Programming = proving?

The Curry-Howard correspondence today

Final lecture

Conclusions, answers to questions, discussion

Xavier Leroy

Collège de France

2019-01-30

I

Summary of the course

The Curry-Howard correspondence

simply-typed λ-calculus	intuitionistic logic
type	proposition
term (program)	proof
reduction (execution)	cut elimination
$A \to B$	implication
$A \times B$	conjunction
$A + B$	disjunction
empty type, unit type	$\perp$, $\top$

Distinct from the "proposition = program" approach
(Church's 1942 λ-calculus, logic programming).

"Implements" the BHK interpretation of intuitionistic logic.

Modern type theories

Richer types and propositions: polymorphism, dependent types, equality.

Syntactic unification between terms and types, controlled by universes.

$\Rightarrow$ Unified formalisms for programming and for proving:
Martin-Löf's type theory, the Calculus of Constructions, Pure Type Systems.

The Curry-Howard-Martin Löf correspondence

Type theory	Set theory	Intuitionistic logic
$A : U$	set	proposition
$A : U$	—	type
$x : A$	element	proof
$0, 1$	$\emptyset$, $\{\emptyset\}$	$\bot$, $\top$
$A \times B$	Cartesian product	conjunction
$A + B$	disjoint union	disjunction
$A \to B$	sets of functions	implications
$x : A \vdash B(x)$	families of sets	predicate
$x : A \vdash b : B(x)$	families of elements	proof under hypothesis
$\Pi x : A.\, B(x)$	product	"for all" quantifier
$\Sigma x : A.\, B(x)$	disjoint sum	"there exists" quantifier
$x =_A y$	equality	equality
$p : x =_A y$	—	equality proof

The Curry-Howard-Martin Löf-Voevodsky correspondence

(From Emily Riehl's presentation at the *Vladimir Voevodsky memorial conference*, 2018)

Type theory	set theory	logic	homotopy theory
$A : U$	set	proposition	space
$A : U$	—	type	—
$x : A$	element	proof	point
$0, 1$	$\emptyset$, $\{\emptyset\}$	$\perp$, $\top$	$\emptyset$, $*$
$A \times B$	Cartesian product	conjunction	product space
$A + B$	disjoint union	disjunction	coproduct
$A \to B$	set of functions	implication	function space
$x : A \vdash B(x)$	family of sets	predicate	fibration
$x : A \vdash b : B(x)$	family of elements	proof under hyp.	section
$\Pi x : A. B(x)$	product	"for all"	space of sections
$\Sigma x : A. B(x)$	disjoint sum	"there exists"	total space
$x =_A y$	equality	equality	path space for A
$p : x =_A y$	—	equality proof	path from x to y

Inductive types and inductive predicates

A general mechanism to define

- data types generated by constructors;
- predicates generated by axioms and rules;

as well as the corresponding recursive functions and inductive proofs.

Extends *mutatis mutandis* to coinduction and to codata.

Towards classical logic

Nice correspondences:

- double negation translations / continuation-passing style (CPS) translations;
- classical laws / control operators (`call/cc`).

A question remains open: what is“the right” calculus to express the computational contents of a classical proof?
(symmetric lambda-calculi, Krivine-style machines, process calculi, interaction nets, etc.)

Transforming programs and proofs

Transforming programs of a language L_1 into a language L_2:
a standard technique in compilation, semantics, and programming.

Transforming propositions and proofs from a logic L_1 to a logic L_2:

- double negation translations;
- intuitionistic forcing;
- parametricity in the style of Bernardy et al;
- syntactic models such as those of Boulier, Pédrot and Tabareau;
- etc.

Effects

The main effects:

- Partiality (general recursion, non-termination).
- Mutability (“in-place” modifications).
- Exceptions, control operators.
- Communications: input-output, shared-memory parallelism, message-passing parallelism.

Monads as a representation for many effects.

Algebraic effects and effect handlers as a more flexible representation of some of these effects.

Program logics to reason about some of these effects (Hoare logic, separation logics, etc).

No generally-applicable correspondence with logic.

A few tools for semantics

Tools more or less inspired by logic to reason about programs and give semantics to programming languages:

- Logical relations, indexed by types or by step counts (*step-indexing*).
- The “topos of trees” and its “later” modality $\triangleright$, to build semantic objects (and reactive programs!) by guarded recursion.

II

Does Curry-Howard make me a better programmer?

The primacy of pure, total, functional programming

At the core of any program, there is a collection of pure, total functions (no state, always terminating).

At the core of any programming language, there should be a pure functional language, preferably typed, preferably guaranteeing termination.

A few reasons:

- These functions are both programs and mathematical definitions, over which we can reason directly (without a program logic).
- “Pure + total” enables static typing with rich types: dependent types, HIT-style equations, etc.
- “Pure + total” enables the language to express proof terms.

Partiality and general recursion

Bad reasons:

- "In order to be Turing-complete."
  (All useful computations are provably terminating.)
- "A Web server must never terminate!"
  (But the processing of every request must terminate $\Rightarrow$ productivity.)

Good reasons:

- Proving the termination of an algorithm can be difficult.
- Coding an algorithm in a normalizing language is even more difficult.
- For many applications, partial correctness is enough.

Beyond termination:

- Guaranteeing worst-case execution time (WCET).
- Guaranteeing a given asymptotic complexity.

Imperative programming and mutable data structures

Bad reasons:

- “A algorithm is a cooking recipe!”
- “That’s the way hardware works!”

Good reasons:

- Many of the fastest known algorithms use mutable state (functional algorithms are slower by a factor $\log n$).
- Low-level systems programming.

Reconciliation:

- Encapsulating mutable state in a pure interface.
- Linearity and control of sharing: separation logic, ownership types, types as permissions, the Rust language.

Objects, classes, inheritance, modules, components, …

Bad reasons:

- "Nature is a class hierarchy."
- Every piece of code must be extensible a posteriori, whatever it costs.

Good reasons:

- Reusing code and its verification.
- Modular decomposition + abstraction barriers.
  (A source of inspiration: algebraic structures.)
- The base mechanisms are well understood: function abstractions (λ), type abstractions ($\exists$), parametric polymorphism ($\forall$).

Beyond the base mechanisms:

- Many higher-level mechanisms, poorly understood, ineffective.

III

Questions and discussions