Formalization of Lumo IR v1.0.0-b144b77 (2026-03-26)

1. Introduction

In this document, we write a Lumo IR, built upon:

- *Algebraic Effects* research done by **Effekt** (Brachthäuser et al. 2020).
- *Linear Resource Calculus*(λ^1) proposed by Perceus (Reinking et al. 2021).
- *Mutable Value Semantics* formalized by **Hylo/Val** (Cordonier et al. 2022; 2021).

2. Reading Typing Rules

We'll use several typing judgments with the following contexts:

- Δ: Variable and its Assigned Types, Borrowed. Must own before use.
- Γ: Variable and its Assigned Types, Owned. Use values exactly once.
- E: Named Capabilities available in this context, implemented in a Bundle.

Judgment forms:

- $\Delta \mid \Gamma; E \vdash^{V} x \Rightarrow A$: The inferred type of value x is A
- $\Delta \mid \Gamma; E \vdash^{V} x \Leftarrow A$: The value x must be type-checked against A
- $\Delta \mid \Gamma; E \vdash^{C} M \Rightarrow \underline{B}$: The inferred type of computation M is $\underline{B}$
- $\Delta \mid \Gamma; E \vdash^{C} M \Leftarrow \underline{B}$: The computation M must be type-checked against $\underline{B}$

Pattern binder notation:

- $\mathrm{bv}(p)$: the set (or ordered list) of variables bound by pattern p.
- $\mathrm{bv}(\_) = \emptyset$.
- $\mathrm{LetSlots}(p)$: variables introduced as **let** slots by pattern p.
- $\mathrm{MutSlots}(p)$: variables introduced as **mut** slots by pattern p.

Pattern syntax:

$$p ::= \_ \mid i(p_0, \cdots, p_n) \mid \textbf{let}\ x \mid \textbf{mut}\ x$$

3. Tags

Tags are second-class names, used for places such as variant constructors and function names inside bundles. They cannot be used alone. For effects, we assume a tag-extraction operation $\mathrm{EffectTag}(e)$ for every effect e (including instantiated effects after generic monomorphization).

4. Types

Lumo IR uses System F with several extensions: higher-order types (ω), recursive types (μ), and a Call-by-Push-Value style separation between values (A) and computations ($\underline{B}$).

$$
\begin{aligned}
K &::= * \mid K \to K \\
A &::= X \mid i \ \textbf{of}\ [A_0, A_1, \cdots, A_n] \mid A + A \mid \textbf{thunk}\ \underline{B} \mid \textbf{mut}\ A \\
&\quad \mid \textbf{forall}\ [\ X : K\ ]\ .A \mid A\ [\ A\ ] \mid \mu X.A \\
\underline{B} &::= \textbf{produce}\ A \mid A \to \underline{B} \mid \Pi_{(i \in I)} \left(i \times \underline{B_i} \right)
\end{aligned}
$$

4.1. Type Formation

We separate value types (`"Type"`) from computation types (`"CType"`).

$$\frac{A : \text{Type}}{\textbf{produce}\ A : \text{CType}}$$

$$\frac{A : \text{Type} \quad \underline{B} : \text{CType}}{(A \to \underline{B}) : \text{CType}}$$

$$\frac{\forall i \in I. \left(\underline{B_i} : \text{CType} \right)}{\Pi_{(i \in I)} \left(i \times \underline{B_i} \right) : \text{CType}}$$

bundle values are named in E and treated as computation-level entities.

5. On Bidirectional Typing

We use Pfenning Recipe for our type system with following rules included.

5.1. Variable Synthesis

Variables synthesize their type directly from the owned context.

$$\frac{x : A \in \Gamma}{\Delta \mid \Gamma; E \vdash^V x \Rightarrow A}\,[\textsc{Var-Synth}]$$

5.2. Annotation Synthesis

A type annotation upgrades checking into synthesis.

$$\frac{\Delta \mid \Gamma; E \vdash^V v \Leftarrow A}{\Delta \mid \Gamma; E \vdash^V (v : A) \Rightarrow A}\,[\textsc{Ann-Synth}]$$

5.3. Value Synth-to-Check

Any synthesized value can be consumed in checking mode.

$$\frac{\Delta \mid \Gamma; E \vdash^V v \Rightarrow A}{\Delta \mid \Gamma; E \vdash^V v \Leftarrow A}\,[\text{V-}\textsc{Synth-Check}]$$

