Mechanized semantics, seventh lecture

Of functions and types: the semantics of a functional language

Xavier Leroy

2020-02-06

Collège de France, chair of software sciences

A change of paradigm

IMP, a toy imperative language

- Running a program = modifying the state
- Basic operation: assignment
- Control structures: conditional, loops
- Data types: first order (e.g. numbers).

FUN, a toy functional language

- Running a program = computing its value.
- Basic operations: function abstraction, function application.
- Control structures: conditional, recursion.
- Data types: higher order (functions as first-class values).

The FUN functional language

A recipe for a functional language

	the lambda-calculus
$+$	a reduction strategy
$+$	primitive data types
$+$	a type system
$=$	a functional language

The lambda-calculus

Terms: $M, N ::= x$ variables

$\mid \lambda x.\, M$ function abstraction ($x \mapsto M$)

$\mid M\ N$ function application

One structural rule: α-conversion (renaming of bound variables)

$$\lambda x.\, M =_\alpha \lambda y.\, M\{x \leftarrow y\} \qquad \text{if } y \text{ not free in } M$$

One computation rule: β-reduction

$$(\lambda x.\, M)\ N \to_\beta M\{x \leftarrow N\}$$

Good properties of reductions

Theorem (Church and Rosser, 1935)

The β-reduction is confluent:
if $M \xrightarrow{*} M_1$ and $M \xrightarrow{*} M_2$, there exists M' such that $M_1 \xrightarrow{*} M'$ and $M_2 \xrightarrow{*} M'$.

$$\begin{tikzcd}[column sep=small]
 & M \arrow[dl, "*"'] \arrow[dr, "*"] & \\
M_1 \arrow[dr, dashed, "*"'] & & M_2 \arrow[dl, dashed, "*"] \\
 & M' &
\end{tikzcd}$$

We say that N is a normal form of M if $M \xrightarrow{*} N \not\rightarrow$

Corollary

The normal form of a term, if it exists, is unique.

Expressiveness of lambda-calculus

Lambda-calculus is Turing-complete.

In particular, via functional encodings, it can express

- All the usual data types: integers, pairs, lists, …

  Example: Church's encoding of natural numbers

$$n \;\equiv\; \lambda f.\, \underbrace{f \circ \cdots \circ f}_{n \text{ times}} \;\equiv\; \lambda f.\, \lambda x.\, \underbrace{f\ (f\ (\cdots\ (f\ x)))}_{n \text{ times}}$$

- General recursion via fixed-point combinators.

  Example: the combinator $Y = \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$ is such that $Y\ F \xrightarrow{*} F\ (Y\ F)$.

Why lambda-calculus is not a good programming language

Little control over termination and complexity

Non-determinism caused by β-reductions that can apply in several places and in any order. Depending on the way β-reductions are performed,

- a computation can diverge or terminate;
- it can terminate quickly or slowly.

Functional encodings of data structures are limited

- Unnatural.
- Generally inefficient.
- Not typable in several standard type systems.

Reduction strategies

Make β-reduction deterministic by restricting where and when it can be performed. Two main choices:

- **Strong vs weak reduction**: can we reduce “under a λ”?
  Weak reduction: a function body is evaluated only after the function is applied.
  Strong reduction: we can simplify the function body before application.
- **Call-by-name vs call-by-value**:
  By value: the argument must be evaluated before being passed to the function.
  By name: the argument is passed as is, not necessarily evaluated.

Specifying a strategy: the “SOS” style

(G. Plotkin, *A structural approach to operational semantics*, 1981, 2004.)

Axioms and inference rules for a relation $M \to M'$
(read: the whole term M reduces into the term M').

Weak call-by-name

$$(\lambda x.\, M)\; N \to M\{x \leftarrow N\}$$

$$\frac{M \to M'}{M\; N \to M'\; N}$$

Weak call-by-value
left to right

$$(\lambda x.\, M)\; v \to M\{x \leftarrow v\}$$

$$\frac{M \to M'}{M\; N \to M'\; N} \qquad \frac{N \to N'}{v\; N \to v\; N'}$$

(Here, values, written v, are just the lambdas: $v ::= \lambda x.\, M$)

Specifying a strategy: via a grammar of contexts

(A. Wright, M. Felleisen, *A Syntactic Approach to Type Soundness*, 1994).

