Software System Design and Implementation

Curry Howard Correspondence
(Curry Howard Isomorphism)

The University of New South Wales
School of Computer Science and Engineering
Sydney, Australia
Let’s go back in time

- Different, equivalent models of computation to address Hilbert’s Entscheidungsproblem
  - Lambda-calculus (Church)
  - Recursive functions (Gödel)
  - Turing machine
The (untyped) lambda calculus

• Functions can be applied to themselves:

\[ \lambda f. f \ f \]

• As a result, we can have non-terminating reduction sequences:

\[ (\lambda f. f \ f)(\lambda f. f \ f) \rightarrow \beta \]
\[ (\lambda f. f \ f)(\lambda f. f \ f) \rightarrow \beta \]
\[ (\lambda f. f \ f)(\lambda f. f \ f) \rightarrow \beta \]
\[ ... \]
Church’s Simply Typed Lambda Calculus

• For the presentation, we add the following functions & data constructors to the lambda calculus as short hand
  
  • ( , ): like the pair data constructor in Haskell
  • fst, snd: like Haskell fst and snd
  • left, right: like Left and Right of the Either type
  • case: similar to case in Haskell, but restricted to Either type

  case x f g = case x of
    Left a   → f a
    Right b → g b
Church’s Simply Typed Lambda Calculus

- Can be encoded in the lambda-calculus

\[(,) = \lambda a. \lambda b. \lambda f. f \ a \ b\]
\[\text{fst} = \lambda a. \lambda b. a\]
\[\text{snd} = \lambda a. \lambda b. b\]
\[\text{Right} = \lambda a. \lambda f. \lambda g. f \ a\]
\[\text{Left} = \lambda a. \lambda f. \lambda g. g \ a\]
\[\text{case} = \lambda a. \lambda f. \lambda g. a \ f \ g\]
Church’s Simply Typed Lambda Calculus

M :: A  
N :: B  

(M, N) :: A * B

read as:
if you can derive
M :: A and N :: B
then
(M,N) :: A * B
is derivable

M :: A * B

fst M :: A

M :: A * B

snd M :: B
Church’s Simply Typed Lambda Calculus

\[
\begin{align*}
M &:: A \\
\text{left } M &:: A + B \\
M &:: B \\
\text{right } M &:: A + B
\end{align*}
\]

\[
\begin{align*}
M &:: A + B \\
K &:: A \rightarrow C \\
H &:: B \rightarrow C \\
\text{case } M \ K \ H &:: C
\end{align*}
\]
Church’s Simply Typed Lambda Calculus

\[ [x :: A] \]
\[ \vdash \]
\[ M :: B \]
\[ \frac{\lambda x. M :: A \Rightarrow B}{\lambda x. M :: A \Rightarrow B} \]

read as:
if we can derive \( M :: B \)
from the assumption \( x :: A \)
then
\( \lambda x. M :: A \Rightarrow B \)
is derivable

\[ \lambda x. M :: A \Rightarrow B \]
\[ N :: A \]
\[ \frac{(\lambda x. M) N :: B}{\lambda x. M :: A \Rightarrow B \quad N :: A} \]
Church’s Simply Typed Lambda Calculus

• The simply typed lambda calculus doesn’t have general recursion:

\[ \lambda f. \ f \ f \] can’t be typed!

• For all well-typed terms
  - reduction terminates
  - reduction does not change the type of a term

• Note: the Y-combinator can be added to make it turing-complete again:

\[ Y = \lambda f. \ (\lambda x. \ f \ (x \ x))\ ((\lambda x. \ f (x \ x)) \ ) \]

\[ Y \ f = f \ (Y \ f) \]

\[ Y :: (A \rightarrow A) \rightarrow A \]
At around the same time, Gerhard Gentzen was working on the logic aspects of the Hilbert program: establishing the consistency of various logics.