5.4. Computation Synth-to-Check

Any synthesized computation can be consumed in checking mode.

$$\frac{\Delta \mid \Gamma; E \vdash^C M \Rightarrow \underline{B}}{\Delta \mid \Gamma; E \vdash^C M \Leftarrow \underline{B}}\,[\text{C-}\textsc{Synth-Check}]$$

5.5. Forall Elimination

Type application instantiates a polymorphic value and synthesizes the instantiated type.

$$\frac{\Delta \mid \Gamma; E \vdash^V u \Rightarrow \textbf{forall}[X : K].A \qquad T : K}{\Delta \mid \Gamma; E \vdash^V u[T] \Rightarrow A[X := T]}\,[\textsc{Forall-Elim-Synth}]$$

6. Data

Lumo IR uses a single concept for constructing values, called **data**, which can cover the following real-world practices:

- `enum`: finite **data** variants
- `struct`: single **data** variant
- `primitives`: modeled as infinite-variant **data**.
  for example: **data** nat $:=: \{x \mid x \in \mathbb{N}\}$

6.1. Definition of Data

$$\textbf{data}\ T = \Sigma_{(i \in I)} i\ \textbf{of}\ [A_0, A_1, \cdots, A_n]$$

The **of** [...] part can be omitted; if omitted, the tag is treated as nullary.

Examples:

$$\textbf{data}\ \text{Nat} = \text{Zero} + \text{Succ}\ \textbf{of}\ [\text{Nat}]$$

$$\textbf{data}\ \text{Bool} = \text{False} + \text{True}$$

$$\textbf{data}\ \text{NatList} = \text{Nil} + \text{Cons}\ \textbf{of}\ [\text{Nat}, \text{NatList}]$$

6.2. Rules of Data

6.2.1. Introduction

Before constructing `data`, we must "own" the parameters — each p_i must be available in Γ_i — and consume contexts sequentially so the constructed data owns its values.

$$\frac{\Delta \mid \Gamma_{i+1}, \cdots, \Gamma_n; E \mid \Gamma_i \vdash^V p_i \Leftarrow A_i \quad (0 \le i \le n)}{\Delta \mid \Gamma_0, \Gamma_1, \cdots, \Gamma_n; E \vdash^V i(p_0, p_1, \cdots, p_n) \Leftarrow i\ \textbf{of}\ [A_0, A_1, \cdots, A_n]}\,[\textsc{Data-Intro}]$$

We can get `data T` by `roll`-ing the expression above with type assertion `(e : T)`.

6.2.2. Elimination

Next we eliminate unrolled data (a sum of variants). **match** requires an owned variable `x`, and each branch is checked under freshly owned bindings introduced by the match pattern. For the bindings, we can use **let** x and for mut data, **mut** x.

$$\frac{\forall i \in I.\ \text{MutSlots}(p_i) = \emptyset \qquad \Delta \mid \Gamma, \text{bv}(p_i); E \vdash^C e_i \Leftarrow \underline{B}}{\Delta \mid \Gamma, x; E \vdash^C \textbf{match}\ x\{p_i \mapsto e_i\} \Leftarrow \underline{B}}\,[\textsc{Data-Elim}]$$

$$\frac{x : \textbf{mut}\ A \in \Gamma \qquad \Delta \mid \Gamma, \text{bv}(p_i); E \vdash^C e_i \Leftarrow \underline{B}}{\Delta \mid \Gamma, x; E \vdash^C \textbf{match}\ x\{p_i \mapsto e_i\} \Leftarrow \underline{B}}\,[\textsc{Data-Elim-Mut}]$$

6.2.3. β-reduction and η-expansion

When a pattern matches, only bound variables are substituted. Wildcard _ drops the matched value and contributes no binding.