One general reduction rule under a context E:

$$\frac{M \to_\varepsilon M' \qquad E \in Ctx}{E[M] \to E[M']}$$

For each strategy, axioms for head reductions $\to_\varepsilon$ and a grammar defining the valid contexts E:

Weak call-by-name

$(\lambda x.\, M)\; N \to_\varepsilon M\{x \leftarrow N\}$

$E ::= [\,] \mid E\; N$

Weak call-by-value

$(\lambda x.\, M)\; v \to_\varepsilon M\{x \leftarrow v\}$

Left to right: $E ::= [\,] \mid E\; N \mid v\; E$

Right to left: $E ::= [\,] \mid E\; v \mid M\; E$

Specifying a strategy: via a natural semantics

Like we already did for IMP, we can summarize finite reduction sequences to a value $M \xrightarrow{*} v \not\to$ by a predicate $M \Downarrow v$, "term M evaluates to value v".

Weak call-by-name:

$$\lambda x.\, M \Downarrow \lambda x.\, M \qquad\qquad \frac{M \Downarrow \lambda x.\, P \qquad P\{x \leftarrow N\} \Downarrow v}{M\; N \Downarrow v}$$

Weak call-by-value:

$$\lambda x.\, M \Downarrow \lambda x.\, M \qquad\qquad \frac{M \Downarrow \lambda x.\, P \qquad N \Downarrow v' \qquad P\{x \leftarrow v'\} \Downarrow v}{M\; N \Downarrow v}$$

Adding primitive data types

A systematic process: add

- new syntactic forms to the grammar of terms;
- new head reduction rules;
- new cases to the grammars of values and of contexts.

Starting point: weak call-by-value.

Terms: $M, N ::= x \mid \lambda x.\, M \mid M\ N$

Values: $v ::= \lambda x.\, M$

Contexts: $E ::= [\,] \mid E\ M \mid v\ E$

Head reduction: $(\lambda x.\, M)\ v \to_\varepsilon M\{x \leftarrow v\}$

Booleans

Terms: $M ::= \ldots \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ M_1\ M_2\ M_3$

Values: $v ::= \ldots \mid \texttt{true} \mid \texttt{false}$

Contexts: $E ::= \ldots \mid \texttt{if}\ E\ M_2\ M_3$

$$\texttt{if true}\ M_2\ M_3 \to_\varepsilon M_2$$

$$\texttt{if false}\ M_2\ M_3 \to_\varepsilon M_3$$

Peano natural numbers

Terms: $M ::= \ldots \mid 0 \mid S\,M \mid \texttt{if0}\ M_1\ M_2\ M_3$

Values: $v ::= \ldots \mid 0 \mid S\,v$

Contexts: $E ::= \ldots \mid S\,E \mid \texttt{if0}\ E\ M_2\ M_3$

$$\texttt{if0}\ 0\ M_2\ M_3 \to_\varepsilon M_2$$

$$\texttt{if0}\ (S\,v)\ M_2\ M_3 \to_\varepsilon M_3\ v$$

Products and sums

$$
\begin{array}{ll}
\text{Terms:} & M ::= \ldots \mid (M_1, M_2) \mid \texttt{fst}\ M \mid \texttt{snd}\ M \\
& \quad \mid \texttt{left}\ M \mid \texttt{right}\ M \mid \texttt{case}\ M\ M_1\ M_2 \\
\text{Values:} & v ::= \ldots \mid (v_1, v_2) \mid \texttt{left}\ v \mid \texttt{right}\ v \\
\text{Contexts:} & E ::= \ldots \mid (E, M) \mid (v, E) \mid \texttt{fst}\ E \mid \texttt{snd}\ E \\
& \quad \mid \texttt{left}\ E \mid \texttt{right}\ E \mid \texttt{case}\ E\ M_2\ M_3
\end{array}
$$

$$
\begin{array}{ll}
\texttt{fst}\ (v_1, v_2) \rightarrow_\varepsilon v_1 & \texttt{case}\ (\texttt{left}\ v)\ M_2\ M_3 \rightarrow_\varepsilon M_2\ v \\
\texttt{snd}\ (v_1, v_2) \rightarrow_\varepsilon v_2 & \texttt{case}\ (\texttt{right}\ v)\ M_2\ M_3 \rightarrow_\varepsilon M_3\ v
\end{array}
$$

Fixed points (general recursion)

$$\begin{array}{ll} \text{Terms:} & M ::= \ldots \mid \texttt{fix}\, M \\ \text{Values:} & v ::= \ldots \mid \texttt{fix}\, v \\ \text{Contexts:} & E ::= \ldots \mid \texttt{fix}\, E \end{array}$$

$$\texttt{fix}\, v_f\ v \rightarrow_\varepsilon v_f\ (\texttt{fix}\, v_f)\ v$$

Mechanizing a functional language and its semantics

See the Coq development `FUN.v`.

