Fun with Curry Howard

Tikhon Jelvis (tikhon@jelv.is)

December 17, 2013

Curry-Howard

- correspondence between **programming languages** and **formal logic systems**
  - programming language $\equiv$ logic
  - program $\equiv$ proof
- shows deep relationship between mathematics and programming

Why

- useful for thinking by analogy—a new perspective on programming
- underlies proof assistants like Coq and Agda
- useful for *practical* programming in Haskell and OCaml
  - GADTs, DataKinds, Type Families...
  - Putting Curry-Howard to Work
    - `http://web.cecs.pdx.edu/~sheard/papers/PutCurryHoward2WorkFinalVersion.ps`

Basic Idea

- type $\equiv$ proposition
- program $\equiv$ proof
- a type is **inhabited** if it has at least one element $\equiv$ proposition with proof
- **unit** is trivially inhabited: () —like $\top$
- **void** is uninhabited: like $\perp$
  - Haskell: `data Void`

Comparing Inference Rules: True

- STLC vs **intuitionistic propositional logic**
- **true** introduction:

$$\frac{}{\top}$$

- **unit** type:

$$\frac{}{() : \textbf{unit}}$$

False

- no way to introduce **false** ($\perp$)
- similarly, no rule for **void** !
- we can "eliminate" **false**:

$$\frac{\perp}{C}$$

- this cannot actually happen!

Implication Introduction

- if we can prove B given A:

$$\frac{A \vdash B}{A \Rightarrow B}$$

- just like rule for abstractions:

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau.e) : \tau \to \tau'}$$

Implication Elimination

- $$\frac{A \Rightarrow B \quad A}{B}$$
- Just like function application:

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

And Introduction

- $$\frac{A \quad B}{A \wedge B}$$
- just like product type:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

And Elimination

- $$\frac{A \wedge B}{A} \quad \frac{A \wedge B}{B}$$
- just like first and second:

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{first } e : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{second } e : \tau_2}$$

Or Introduction

- $$\frac{A}{A \vee B} \quad \frac{B}{A \vee B}$$
- just like sum type:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{left } e : \tau_1 + \tau_2} \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{right } e : \tau_1 + \tau_2}$$

Or Elimination

- $$\frac{A \vdash C \quad B \vdash C \quad A \vee B}{C}$$
- just like pattern matching (case):

$$\frac{\Gamma \ \vdash \ e : \tau_1 + \tau_2 \quad \Gamma,\ x : \tau_1 \ \vdash \ e_1 : \tau' \quad \Gamma,\ y : \tau_2 \ \vdash \ e_2 : \tau'}{\Gamma \vdash (\text{case } e \text{ of } x \rightarrow e_1 \parallel y \rightarrow e_2) : \tau'}$$

Constructive Logic

- we did not talk about $\neg$ and Curry-Howard
- functional programming does not generally deal with $\neg$
- functional programming corresponds to **intuitionistic** or **constructive** logic
  - logic system *without* the **law of the excluded middle**

$$\forall x . x \vee \neg x$$

Negation

- what does it mean for $\neg x$ to be true?

$$\neg x \equiv x \Rightarrow \bot$$

- beacuse only

$$\bot \Rightarrow \bot$$

- we can't directly write programs/proofs with this idea

Exceptions

- control flow for handling errors
- does not play well with proving things!

$$\frac{\Gamma \vdash e : \mathbf{exn}}{\text{raise } e : \tau}$$

- we could even have:

$$\text{raise } e : \mathbf{void}$$

- raise does not return to context

Catching Exceptions

- very similar to pattern matching

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \mathbf{exn} \vdash e_2 : \tau}{\Gamma \vdash (\mathsf{try}\ e_1\ \mathsf{with}\ x \Rightarrow e_2) : \tau}$$

- error handler and body have the same type
- exceptions *not* encoded in type system
- good example of isolating the design of a language feature

Generalizing Exceptions

- we can generalize exceptions with **continuations**
- a continuation is a “snapshot” of the current execution
  - can be resumed multiple times
- `callCC` is a very powerful construct for control flow

Continuations

- very versatile
  - exceptions
  - threads
  - coroutines
  - generators
  - backtracking

Basic Idea

- control what happens "next" as a program evaluates
- the next step (continuation) is reified as a function
- the continuation is a *first class value*
  - pass it around
  - call it multiple times—or **none**
  - be happy