Gentzen introduced two new formulations of logic, which remain the main ones used to this day:

- Sequent calculus
- Natural deduction
Natural Deduction

• Rules come in pairs: introduction and elimination

\[ \frac{A \quad B}{A \land B} \land-I \]

\[ \frac{A \land B}{A} \land-E_1 \]

\[ \frac{A \land B}{B} \land-E_2 \]

\[ \frac{A \land B}{B \land A} \land-E_2 \]

\[ \frac{A \land B}{B} \land-E_1 \]

\[ \frac{A \land B}{A} \land-I \]
Natural Deduction

- χ-introduction and elimination

\[ \begin{align*}
A & \rightarrow C \\
\lor-E
\end{align*} \]
Natural Deduction

• Implication

\[
\begin{align*}
[A] \\
\vdots \\
\therefore B \\
A \rightarrow B &
\end{align*}
\]

\[
\begin{align*}
A \rightarrow B \\
A &
\end{align*}
\]

\[
\begin{align*}
\therefore \neg I \\
B &
\end{align*}
\]

\[
\begin{align*}
\rightarrow E \\
B &
\end{align*}
\]
Proof normalisation

- Gentzen observed that all proofs for propositional logic can be normalised, so they only contain sub formulas of premise or conclusion:
Curry Howard Isomorphism

- In 1934, Curry observed a relationship between logic implication $A \Rightarrow B$ and function types $A \rightarrow B$

- Howard realised in 1969 that this connection is much deeper
Curry Howard Isomorphism

\[
\begin{align*}
M &: A \\
N &: B \\
(M, N) &: A \times B \\
A &\land B \\
A &\land B
\end{align*}
\]

\[
\begin{align*}
M &: A \times B \\
\text{fst } M &: A \\
A &\land B \\
A &\land B
\end{align*}
\]

\[
\begin{align*}
M &: A \times B \\
\text{snd } M &: B \\
A &\land B \\
B
\end{align*}
\]
M :: A

\[ \text{left } M :: A + B \]

M :: B

\[ \text{right } M :: A + B \]

A

\[ A \lor B \]

B

\[ A \lor B \]

M :: A + B

K :: A \rightarrow C

H :: B \rightarrow C

case M K H :: C

A \lor B

A \Rightarrow C

B \Rightarrow C

C
\[ \frac{[x :: A] \quad M :: B}{\lambda x. M :: A \to B} \]

\[ \frac{[A] \quad B \quad A \to B}{\Rightarrow \text{I}} \]

\[ \frac{\lambda x. M :: A \to B \quad N :: A}{(\lambda x. M) N :: B} \]

\[ \frac{A \to B \quad A \quad \Rightarrow \text{E}}{B} \]
\[
\begin{align*}
\text{x :: A} \times \text{B} & \quad \text{x :: A} \times \text{B} \\
\text{snd x :: B} & \quad \text{fst x :: A} \\
(\text{snd x, fst x}) & \quad \text{:: B} \times \text{A}
\end{align*}
\]
• Proof normalisation corresponds to evaluation!
Curry Howard Isomorphism

- Howard proposed extension for for-all and existentially quantified types (now known as dependent types) to predicate logic
  - de Bruijn’s Automath
  - Martin-Löf’s type theory (Agda, Idris)
  - PRL, nuPRL
  - Coquant and Huet’s calculus of constructions (Coq proof assistant)
Curry Howard Isomorphism

• In short, it is the observation that
  • propositions can be viewed as types
  • programs as their (constructive) proof
  • proof normalisation as program evaluation
Curry Howard Isomorphism

- The pattern of logicians/computer scientist discovering the same system independently has repeated since then multiple times:
  - Second order lambda calculus (Jean-Yves Girard, John Reynolds), basis for Java, C#
  - Principal type inference, by Roger Hindley and Robin Milner (e.g., Haskell)
  - Existential quantification in second order logic as basis for abstraction (John Mitchell, Gordon Plotkin)
  - Girard’s linear logic, linear types
  - ...?