The basic tools are the same as for IMP:

- Inductive types for abstract syntax.
- Inductive predicates for reduction and evaluation relations.

A delicate issue: α-conversion

$$\lambda x.\, M =_\alpha \lambda y.\, M\{x \leftarrow y\} \qquad \text{if } y \text{ not free in } M$$

It is not obvious how to consider terms modulo α-conversion, that is, equal up to a renaming of bound variables.

Making do without alpha-conversion

The development `FUN.v` represents terms without implicit renaming of bound variables:

$$\texttt{Abs("x", Var "x")} \neq \texttt{Abs("y", Var "y")}$$

This is a problem to define substitution $M\{x \leftarrow N\}$:
the naive definition

$$(\lambda y.\, M)\{x \leftarrow N\} = \lambda y.\, (M\{x \leftarrow N\})$$

is vulnerable to variable capture.

For example $(\lambda y.\, x)\{x \leftarrow y\}$ is computed as $\lambda y.\, y$ ✘

Making do without alpha-conversion

The naive definition of substitution

$$(\lambda y.\, M)\{x \leftarrow N\} = \lambda y.\, (M\{x \leftarrow N\})$$

is correct if the term N is closed, i.e. without free variables.

(If N is closed, $\lambda y \ldots N \ldots$ cannot capture a y free in N.)

Fortunately, reducing a closed term (a complete program) produces only closed terms:

$$\underbrace{Prog}_{\text{closed}} \to \cdots \to \underbrace{(\lambda x.\, M)}_{\text{closed}} \; \underbrace{N}_{\text{closed}} \to \underbrace{M\{x \leftarrow N\}}_{\text{closed}} \to \cdots$$

Hence, the semantics we obtain is valid only for complete programs.

A type system with simple types

Absurd programs

“Don’t compare apples with oranges.”

“On n’additionne pas des choux et des carottes.”

When we enrich lambda-calculus with data types such as Booleans, absurd terms appear:

$\texttt{true}\ (\lambda x.\, x)$	(a Boolean used as if it were a function)
$\texttt{if}\ (\lambda x.\, x)\ M\ M'$	(a function used as if it were a Boolean)

Dynamic typing, static typing

Dynamic typing:

detect and report these absurdities during execution

$$(\lambda b.\, \texttt{if}\ b\ M\ M')\ (\lambda x.\, x) \to \texttt{if}\ (\lambda x.\, x)\ M\ M' \to \texttt{ERROR}$$

Static typing:

analyze terms before execution to “statically” reject the terms that are not well typed.

- ✔ $\lambda b : \texttt{bool}.\, \texttt{if}\ b\ \texttt{false}\ \texttt{true} \ : \ \texttt{bool} \to \texttt{bool}$
- ✘ $(\lambda b : \texttt{bool}.\, \texttt{if}\ b\ \texttt{false}\ \texttt{true})\ (\lambda x.\, x)$
- ✘ $\lambda b : \texttt{bool} \to \texttt{bool}.\, \texttt{if}\ b\ \texttt{false}\ \texttt{true}$

A static type system

A type algebra, for example Church's simple types

$$\begin{array}{llll} \text{Types:} & \tau, \sigma ::= \texttt{bool} & & \text{base type} \\ & \quad\quad\;\; \mid \sigma \to \tau & & \text{type of functions from } \sigma \text{ to } \tau \end{array}$$

Typing rules that define a relation $\Gamma \vdash M : \tau$

read: "in context Γ term M is well typed and has type τ".

The context Γ is a list of assumptions $x_1 : \tau_1, \ldots, x_n : \tau_n$ associating each free variable x_i with its type τ_i.