$$\begin{gathered}\mathrm{binds}(\_, v) = [] \\ \mathrm{binds}(\textbf{let}\ x, v) = [x := v] \\ \mathrm{binds}(\textbf{mut}\ x, v) = [x := v] \\ \mathrm{binds}(i(p_0, \cdots, p_n), i(v_0, \cdots, v_n)) = \mathrm{binds}(p_0, v_0), \cdots, \mathrm{binds}(p_n, v_n)\end{gathered}$$

$$\begin{gathered}\textbf{match}\ i(v_0, v_1, \cdots, v_n)\{i(p_0, p_1, \cdots, p_n) \mapsto e_i\}_{i \in I}\ [\textsc{Data-Beta}] \\ \longrightarrow e_{i[\mathrm{binds}(i(p_0, p_1, \cdots, p_n), i(v_0, v_1, \cdots, v_n))]}\end{gathered}$$

Reconstructing the same data through **match** yields the original value.

$$\begin{gathered}\textbf{match}\ v\{i(\textbf{let}\ x_0, \textbf{let}\ x_1, \cdots, \textbf{let}\ x_n) \mapsto i(x_0, x_1, \cdots, x_n)\}_{i \in I}\ [\textsc{Data-Eta}] \\ \longrightarrow v\end{gathered}$$

7. Recursive Types

Recursive types are introduced by **roll** and eliminated by **unroll**. It is crucial, to represent recursive **data** types.

7.1. Introduction / Elimination

$$\frac{\Delta \mid \Gamma; E \vdash^V v \Leftarrow A[X := \mu X.A]}{\Delta \mid \Gamma; E \vdash^V \textbf{roll}\ v \Leftarrow \mu X.A}[\textsc{Mu-Intro}]$$

$$\frac{\Delta \mid \Gamma; E \vdash^V v \Rightarrow \mathrm{mu}\ X.A}{\Delta \mid \Gamma; E \vdash^V \textbf{unroll}\ v \Rightarrow A[X := \mathrm{mu}\ X.A]}[\textsc{Mu-Elim}]$$

7.2. β-reduction

$$\begin{gathered}\textbf{unroll}\ (\textbf{roll}\ v)\ [\textsc{Mu-Beta}] \\ \longrightarrow v\end{gathered}$$

8. Function

Functions are computations transforming input data into output data while having capabilities to utilize.

8.1. User-level Syntax

The user only sees the surface-level syntax:

$$\textbf{fn}\ f[X_0 : K_0, \cdots, X_n : K_n](p_0 : A_0, \cdots, p_m : A_m) : \underline{B}\ /\ \rho := M$$

where

$$p ::= \_ \mid i(p_0, \cdots, p_n) \mid \textbf{let}\ x \mid \textbf{mut}\ x$$

Also, we have the sibling call syntax:

$$\mathrm{Call} ::= u(\overline{e})$$

$$u ::= f \mid f\left[\overline{A}\right]$$

Here each argument position in $\overline{e}$ may be filled by either an ordinary expression argument or an explicit slot argument (**let** x or **mut** x). We keep the metavariable e for both forms in the call rules below.

TODO: Slot semantics and metatheory for let/mut parameters are deferred.

8.1.1. Elaboration of User-level Syntax

The user-level syntax mixes several concerns, so we elaborate it step by step.

8.1.1.1. Let Elaboration

In fact, named **fn** is just an alias for **let**. We can define **fn** without giving a name.

$$
\begin{array}{l}
\textbf{fn}\ f[X_0 : K_0, \cdots, X_n : K_n](\\
\quad p_0 : A_0, \cdots, p_m : A_m\\
) : \underline{B} \ / \ \rho := M\\
\\
\quad \textbf{let}\ f =\\
\quad\quad \textbf{produce}\ (\\
\quad\quad\quad \textbf{fn}[X_0 : K_0, \cdots, X_n : K_n](\\
\rightsquigarrow \quad\quad\quad\quad p_0 : A_0, \cdots, p_m : A_m\\
\quad\quad\quad) : \underline{B} \ / \ \rho := M)\\
\quad\quad) \text{ in } f
\end{array}
\qquad \text{[Fn-Let-Elab]}
$$

8.1.1.2. Type Parameters Elaboration

Next we'll elaborate type parameters using **forall**. Sequentially transform $X_i : K_i$ into **forall**$[X_i : K_i]$ syntax.

$$
\begin{array}{l}
\textbf{fn}[X_0 : K_0, \cdots, X_n : K_n](\\
\quad p_0 : A_0, \cdots, p_m : A_m\\
) : \underline{B} := M\\
\\
\quad \textbf{forall}[X_0 : K_0].\\
\quad \vdots\\
\quad \textbf{forall}[X_n : K_n].\\
\rightsquigarrow \quad\quad \textbf{fn}(\\
\quad\quad\quad p_0 : A_0, \cdots, p_m : A_m\\
\quad\quad) : \underline{B} := M
\end{array}
\qquad \text{[Fn-Type-Param-Elab]}
$$

