boldsymbol{\alpha} \Rightarrow T) \wedge (T \Rightarrow \boldsymbol{\alpha})))$ is false because there exists a boolean value for $\boldsymbol{\alpha}$ that falsifies it. In particular, when $\boldsymbol{\alpha} = F$ we have

$$\begin{gathered} T \Rightarrow ((F \Rightarrow T) \wedge (T \Rightarrow F)) \\ T \Rightarrow (T \wedge F) \\ T \Rightarrow F \\ F \end{gathered}$$

The consequent is false too; for example, when $\boldsymbol{\beta} = \boldsymbol{\eta} = F$ we have $F \vee F = F$. Therefore proposition 7 has the form $F \Rightarrow F$, which is true, so it is inhabited. In general you can decide the truth of any first-order propositional sentence by carrying out the algorithm you developed for Assignment 1.

Since τ is inhabited, we must now search for an expression that has type τ. The easiest way to proceed is to imagine the type as an abstract syntax tree and work from the top down. Type τ has the form $\tau_1 \to \tau_2$ where $\tau_1 = \forall\alpha.(\mathit{unit} \to ((\alpha \to \mathit{int}) \times (\mathit{int} \to \alpha)))$ and $\tau_2 = \forall\beta.\forall\eta.(\beta + \eta)$. To construct an expression having type $\tau_1 \to \tau_2$, it suffices to write a function of the form

$$\lambda x{:}\tau_1 \,.\, e_2$$

where e_2 has type τ_2. Note that we **never inhabit the type on the left of an arrow**; that's the function's parameter type, and it's given to you for free. Inhabiting a function (arrow) type requires constructing a function body that inhabits the type τ_2 on the **right** of the arrow. Type τ_2 has the form $\forall\beta.\tau_3$ where $\tau_3 = \forall\eta.(\beta + \eta)$. To construct an expression having type $\forall\beta.\tau_3$, it suffices to write a polymorphic function of the form $\Lambda\beta.e_3$ where e_3 has type τ_3:

$$\lambda x : \big(\forall\alpha.(\mathit{unit} \to ((\alpha \to \mathit{int}) \times (\mathit{int} \to \alpha)))\big) \,.\, \Lambda\beta \,.\, e_3$$

Repeating this one more time, we have an expression of the form

$$\lambda x : \big(\forall\alpha.(\mathit{unit} \to ((\alpha \to \mathit{int}) \times (\mathit{int} \to \alpha)))\big) \,.\, \Lambda\beta \,.\, \Lambda\eta \,.\, e_4 \tag{8}$$

where e_4 needs to have type $\beta + \eta$.

At this point we have two choices: A type of the form $\tau_5 + \tau_6$ can be inhabited by either a first-injection $\texttt{in}_1^{\tau_5+\tau_6} e_5$ (where e_5 has type τ_5) or second-injection $\texttt{in}_2^{\tau_5+\tau_6} e_6$ (where e_6 has type τ_6). If there is an obvious way to inhabit τ_5 or τ_6, then use $\texttt{in}_1$ or $\texttt{in}_2$, respectively. But in this case there's no obvious way to inhabit either $\tau_5 = \beta$ or $\tau_6 = \eta$, which are both type variables that denote aribtrary (unknown) types. The solution is to leverage argument x. We know from our analysis of proposition $I(\tau)$ that x's type is uninhabited. This means that x is extremely powerful. It is an argument type that cannot ever really exist, making the function we are constructing uncallable. If it ever did exist, one could do impossible things with it, such as inhabiting types that we know are actually uninhabited.

To use x, we must *destruct* (i.e., "use") an expression with type of the form $\forall\alpha.\tau_7$, where $\tau_7 = \mathit{unit} \to ((\alpha \to \mathit{int}) \times (\mathit{int} \to \alpha))$. Destructing a $\forall$-typed expression entails polymorphic instantiation: $x[\tau']$, where τ' can be any type we wish. This yields an expression of the form of τ_7 but with all α's replaced with τ'. Since we're seeking an expression of the form $\beta + \eta$, let's choose $\tau' = \beta + \eta$:

$$x[\beta + \eta]$$

This expression has type $\mathit{unit} \to (((\beta + \eta) \to \mathit{int}) \times (\mathit{int} \to (\beta + \eta)))$. An expression with a type of the form $\mathit{unit} \to \tau_8$ is a function that expects a unit-typed argument, so to destruct it we must apply it to the unit value ():

$$x[\beta + \eta]()$$

This new expression has type $((\beta + \eta) \to \mathit{int}) \times (\mathit{int} \to (\beta + \eta))$, which is a pair-type. If we apply π_1 to it, we get an expression with the type on the left of the $\times$; and if we apply π_2 to it, we get an expression with the type on the right of the $\times$. The type on the left of the $\times$ is useless to us, but the one on the right is a function that returns something of the type we want; so let's use π_2:

$$\pi_2(x[\beta + \eta]())$$

This has type $\mathit{int} \to (\beta + \eta)$, which is a function expecting an integer argument. To destruct it, apply it to any integer argument:

$$\pi_2(x[\beta + \eta]())3$$

This has type $\beta + \eta$, which is the type we've been seeking. Plugging it into expression 8, we arrive at our final answer:

$$\lambda x : \big(\forall\alpha.(\mathit{unit} \to ((\alpha \to \mathit{int}) \times (\mathit{int} \to \alpha)))\big) \;.\; \Lambda\beta \;.\; \Lambda\eta \;.\; (\pi_2(x[\beta + \eta]())3)$$