Typing rules for simple types

The simply-typed lambda-calculus:

$$\frac{\Gamma = \ldots, x : \tau, \ldots}{\Gamma \vdash x : \tau}\ \text{(Var)} \qquad \frac{x \notin \mathit{Dom}(\Gamma) \qquad \Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.\, M : \sigma \to \tau}\ \text{(Abs)}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}\ \text{(App)}$$

Extension with Booleans:

$$\Gamma \vdash \texttt{true} : \texttt{bool}\ \text{(Cst)} \qquad \Gamma \vdash \texttt{false} : \texttt{bool}\ \text{(Cst)}$$

$$\frac{\Gamma \vdash M : \texttt{bool} \qquad \Gamma \vdash N : \tau \qquad \Gamma \vdash P : \tau}{\Gamma \vdash \texttt{if}\ M\ N\ P : \tau}\ \text{(If)}$$

Type soundness

Well-typed programs do not go wrong. (R. Milner)

A type system is sound if no program that is well typed in the empty context can "go wrong", i.e. produce a run-time error such as `true` $(\lambda x.x)$.

Formulated in terms of reduction sequences:

$$\begin{array}{rl} \text{Normal termination:} & M \to \cdots \to v \in \mathit{Val} \\ \text{Abnormal termination (going wrong):} & M \to \cdots \to N \not\to, N \notin \mathit{Val} \\ \text{Divergence:} & M \to \cdots \to M' \to \cdots \end{array}$$

Type soundness = if $\emptyset \vdash M : \tau$, the "going wrong" case is impossible.

(Normalization = if $\emptyset \vdash M : \tau$, the "divergence" case is impossible.)

Various ways to prove type soundness

Using a denotational semantics: (1975–1985)

(D. MacQueen, G. Plotkin, R. Sethi, *An ideal model for recursive polymorphic types*, 1986)

- Write a denotational semantics $[\![M]\!]$ where the domain of denotations contains a special element $\mathtt{err}$.
  For example: $D \simeq Bool_\perp + [D \to D] + \{\mathtt{err}\}_\perp$.
- Interpret types τ as sets $[\![\tau]\!]$ not containing $\mathtt{err}$.
- Show that if $\emptyset \vdash M : \tau$, then $[\![M]\!] \in [\![\tau]\!]$

Various ways to prove type soundness

Using a denotational semantics: (1975–1985)

Using a natural semantics: (1980–1995)

(M. Tofte, *Operational semantics and polymorphic type inference*, PhD Edinburgh, 1988)

- Write two natural semantics: $M \Downarrow v$ for normal termination, $M \Downarrow \texttt{err}$ for abnormal termination (going wrong).
- Show that if $\emptyset \vdash M : \tau$, then $M \not\Downarrow \texttt{err}$, and $M \Downarrow v \Rightarrow v \in \tau$.

Various ways to prove type soundness

Using a denotational semantics: (1975–1985)

Using a natural semantics: (1980–1995)

Using a reduction semantics: (since 1995)

(A. Wright et M. Felleisen, *A syntactic approach to type soundness*, 1994)

- Show two properties of reductions:
  progress and preservation.

The progress property

Show that a well-typed program does not go wrong immediately.

Theorem (Progress)

If $\emptyset \vdash M : \tau$, either M is a value or M can reduce ($M \to N$ for some N).

Uses a lemma that determines the shapes of values according to their types.

Lemma (Canonical forms)

Let v be a value.
If $\emptyset \vdash v : \sigma \to \tau$ then v is of the shape $\lambda x.\, M$.
If $\emptyset \vdash v : \texttt{bool}$ then v is `true` or `false`.

The preservation property (*subject reduction*)

Well-typedness is preserved by reduction steps.

Theorem (Preservation)

If $\Gamma \vdash M : \tau$ *and* $M \to N$ *then* $\Gamma \vdash N : \tau$.

Uses a substitution lemma and a weakening lemma.

Lemma (Typing is stable by substitution)

If $\Gamma, x : \sigma, \Gamma' \vdash M : \tau$ *and* $\Gamma \vdash N : \sigma$ *then* $\Gamma, \Gamma' \vdash M\{x \leftarrow N\} : \tau$.

Lemma (Weakening)

If $\Gamma \vdash M : \tau$ *then* $\Gamma, \Gamma' \vdash M : \tau$.

Type soundness

Well-typed programs do not go wrong.