8.1.1.3. Spine Elaboration

Now let's elaborate the rest – spine of function – at once.

$$
\begin{array}{l}
\textbf{fn}\ (p_0 : A_0, \cdots, p_m : A_m) : \underline{B} := M\\
\quad\quad \textbf{thunk}\ (\\
\quad\quad\quad \textbf{lambda}\ (p_0 : A_0).\\
\quad\quad\quad \vdots\\
\rightsquigarrow \quad\quad\quad \textbf{lambda}\ (p_m : A_m).\\
\quad\quad\quad\quad (M : \underline{B})\\
\quad\quad)
\end{array}
\qquad \text{[Fn-Elab-Mono]}
$$

8.1.1.4. Call Elaboration

$$\frac{u(\overline{e})}{\rightsquigarrow\ (\textbf{force}\ u)(e_0)\cdots(e_n)}\ [\textsc{Call-Elab}]$$

Note that you can pass $\textbf{let}\ x$ or $\textbf{mut}\ x$ slots.

9. CBPV Instructions

9.1. bundle (Computation-Level Π)

$\Pi_{(i\in I)}\left(i \times \underline{B_i}\right)$ is a finite computation-level record keyed by tags.

9.1.1. Bundle Introduction

Check each field and assemble them into one bundle.

$$\frac{\Delta \mid \Gamma_{i+1},\cdots,\Gamma_n; E \mid \Gamma_i \vdash^C M_i \Leftarrow \underline{B_i} \quad (0 \le i \le n)}{\Delta \mid \Gamma_0,\Gamma_1,\cdots,\Gamma_n; E \vdash^C \textbf{bundle}\ \{i_0 \mapsto M_0, i_1 \mapsto M_1, \cdots, i_n \mapsto M_n\} \Leftarrow \Pi_{(i\in I)}\left(i \times \underline{B_i}\right)}\ [\textsc{Pi-Intro}]$$

9.1.2. Bundle Elimination

Projecting a field from a synthesized bundle yields that field type.

$$\frac{\Delta \mid \Gamma; E \vdash^C M \Rightarrow \Pi_{(i\in I)}\left(i \times \underline{B_i}\right) \qquad j \in I}{\Delta \mid \Gamma; E \vdash^C M.j \Rightarrow \underline{B_j}}\ [\textsc{Pi-Elim}]$$

9.1.3. Beta Reduction

Projecting from a literal bundle returns the corresponding field.

$$(\textbf{bundle}\ \{i_0 \mapsto M_0, i_1 \mapsto M_1, \cdots, i_n \mapsto M_n\}).j \longrightarrow M_j\ [\textsc{Pi-Beta}]$$

9.2. Lambda

The function type $A \to \underline{B}$ is made out of **lambda**.

9.2.1. Introduction

$$\frac{\Delta \mid \Gamma, x : A; E \vdash^C M \Leftarrow \underline{B}}{\Delta \mid \Gamma; E \vdash^C \textbf{lambda}\ x.M \Leftarrow A \to \underline{B}}\ [\textsc{Lambda-Intro}]$$

Also you can use slot-bindings here:

TODO: Formal proof obligations for slot-bindings (**let**/**mut** parameters) will be specified later.

$$\frac{\Delta \mid \Gamma, x : A; E \vdash^C M \Leftarrow \underline{B} \qquad \text{assign-count}(M, x) \ge 1}{\Delta \mid \Gamma; E \vdash^C \textbf{lambda}\ (\textbf{let}\ x).M \Leftarrow A \to \underline{B}}\ [\textsc{Lambda-Let-Intro}]$$

$$\frac{\Delta \mid \Gamma, x : A; E \vdash^C M \Leftarrow \underline{B}}{\Delta \mid \Gamma; E \vdash^C \textbf{lambda}\ (\textbf{mut}\ x).M \Leftarrow A \to \underline{B}}\ [\textsc{Lambda-Mut-Intro}]$$

9.2.2. Elimination

Application consumes the argument and runs the function computation.