Example

$$e_1 + e_2$$
$$\bullet$$

- split into current value (e_1) and "continuation":

$$\bullet + e_2$$

- we could get the continuation as a function:

$$\lambda x.x + e_2$$

callCC

- introduce a new primitive for getting **current continuation**
- callCC —"call with current continuation"
- continuation as *function*
  - calling continuation causes callCC to return
- calls a function with a function...
  - "body" function gets "continuation" function as argument

callCC Example

$$\underset{\bullet}{e_1} + e_2$$

- get continuation out:

$$\text{callCC } k \text{ in } \mathit{body} + e_2$$

- *body* gets $\bullet + e_2$ as k
- original expression *doesn't return*
- calling k is like original expression returning

Early Exit

- we can use continuations to return from an expression early
- like a hypothetical `(return 1) + 10` in a C-like language

$$\text{callCC } \mathit{exit} \text{ in } (\mathit{exit}\ 1) + 10$$

- entire expression evaluates to 1
- similar to exception handling

Types

- we can think of callCC with this type:

$$\mathit{callCC} : ((\tau \to \sigma) \to \tau) \to \tau$$

- note how σ is never used—it can be anything including $\perp$
- $((\tau \to \sigma) \to \tau) \to \tau$ implies the law of the excluded middle
- callCC turns our logic into a classical one!

Negation Again

- remember that $\neg x \equiv x \Rightarrow \perp$
- in $((\tau \to \sigma) \to \tau) \to \tau$, σ is not used
- this means σ can be $\perp$!

$$((\tau \to \perp) \to \tau) \to \tau$$

$$(\neg\tau \to \tau) \to \tau$$

Peirce's Law

- $((\tau \to \sigma) \to \tau) \to \tau$ as an axiom is equivalent to the law of the excluded middle as an axiom
- callCC moves our language from a constructive logic to a classical logic
- a nice proof of this equivalence
- side-note: apparently "Peirce" is pronounced more like "purse"

Continuation-Passing Style

- we can emulate callCC by cleverly structuring our program
- every continuation is explicitly represented as a callback
- this is **continuation-passing style** (CPS)
- used in node.js for concurrency (non-blocking operations)
- normal code can be systematically compiled to CPS

CPS Example

$$add\ x\ y = x + y$$

- CPS version:

$$add\ x\ y\ k = k(x + y)$$

- k is the continuation—a function to call after finishing
  - k is the conventional name for "callback" or "continuation"

CPS Example Usage

$$add\ 1\ (add\ 2\ 3)$$

- CPS-transformed:
$$add\ 2\ 3\ (\lambda x.add\ 1\ x\ (\lambda y.y))$$

- functions never return—call continuation instead
- access result with a $\lambda x.x$ continuation
- callCC just gives access to k

Double Negation Translation

- CPS means we can emulate callCC
- similarly, we can *embed* classical logic into constructive logic
  - called **double negation translation**
- for ever provable proposition ϕ in classical logic, we can prove $\neg\neg\phi$ in constructive logic
  - in constructive logic, $\phi \equiv \neg\neg\phi$ does not necessarily hold

Double Negation Translation Intuition

- $\neg\neg\phi$ is like proving "ϕ does not lead to a contradiction"
- *not* a constructive proof for ϕ because we have not constructed an example of ϕ
- a classical proof can be an example that "ϕ does not lead to a contradiction"

Double Negation and CPS

- CPS transform $\equiv$ double negation
- remember: $\neg x \equiv (x \to \bot)$
- for a constant (say 3), the CPS version is:

$$\lambda k.k(3)$$

- we go from $3 : \mathbf{int}$ to:

$$((\mathbf{int} \to \sigma) \to \sigma)$$

- σ can be anything

Double Negation and CPS

- same trick as before: take σ to be $\bot$:

$$((\textbf{int} \to \bot) \to \bot)$$

- now translate to $\neg$:

$$(\neg\textbf{int} \to \bot)$$

$$\neg(\neg\textbf{int})$$

- since CPS doesn't usually use $\bot$, it's a bit more general

Curry-Howard Conclusion

- programming languages $\equiv$ logic systems
- programs $\equiv$ proofs
- functional $\equiv$ intuitionistic
- imperative $\equiv$ classical
  - "imperative" means exceptions, callCC or similar