Let M be a closed, well-typed program: $\emptyset \vdash M : \tau$.

Assume that M goes wrong:

$$M \to \cdots \to N \not\to, N \notin \mathit{Val}$$

By (iterated) preservation, $\emptyset \vdash N : \tau$.

By progress, either N is a value or N reduces.

Contradiction!

Intrinsically-typed terms

Two views of typing

The “extrinsic” view, in the style of Curry:

- Abstract syntax and semantics are defined independently of the type system.
- The type system is a “filter” (a static analysis) that eliminates problematic terms.

Then “intrinsic” view, in the style of Church:

- The type system participates in the definition of the terms of the language. E.g. Church’s simply-typed lambda-calculus:

$$M_\tau ::= x_\tau \mid (\lambda x_\sigma .\, M_\tau)_{\sigma\to\tau} \mid (M_{\sigma\to\tau}\; N_\sigma)_\tau$$

- Semantics is defined on well-typed terms only.

Dependent types and intrinsic typing

Church's intrinsic view can be expressed using dependent types (Coq, Agda, …) or generalized algebraic data types (GADTs) (Haskell, OCaml).

The type of terms $\mathtt{term}\ \Gamma\ \tau$ is parameterized by a typing context Γ and a type expression τ.

$$\begin{aligned}
\mathtt{Const} &: \mathtt{bool} \to \mathtt{term}\ \Gamma\ \mathtt{Bool} \\
\mathtt{Cond} &: \mathtt{term}\ \Gamma\ \mathtt{Bool} \to \mathtt{term}\ \Gamma\ \tau \to \mathtt{term}\ \Gamma\ \tau \to \mathtt{term}\ \Gamma\ \tau \\
\mathtt{App} &: \mathtt{term}\ \Gamma\ (\mathtt{Fun}\ \sigma\ \tau) \to \mathtt{term}\ \Gamma\ \sigma \to \mathtt{term}\ \Gamma\ \tau \\
\mathtt{Abs} &: \mathtt{term}\ (\sigma :: \Gamma)\ \tau \to \mathtt{term}\ \Gamma\ (\mathtt{Fun}\ \sigma\ \tau) && (?) \\
\mathtt{Var} &: \mathtt{var}\ \Gamma\ \tau \to \mathtt{term}\ \Gamma\ \tau && (?)
\end{aligned}$$

Representing variables

In the intrinsic approach, a variable designates one of the typing assumptions in the context. This assumption determines the type of the variable. There should be no way to mention a variable that is not described in the context!

Designating variables by names:
feasible, but can raise problems with renaming.

Designating variables by positions:
quite natural: context $\approx$ list, assumption $\approx$ position in the list.
It is de Bruijn's notation (1972)!

de Bruijn's notation

(N. de Bruijn, *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation*, 1972.)

Instead of identifying variables by names, de Bruijn's notation identifies them by their positions relative to the λ-abstractions that bind them.

$$\begin{array}{llllll} \lambda \mathtt{x}. & (\lambda \mathtt{y}. & \mathtt{y} & \mathtt{x}) & \mathtt{x} \\ & & | & | & | \\ \lambda . & (\lambda . & \underline{1} & \underline{2}) & \underline{1} \end{array}$$

$\underline{n}$ is the variable bound by the n-th enclosing λ.

Two α-convertible terms are equal in de Bruijn's notation: $\lambda x.\, x$ and $\lambda y.\, y$ are both représented as $\lambda.\, \underline{1}$

Intrinsically-typed de Bruijn's notation

A context Γ is a list of types $\tau_1 :: \cdots :: \tau_n :: \mathit{nil}$ where τ_i is the type of the variable having de Bruijn index i.

The type $\mathtt{var}\ \Gamma\ \tau$ of variables of type τ in context Γ is isomorphic to the integers between 1 and the size n of Γ.

