Programming = proving?
The Curry-Howard correspondence today

Final lecture

Conclusions, answers to questions, discussion

Xavier Leroy

Collège de France

2019-01-30
Summary of the course
The Curry-Howard correspondence

<table>
<thead>
<tr>
<th>simply-typed λ-calculus</th>
<th>intuitionistic logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>proposition</td>
</tr>
<tr>
<td>term (program)</td>
<td>proof</td>
</tr>
<tr>
<td>reduction (execution)</td>
<td>cut elimination</td>
</tr>
<tr>
<td>$A \rightarrow B$</td>
<td>implication</td>
</tr>
<tr>
<td>$A \times B$</td>
<td>conjunction</td>
</tr>
<tr>
<td>$A + B$</td>
<td>disjunction</td>
</tr>
<tr>
<td>empty type, unit type</td>
<td>$\bot, \top$</td>
</tr>
</tbody>
</table>

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.

⇒ 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

<table>
<thead>
<tr>
<th>Type theory</th>
<th>Set theory</th>
<th>Intuitionistic logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A : U$</td>
<td>set</td>
<td>proposition</td>
</tr>
<tr>
<td>$A : U$</td>
<td>—</td>
<td>type</td>
</tr>
<tr>
<td>$x : A$</td>
<td>element</td>
<td>proof</td>
</tr>
<tr>
<td>0, 1</td>
<td>$\emptyset, {\emptyset}$</td>
<td>$\bot, \top$</td>
</tr>
<tr>
<td>$A \times B$</td>
<td>Cartesian product</td>
<td>conjunction</td>
</tr>
<tr>
<td>$A + B$</td>
<td>disjoint union</td>
<td>disjunction</td>
</tr>
<tr>
<td>$A \rightarrow B$</td>
<td>sets of functions</td>
<td>implications</td>
</tr>
<tr>
<td>$x : A \vdash B(x)$</td>
<td>families of sets</td>
<td>predicate</td>
</tr>
<tr>
<td>$x : A \vdash b : B(x)$</td>
<td>families of elements</td>
<td>proof under hypothesis</td>
</tr>
<tr>
<td>$\prod x : A. B(x)$</td>
<td>product</td>
<td>“for all” quantifier</td>
</tr>
<tr>
<td>$\Sigma x : A. B(x)$</td>
<td>disjoint sum</td>
<td>“there exists” quantifier</td>
</tr>
<tr>
<td>$x =_A y$</td>
<td>equality</td>
<td>equality</td>
</tr>
<tr>
<td>$p : x =_A y$</td>
<td>—</td>
<td>equality proof</td>
</tr>
</tbody>
</table>
# The Curry-Howard-Martin Lőf-Voevodsky correspondence

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

<table>
<thead>
<tr>
<th>Type theory</th>
<th>set theory</th>
<th>logic</th>
<th>homotopy theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A : U$</td>
<td>set</td>
<td>proposition</td>
<td>space</td>
</tr>
<tr>
<td>$A : U$</td>
<td>—</td>
<td>type</td>
<td>—</td>
</tr>
<tr>
<td>$x : A$</td>
<td>element</td>
<td>proof</td>
<td>point</td>
</tr>
<tr>
<td>0, 1</td>
<td>$\emptyset, {\emptyset}$</td>
<td>$\bot, \top$</td>
<td>$\emptyset, *$</td>
</tr>
<tr>
<td>$A \times B$</td>
<td>Cartesian product</td>
<td>conjunction</td>
<td>product space</td>
</tr>
<tr>
<td>$A + B$</td>
<td>disjoint union</td>
<td>disjunction</td>
<td>coproduct</td>
</tr>
<tr>
<td>$A \rightarrow B$</td>
<td>set of functions</td>
<td>implication</td>
<td>function space</td>
</tr>
<tr>
<td>$x : A \vdash B(x)$</td>
<td>family of sets</td>
<td>predicate</td>
<td>fibration</td>
</tr>
<tr>
<td>$x : A \vdash b : B(x)$</td>
<td>family of elements</td>
<td>proof under hyp.</td>
<td>section</td>
</tr>
<tr>
<td>$\Pi x : A. B(x)$</td>
<td>product</td>
<td>“for all”</td>
<td>space of sections</td>
</tr>
<tr>
<td>$\Sigma x : A. B(x)$</td>
<td>disjoint sum</td>
<td>“there exists”</td>
<td>total space</td>
</tr>
<tr>
<td>$x \equiv_A y$</td>
<td>equality</td>
<td>equality</td>
<td>path space for $A$</td>
</tr>
<tr>
<td>$p : x \equiv_A y$</td>
<td>—</td>
<td>equality proof</td>
<td>path from $x$ to $y$</td>
</tr>
</tbody>
</table>
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.
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.
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 ⇒ 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.
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 (\(\lambda\)), type abstractions (\(\exists\)), parametric polymorphism (\(\forall\)).

Beyond the base mechanisms:

- Many higher-level mechanisms, poorly understood, ineffective.
Questions and discussions