$$\frac{\Delta \mid \Gamma_1; E \vdash^C f \Rightarrow A \to \underline{B} \quad \Delta \mid \Gamma_2; E \vdash^V v \Leftarrow A}{\Delta \mid \Gamma_1, \Gamma_2; E \vdash^C f(v) \Rightarrow \underline{B}}\,[\textsc{Lambda-Elim}]$$

9.2.3. β-reduction and η-expansion

$$(\textbf{lambda}\ x.M)(v) \longrightarrow M[x := v]\ [\textsc{Lambda-Beta}]$$

$$\textbf{lambda}\ x.f(x) \longrightarrow f\ [\textsc{Lambda-Eta}]$$

9.3. Thunk and Force

thunk packages a computation as a value, and **force** restores the computation.

9.3.1. Type Formation

$$\textbf{thunk}\ \underline{B} : \text{Type}$$

9.3.2. Typing Rules

$$\frac{\Delta \mid \Gamma; E \vdash^C M \Leftarrow \underline{B}}{\Delta \mid \Gamma; E \vdash^V \textbf{thunk}\ M \Leftarrow \textbf{thunk}\ \underline{B}}\,[\textsc{Thunk-Intro}]$$

$$\frac{\Delta \mid \Gamma; E \vdash^V v \Rightarrow \textbf{thunk}\ \underline{B}}{\Delta \mid \Gamma; E \vdash^C \textbf{force}\ v \Rightarrow \underline{B}}\,[\textsc{Force-Elim}]$$

9.3.3. β-reduction and η-expansion

$$\textbf{force}\ (\textbf{thunk}\ M) \longrightarrow M\ [\textsc{Thunk-Beta}]$$

$$\textbf{thunk}\ (\textbf{force}\ v) \longrightarrow v\ [\textsc{Thunk-Eta}]$$

9.4. Produce and Let

produce injects a value into the computation type $\textbf{produce}\ A$.

$$\frac{\Delta \mid \Gamma; E \vdash^V v \Leftarrow A}{\Delta \mid \Gamma; E \vdash^C \textbf{produce}\ v \Leftarrow \textbf{produce}\ A}\,[\textsc{Produce-Intro}]$$

$$\frac{\Delta \mid \Gamma_1; E \vdash^C M \Rightarrow \textbf{produce}\ A \quad \Delta \mid \Gamma_2, x : A; E \vdash^C N \Rightarrow \underline{B}}{\Delta \mid \Gamma_1, \Gamma_2; E \vdash^C \textbf{let}\ x = M\ \text{in}\ N \Rightarrow \underline{B}}\,[\textsc{Produce-Elim}]$$

9.4.1. β-reduction

$$\textbf{let}\ x = \textbf{produce}\ v\ \text{in}\ N \longrightarrow N[x := v]\ [\textsc{Produce-Beta}]$$

10. Algebraic Effects

Algebraic effects are expressed with $\textbf{perform}_{\text{eff}}$ and interpreted by $\textbf{handle}_{\text{eff}}$.

10.1. User-level Syntax

Effects are computations passed naturally.

10.1.1. Effect Definition

You group compuations by the namespace, and can utilize same syntax as **fn**.

$$\begin{array}{l}\textbf{effect}\ \text{Test}\ \{\\ \quad \textbf{fn}\ \text{argless}() : \text{Ret1}\\ \quad \textbf{fn}\ \text{generic}[A : *]() : \text{Ret1}\\ \}\end{array}$$

So the effect declaration becomes a bundle of elaborated operation types:

$$\begin{array}{l}\textbf{effect}\ e\{\\ \quad \textbf{fn}\ \text{op}_0\left[X_{0,0} : K_{0,0}, \cdots, X_{0,n_0} : K_{0,n_0}\right]\left(p_{0,0} : A_{0,0}, \cdots, p_{0,m_0} : A_{0,m_0}\right) : B_0\\ \quad \vdots\\ \quad \textbf{fn}\ \text{op}_{n\left[X_{n,0}:K_{n,0},\cdots,X_{n,n_n}:K_{n,n_n}\right]}\left(p_{n,0} : A_{n,0}, \cdots, p_{n,m_n} : A_{n,m_n}\right) : B_n\\ \}\end{array}$$