This type is generated by two constructors:

$$\begin{array}{lcll} \mathtt{V1} & : & \mathtt{var}\ (\tau :: \Gamma)\ \tau & \text{(one)} \\ \mathtt{VS} & : & \mathtt{var}\ \Gamma\ \tau \to \mathtt{var}\ (\sigma :: \Gamma)\ \tau & \text{(successor)} \end{array}$$

Derived definitions:

$$\begin{array}{lclcl} \mathtt{V2} & = & \mathtt{VS}\ \mathtt{V1} & : & \mathtt{var}\ (\tau_1 :: \tau_2 :: \Gamma)\ \tau_2 \\ \mathtt{V3} & = & \mathtt{VS}\ \mathtt{V2} & : & \mathtt{var}\ (\tau_1 :: \tau_2 :: \tau_3 :: \Gamma)\ \tau_3 \end{array}$$

A denotational semantics for intrinsically-typed terms

We can define an interpretation of FUN type expressions as Coq types:

$$[\![\mathtt{Bool}]\!] = \mathtt{bool} \qquad\qquad [\![\mathtt{Fun}\ \sigma\ \tau]\!] = [\![\sigma]\!] \to [\![\tau]\!]$$

Typing contexts become the Coq types for evaluation environments that associate a value to each variable of the context:

$$[\![nil]\!] = \mathtt{unit} \qquad\qquad [\![\tau :: \Gamma]\!] = [\![\tau]\!] * [\![\Gamma]\!]$$

We can, then, interpret a term $a : \mathtt{term}\ \Gamma\ \tau$ as a Coq function environment $\mapsto$ value:

$$[\![a]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$$

A denotational semantics for intrinsically-typed terms

$$
\begin{aligned}
[\![\mathtt{Var}\ \mathtt{V1}]\!]\ e &= \mathtt{fst}(e) \\
[\![\mathtt{Var}\ (\mathtt{VS}\ v)]\!]\ e &= [\![\mathtt{Var}\ v]\!]\ (\mathtt{snd}\ e) \\
[\![\mathtt{Abs}\ a]\!]\ e &= \mathtt{fun}\ x \Rightarrow [\![a]\!]\ (x, e) \\
[\![\mathtt{App}\ a_1\ a_2]\!]\ e &= ([\![a_1]\!]\ e)\ ([\![a_2]\!]\ e) \\
[\![\mathtt{Const}\ b]\!]\ e &= b \\
[\![\mathtt{Cond}\ a_1\ a_2\ a_3]\!]\ e &= \mathtt{if}\ [\![a_1]\!]\ e\ \mathtt{then}\ [\![a_2]\!]\ e\ \mathtt{else}\ [\![a_3]\!]\ e
\end{aligned}
$$

This defines a Coq function that is well-typed and total
$\Rightarrow$ type soundness and normalization hold “by construction”.

The equations of denotational semantics are satisfied.

Compatible with reductions: if $a \rightarrow a'$ then $[\![a]\!] = [\![a']\!]$.

Limitations of the intrinsic approach

The features of the object language (FUN) must be available or encodable in the host language (Coq).

- Effects (including divergence) $\Rightarrow$ monadic encoding.
- Subtyping $\Rightarrow$ coercions $[\![\mathit{subtype}]\!] \to [\![\mathit{supertype}]\!]$.
- Impredicative polymorphism (system F) $\Rightarrow$ Coq's option `-impredicative-set`.

The host language must have inductive families (GADTs) and preferably full dependent types $\Rightarrow$ excludes HOL, PVS, …

We explain simple languages (such as FUN) in terms of a more complex language (OCaml, Haskell, Agda, Coq).

Summary

Summary

Functional languages (syntax, semantics, typing) mechanize very well, generally speaking…

… modulo a few difficulties to account for bound variables and alpha-conversion (equivalence up to renaming of bound variables).

Many type systems have been mechanized, including advanced features such as

- Subtype polymorphism (e.g. $\texttt{bool} <: \texttt{int}$)
- Parametric polymorphism (e.g. $\forall\alpha.\ \alpha \to \alpha$)
- Dependent types (e.g. $\texttt{term}\ \Gamma\ \tau$)

The next lecture reconsiders the latter two from a logical perspective (that of type theory).

References

References

Two textbooks on typed functional languages:

- Benjamin Pierce. *Types and Programming Languages.* MIT Press, 2002.
- Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, 2016.

Mechanizations of typed functional languages:

- Extrinsic approach, in Coq: Benjamin Pierce et al, *Software Foundations, volume 2: Programming Languages Foundations*, `https://softwarefoundations.cis.upenn.edu/`.
- Intrinsic approach, in Agda: Philip Wadler, *Programming Language Foundations in Agda*, `https://plfa.github.io/`