This expression has type τ, proving that τ is inhabited. □

It is helpful to generalize the lessons learned from this exercise to build a table of how to construct and destruct expressions of various types:

Type operator	**Construct using...**	**Destruct using...**
int	3	–
bool	`true`	–
unit	()	–
$\to$	λ	$e_1 e_2$ (application)
$\times$	(e_1, e_2)	π_1 or π_2
$+$	$\mathtt{in}_1$ or $\mathtt{in}_2$	`case..of`
$\forall$	Λ	$e[\tau]$

To inhabit a type, start by using the construct column recursively until you start to get stuck. At any stuck points, look at the set of in-scope variables and their types to see whether you can destruct any of them to make progress.

To help you practice, here are two more sample exercises and their solutions:

Exercise 2. *Inhabit the following type if possible:* $\forall\alpha.\forall\beta.\forall\eta.((\alpha \times \beta) \to \eta) \to (\alpha \to \beta \to \eta)$

Solution. Applying algorithm I yields the following propositional sentence:

$$\boldsymbol{\forall\alpha}.\boldsymbol{\forall\beta}.\boldsymbol{\forall\eta}.((\boldsymbol{\alpha} \wedge \boldsymbol{\beta}) \Rightarrow \boldsymbol{\eta}) \Rightarrow (\boldsymbol{\alpha} \Rightarrow \boldsymbol{\beta} \Rightarrow \boldsymbol{\eta})$$

Remember that $\to$ and $\Rightarrow$ are always right-associative; so $\alpha \to \beta \to \eta$ is $\alpha \to (\beta \to \eta)$ and $\boldsymbol{\alpha} \Rightarrow \boldsymbol{\beta} \Rightarrow \boldsymbol{\eta}$ is $\boldsymbol{\alpha} \Rightarrow (\boldsymbol{\beta} \Rightarrow \boldsymbol{\eta})$. With that in mind, we see that the sentence is true for all $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\eta}$. You can either establish this using a truth table, or you can interpret it as a theorem: It says that if $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ together imply $\boldsymbol{\eta}$, then $\boldsymbol{\alpha}$ implies that $\boldsymbol{\beta}$ implies $\boldsymbol{\eta}$, which is true. (It's a statement of currying from functional programming!) Here is an inhabitant of the type:

$$\Lambda\alpha\ .\ \Lambda\beta\ .\ \Lambda\eta\ .\ \lambda x{:}(\alpha \times \beta) \to \eta\ .\ \lambda y{:}\alpha\ .\ \lambda z{:}\beta\ .\ x(y, z)$$

$\square$

Exercise 3. *Inhabit the following type if possible:* $\forall\alpha.((\forall\beta.\beta \to (\alpha + \beta)) \to \mathit{void})$.

Solution. Applying algorithm I yields the following propositional sentence:

$$\boldsymbol{\forall\alpha}.((\boldsymbol{\forall\beta}.\boldsymbol{\beta} \Rightarrow (\boldsymbol{\alpha} \vee \boldsymbol{\beta})) \Rightarrow (\boldsymbol{\forall\alpha}.\boldsymbol{\alpha}))$$

This sentence contains two different propositional variables both named $\boldsymbol{\alpha}$, which I'll call $\boldsymbol{\alpha}_1$ and $\boldsymbol{\alpha}_2$ respectively:

$$\boldsymbol{\forall\alpha}_1.((\boldsymbol{\forall\beta}.\boldsymbol{\beta} \Rightarrow (\boldsymbol{\alpha}_1 \vee \boldsymbol{\beta})) \Rightarrow (\boldsymbol{\forall\alpha}_2.\boldsymbol{\alpha}_2))$$

This sentence is false because $\boldsymbol{\alpha}_1 = T$ falsifies it. In particular, when $\boldsymbol{\alpha}_1 = T$, antecedent $\boldsymbol{\forall\beta}.\boldsymbol{\beta} \Rightarrow (T \vee \boldsymbol{\beta})$ is true yet consequent $\boldsymbol{\forall\alpha}_2.\boldsymbol{\alpha}_2$ is false. (It is falsified by $\boldsymbol{\alpha}_2 = F$.) Since the sentence is false, the type is uninhabited. $\square$

Remember, the point to solving these type inhabitation problems is to understand why converting them to propositional sentences actually works, not to memorize a list of tricks for solving them (which probably won't generalize to new problems and won't help you understand the Curry-Howard isomorphism). When practicing, you should therefore try to understand why propositional sentence $I(\tau)$ is true, and then leverage that understanding to guide your search for an inhabitant of τ. Such an approach will make future problems quite easy to solve and gain you an appreciation for the connection between computer science and abstract logic. Solving type inhabitation exercises in this way shows your mastery of several skills:

- comprehension of the typing rules for System F,
- ability to write syntactically correct System F expressions,
- understanding of how type operators relate to logical operators, and
- ability to relate propositional logical reasoning to program correctness.