$$\begin{array}{l}e : \Pi_{(i\in I)}\Big(\\ \quad \text{op}_i \times \Big(\\ \quad\quad \textbf{forall}\left[X_{i,0} : K_{i,0}\right].\\ \quad\quad \vdots\\ \quad\quad \textbf{forall}\left[X_{i,n_i} : K_{i,n_i}\right].\Big(\\ \rightsquigarrow \quad\quad\quad A_{i,0} \to\\ \quad\quad\quad \ldots \to\\ \quad\quad\quad A_{i,m_i} \to\\ \quad\quad\quad\quad \underline{B_i}\\ \quad \Big)\Big)\\ \Big)\end{array} \qquad \text{[Effect-Def-Elab]}$$

10.2. Effect Operation Invocation

When we have the operation eff in our effect handler context, we can invoke $\textbf{perform}_{\text{eff}}$.

$$\frac{\text{eff} : \underline{B} \in E}{\Delta \mid \Gamma; E \vdash^C \textbf{perform}_{\text{eff}} \Rightarrow \underline{B}}\,[\text{Perform}]$$

10.3. Handlers

Of course you can pass the handler so the context takes care of it.

$$\frac{\begin{array}{c}\Delta \mid \Gamma_1; E \vdash^C M \Leftarrow \underline{B}\\ \Delta \mid \Gamma_2; E, (\text{eff} : \underline{B}) \vdash^C N \Leftarrow \underline{C}\end{array}}{\Delta \mid \Gamma_1, \Gamma_2; E \vdash^C \textbf{handle}_{\text{eff}}\ \text{with}\ M\ \text{in}\ N \Leftarrow \underline{C}}\,[\text{Handler-Intro}]$$

10.4. Operational Equations

$$\mathbf{handle}_{\text{eff}}\ \text{with}\ M\ \text{in}\ \mathbf{produce}\ v \longrightarrow \mathbf{produce}\ v \quad [\textsc{Handle-Unit}]$$

$$\mathbf{handle}_{\text{eff}}\ \text{with}\ M\ \text{in}\ \mathbf{perform}_{\text{eff}} \longrightarrow M \quad [\textsc{Handle-Perform-Beta}]$$

11. Meta-Theory Roadmap

Priority lemmas.

11.1. Substitution

Goal: prove substitution while preserving linear/accounting side conditions.

Expected effect: enables modular proofs by replacing variables with well-typed terms safely.

11.2. Preservation

Goal: prove that one-step reduction preserves typing and resource/effect invariants.

Expected effect: guarantees type/resource invariants are maintained by every reduction step.

11.3. Progress

Goal: prove that well-typed closed programs are either terminal forms or can reduce.

Expected effect: rules out stuck well-typed closed programs (except designated terminal forms).

11.4. NbE Soundness

Goal: prove soundness of NbE-based definitional equality.

Expected effect: provides a trustworthy basis for definitional equality checks.

11.5. NbE (Normalization by Evaluation)

TODO: NbE formalization is deferred.

Expected benefits:

- Canonical decision procedure for definitional equality.
- Cleaner convertibility checks in type checking.
- Better foundation for future optimizations (normalization-driven simplification).

This document is the base spec for machine-checked proofs.

12. References

Bibliography

Brachthäuser, Jonathan Immanuel, Philipp Schuster, and Klaus Ostermann. 2020. "Effekt: Capability-Passing Style for Type- and Effect-Safe, Extensible Effect Handlers in Scala." *Journal of Functional Programming* 30 : e8. https://doi.org/10.1017/S0956796820000027.

Cordonier, Raphael, Theo Winterhalter, Matthieu Casanova, Sébastien Doeraene, Paolo G. Giarrusso, and Frédéric Bour. 2021. "Native Implementation for Mutable Value Semantics.". https://arxiv.org/abs/2106.12678.

Cordonier, Raphael, Theo Winterhalter, Matthieu Casanova, Sébastien Doeraene, Paolo G. Giarrusso, and Frédéric Bour. 2022. "Implementation Strategies for Mutable Value Semantics." *Journal of Object Technology* 21 (2): 2:1–24. https://doi.org/10.5381/jot.2022.21.2.a2.

Reinking, Alex, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. “Perceus: Garbage-Free Reference Counting with Reuse.” In “Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.” Special issue, *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 96–111. https://doi.org/10.1145/3453483.3454032.